Imperial College London
Department of Computing

Semantic Types for Class-based Objects

Reuben N. S. Rowe

July 2012

Supervised by Dr. Steffen van Bakel

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Abstract

We investigate semantics-based type assignment for class-based object-oriented programming. Our motivation is developing a theoretical basis for practical, expressive, type-based analysis of the functional behaviour of object-oriented programs. We focus our research using Featherweight Java, studying two notions of type assignment:- one using *intersection* types, the other a 'logical' restriction of recursive types.

We extend to the object-oriented setting some existing results for intersection type systems. In doing so, we contribute to the study of denotational semantics for object-oriented languages. We define a model for Featherweight Java based on *approximation*, which we relate to our intersection type system via an Approximation Result, proved using a notion of reduction on typing derivations that we show to be strongly normalising. We consider restrictions of our system for which type assignment is decidable, observing that the implicit recursion present in the class mechanism is a limiting factor in making practical use of the expressive power of intersection types.

To overcome this, we consider type assignment based on recursive types. Such types traditionally suffer from the inability to characterise convergence, a key element of our approach. To obtain a semantic system of recursive types for Featherweight Java we study Nakano's systems, whose key feature is an approximation modality which leads to a 'logical' system expressing both functional behaviour and convergence. For Nakano's system, we consider the open problem of type inference. We introduce *insertion* variables (similar to the expansion variables of Kfoury and Wells), which allow to infer when the approximation modality is required. We define a type inference procedure, and conjecture its soundness based on a technique of Cardone and Coppo. Finally, we consider how Nakano's approach may be applied to Featherweight Java and discuss how intersection and logical recursive types may be brought together into a single system.

I dedicate this thesis to the memory of my grandfather, David W. Hyam, who I very much wish was here to see it.

Acknowledgements

I would like to acknowledge the help, input and inspiration of a number of people who have all helped me, in their own larger and smaller ways, to bring this thesis into being.

Firstly, my supervisor, Steffen, deserves my sincere thanks for all his guidance over the past five years. If it weren't for him, I would never have embarked upon this line of research that I have found so absolutely fascinating. Also, if it weren't for him I would probably have lost myself in many unnecessary details – his gift for being able to cut through to the heart of many a problem has been invaluable when I couldn't see the wood for the trees.

I would also like to thank my second supervisor, Sophia Drossopoulou, who has always been more than willing to offer many insightful suggestions and opinions, and above all a friendly ear.

I owe a huge debt to my parents, Peter and Catherine, and all of my family who have been so supportive and encouraging of my efforts. They have always brought me up to believe that I can achieve everything that I put my mind to, and without them I would never have reached this point. I must thank, in particular, my late aunt Helen whose financial legacy, in part, made my higher education aspirations a reality.

My wonderful girlfriend Carlotta has shared some of this journey with me, and this thesis is just as much hers as it is mine, having borne my distractions and preoccupations with grace. Her encouragement and faith in me has carried me through more than a few difficult days.

I thank Jayshan Raghunandan, Ioana Boureanu, Juan Vaccari and Alex Summers for their friendship, interesting discussions, and for making the start of my PhD such an enjoyable experience. I especially thank Alex, who may be one of the most intelligent, friendly and modest people I have met.

Finally, I would like to extend my appreciation to the SLURP group, and the staff and students of the Imperial College Department of Computing, who have all contributed to my wider academic environment.

Contents

1. Introduction

Type theory constitutes a form of abstract reasoning, or interpretation of programs. It provides a way of classifying programs according to the kinds of values they compute [88]. More generally, type systems specify schemes for associating syntactic entities (types) with programs, in such a way that they reflect abstract properties about the behaviour of those programs. Thus, typeability effectively *guarantees* well-behavedness, as famously stated by Milner when he said that "Well-typed programs can't go wrong" [79], where 'wrong' is a semantic concept defined in that paper.

In type theory, systems in the *intersection type discipline* (ITD) stand out as being able to express both the functional behaviour of programs and their termination properties. Intersection types were first introduced in [40] as an extension to Curry's basic functionality theory for the Lambda Calculus (λ-calculus or LC) [45]. Since then, the intersection type approach has been extended to many different models of computation including Term Rewriting Systems (TRS) [14, 15], sequent calculi [101, 102], object calculi [18, 13], and concurrent calculi [37, 87] proving its versatility as an analytical technique for program verification. Furthermore, intersection types have been put to use in analysing not just termination properties but in dead code analysis [47], strictness analysis [70], and control-flow analysis [17]. It is obvious, then, that intersection types have a great potential as a basis for expressive, type-based analysis of programs.

The expressive power of intersection types stems from their deep connection with the mathematical, or denotational, semantics of programming languages [96, 97]. It was first demonstrated in [20] that the set of intersection types assignable to any given term forms a filter, and that the set of such filters forms a domain, which can be used to give a denotational model to the λ-calculus. Denotational models for λ-calculus were connected with a more 'operational' view of computation in [105] via the concept of *approximant*, which is a term approximating the final result of a computation. Approximants essentially correspond to Böhm trees [19], and a λ-model can be given by considering the interpretation of a term to be the set of all such approximations of its (possibly infinite) normal form. Intersection types have been related to these *approximation* semantics (see e.g. [95, 9, 15]) through approximation *results*. These results consider the typeability of approximants and relate the typeability of a term with the typeability of its approximants, showing that every intersection type that can be assigned to a term can also be assigned to one of its approximants and vice-versa. This general result relates intersection types to the operational behaviour of terms, and shows that intersection types completely characterise the behavioural properties of programs.

The object-oriented paradigm (OO) is one of the principal styles of programming in use today. Object-oriented concepts were first introduced in the 1960s by the language Simula [46], and since then have been incorporated and extended by many programming languages from Smalltalk [60], C++ [98], Java [61] and ECMAscript (or Javascript) [68], through to C# [69], Python [103], Ruby [1] and Scala [56], amongst many others. The basic premise is centred on the concept of an *object*, which is an entity that binds together state (in the form of data fields) along with the functions or operations that act upon it,

such operations being called *methods*. Computation is mediated and carried out by objects through the act of sending messages to one another which *invoke* the execution of their methods.

Initial versions of object-oriented concepts were *class*-based, a style in which programmers write classes that act as fixed templates, which are *instantiated* by individual objects. This style facilitates a notion of specialisation and sharing of behaviour (methods) through the concept of *inheritance* between classes. Later, a pure object, or prototype-based approach was developed in which methods and fields can be added to (and even removed from) individual objects at any time during execution. Specialisation and behaviour-sharing is achieved in this approach via *delegation* between objects. Both class-based and object-based approaches have persisted in popularity. A second dichotomy exists in the object-oriented world, which is that between (strongly) typed and untyped (or *dynamically* typed) languages. Strongly typed oo languages provide the benefit that guarantees can be given about the behaviour of the programs written in them. From the outset, class-based oo languages have been of the typed variety; since objects must adhere to a pre-defined template or interface, classes naturally act as types that specify the (potential) behaviour of programs, as well as being able to classify the values resulting from their execution, i.e. objects. As object-oriented programmers began to demand a more flexible style, object-based languages were developed which did not impose the uncompromising rigidity of a type system.

From the 1980s onwards, researchers began to look for ways of describing the object-oriented style of computation from a theoretical point of view. This took place from both an operational perspective, as well as a (denotational) semantic one. For example, Kamin [72] considered a denotational model for Smalltalk, while Reddy worked on a more language-agnostic denotational approach to understanding objects [92]. They subsequently unified their approaches [73]. On the other hand, a number of operational models were developed, based on extending the λ-calculus with records and interpreting or encoding objects and object-oriented features in these models. These notably include work by Cardelli [31, 33, 32], Mitchell [81], Cook and Palsberg [39], Fisher et al. [58, 59], Pierce et al. [89, 63], and Abadi, Cardelli and Viswanathan [3, 104]. As well as to give an operational account of oo, the aim of this work was also to understand the object-oriented paradigm on a more fundamental, *type-theoretic* level. Many of these operational models have been accompanied by a denotational approach in which the semantics of both terms and types are closely linked, and related to System F-typed λ-models.

While this was a largely successful programme, and led to a much deeper theoretical understanding of object-oriented concepts, the encoding-based approach proved a complex one requiring, at times, attention to 'low-level' details. This motivated Abadi and Cardelli to develop the ς-calculus, in which objects and object-oriented mechanisms were 'first-class' entities [2]. Abadi and Cardelli also defined a denotational PER model for this calculus, which they used to show that well-typed expressions do not correspond to the Error value in the semantic domain, i.e. do not go "wrong". Similar to this, Bruce [27] and Castagna [36] have also defined typed calculi with object-oriented primitives.

While these calculi represent comprehensive attempts to capture the plethora of features found in object-oriented languages, they are firmly rooted in the object-based approach to oo. They contain many features (e.g. method override) which are not expressed in the class-based variant. An alternative model specifically tailored to the class-based approach was developed in Featherweight Java (FJ) [66]. This has been used as the basis for investigating the theoretical aspects of many proposed extensions to class-based mechanisms (e.g. [65, 54, 21, 76]). FJ is a purely operational model, however, and it must be remarked that there has been relatively little work in treating class-based oo from a denotational

position. Studer [99] defined a semantics for Featherweight Java using a model based on Feferman's Explicit Mathematics formalism [57], but remarks on the weakness of the model. Alves-Foss [4] has done work on giving a denotational semantics to the full Java language. His system is impressively comprehensive but, as far as we can see, it is not used for any kind of analysis - at least not in [4]. Burt, in his PhD thesis [30], builds a denotational model for a stateful, featherweight model of Java based on game semantics, via a translation to a PCF-like language.

Despite the great wealth of semantic and type-theoretic research into the foundations of object-oriented programming, the intersection type approach has not been brought to bear on this problem until more recently. De'Liguoro and van Bakel have defined and developed an intersection type system for the ς-calculus, and show that it gives rise to a denotational model [11]. The key aspect of their system is that it assigns intersection types to *typed* ς-calculus terms. As such, their intersection types actually constitute logical predicates for typed terms. They also capture the notion of contextual equivalence, and characterise the convergence of terms which is shown by considering a realizability interpretation of intersection types.

In this thesis, we continue that program of research by applying the intersection type discipline to the *class-based* variant of oo, as expressed in the operational model FJ. Our approach will be to build an approximation-based denotational model, and show an approximation result for an intersection type assignment system. Thus, we aim to develop a type-based characterisation of the computational behaviour of class-based object-oriented programs. Our technique for showing such an approximation result will be based upon defining a notion of reduction for intersection type assignment *derivations* and showing it to be strongly normalising, a technique which has been employed for example in [15, 10]. This notion of reduction can be seen as an analogue of cut-elimination in formal logics. Using this result, we show that our intersection type system characterises the convergence of terms, as well as providing an analysis of functional behaviour.

One of our motivations for undertaking this programme of research is to develop a strong theoretical basis for the development of practical and expressive tools that can both help programmers to reason about the code that they write, and verify its correct behaviour. To that end, a significant part of this research pertains to type *inference*, which is the primary mechanism for implementing type-based program analysis. The strong expressive capabilities of the intersection type discipline are, in a sense, *too* powerful: since intersection types completely characterise strongly normalising terms, full type assignment is undecidable. The intersection type discipline has the advantage, however, that decidable restrictions exist which preserve the strong semantic nature of type assignment. We investigate such a restriction for our system and show it to be decidable by giving a principal typings result. We observe, however, that it is not entirely adequate for the *practical* analysis of class-based oo programs: the implicit recursive nature of the class mechanism means that we cannot infer informative types for 'typically' object-oriented programs.

To enhance the practicality of our type analysis we look to a 'logical' variant of recursive types, due to Nakano [83, 84], which is able to express the convergence properties of terms through the use of a modal type operator $\bullet$, or 'bullet', that constrains the folding of certain recursive types during assignment. This allows their incorporation into the semantic framework given by our intersection type treatment. Nakano's system is presented for the λ-calculus and leaves unanswered the question of the decidability of its type assignment relation. Furthermore, although he does discuss its potential applicability to the

analysis of oo programs, details of how this may be achieved are elided.

We address each of these two issues in turn. First, we consider a unification-based type inference procedure. We are inspired by the *expansion variables* of Kfoury and Wells [75], used to facilitate type inference for intersection types, and introduce *insertion* variables which we use to infer when the modal bullet operator is required to unify two types. In an extension of a technique due to Cardone and Coppo [35], we define our unification procedure through a derivability relation on unification judgements which we argue is decidable, thus leading to a terminating unification algorithm. Secondly, we give a type system which assigns logical recursive types to FJ programs. We do not present formal results for that system in this thesis, leaving the proof of properties such as convergence and approximation for future work. We discuss the typeability of various illustrative examples using this system, as well as how we might extend the type inference algorithm from the λ-calculus setting to the object-oriented one. Finally, we consider how to incorporate both intersection types and logical recursive types within a single type system.

Outline of the Thesis

This thesis naturally splits into two parts - chapters 2 through to 7 are concerned with intersection type assignment, while chapters 8 to 10 deal with Nakano's logical recursive types and how they can be applied to the object-oriented paradigm.

In Chapter 2, we give a short introduction to the intersection type discipline, as it applies to Lambda Calculus and the object ς-calculus, reviewing the main results admitted by the intersection type systems for these computational models. Chapter 3 presents the class-based model of object-orientation that we focus on - Featherweight Java - and defines a system for assigning intersection types to Featherweight Java Programs. The main result of this chapter is that assignable types are preserved under conversion. We continue, in Chapter 4, by considering a notion of reduction on intersection type derivations and proving it to be strongly normalising. This lays the groundwork for our Approximation Result which links our notion of type assignment with the denotational semantics of programs, and forms the subject of Chapter 5. In Chapter 6 we consider some example programs and how to type them using our intersection type system, including an encoding of Combinatory Logic. We also make a detailed comparison between the intersection type system and the nominally-based approach to typing class-based oo. We finish the first part of the thesis by considering, in Chapter 7, a type inference procedure.

The inadequacies of intersection type inference suggest a different approach to typing object-oriented programs using recursive types, which we investigate in the second half of the thesis. We begin by giving an explanation of the 'illogical' nature of conventional systems of recursive types, and reviewing Nakano's modal logic-inspired systems of recursive types in Chapter 8. In Chapter 9 we describe a procedure for inferring types in a variant of Nakano's system. We sketch a proof of its decidability and consider examples suggesting the generality of our approach. Lastly, in Chapter 10, we describe how this can be applied to oo by defining a type system assigning Nakano-style recursive types to Featherweight Java. We revisit the example programs of Chapter 6 and demonstrate how the system of recursive types handles them. We also consider how Nakano types might be integrated with intersection types. We conclude the thesis in Chapter 11, giving a summary of the contributions of our work, and discussing how it may be extended in the future.

Notational Preliminaries

Throughout the thesis we will make heavy use of the following notational conventions for dealing with sequences of syntactic entities.

1. A sequence s of n elements $a_1, \ldots, a_n$ is denoted by $\vec{a}_n$; the subscript can be omitted when the exact number of elements in the sequence is not relevant.
2. We write $a \in \vec{a}_n$ whenever there exists some $i \in \{1, \ldots, n\}$ such that $a = a_i$. Similarly, we write $a \notin \vec{a}_n$ whenever there does *not* exist an $i \in \{1, \ldots, n\}$ such that $a = a_i$.
3. We use $\overline{n}$ (where n is natural number) to represent the sequence $1, \ldots, n$.
4. For a constant term $\mathtt{c}, \vec{\mathtt{c}}_n$ represents the sequence of n occurrences of $\mathtt{c}$.
5. The empty sequence is denoted by ϵ, and concatenation on sequences by $s_1 \cdot s_2$.

Part I.

Simple Intersection Types

2. The Intersection Type Discipline

In this chapter, we will give a brief overview of the main details and relevant results of the intersection type discipline by presenting an intersection type system for the λ-calculus. We will also present a (restricted version) of the intersection type system of [13] for the ς-calculus, with the aim of better placing our research in context, and to be able to make comparisons later on.

Intersection types were first developed for the λ-calculus in the late '70s and early '80s by Coppo and Dezani [41] and extended in, among others, [42, 20]. The motivation was to extend Curry's basic functional type theory [45] in order to be able to type a larger class of 'meaningful' terms; that is, all terms with a head normal form.

The basic idea is surprisingly simple: allowing term variables to be assigned more than one type. This ostensibly modest extension belies a greater generality since the different types that we are now allowed to assign to term variables need not be unifiable - that is, they are allowed to be fundamentally different. For example, we may allow to assign to a variable both a type variable φ (or a ground type) *and* a function type whose domain is that very type variable (e.g. $\varphi \to \sigma$). This is interpreted in the functional theory as meaning that the variable denotes both a function and an argument that can be provided to that function. In other words, it allows to type the *self-application* $x\,x$. This leads to great expressive power: using intersection types, all and only strongly normalising terms can be given a type. By adding a type constant ω, assignable to all terms, the resulting system is able to characterise strongly normalising, weakly normalising, and head normalising terms.

2.1. Lambda Calculus

The λ-calculus, first introduced by Church in the 1930s [38], is a model of computation at the core of which lies the notion of function. It has two basic notions: (function) abstraction and (function) application, and from these two elements arises a model which fully captures the notion of computability (it is able to express all computable functions). The λ-calculus forms the basis on the functional programming paradigm, and languages such as ML [80] are based directly upon it.

Definition 2.1 (λ-terms)**.** *Terms M, N, etc., in the λ-calculus are built from a set of term* variables *(ranged over by x, y, z, etc.), a term constructor λ which* abstracts *over a named variable, and the* application *of one term to another.*

$$M, N \quad ::= \quad x \quad | \quad (\lambda x.M) \quad | \quad (M\,N)$$

Repeated abstractions can be abbreviated (i.e. $\lambda x.\lambda y.\lambda z.M$ is written as $\lambda xyz.M$) and left-most, outer-most brackets in function applications can be omitted.

In the term $\lambda x.M$, the variable x is said to be *bound*. If a variable does not appear within the scope of a λ that names it, then the variable is said to be *free*. The notation $M[N/x]$ denotes the λ-term obtained

by replacing all the (free) occurrences of x in M by N. During this substitution, the free variables of N should not inadvertently become bound, and if necessary the free variables of N and the bound variables of M can be (consistently) renamed so that they are separate (this process is called α-conversion).

Computation is then expressed as a formal *reduction* relation, called β-reduction, over terms. The basic operation of computation is to reduce terms of the form $(\lambda x.M)N$, called *redexes* (or reducible expressions), by substituting the term N for all occurrences of the bound variable x in M.

Definition 2.2 (β-reduction)**.** *The reduction relation $\rightarrow_\beta$, called β-reduction, is the smallest preorder on λ-terms satisfying the following conditions:*

$$(\lambda x.M)N \rightarrow_\beta M[N/x]$$

$$M \rightarrow_\beta N \Rightarrow \begin{cases} P\,M \rightarrow_\beta P\,N \\ M\,P \rightarrow_\beta N\,P \\ \lambda x.M \rightarrow_\beta \lambda x.N \end{cases}$$

This reduction relation induces an equivalence on λ-terms, called β-equivalence or β-convertibility, and this equivalence captures a certain notion of equality between functions. In one sense, the study of the λ-calculus can be seen as the study of this equivalence.

Definition 2.3 (β-equivalence)**.** *The equivalence relation $=_\beta$ is the smallest equivalence relation on λ-terms satisfying the condition:*

$$M \rightarrow_\beta N \Rightarrow M =_\beta N$$

The reduction behaviour of λ-terms can be characterised using variations on the concept of *normal form*, expressing when the result of computation has been achieved.

Definition 2.4 (Normal Forms and Normalisability)**.**

1. *A term is in* head-normal form *if it is in the form $\lambda x_1 \cdots x_n.yM_1 \cdots M_{n'}$ $(n, n' \geq 0)$. A term is in* weak *head normal form if it is of the form $\lambda x.M$.*

2. *A term is in* normal form *if it does not contain a redex. Terms in normal form can be defined by the grammar:*

$$N \quad ::= \quad x \quad | \quad \lambda x.N \quad | \quad xN_1 \cdots N_n \quad (n \geq 0)$$

   By definition, a term in normal form is also in head-normal form.

3. *A term is (weakly)* head normalisable *whenever it has a (weak) head normal form, i.e. if there exists a term N in (weak) head normal form such that $M =_\beta N$.*

4. *A term is* normalisable *whenever it has a normal form. A term is* strongly *normalisable whenever it does not have any infinite reduction sequence*

$$M \rightarrow_\beta M' \rightarrow_\beta M'' \rightarrow_\beta \ldots$$

   Notice that by definition, all strongly normalisable terms are normalisable, and all normalisable terms are head-normalisable.

Intersection types are formed using the type constructor $\cap$. The intersection type system that we will present here is actually the *strict* intersection type system of van Bakel [7], which only allows

intersections to occur on the left-hand sides of function types. This represents a restricted type language with respect to e.g. [20], but is still fully expressive.

Definition 2.5 (Intersection Types [7, Def. 2.1])**.** *The set of intersection types (ranged over by ϕ, ψ) and its (strict) subset of* strict *intersection types (ranged over by σ, τ) are defined by the following grammar:*

$$\begin{array}{lcl} \sigma,\tau & ::= & \varphi \quad | \quad \phi \to \sigma \\ \phi,\psi & ::= & \sigma_1 \cap \ldots \cap \sigma_n \quad (n \geq 0) \end{array}$$

where φ ranges over a denumerable set of type variables*; we will use the notation ω as a shorthand for $\sigma_1 \cap \ldots \cap \sigma_n$ where $n = 0$, i.e. the* empty *intersection.*

Intersection types are assigned to λ-terms as follows:

Definition 2.6 (Intersection Type Assignment [7, Def. 2.2])**.**

1. *A* type statement *is of the form $M : \phi$ where M is a λ-term and ϕ is an intersection type. The term M is called the* subject *of the statement.*

2. *A* basis *B is a finite set of type statements such that the subject of each statement is a* unique *term variable. We write $B, x : \phi$ for the basis $B \cup \{x : \phi\}$ where x does not appear as the subject of any statement in B.*

3. *Type assignment $\vdash$ is a relation between bases and type statements, and is defined by the following natural deduction system.*

$$(\cap E) : \frac{}{B, x : \sigma_1 \cap \ldots \cap \sigma_n \vdash x : \sigma_i}\ (n > 0, 1 \leq i \leq n) \qquad (\to I) : \frac{B, x : \phi \vdash M : \sigma}{B \vdash \lambda x.M : \phi \to \sigma}$$

$$(\cap I) : \frac{B \vdash M : \sigma_1 \quad B \vdash M : \sigma_n}{B \vdash M : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 0) \qquad (\to E) : \frac{B \vdash M : \phi \to \sigma \quad B \vdash N : \phi}{B \vdash MN : \sigma}$$

We point out that, alternatively, ω could be defined to be a type *constant*. Defining it to be the empty intersection, however, simplifies the presentation of the type assignment rules, in that we can combine the rule that assigns ω to any arbitrary term with the intersection introduction rule $(\cap I)$, of which it is now just a special case. Another justification for defining it to be the empty intersection is semantic: when considering an interpretation $[\![\cdot]\!]$ of types as the set of λ-terms to which they are assignable, we have the property that for all strict types σ_1, …, σ_n

$$[\![\sigma_1 \cap \ldots \sigma_n]\!] \subseteq [\![\sigma_1 \cap \ldots \sigma_{n-1}]\!] \subseteq \ldots \subseteq [\![\sigma_1]\!]$$

It is natural to extend this sequence with $[\![\sigma_1]\!] \subseteq [\![\ ]\!]$, and therefore to define that the semantics of the empty intersection is the entire set of λ-terms; this is justified, since via the rule $(\cap I)$ we have $B \vdash M : \omega$ for all terms M.

In the intersection type discipline, types are preserved under conversion, an important semantic property.

Theorem 2.7 ([7, Corollary 2.11])**.** *Let $M =_\beta N$; then $B \vdash M : \sigma$ if and only if $B \vdash N : \sigma$.*

As well as expressing a basic functionality theory (i.e. a theory of λ-terms as functions), intersection type systems for λ-calculus also capture the termination, or convergence properties of terms.

Theorem 2.8 (Characterisation of Convergence, [7, Corollary 2.17 and Theorem 3.29])**.**

1. *$B \vdash M : \sigma$ with $\sigma \neq \omega$ if and only if M has a head-normal form.*
2. *$B \vdash M : \sigma$ with $\sigma \neq \omega$ and B not containing ω if and only if M has a normal form.*
3. *$B \vdash M : \sigma$ without ω begin used at all during type assignment if and only if M is strongly normalisable.*

As mentioned in the introduction, the intersection type discipline gives more than just a termination analysis and a theory of functional equality. By considering an approximation semantics for λ-terms, we see a deep connection between intersection types and the computational behaviour of terms.

The notion of approximant was first introduced by Wadsworth in [105]. Essentially, approximants are partially evaluated expressions in which the locations of incomplete evaluation (i.e. where reduction *may* still take place) are explicitly marked by the element $\bot$; thus, they *approximate* the result of computations; intuitively, an approximant can be seen as a 'snapshot' of a computation, where we focus on that part of the resulting program which will no longer change (i.e. the observable *output*).

Definition 2.9 (Approximate λ-Terms [10, Def. 4.1])**.** 1. *The set of* approximate *λ-terms is the conventional set of λ-terms extended with an extra constant, $\bot$. It can be defined by the following grammar:*

$$M, N \quad ::= \quad \bot \quad | \quad x \quad | \quad (\lambda x.M) \quad | \quad (M\,N)$$

Notice that the set of λ-terms is a subset of the set of approximate λ-terms.

2. *The reduction relation $\rightarrow_\beta$ is extended to approximate terms by the following rules*

$$\lambda x.\bot \rightarrow_{\beta\bot} \bot \qquad \bot\, M \rightarrow_{\beta\bot} \bot$$

3. *The set of* normal forms *with respect to the extended reduction relation $\rightarrow_{\beta\bot}$ is characterised by the following grammar:*

$$A \quad ::= \quad \bot \quad | \quad \lambda x.A \quad (A \neq \bot) \quad | \quad x A_1 \ldots A_n \quad (n \geq 0)$$

Approximants are approximate normal forms which match the structure of a λ-term up to occurrences of $\bot$. Since, for approximate normal forms, no further reduction is possible, their structure is fixed. This means that they (partially) represent the normal form of a λ-term and thus, they 'approximate' the output of the computation being carried out by the term.

Definition 2.10 (Approximants [10, Def. 4.2])**.** 1. *The relation $\sqsubseteq$ is defined as the smallest relation on approximate λ-terms satisfying the following:*

$$\bot \sqsubseteq M \qquad (\textit{for all } M)$$

$$M \sqsubseteq N \Rightarrow \lambda x.M \sqsubseteq \lambda x.N$$

$$M \sqsubseteq N \;\&\; M' \sqsubseteq N' \Rightarrow M\,M' \sqsubseteq N\,N'$$

2. *The set of* approximants *of a λ-term M is denoted by $\mathcal{A}(M)$, and is defined by* $\mathcal{A}(M) = \{A \mid \exists N.M =_\beta N \ \& \ A \sqsubseteq N\}$.

Notice that if two terms are equivalent, $M =_\beta N$, then they have the same set of approximants $\mathcal{A}(M) = \mathcal{A}(N)$. Thus, we can give a semantics of λ-calculus by interpreting a term by its set of approximants.

We can define a notion of intersection type assignment for approximate λ-terms (and thus approximants themselves), with little difficulty: exactly the same rules can be applied, we simply allow approximate terms to appear in the type statements. Since we do not add a specific type assignment rule for the new term $\bot$, this means that the only type that can be assigned to $\bot$ is ω, the empty intersection. Equipped with a notion of type assignment for approximants, the intersection type system admits an *Approximation Result*, which links intersection types with approximants:

Theorem 2.11 (Approximation Result, [7, Theorem 2.22(ii)])**.** *$B \vdash M : \sigma$ if and only if there exists some $A \in \mathcal{A}(M)$ such that $B \vdash A : \sigma$.*

This result states that every type which can be assigned to a term can also be assigned to one of its approximants. This is a powerful result because it shows that the intersection types assignable to a term actually *predict* the outcome of the computation, the normal form of the term. To see how they achieve this, recall that we said the intersection type assignment system is syntax-directed. This means that for each different form that a type may take (e.g. function type, intersection, etc.) there is exactly one rule which assigns that form of type to a λ-term. Thus, the structure of a type exactly dictates the structure of the approximate normal form that it can be assigned to.

2.2. Object Calculus

The ς-calculus [2] was developed by Abadi and Cardelli in the 1990s with the objective of providing a minimal, fundamental calculus capable of modelling as many features found in object-oriented languages as possible. It is fundamentally an *object-based* calculus, and incorporates the ability to directly update objects by adding and overriding methods as a primitive operation, however it is capable of modelling the class mechanism showing that, in essence, objects are more fundamental than classes. Starting from an untyped calculus, Abadi and Cardelli define a type system of several tiers, ranging from simple, first order system of object types through to a sophisticated second order system with subtyping, as well as developing an equational theory for objects. Using their calculus, they successfully gave a comprehensive theoretical treatment to complex issues in object-oriented programming.

The full type system of Abadi and Cardelli is extensive, and here we only present a subset which is sufficient to demonstrate its basic character and how intersection types have been applied to it.

Definition 2.12 (ς-calculus Syntax)**.** *Let l range over a set of (method) labels. Also, let x, y, z range over a set of term variables and X, Y, Z range over a set of type variables. Types and terms in the ς-calculus*

are defined as follows:

Types

$$A, B \;::=\; X \;\mid\; [l_1{:}B_1, \ldots, l_n{:}B_n] \quad (n \geq 0) \;\mid\; A \to B \;\mid\; \mu X.A$$

Terms

$$\begin{aligned} a, b, c \;::=\;& x \;\mid\; \lambda x^A.b \;\mid\; a\,b \\ \mid\;& [l_1{:}\varsigma(x^{A_1})b_1, \ldots, l_n{:}\varsigma(x^{A_n})b_n] \\ \mid\;& a.l \;\mid\; a.l \Leftarrow \varsigma(x^A)b \\ \mid\;& \mathsf{fold}(A, a) \;\mid\; \mathsf{unfold}(a) \end{aligned}$$

Values

$$v \;::=\; [l_1{:}\varsigma(x^{A_1})b_1, \ldots, l_n{:}\varsigma(x^{A_n})b_n] \;\mid\; \lambda x.a$$

We use $[l_i{:}B_i{}^{\,i \in 1..n}]$ *to abbreviate the type* $[l_1{:}B_1, \ldots, l_n{:}B_n]$*, and* $[l_i{:}\varsigma(x^{A_i})b_i{}^{\,i \in 1..n}]$ *to abbreviate the term* $[l_1{:}\varsigma(x^{A_1})b_1, \ldots, l_n{:}\varsigma(x^{A_n})b_n]$*, where we assume that each label* l_i *is distinct.*

Thus, we have objects $[l_i{:}\varsigma(x^{A_i})b_i{}^{\,i \in 1..n}]$ which are collections of methods of the form $\varsigma(x^A)b$. Methods can be invoked by the syntax $a.l$, or overridden with a new method using the syntax $a.l \Leftarrow \varsigma(x^A)b$. Like λ, ς is a *binder*, so the term variable x is bound in the method $\varsigma(x^A)b$. The ς binder plays a slightly different role, however, which is to refer to the object that contains the method (the self, or receiver) within the body of the method itself. The intended semantics of this construction is that when a method is invoked, using the syntax $[l_i{:}\varsigma(x^{A_i})b_i{}^{\,i \in 1..n}].l_i$, the result is given by returning the method body and replacing all occurrences of the self-bound variable by the object on which the method was invoked. We will see this more clearly when we define the notion of reduction below.

In this presentation, the syntax of the λ-calculus is embedded into the ς-calculus, and so we more precisely be said to be presenting the $\varsigma\lambda$-calculus. Embedding the λ-calculus does not confer any additional expressive power, however, since it can be encoded within the pure ς-calculus. For convenience, though, we will use the embedded, rather than the encoded, λ-calculus. Then, λ-abstractions can be used to model methods which take arguments. Fields can be modelled as methods which do not take arguments. For simplicity, we have not included any term constants in this presentation, although these are incorporated in the full treatment, and may contain elements such as numbers, boolean values, etc. Recursive types $\mu X.A$ can be used to type objects containing methods which return self, an important feature in the object-oriented setting. Notice that folding and unfolding of recursive types is syntax-directed, using the terms $\mathsf{fold}(A, a)$ and $\mathsf{unfold}(a)$.

The ς-calculus is a typed calculus in which types are embedded into the syntax of terms. An *un*typed version of the calculus can be obtained simply by erasing this type information. As with the λ-calculus, in the ς-calculus we have notion of free and bound variables, and of substitution which again drives reduction. For uniformity of notation, we will denote substitution in the ς-calculus in the same way as we did for λ-calculus in the previous section. Specifically, the notation $a[b/x]$ will denote the term obtained by replacing all the free occurrence of the term variable x in the term a by the term b. Similarly, the type constructor μ is a binder of type variables X, and we assume the same notation to denote substitution of types.

Definition 2.13 (Reduction)**.** *1. An* evaluation context *is a term with a hole* [_]*, and is defined by the*

following grammar:

$$\mathcal{E}[\_] \quad ::= \quad \_ \quad | \quad \mathcal{E}[\_].l \quad | \quad \mathcal{E}[\_].l \Leftarrow \varsigma(x^A)b$$

$\mathcal{E}[a]$ denotes filling the hole in $\mathcal{E}$ with a.

2. *The one-step reduction relation $\to$ on terms is the smallest binary relation defined by the following rules:*

$$\begin{aligned}
(\lambda x^A.a)b &\to a[b/x] \\
[l_i{:}\varsigma(x^{A_i})b_i{}^{i\in 1..n}].l_j &\to b_j[[l_i{:}\varsigma(x^{A_i})b_i{}^{i\in 1..n}]/x_j] \quad (1 \le j \le n) \\
[l_i{:}\varsigma(x^{A_i})b_i{}^{i\in 1..n}].l_j \Leftarrow \varsigma(x^A)b &\to \\
&[l_1{:}\varsigma(x_1^{A_1})b_1, \ldots, l_j{:}\varsigma(x^A)b, \ldots, l_n{:}\varsigma(x_n^{A_n})b_n] \quad (1 \le j \le n) \\
a \to b &\Rightarrow \mathcal{E}[a] \to \mathcal{E}[b]
\end{aligned}$$

3. *The relation $\to^*$ is the reflexive and transitive closure of $\to$.*

4. *If $a \to^* v$ then we say that a* converges *to the value v, and write $a \Downarrow v$.*

Types are now assigned to terms as follows.

Definition 2.14 (ς-calculus Type Assignment)**.** 1. *A type statement is of the form $a : A$ where a is a term and A is a type. The term a is called the* subject *of the statement.*

2. *An environment E is a finite set of type statements in which the subject of each statement is a* unique *term variable. The notation $E, x : A$ stands for the environment $E \cup \{x : A\}$ where x does not appear as the subject of any statement in E.*

3. *Types assignment is relation $\vdash$ between environments and type statements, and is defined by the following natural deduction system:*

(*Val x*) :
$$\frac{}{E, x{:}A \vdash x : A}$$

(*Val Object*) : (*where* $A = [l_i{:}B_i{}^{i\in 1..n}]$)
$$\frac{E, x : A \vdash b_i : B_i \quad (\forall\, 1 \le i \le n)}{E \vdash [l_i{:}\varsigma(x^A)b_i{}^{i\in 1..n}] : A}$$

(*Val Select*) :
$$\frac{E \vdash a : [l_i{:}B_i{}^{i\in 1..n}]}{E \vdash a.l_j : B_j}\,(1 \le j \le n)$$

(*Val Override*) : (*where* $A = [l_i{:}B_i{}^{i\in 1..n}]$)
$$\frac{E \vdash a : A \quad E, x : A \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x^A)b : A}\,(1 \le j \le n)$$

(*Val Fun*) :
$$\frac{E, x{:}B \vdash b : C}{E \vdash \lambda x^B.b : B \to C}$$

(*Val App*) :
$$\frac{E \vdash a : B \to C \quad E \vdash b : B}{E \vdash ab : C}$$

(*Val Fold*) :
$$\frac{E \vdash a : A[\mu X.A/X]}{E \vdash \mathit{fold}(\mu X.A, a) : \mu X.A}$$

(*Val Unfold*) :
$$\frac{E \vdash a : \mu X.A}{E \vdash \mathit{unfold}(a) : A[\mu X.A/X]}$$

Abadi and Cardelli show that this type assignment system has the subject *reduction* property, so assignable types are preserved by reduction. Thus, typeable terms do not ‘get stuck’.

Theorem 2.15 ([13, Theorem 1.17])**.** *If $E \vdash a : A$ and $a \to b$, then $E \vdash b : A$.*

It does not, however, preserve typeability under *expansion*.

Over several papers [48, 49, 11, 12, 13], van Bakel and de'Liguoro demonstrated how the intersection type discipline could be applied to the ς-calculus. Like the previous systems of intersection types for λ-calculus and TRS, their system for the object calculus gives rise to semantic models and a characterisation of convergence. They also use their intersection type discipline to give a treatment of observational equivalence for objects. A key aspect of that work was that the intersection type system was defined as an *additional* layer on top the existing object type system of Abadi and Cardelli. This is in contrast to the approach taken for λ-calculus and TRS, in which the intersection types are utilised as a standalone type system to replace (or rather, extend) the previous Curry-style type systems. For this reason, de'Liguoro and van Bakel dubbed their intersection types 'predicates', since they constituted an extra layer of logical information about terms, over and above the existing 'types'.

Definition 2.16 (ς-calculus Predicates)**.** *1. The set of predicates (ranged over by ϕ, ψ, etc.) and its subset of* strict *predicates (ranged over by σ, τ, etc.) are defined by the following grammar:*

$$\begin{array}{lcl} \sigma,\tau & ::= & \omega \quad | \quad (\phi \to \sigma) \quad | \quad \langle l{:}\sigma \rangle \quad | \quad \mu(\sigma) \\ \phi,\psi & ::= & \sigma_1 \cap \ldots \cap \sigma_n \quad (n \geq 1) \end{array}$$

2. The subtyping relation $\leq$ is defined as the least preorder on predicates satisfying the following conditions:

a) $\sigma \leq \omega$, *for all* σ;
b) $\sigma_1 \cap \ldots \cap \sigma_n \leq \sigma_i$ *for all* $1 \leq i \leq n$;
c) $\phi \leq \sigma_i$ *for each* $1 \leq i \leq n \Rightarrow \phi \leq \sigma_1 \cap \ldots \cap \sigma_n$;
d) $(\sigma \to \omega) \leq (\omega \to \omega)$ *for all* σ;
e) $\sigma \leq \tau$ *and* $\psi \leq \phi \Rightarrow (\phi \to \sigma) \leq (\psi \to \tau)$;
f) $\sigma \leq \tau \Rightarrow \langle l{:}\sigma \rangle \leq \langle l{:}\tau \rangle$ *for any label* l.

Notice that this predicate language differs from that of the intersection type system we presented for the λ-calculus above. Here, ω is a separate type constant, and is treated as a strict type. We also have that types of the form $\sigma \to \omega$ are *not* equivalent to the type ω itself, which differs from the usual equivalence and subtyping relations defined for intersection types in the λ-calculus. Predicates and subtyping are defined this way for the ς-calculus because the reduction relation is *lazy* - i.e. no reduction occurs under ς (or λ) binders. Thus objects (and abstractions) are considered to be values, and even if invoking a method (or applying a term to an abstraction) does not return a result.

The predicate assignment system, then, assigns predicates to typeable terms. Part of van Bakel and de'Liguoro's work was to consider the relationship between their logical predicates and the types of the ς-calculus, and so they also study a notion of predicate assignment for types, which defines a family of predicates for each type. We will not present this aspect of their work here, as it does not relate to our research which is not currently concerned with the relationship between intersection types and the existing (nominal class) types for object-oriented programs.

Definition 2.17 (Predicate Assignment)**.** *1. A* predicated *type statement is of the form $a : A : \phi$, where a is a term, A is a type and ϕ is a predicate. The term a is called the* subject *of the statement.*

2. *A* predicated *environment,* Γ*, is a sequence of predicated type statements in which the subject of each statement is a* unique *term variable. The notation* $\Gamma, x : A : \phi$ *stands for the predicated environment* $\Gamma \cup \{x : A : \phi\}$ *where* x *does not appear as the subject of any statement in* Γ.

3. $\widehat{\Gamma}$ *denotes the environment obtained by discarding the predicate information from each statement in* Γ*, ie* $\widehat{\Gamma} = \{x : A \mid \exists \phi \,.\, x : A : \phi \in \Gamma\}$.

4. *Predicate assignment* $\vdash$ *is a relation between predicated environments and predicate type statements, and is defined by the following natural deduction system, in which we take* $A = [l_i{:}B_i{}^{\,i \in 1..n}]$*:*

$$(\textit{Val } x): \quad \frac{}{\Gamma, x : B : \sigma_1 \cap \ldots \cap \sigma_n \vdash x : B : \sigma_i}\ (n \geq 1, 1 \leq i \leq n)$$

$$(\omega): \quad \frac{\widehat{\Gamma} \vdash a : B}{\Gamma \vdash a : B : \omega} \qquad (\cap I): \quad \frac{\Gamma \vdash a : B : \sigma_i \quad (\forall 1 \leq i \leq n)}{\Gamma \vdash a : B : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 1)$$

$$(\textit{Val Fun}): \quad \frac{\Gamma, x : B : \phi \vdash b : C : \sigma}{E \vdash \lambda x^B.b : B \to C : \phi \to \sigma} \qquad (\textit{Val Object}): \quad \frac{\Gamma, x : A : \phi_i \vdash b_i : B_i : \sigma_i \quad (\forall\, 1 \leq i \leq n)}{\Gamma \vdash [l_i{:}\varsigma(x^A)b_i{}^{\,i \in 1..n}] : A : \langle l_j{:}\phi_j \to \sigma_j \rangle}\ (1 \leq j \leq n)$$

$$(\textit{Val App}): \quad \frac{\Gamma \vdash a : B \to C : \phi \to \sigma \quad \Gamma \vdash b : B : \phi}{\Gamma \vdash ab : C : \sigma} \qquad (\textit{Val Select}): \quad \frac{\Gamma \vdash a : A : \langle l_j{:}\phi \to \sigma \rangle \quad \Gamma \vdash a : A : \phi}{\Gamma \vdash a.l_j : B_j : \sigma}\ (1 \leq j \leq n)$$

$$(\textit{Val Fold}): \quad \frac{\Gamma \vdash a : A[\mu X\,.\,A/X] : \sigma}{\Gamma \vdash \mathit{fold}(\mu X\,.\,A,\, a) : \mu X\,.\,A : \mu(\sigma)} \qquad (\textit{Val Update}_1): \quad \frac{\Gamma \vdash a : A : \sigma \quad \Gamma, x : A : \phi \vdash b : B_j : \tau}{E \vdash a.l_j \Leftarrow \varsigma(x^A)b : A : \langle l_j{:}\phi \to \tau \rangle}\ (1 \leq j \leq n)$$

$$(\textit{Val Unfold}): \quad \frac{\Gamma \vdash a : \mu X\,.\,A : \mu(\sigma)}{\Gamma \vdash \mathit{unfold}(a) : A[\mu X\,.\,A/X] : \sigma} \qquad (\textit{Val Update}_2): \quad \frac{\Gamma \vdash a : A : \langle l_i{:}\sigma \rangle \quad \widehat{\Gamma}, x : A \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x^A)b : A : \langle l_i{:}\sigma \rangle}\ (1 \leq i \neq j \leq n)$$

The predicate system displays the usual type preservation results for intersection type systems, although since the system only assigns predicate to *typeable* terms, the subject expansion result only holds modulo typeability.

Theorem 2.18 ([13, Theorems 4.3 and 4.6])**.** 1. *If* $\Gamma \vdash a : A : \sigma$ *and* $a \to b$*, then* $\Gamma \vdash b : A : \sigma$.

2. *If* $\Gamma \vdash b : A : \sigma$ *and* $a \to b$ *with* $\widehat{\Gamma} \vdash a : A$*, then* $\Gamma \vdash a : A : \sigma$.

To show that the predicate system characterises the convergence of (typeable) terms, a *realizability* interpretation of types as sets of closed (typeable) terms is given.

Definition 2.19 (Realizability Interpretation)**.** *The* realizability interpretation *of the predicate* σ *is a set* $[\![\sigma]\!]$ *of closed terms defined by induction over the structure of predicates as follows:*

1. $[\![\omega]\!] = \{a \mid \emptyset \vdash a : A \text{ for some } A\}$
2. $[\![\phi \to \sigma]\!] = \{a \mid \emptyset \vdash a : A \to B \;\&\; (a \Downarrow \lambda x^A.b \Rightarrow \forall c \in [\![\phi]\!].\emptyset \vdash c : A \Rightarrow b[c/x] \in [\![\sigma]\!])\}$

3. $[\![\langle l{:}\phi \to \sigma\rangle]\!] = \{a \mid \emptyset \vdash a : A \;\&\; (a \Downarrow [l_i{:}\varsigma(x^A)b_i{}^{i \in 1..n}] \Rightarrow \exists 1 \leq j \leq n.l = l_j \;\&\; \forall c \in [\![\phi]\!].\emptyset \vdash c : A \Rightarrow b_j[c/x] \in [\![\sigma]\!])\}$, *where* $A = [l_i{:}B_i{}^{i \in 1..n}]$
4. $[\![\mu(\sigma)]\!] = \{a \mid \emptyset \vdash a : \mu X.A \;\&\; (a \to^* \mathsf{fold}(\mu X.A, b) \Rightarrow b \in [\![\sigma]\!])\}$
5. $[\![\sigma_1 \cap \ldots \cap \sigma_n]\!] = [\![\sigma_1]\!] \cap \ldots \cap [\![\sigma_n]\!]$

This interpretation admits a realizability theorem: that given a typeable term, if we substitute variables by terms in the interpretation of their assumed types, we obtain a (necessarily closed) term in the interpretation of the original term's type.

Theorem 2.20 (Realizability Theorem, [13, Theorem 6.5])**.** *Let ϑ be a substitution of term variables for terms and $\vartheta(a)$ denote the result of applying ϑ to the term a; if $\Gamma \vdash b : A : \sigma$ and $\vartheta(x) \in [\![\phi]\!]$ for all $x : B : \phi \in \Gamma$, then $\vartheta(b) \in [\![\sigma]\!]$.*

A characterisation of convergent (typeable and closed) terms then follows as a corollary since, on the one hand all values can be assigned a non-trivial predicate (i.e. not ω) which is preserved by expansion, and on the other hand a straightforward induction on the structure of predicates that if $a \in [\![\sigma]\!]$ then a converges.

Corollary 2.21 (Characterisation of Convergence, [13, Corollary 6.6])**.** *Let a be any closed term such that $\vdash a : A$ for some type A; then $a \Downarrow v$ for some v if and only if $\vdash a : A : \sigma$ for some non-trivial predicate σ.*

3. Intersection Types for Featherweight Java

3.1. Featherweight Java

Featherweight Java [66], or FJ, is a calculus specifying the operational semantics of a minimal subset of Java. It was defined with the purpose of succinctly capturing the core features of a class-based object-oriented programming languages, and with the aim of providing a setting in which the formal study of class-based object-oriented features could be more easily carried out.

Featherweight Java incorporates a native notion of classes. A class represents an abstraction encapsulating both data (stored in *fields*) and the operations to be performed on that data (encoded as *methods*). Sharing of behaviour is accomplished through the *inheritance* of fields and methods from parent classes. Computation is mediated via *objects*, which are *instances* of these classes, and interact with one another by *calling* (also called *invoking*) methods on each other and accessing each other's (or their own) fields. Featherweight Java also includes the concept of *casts*, which allow the programmer to insert runtime type checks into the code, and are used in [66] to encode *generics* [25].

In this section, we will define a variant of Featherweight Java, which we simplify by removing casts. For this reason we call our calculus $\text{FJ}^{¢}$. Also, since the notion of constructors in the original formulation of FJ was not associated with any operational behaviour (i.e. constructors were purely syntactic), we leave them as implicit in our formulation. We use familiar meta-variables in our formulation to range over class names (C and D), field names or identifiers (f), method names (m) and variables (x). We distinguish the class name `Object` (which denotes the root of the class inheritance hierarchy in all programs) and the variable `this`, used to refer to the receiver object in method bodies.

Definition 3.1 ($\text{FJ}^{¢}$ Syntax)**.** *$\text{FJ}^{¢}$ programs P consist of a* class table *$\mathcal{CT}$, comprising the* class declarations, *and an* expression *e to be run (corresponding to the body of the* `main` *method in a real Java program). They are defined by the grammar:*

$$
\begin{array}{rcll}
e & ::= & x \mid \texttt{new } C(\vec{e}) \mid e.f \mid e.m(\vec{e}) & \\
fd & ::= & C\ f; & \\
md & ::= & D\ m(C_1\ x_1, \ldots, C_n\ x_n)\ \{\ \texttt{return } e;\ \} & \\
cd & ::= & \texttt{class } C \texttt{ extends } C'\ \{\ \overrightarrow{fd}\ \overrightarrow{md}\ \} & (C \neq \texttt{Object}) \\
\mathcal{CT} & ::= & \overrightarrow{cd} & \\
P & ::= & (\mathcal{CT}, e) &
\end{array}
$$

The remaining concepts that we will define below are dependent, or more precisely parametric on a given class table. For example, the reduction relation we will define uses the class table to look up fields and method bodies in order to direct reduction. Our type assignment system will do similar. Thus, there is a reduction relation and type assignment system *for each program*. However, since the class table is a fixed entity (i.e. it is not changed during reduction, or during type assignment), it will be left as

an implicit parameter in the definitions that follow. This is done in the interests of readability, and is a standard simplification in the literature (e.g. [66]).

Here, we also point out that we only consider programs which conform to some sensible well-formedness criteria: that there are no cycles in the inheritance hierarchy, and that fields and methods in any given branch of the inheritance hierarchy are uniquely named. An exception is made to allow the redeclaration of methods, providing that only the *body* of the method differs from the previous declaration. This is the class-based version of method *override*, which is to be distinguished from the object-based version that allows method bodies to be redefined on a per-object basis. Lastly, the method bodies of well-formed programs only use the variables which are declared as formal parameters in the method declaration, apart from the distinguished self variable, `this`.

We define the following functions to look up elements of the definitions given in the class table.

Definition 3.2 (Lookup Functions)**.** *The following lookup functions are defined to extract the names of fields and bodies of methods belonging to (and inherited by) a class.*

1. *The following functions retrieve the name of a class, method or field from its definition:*

$$\begin{aligned} \mathcal{CN}(\texttt{class } C \texttt{ extends } D \texttt{ \{ } \vec{fd}\ \vec{md} \texttt{ \})} &= C \\ \mathcal{FN}(C\ f) &= f \\ \mathcal{MN}(D\ m(C_1\ x_1,\ldots,C_n\ x_n)\texttt{ \{ return } e\texttt{; \}}) &= m \end{aligned}$$

2. *In an abuse of notation, we will treat the* class table, $\mathcal{CT}$*, as a partial map from class names to class definitions:*

$$\mathcal{CT}(C) = cd \quad \text{if and only if } cd \in \mathcal{CT} \text{ and } \mathcal{CN}(cd) = C$$

3. *The list of fields belonging to a class* `C` *(including those it inherits) is given by the function* $\mathcal{F}$*, which is defined as follows:*
   a) $\mathcal{F}(\texttt{Object}) = \epsilon$.
   b) $\mathcal{F}(C) = \mathcal{F}(C') \cdot \vec{f}_n$, *if* $\mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \texttt{ \{ } \vec{fd}_n\ \vec{md} \texttt{ \}}$ *and* $\mathcal{FN}(fd_i) = f_i$ *for all* $i \in \overline{n}$.

4. *The function* $\mathcal{M}\mathrm{b}$*, given a class name* `C` *and method name* `m`*, returns a tuple* $(\vec{x}, e)$*, consisting of a sequence of the method's formal parameters and its body:*
   a) *if* $\mathcal{CT}(C)$ *is undefined then so is* $\mathcal{M}\mathrm{b}(C,m)$*, for all* `m` *and* `C`.
   b) $\mathcal{M}\mathrm{b}(C,m) = (\vec{x}_n, e)$, *if* $\mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \texttt{ \{ } \vec{fd}\ \vec{md} \texttt{ \}}$ *and there is a method* $C_0\ m(C_1\ x_1,\ldots,C_n\ x_n)\texttt{ \{ return } e\texttt{; \}} \in \vec{md}$ *for some* C_0 *and* $\vec{C}_n$.
   c) $\mathcal{M}\mathrm{b}(C,m) = \mathcal{M}\mathrm{b}(C',m)$, *if* $\mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \texttt{ \{ } \vec{fd}\ \vec{md} \texttt{ \}}$ *and* $\mathcal{MN}(md) \neq m$ *for all* $md \in \vec{md}$.

5. *The function* VARS *returns the set of variables used in an expression.*

Substitution is the basic mechanism for reduction also in our calculus: when a method is invoked on an object (the *receiver*) the invocation is replaced by the body of the method that is called, and each of the variables is replaced by a corresponding argument.

Definition 3.3 (Reduction)**.** *1. A* term substitution $\mathsf{S} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ *is defined in the standard way as a total function on expressions that systematically replaces all occurrences of the variables x_i by their corresponding expression e_i. We write e^{S} for $\mathsf{S}(e)$.*

2. The reduction relation $\rightarrow$ is the smallest relation on expressions satisfying:

$$\begin{array}{rcll} \texttt{new } C(\vec{e}_n).f_i & \rightarrow & e_i & \text{if } \mathcal{F}(C) = \vec{f}_n \text{ and } i \in \overline{n} \\ \texttt{new } C(\vec{e}).m(\vec{e'}_n) & \rightarrow & e^{\mathsf{S}} & \text{if } \mathcal{M}b(C,m) = (\vec{x}_n, e) \\ & & \multicolumn{2}{l}{\text{where } \mathsf{S} = \{\ \texttt{this} \mapsto \texttt{new } C(\vec{e}),\ x_1 \mapsto e'_1, \ldots,\ x_n \mapsto e'_n\ \}} \end{array}$$

3. We add the usual congruence rules for allowing reduction in subexpressions.

4. If $e \rightarrow e'$, then e is the redex *and e' the* contractum*.*

5. The reflexive and transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

This notion of reduction is *confluent*, which is easily shown by a 'colouring' argument (as done in [19] for LC).

3.2. Intersection Type Assignment

In this section we will defined a type assignment system following in the *intersection type discipline*; it is influenced by the predicate system for the object calculus [13], and is ultimately based upon the strict intersection type system for LC (see [9] for a survey). Our types can be seen as describing the capabilities of an expression (or rather, the object to which that expression evaluates) in terms of (1) *the operations that may be performed on it (i.e. accessing a field or invoking a method), and (2) the* outcome *of performing those operations,* where dependencies between the inputs and outputs of methods are tracked using (type) variables. In this way they express detailed properties about the contexts in which expressions can be safely used. More intuitively, they capture a certain notion of *observational equivalence*: two expressions with the same (non-empty) set of assignable types will be observationally indistinguishable. Our types thus constitute *semantic predicates* describing the functional behaviour of expressions.

We call our types 'simple' because they are essentially function types, of a similar order to the types used in the simply typed Lambda Calculus.

Definition 3.4 (Simple Intersection Types)**.** *The set of* FJ$^{\cent}$ simple intersection types *(ranged over by ϕ, ψ) and its subset of* strict *simple intersection types (ranged over by σ) are defined by the following grammar (where φ ranges over a denumerable set of type* variables*, and C ranges over the set of class names):*

$$\begin{array}{rcl} \sigma & ::= & \varphi \mid C \mid \langle f{:}\sigma \rangle \mid \langle m{:}(\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad (n \geq 0) \\ \phi, \psi & ::= & \omega \mid \sigma \mid \phi \cap \psi \end{array}$$

We may abbreviate method types $\langle m{:}(\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle$ by writing $\langle m{:}(\vec{\phi}_n) \rightarrow \sigma \rangle$.

The key feature of our types is that they may group information about many operations together into *intersections* from which any specific one can be selected for an expression as demanded by the context in which it appears. In particular, an intersection may combine two or more different analyses (in the sense that they are not unifiable) of the *same* field or method. Types are therefore not records: records

can be characterised as intersection types of the shape $\langle l_1 : \sigma_1 \rangle \cap \ldots \cap \langle l_n : \sigma_n \rangle$ where all σ_i are intersection free, and all labels l_i are distinct; in other words, records are intersection types, but not vice-versa.

In the language of intersection type systems, our types are *strict* (in the sense of [7]), since they must describe the outcome of performing an operation in terms of another *single* operation rather than an intersection. We include a type constant for each class, which we can use to type objects when a more detailed analysis of the object's fields and methods is not possible. This may be because the object does not contain any fields or methods (as is the case for `Object`) or more generally because no fields or methods can be safely invoked. The type constant ω is a *top* (maximal) type, assignable to all expressions.

We also define a subtype relation that facilitates the selection of individual behaviours from intersections.

Definition 3.5 (Subtyping)**.** *The subtype relation $\leqslant$ is the smallest preorder satisfying the following conditions:*

$$\phi \leqslant \omega \quad \text{for all } \phi \qquad \phi \cap \psi \leqslant \phi$$
$$\phi \leqslant \psi \;\&\; \phi \leqslant \psi' \;\Rightarrow\; \phi \leqslant \psi \cap \psi' \qquad \phi \cap \psi \leqslant \psi$$

We write $\sim$ for the equivalence relation generated by $\leqslant$, extended by

1. $\langle \mathtt{f} : \sigma \rangle \sim \langle \mathtt{f} : \sigma' \rangle$*, if* $\sigma \sim \sigma'$*;*
2. $\langle \mathtt{m} : (\phi_1, \ldots, \phi_n) \to \sigma \rangle \sim \langle \mathtt{m} : (\phi'_1, \ldots, \phi'_n) \to \sigma' \rangle$*, if* $\sigma \sim \sigma'$ *and* $\phi'_i \sim \phi'_i$ *for all* $i \in \overline{n}$*.*

Notice that $\phi \cap \omega \sim \omega \cap \phi \sim \phi$.

We will consider types modulo $\sim$; in particular, all types in an intersection are different and ω does not appear in an intersection. It is easy to show that $\cap$ is associative and commutative with respect to $\sim$, so we will abuse notation slightly and write $\sigma_1 \cap \ldots \cap \sigma_n$ (where $n \geq 2$) to denote a general intersection, where each σ_i is distinct and the order is unimportant. In a further abuse of notation, $\phi_1 \cap \ldots \cap \phi_n$ will denote the type ϕ_1 when $n = 1$, and ω when $n = 0$.

Definition 3.6 (Type Environments)**.**

1. *A* type statement *is of the form $\mathtt{e} : \phi$, where $\mathtt{e}$ is called the* subject *of the statement.*
2. *An environment Π is a set of type statements with (distinct) variables as subjects; $\Pi, x{:}\phi$ stands for the environment $\Pi \cup \{x{:}\phi\}$ where x does not appear as the subject of any statement in Π.*
3. *We extend the subtyping relation to environments by: $\Pi' \leqslant \Pi$ if and only if for all statements $x{:}\phi \in \Pi$ there is a statement $x{:}\phi' \in \Pi'$ such that $\phi' \leqslant \phi$.*
4. *If $\vec{\Pi}_n$ is a sequence of environments, then $\bigcap \vec{\Pi}_n$ is the environment defined as follows: $x{:}\phi_1 \cap \ldots \cap \phi_m \in \bigcap \vec{\Pi}_n$ if and only if $\{x{:}\phi_1, \ldots, x{:}\phi_m\}$ is the non-empty set of all statements in the union of the environments that have x as the subject.*

Notice that, as for types themselves, the intersection of environments is a subenvironment of each individual environment in the intersection.

Lemma 3.7. *Let $\vec{\Pi}_n$ be type environments; then $\bigcap \vec{\Pi}_n \leqslant \Pi_i$ for each $i \in \overline{n}$.*

Proof. Directly by Definitions 3.6(4) and 3.5. □

We will now define our notion of intersection type assignment for FJ$^{\mathfrak{C}}$.

$$(\textsc{var}):\ \frac{}{\Pi, x{:}\phi \vdash x : \sigma}\ (\phi \leqslant \sigma) \qquad (\omega):\ \frac{}{\Pi \vdash e : \omega} \qquad (\textsc{join}):\ \frac{\Pi \vdash e : \sigma_1 \ \ldots\ \Pi \vdash e : \sigma_n}{\Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2)$$

$$(\textsc{fld}):\ \frac{\Pi \vdash e : \langle f{:}\sigma \rangle}{\Pi \vdash e.f : \sigma} \qquad (\textsc{invk}):\ \frac{\Pi \vdash e : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle \quad \Pi \vdash e_1 : \phi_1 \ \ldots\ \Pi \vdash e_n : \phi_n}{\Pi \vdash e.m(\vec{e}_n) : \sigma}$$

$$(\textsc{obj}):\ \frac{\Pi \vdash e_1 : \phi_1 \quad \ldots \quad \Pi \vdash e_n : \phi_n}{\Pi \vdash \texttt{new}\ C(\vec{e}_n) : C}\ (\mathcal{F}(C) = \vec{f}_n)$$

$$(\textsc{newF}):\ \frac{\Pi \vdash e_1 : \phi_1 \quad \ldots \quad \Pi \vdash e_n : \phi_n}{\Pi \vdash \texttt{new}\ C(\vec{e}_n) : \langle f_i{:}\sigma \rangle}\ (\mathcal{F}(C) = \vec{f}_n, i \in \overline{n}, \sigma = \phi_i, n \geq 1)$$

$$(\textsc{newM}):\ \frac{\{\texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash e_b : \sigma \quad \Pi \vdash \texttt{new}\ C(\vec{e}) : \psi}{\Pi \vdash \texttt{new}\ C(\vec{e}) : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle}\ (\mathcal{M}b(C, m) = (\vec{x}_n, e_b))$$

Figure 3.1.: Predicate Assignment for FJ$^{\text{¢}}$

Definition 3.8 (Intersection Type Assignment)**.** *Intersection type assignment for* FJ$^{\text{¢}}$ *is defined by the natural deduction system given in Figure 3.1.*

The rules of our type assignment system are fairly straightforward generalisations to oo of the rules of the strict intersection type assignment system for LC: e.g. (FLD) and (INVK) are analogous to ($\to E$); (NEWF) and (NEWM) are a form of ($\to I$); and (OBJ) can be seen as a universal (ω)-like rule for *objects* only. Notice that objects `new C()` without fields can be dealt with by both the (NEWM) and (OBJ) rules, and then the environment can be anything, as is also the case with the (ω) rule.

The only non-standard rule from the point of view of similar work for term rewriting and traditional nominal oo type systems is (NEWM), which derives a type for an object that presents an analysis of a method. It makes sense however when viewed as an abstraction introduction rule. Like the corresponding LC typing rule ($\to I$), the analysis involves typing the body of the abstraction (i.e. the method body), and the assumptions (i.e. requirements) on the formal parameters are encoded in the derived type (to be checked on invocation). However, a method body may also make requirements on the *receiver*, through the use of the variable `this`. In our system we check that these hold *at the same time* as typing the method body, so-called *early self typing*, whereas with *late* self typing (as used in [13]) we would check the type of the receiver at the point of invocation. This checking of requirements on the object itself is where the expressive power of our system resides. If a method calls itself recursively, this recursive call must be checked, but – crucially – carries a *different* type if a valid derivation is to be found. Thus only recursive calls which terminate at a certain point (i.e. which can be assigned ω, and thus ignored) will be permitted by the system.

We discuss several extended examples of type assignment using this system in Chapter 6.

3.3. Subject Reduction & Expansion

As is standard for intersection type assignment systems, our system exhibits both subject reduction *and* subject expansion. We first show a *weakening* lemma, which allows to increase the typing environment where necessary, and will be used in the proof of subject expansion.

Lemma 3.9 (Weakening)**.** *Let* $\Pi' \leqslant \Pi$*; then* $\Pi \vdash e : \phi \Rightarrow \Pi' \vdash e : \phi$

Proof. By easy induction on the structure of derivations. The base case of (ω) follows immediately, and for (VAR) it follows by transitivity of the subtype relation. The other cases follow easily by induction. □

We also need to show replacement and expansion lemmas. The replacement lemma states that, for a typeable expression, if we replace all its variables by appropriately typed expressions (i.e. typeable using the same type assumed for the variable being replaced) then the result can be assigned the same type as the original expression. The extraction lemma states the opposite: if the result of substituting expressions for variables is typeable, then we can also type the substituting and original expressions.

Lemma 3.10. *1.* ***(Replacement)*** *If* $\{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash e : \phi$ *and there exists* Π *and* $\overline{e}_n$ *such that* $\Pi \vdash e_i : \phi_i$ *for each* $i \in \overline{n}$*, then* $\Pi \vdash e^{\mathsf{S}} : \phi$ *where* $\mathsf{S} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$.

2. ***(Extraction)*** *Let* $\mathsf{S} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ *be a term substitution and* e *be an expression with* $\textsc{vars}(e) \subseteq \{x_1, \ldots, x_n\}$*, if* $\Pi \vdash e^{\mathsf{S}} : \phi$*, then there is some* $\vec{\phi}_n$ *such that* $\Pi \vdash e_i : \phi_i$ *for each* $i \in \overline{n}$ *and* $\{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash e : \phi$.

Proof. 1. By induction on the structure of derivations.

(ω): Immediate.

(VAR): Then $e = x_i$ for some $i \in \overline{n}$ and $e^{\mathsf{S}} = e_i$. Also, $\phi = \sigma$ with $\phi_i \leqslant \sigma$, thus $\phi_i = \sigma_1 \cap \ldots \cap \sigma_n$ and $\sigma = \sigma_j$ for some $j \in \overline{n}$. Since $\Pi \vdash e_i : \phi_i$ it follows from rule (JOIN) that $\Pi \vdash e_i : \sigma_k$ for each $k \in \overline{n}$. So, in particular, $\Pi \vdash e_i : \sigma_j$.

(FLD), (JOIN), (INVK), (OBJ), (NEWF), (NEWM): These cases follow straightforwardly by induction.

2. Also by induction on the structure of derivations.

(ω): By the (ω) rule, $\Pi \vdash e_i : \omega$ for each $i \in \overline{n}$ and $\{x_1{:}\omega, \ldots, x_n{:}\omega\} \vdash e : \omega$.

(VAR): Then ϕ is a strict type (hereafter called σ), and $x{:}\psi \in \Pi$ with $\psi \leqslant \sigma$. Also, it must be that $e = x_i$ for some $i \in \overline{n}$ and $e_i = x$. We then take $\phi_i = \sigma$ and $\phi_j = \omega$ for each $j \in \overline{n}$ such that $j \neq i$. By assumption $\Pi \vdash x : \sigma$ (that is $\Pi \vdash e_i : \phi_i$). Also, by the (ω) rule, we can derive $\Pi \vdash e_j : \omega$ for each $j \in \overline{n}$ such that $j \neq i$. Lastly, by (VAR) we have $\{x_1{:}\omega, \ldots, x_i{:}\sigma, \ldots, x_n{:}\omega\} \vdash x_i : \sigma$.

(NEWF): Then $e^{\mathsf{S}} = \mathtt{new}\ C(\overline{e'}_{n'})$ and $\phi = \langle f{:}\sigma \rangle$ with $\mathcal{F}(C) = \overline{f}_{n'}$ and $f = f_j$ for some $j \in \overline{n'}$. Also, there is $\vec{\phi}_{n'}$ such that $\Pi \vdash e'_{k'} : \phi_{k'}$ for each $k' \in \overline{n'}$, and $\sigma \leqslant \phi_j$. There are two cases to consider for e:

a) $e = x_i$ for some $i \in \overline{n}$. Then $e_i = \mathtt{new}\ C(\overline{e'}_{n'})$. Take $\phi_i = \langle f{:}\sigma \rangle$ and $\phi_k = \omega$ for each $k \in \overline{n}$ such that $k \neq i$. By assumption we have $\Pi \vdash \mathtt{new}\ C(\overline{e'}_{n'}) : \langle f{:}\sigma \rangle$ (that is $\Pi \vdash e_i : \phi_i$). Also, by rule (ω) $\Pi \vdash e_k : \omega$ for each $k \in \overline{n}$ such that $k \neq i$, and lastly by rule (VAR) $\Pi' \vdash x_i : \langle f{:}\sigma \rangle$ where $\Pi' = \{x_1{:}\omega, \ldots, x_i{:}\langle f{:}\sigma \rangle, \ldots, x_n{:}\omega\}$.

b) $e = \mathtt{new}\ C(\overline{e''}_{n'})$ with $e''^{\mathsf{S}}_{k'} = e'_{k'}$ for each $k' \in \overline{n'}$. Notice that $\textsc{vars}(e''_{k'}) \subseteq \textsc{vars}(e) \subseteq \{x_1, \ldots, x_n\}$ for each $k' \in \overline{n'}$. So, by induction, for each $k' \in \overline{n'}$ there is $\overrightarrow{\phi_{k'}}_n$ such that $\Pi \vdash e_i : \phi_{k',i}$ for each $i \in \overline{n}$ and $\Pi_{k'} \vdash e''_{k'} : \phi_{k'}$ where $\Pi_{k'} = \{x_1{:}\phi_{k',1}, \ldots, x_n{:}\phi_{k',n}\}$. Let the environment $\Pi' = \bigcap \overrightarrow{\Pi}_{n'}$, that is $\Pi' = \{x_1{:}\phi_{1,1} \cap \ldots \cap \phi_{n',1}, \ldots, x_n{:}\phi_{1,n} \cap \ldots \cap \phi_{n',n}\}$. Notice that $\Pi' \leqslant \Pi_{k'}$ for each $k' \in \overline{n'}$, so by Lemma 3.9 $\Pi' \vdash e''_{k'} : \phi_{k'}$ for each $k \in \overline{n'}$. Then by the (NEWF) rule, $\Pi' \vdash \mathtt{new}\ C(\overline{e''}_{n'}) : \langle f{:}\sigma \rangle$ and so by (JOIN) we can derive $\Pi \vdash e_i : \phi_{1,i} \cap \ldots \cap \phi_{n',i}$ for each $i \in \overline{n}$.

(FLD), (JOIN), (INVK), (OBJ), (NEWM): These cases are similar to (NEWF).

□

We can now prove subject reduction, or soundness, as well as subject expansion, or completeness.

Theorem 3.11 (Subject reduction and expansion)**.** *Let* $e \to e'$*; then* $\Pi \vdash e' : \phi$ *if and only if* $\Pi \vdash e : \phi$*.*

Proof. By double induction - the outer induction on the definition of $\to$ and the inner on the structure of types. For the outer induction, we show the cases for the two forms of redex and one inductive case (the others are similar). For the inner induction, we show only the case that ϕ is strict; when $\phi = \omega$ the result follows immediately since we can always type both e and e' using the (ω) rule, and when ϕ is an intersection the result follows trivially from the inductive hypothesis and the (JOIN) rule.

($\mathcal{F}(C) = \vec{f}_n \Rightarrow \texttt{new } C(\vec{e}_n).f_j \to e_j,\ j \in \overline{n}$):

- **(if)**: We begin by assuming $\Pi \vdash \texttt{new } C(\vec{e}_n).f_j : \sigma$. The last rule applied in this derivation must be (FLD) so $\Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f_j : \sigma \rangle$. This is turn must have been derived using the (NEWF) rule and so there are $\phi_1, \ldots, \phi_n$ such that $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. Furthermore $\sigma \leqslant \phi_j$ and so it must be that $\phi_j = \sigma$. Thus $\Pi \vdash e_j : \sigma$.
- **(only if)**: We begin by assuming $\Pi \vdash e_j : \sigma$. Notice that using (ω) we can derive $\Pi \vdash e_i : \omega$ for each $i \in \overline{n}$ such that $i \neq j$. Then, using the (NEWF) rule, we can derive $\Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f_j : \sigma \rangle$ and by (FLD) also $\Pi \vdash \texttt{new } C(\vec{e}_n).f_j : \sigma$.

($\mathcal{M}b(C,m) = (\vec{x}_n, e_b) \Rightarrow \texttt{new } C(\vec{e'}).m(\vec{e}_n) \to {e_b}^{\mathsf{S}}$):
where $\mathsf{S} = \{\texttt{this} \mapsto \texttt{new } C(\vec{e'}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$.

- **(if)**: We begin by assuming $\Pi \vdash \texttt{new } C(\vec{e'}).m(\vec{e}_n) : \sigma$. The last rule applied in the derivation must be (INVK), so there is $\vec{\phi}_n$ such that we can derive $\Pi \vdash \texttt{new } C(\vec{e'}) : \langle m : (\vec{\phi}_n) \to \sigma \rangle$ and $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. Furthermore, the last rule applied in the derivation of $\Pi \vdash \texttt{new } C(\vec{e'}) : \langle m : (\vec{\phi}_n) \to \sigma \rangle$ must be (NEWM) and so there is some type ψ such that $\Pi \vdash \texttt{new } C(\vec{e'}) : \psi$ and $\Pi' \vdash e_b : \sigma$ where $\Pi' = \{\texttt{this}{:}\psi, x_1{:}\phi_i, \ldots, x_n{:}\phi_n\}$. Then from Lemma 3.10(1) it follows that $\Pi \vdash {e_b}^{\mathsf{S}} : \sigma$.
- **(only if)**: We begin by assuming that $\Pi \vdash {e_b}^{\mathsf{S}} : \sigma$. Then by Lemma 3.10(2) it follows that there is ψ, $\vec{\phi}_n$ such that $\Pi' \vdash e_b : \sigma$ where the environment $\Pi' = \{\texttt{this}{:}\psi, x_1{:}\phi_i, \ldots, x_n{:}\phi_n\}$ with $\Pi \vdash \texttt{new } C(\vec{e'}) : \psi$ and $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. By the (NEWM) rule we can then derive $\Pi \vdash \texttt{new } C(\vec{e'}) : \langle m : (\vec{\phi}_n) \to \sigma \rangle$, and by the (INVK) rule that $\Pi \vdash \texttt{new } C(\vec{e'}).m(\vec{e}_n) : \sigma$.

($e \to e' \Rightarrow e.f \to e'.f$):

- **(if)**: We begin by assuming that $\Pi \vdash e.f : \sigma$. The last rule applied in the derivation must be (FLD) and so we have that $\Pi \vdash e : \langle f : \sigma \rangle$. By the inductive hypothesis it follows that $\Pi \vdash e' : \langle f : \sigma \rangle$, and so by (FLD) that $\Pi \vdash e'.f : \sigma$.
- **(only if)**: We begin by assuming that $\Pi \vdash e'.f : \sigma$. The last rule applied in the derivation must be (FLD) and so we have that $\Pi \vdash e' : \langle f : \sigma \rangle$. By the inductive hypothesis it follows that $\Pi \vdash e : \langle f : \sigma \rangle$, and so by (FLD) that $\Pi \vdash e.f : \sigma$.

□

4. Strong Normalisation of Derivation Reduction

In this chapter we will lay the foundations for our main result linking type assignment with semantics: the approximation result, presented in the next chapter. This result shows the deep relationship between the intersection types assignable to an expression and its reduction behaviour, and this link is rooted in the notion we define in this chapter - that of a *reduction* relation on *derivations*. Through this relation, the coupling between typeability, as witnessed by derivations, and the computational behaviour of programs, which is modelled via reduction, is made absolutely explicit.

The approximation result, and the various characterisations of the reduction behaviour of expressions, follows from the fact that the reduction relation on intersection type derivations is *strongly normalising*, i.e. terminating. We will show that this is the case using Tait's *computability* technique [100]. The general technique of showing approximation using derivation reduction has also been used in the context of the TRS [15] and λ-calculus [10].

Our notion of *derivation reduction* is essentially a form of cut-elimination on type derivations [91]. The two 'cut' rules in our type system are (NEWF) and (NEWM), and they are eliminated from derivations using the following transformations:

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_n}{\Pi \vdash e_1 : \phi_1 \quad \ldots \quad \Pi \vdash e_n : \phi_n}}{\dfrac{\Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f_i : \sigma \rangle}{\Pi \vdash \texttt{new } C(\vec{e}_n).f_i : \sigma}} \quad \rightarrow_{\mathfrak{D}} \quad \dfrac{\mathcal{D}_i}{\Pi \vdash e_i : \sigma}$$

$$\dfrac{\dfrac{\mathcal{D}_b}{\{\texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash e_b : \sigma} \quad \dfrac{\mathcal{D}_{\text{self}}}{\Pi \vdash \texttt{new } C(\vec{e}^{\,\prime}) : \psi}}{\begin{array}{c}\Pi \vdash \texttt{new } C(\vec{e}^{\,\prime}) : \langle m : (\vec{\phi}_n) \to \sigma \rangle \\ \vdots \\ \dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_n}{\Pi \vdash e_1 : \phi_1 \quad \ldots \quad \Pi \vdash e_n : \phi_n}}{\Pi \vdash \texttt{new } C(\vec{e}^{\,\prime}).m(\vec{e}_n) : \sigma}\end{array}} \quad \rightarrow_{\mathfrak{D}} \quad \dfrac{\mathcal{D}_b{}^{S}}{\Pi \vdash e_b{}^{\mathsf{S}} : \sigma}$$

where $\mathcal{D}_b{}^{S}$ is the derivation obtained from $\mathcal{D}_b$ by replacing all sub-derivations of the form $\langle \text{VAR} \rangle :: \Pi, x_i{:}\phi_i \vdash x_i : \sigma$ by appropriately typed sub-derivations of $\mathcal{D}_i$, and sub-derivations of the form $\langle \text{VAR} \rangle :: \Pi, \texttt{this}{:}\psi \vdash \texttt{this} : \sigma$ by appropriately typed sub-derivations of $\mathcal{D}_{\text{self}}$. Similarly, $e_b{}^{\mathsf{S}}$ is the expression obtained from e_b by replacing each variable x_i by the expression e_i, and the variable `this` by $\texttt{new } C(\vec{e}^{\,\prime})$.

This reduction creates exactly the derivation for a contractum as suggested by the proof of the subject reduction, but is explicit in all its details, which gives the expressive power to show the approximation result. An important feature of derivation reduction is that sub-derivations of the form $\langle \omega \rangle :: \Pi \vdash e : \omega$ do *not* reduce, although e might; that is, they are already in normal form. This is crucial for the strong normalisability of derivation reduction, since it decouples the reduction of a derivation from the possibly infinite reduction sequence of the expression which it types.

To formalise this notion of derivation reduction, it will be convenient to introduce a notation for describing and specifying the structure of derivations.

Definition 4.1 (Notation for Derivations)**.** *The meta-variable $\mathcal{D}$ ranges over derivations. We will use the notation $\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: \Pi \vdash e : \phi$ to represent the derivation concluding with the judgement $\Pi \vdash e : \phi$ where the last rule applied is r and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are the (sub) derivations for each of that rule's premises. In an abuse of notation, we may sometimes write $\mathcal{D} :: \Pi \vdash e : \phi$ for $\mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: \Pi \vdash e : \phi$ when the structure of $\mathcal{D}$ is not relevant or is implied by the context, and also write $\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle$ when the conclusion of the derivation is similarly irrelevant or implied.*

We also introduce some further notational concepts to aid us. The first of these is the notion of *position* within an expression or derivation. We then extend expressions and derivations with a notion of placeholder, so that we can refer to and reason about specific subexpressions and subderivations.

Definition 4.2 (Position)**.** *The* position *p of one (sub) expression – similarly of one (sub) derivation – within another is a non-empty sequence of integers:*

1. *Positions within expressions are defined inductively as follows:*
   - *i) The position of an expression e within* itself *is 0.*
   - *ii) If the position of e' within e is p, then the position of e' within $e.f$ is $0 \cdot p$.*
   - *iii) If the position of e' within e is p, then the position of e' within $e.m(\vec{e})$ is $0 \cdot p$.*
   - *iv) For a sequence of expressions $\vec{e}_n$, if the position of e' within some e_j is p, then the position of e' within $e.m(\vec{e})$ is $j \cdot p$.*
   - *v) For a sequence of expressions $\vec{e}_n$, if the position of e' within some e_j is p, then the position of e' within* `new C(`$\vec{e}$`)` *is $j \cdot p$.*
2. *Positions within derivations are defined inductively as follows:*
   - *i) The position of a derivation $\mathcal{D}$ within itself is 0.*
   - *ii) For $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}'', \text{NEWM} \rangle$, if the position of $\mathcal{D}'$ within $\mathcal{D}''$ is p then so is the position of $\mathcal{D}'$ within $\mathcal{D}$.*
   - *iii) For $\mathcal{D} = \langle \vec{\mathcal{D}}_n, \text{JOIN} \rangle$, if the position of $\mathcal{D}'$ within $\mathcal{D}_j$ is p for some $j \in \overline{n}$ then so is position of $\mathcal{D}'$ within $\mathcal{D}$.*
   - *iv) For $\mathcal{D} = \langle \mathcal{D}'', \text{FLD} \rangle$, if the position of $\mathcal{D}'$ within $\mathcal{D}''$ is p then the position of $\mathcal{D}'$ within $\mathcal{D}$ is $0 \cdot p$.*
   - *v) For $\mathcal{D} = \langle \mathcal{D}'', \vec{\mathcal{D}}_n, \text{INVK} \rangle$, if the position of $\mathcal{D}'$ within $\mathcal{D}''$ is p the the position of $\mathcal{D}'$ within $\mathcal{D}$ is $0 \cdot p$.*
   - *vi) For $\mathcal{D} = \langle \mathcal{D}'', \vec{\mathcal{D}}_n, \text{INVK} \rangle$, if the position of $\mathcal{D}'$ within $\mathcal{D}_j$ is p for some $j \in \overline{n}$ then the position of $\mathcal{D}'$ within $\mathcal{D}$ is $j \cdot p$.*
   - *vii) For $\mathcal{D} = \langle \vec{\mathcal{D}}_n, \text{OBJ} \rangle$, if the position of $\mathcal{D}'$ within $\mathcal{D}_j$ is p for some $j \in \overline{n}$ then the position of $\mathcal{D}'$ within $\mathcal{D}$ is $j \cdot p$.*
   - *viii) For $\mathcal{D} = \langle \vec{\mathcal{D}}_n, \text{NEWF} \rangle$, if the position of $\mathcal{D}'$ within $\mathcal{D}_j$ is p for some $j \in \overline{n}$ then the position of $\mathcal{D}'$ within $\mathcal{D}$ is $j \cdot p$.*

   Notice that due to the (JOIN) *rule, positions in derivations are not necessarily unique.*
3. *We define the following terminology:*

- *If the position of e' ($\mathcal{D}'$) within e ($\mathcal{D}$) is p, then we say that e' ($\mathcal{D}'$)* appears at position *p within e ($\mathcal{D}$).*
- *If there exists some e' ($\mathcal{D}'$) that appears in position p within e ($\mathcal{D}$), then we say that position p* exists within *e ($\mathcal{D}$).*

Definition 4.3 (Expression Contexts)**.** *1. An* expression context *$\mathfrak{C}$ is an expression containing a 'hole' (denoted by $[\,]$) defined by the following grammar:*

$$\begin{aligned}\mathfrak{C} \;::=\; & [\,] \mid \mathfrak{C}.f \mid \mathfrak{C}.m(\vec{e}) \mid \\ & e.m(\ldots,e_{i-1},\mathfrak{C},e_{i+1},\ldots) \mid \texttt{new } C(\ldots,e_{i-1},\mathfrak{C},e_{i+1},\ldots)\end{aligned}$$

2. *$\mathfrak{C}[e]$ denotes the expression obtained by replacing the hole in $\mathfrak{C}$ with e.*
3. *We write $\mathfrak{C}_p$ to indicate that the hole in $\mathfrak{C}$ appears at position p.*
4. *Contexts $\mathfrak{C}_p$ where $p = \vec{0}_n$ are called* neutral*; by extension,* expressions *of the form $\mathfrak{C}[x]$ where $\mathfrak{C}$ is neutral are also neutral.*

Definition 4.4 (Derivation Contexts)**.** *1. A* derivation context *$\mathfrak{D}_{(p,\sigma)}$ is a derivation concluding with a statement assigning a strict type to a neutral context, in which the hole appears at position p and has type σ. We abuse the notation for derivations in order to more easily formalise the notion of derivation context:*

a) $\mathfrak{D}_{(0,\sigma)} = \langle [\,] \rangle :: \Pi \vdash [\,] : \sigma$ is a derivation context.

b) If $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C} : \langle f : \sigma' \rangle$ is a derivation context, then $\mathfrak{D}'_{(0\cdot p,\sigma)} = \langle \mathfrak{D}, \text{FLD} \rangle :: \Pi \vdash \mathfrak{C}.f : \sigma'$ is also a derivation context.

c) if $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C} : \langle m : (\vec{\phi}_n) \to \sigma' \rangle$ is a derivation context and $\vec{\mathcal{D}}_n$ is a sequence of derivations such that $\mathcal{D}_i :: \Pi \vdash e : \phi_i$ for each $i \in \overline{n}$, then $\mathfrak{D}'_{(0\cdot p,\sigma)} = \langle \mathfrak{D}, \vec{\mathcal{D}}_n, \text{INVK} \rangle :: \Pi \vdash \mathfrak{C}.m(\vec{e}_n) : \sigma'$ is also a derivation context.

2. *For a derivation $\mathcal{D} :: \Pi \vdash e : \sigma$ and derivation context $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C} : \sigma'$, we write $\mathfrak{D}_{(p,\sigma)}[\mathcal{D}] :: \Pi \vdash \mathfrak{C}[e] : \sigma'$ to denote the derivation obtained by replacing the hole in $\mathfrak{D}$ by $\mathcal{D}$.*

We now define an explicit *weakening* operation on derivations, which is also extended to derivation contexts. This will be crucial in defining our notion of *computability* which we will use to show that derivation reduction is strongly normalising.

Definition 4.5 (Weakening)**.** *A* weakening*, written $[\Pi' \trianglelefteq \Pi]$ where $\Pi' \trianglelefteq \Pi$, is an operation that replaces environments by sub-environments. It is defined on derivations and derivation contexts as follows:*

1. *For derivations $\mathcal{D} :: \Pi \vdash e : \phi$, $\mathcal{D}[\Pi' \trianglelefteq \Pi]$ is defined as the derivation $\mathcal{D}'$ of exactly the same shape as $\mathcal{D}$ such that $\mathcal{D}' :: \Pi' \vdash e : \phi$.*
2. *For derivation contexts $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C}_p : \phi$, $\mathfrak{D}_{(p,\sigma)}[\Pi' \trianglelefteq \Pi]$ is defined as the derivation context $\mathfrak{D}'_{(p,\sigma)}$ of exactly the same shape as $\mathfrak{D}_{(p,\sigma)}$ such that $\mathfrak{D}'_{(p,\sigma)} :: \Pi' \vdash \mathfrak{C}_p : \phi$.*

The following two basic properties of the weakening operation on derivations will be needed later when showing that it preserves computability.

Lemma 4.6. *Let* Π_1, Π_2, Π_3 *and* Π_4 *be type environments such that*

- $\Pi_2 \preccurlyeq \Pi_1$, *and* $\Pi_3 \preccurlyeq \Pi_1$*;*
- $\Pi_4 \preccurlyeq \Pi_2$, *and* $\Pi_4 \preccurlyeq \Pi_3$*;*

and $\mathcal{D}$ *be a derivation such that* $\mathcal{D} :: \Pi_1 \vdash e : \phi$*. Then*

1. $\mathcal{D}[\Pi_2 \preccurlyeq \Pi_1][\Pi_4 \preccurlyeq \Pi_2] = \mathcal{D}[\Pi_4 \preccurlyeq \Pi_1]$.
2. $\mathcal{D}[\Pi_2 \preccurlyeq \Pi_1][\Pi_4 \preccurlyeq \Pi_2] = \mathcal{D}[\Pi_3 \preccurlyeq \Pi_1][\Pi_4 \preccurlyeq \Pi_3]$.

Proof. Directly by Definition 4.5. □

We also show the following two properties of weakening for derivation contexts and substitutions, which will be used in the proof of Lemma 4.28 to show that computability is preserved by derivation expansion.

Lemma 4.7. *Let* $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C}_p : \phi$ *be a derivation context and* $\mathcal{D} :: \Pi \vdash e : \sigma$ *be a derivation. Also, let* $[\Pi' \preccurlyeq \Pi]$ *be a weakening. Then*

$$\mathfrak{D}_{(p,\sigma)}[\mathcal{D}][\Pi' \preccurlyeq \Pi] = \mathfrak{D}_{(p,\sigma)}[\Pi' \preccurlyeq \Pi][\mathcal{D}[\Pi' \preccurlyeq \Pi]]$$

Proof. By easy induction on the structure of derivation contexts. □

We now define two important sets of derivations, the strong and ω-safe derivations. The idea behind these kinds of derivation is to restrict the use of the (ω) rule in order to preclude non-termination (i.e. guarantee normalisation). In strong derivations, we do not allow the (ω) rule to be used at all. This restriction is relaxed slightly for ω-safe derivations in that ω may be used to type the arguments to a method call. The idea behind this is that when those arguments disappear during reduction it is 'safe' to type them with ω since non-termination at these locations can be ignored. We will show later that our definitions do indeed entail the desired properties, since expressions typeable using strong derivations are strongly normalising, and expressions which can be typed with ω-safe derivations using an ω-safe environment, while not necessarily being strongly normalising, have a normal form.

Definition 4.8 (Strong Derivations)**.** 1. *Strong derivations are defined inductively as follows:*

- *Derivations of the form* $\langle \textsc{var} \rangle$ *are strong.*
- *Derivations of the form* $\langle \overrightarrow{\mathcal{D}_n}, \textsc{join} \rangle$, $\langle \overrightarrow{\mathcal{D}_n}, \textsc{obj} \rangle$ *and* $\langle \overrightarrow{\mathcal{D}_n}, \textsc{newf} \rangle$ *are strong, if each derivation* $\mathcal{D}_i$ *is strong.*
- *Derivations of the form* $\langle \mathcal{D}, \textsc{fld} \rangle$ *are strong, if* $\mathcal{D}$ *is strong.*
- *Derivations of the form* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \textsc{invk} \rangle$ *are strong, if* $\mathcal{D}$ *is strong and also each derivation* $\mathcal{D}_i$ *is strong.*
- *Derivations of the form* $\langle \mathcal{D}, \mathcal{D}', \textsc{newm} \rangle$ *are strong, if both* $\mathcal{D}$ *and* $\mathcal{D}'$ *are strong.*

2. *We call a type* ϕ *strong if it does not contain* ω*; we call a type environment* Π *strong if for all* $x{:}\phi \in \Pi$*,* ϕ *is strong.*

Notice that a strong derivation need not derive a strong type. This is due to that fact that a strong derivation is not required to use a strong type environment. For example, if the type ϕ of a variable x in the type environment Π contains ω, then a non-strong type may be derived for x using the (VAR) rule. Similarly, if a formal parameter x does not appear in the body of some method m, then that method body

may be typed using an environment that associates ω with x; then, using the (NEWM) rule, a method type containing ω may be derived for a `new C(e⃗)` expression, for a class C containing method m. The crucial feature of strong derivations is that they cannot derive ω as a type for an expression. Furthermore, while a strong (sub)derivation may derive a method type containing ω as an argument type, the *invocation* of that method cannot then be typed with a strong derivation, since no expression passed as that argument can be assigned ω in a subderivation. This restriction is relaxed for ω-safe derivations, which are defined as follows.

Definition 4.9 (ω-safe Derivations)**.** *1. ω-safe derivations are defined inductively as follows:*

- *Derivations of the form $\langle$VAR$\rangle$ are ω-safe.*
- *Derivations of the form $\langle\overrightarrow{\mathcal{D}}_n, \text{JOIN}\rangle$, $\langle\overrightarrow{\mathcal{D}}_n, \text{OBJ}\rangle$ and $\langle\overrightarrow{\mathcal{D}}_n, \text{NEWF}\rangle$ are ω-safe, if each derivation $\mathcal{D}_i$ is ω-safe.*
- *Derivations of the form $\langle\mathcal{D}, \text{FLD}\rangle$ are ω-safe, if $\mathcal{D}$ is ω-safe.*
- *Derivations of the form $\langle\mathcal{D}, \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle$ are ω-safe, if $\mathcal{D}$ is ω-safe and for each $\mathcal{D}_i$ either $\mathcal{D}_i$ is ω-safe or $\mathcal{D}_i$ is of the form $\langle\omega\rangle :: \Pi \vdash e : \omega$.*
- *Derivations of the form $\langle\mathcal{D}, \mathcal{D}', \text{NEWM}\rangle$ are ω-safe, if both $\mathcal{D}$ and $\mathcal{D}'$ are ω-safe.*

2. We call an environment Π ω-safe if, for all $x{:}\phi \in \Pi$, $\phi = \omega$ or ϕ is strong.

Continuing with the definition of derivation reduction we point out that, just as substitution is the main engine for reduction on expressions, a notion of substitution for derivations will form the basis of derivation reduction. The notion of derivation substitution essentially replaces (sub)derivations of the form $\langle$VAR$\rangle :: \Pi \vdash x : \sigma$ by derivations $\mathcal{D} :: \Pi' \vdash e : \sigma$. This is illustrated in the following example.

Example 4.10 (Derivation Reduction)**.** *Consider the derivations below for two expressions e_1 and e_2:*

$$\frac{\boxed{\mathcal{D}_1}}{\Pi \vdash e_1 : \langle m{:}(\sigma_1 \cap \sigma_2) \to \tau\rangle} \qquad \frac{\boxed{\mathcal{D}_2'}\ \Pi \vdash e_2 : \sigma_1 \quad \boxed{\mathcal{D}_2''}\ \Pi \vdash e_2 : \sigma_2}{\mathcal{D}_2 :: \Pi \vdash e_2 : \sigma_1 \cap \sigma_2}$$

and also the following derivation $\mathcal{D}$ of the method invocation $x.m(y)$, where the environment $\Pi' = \{x{:}\langle m{:}(\sigma_1 \cap \sigma_2) \to \tau\rangle, y{:}\sigma_1 \cap \sigma_2, \}$:

$$\frac{\dfrac{}{\Pi' \vdash x : \langle m{:}(\sigma_1 \cap \sigma_2) \to \tau\rangle} \qquad \dfrac{\dfrac{}{\Pi' \vdash y : \sigma_1} \quad \dfrac{}{\Pi' \vdash y : \sigma_2}}{\Pi \vdash y : \sigma_1 \cap \sigma_2}}{\mathcal{D} :: \Pi' \vdash x.m(y) : \tau}$$

Let $\mathcal{S}$ denote the derivation substitution $\{x \mapsto \mathcal{D}_1, y \mapsto \mathcal{D}_2\}$; then the result of substituting $\mathcal{D}_1$ for x and $\mathcal{D}_2$ for y in $\mathcal{D}$ is the following derivation, where instances of the (VAR) *rule in $\mathcal{D}$ have been replaced by the appropriate (sub) derivations in $\mathcal{D}_1$ and $\mathcal{D}_2$:*

$$\frac{\dfrac{\boxed{\mathcal{D}_1}}{\Pi \vdash e_1 : \langle m{:}(\sigma_1 \cap \sigma_2) \to \tau\rangle} \qquad \dfrac{\boxed{\mathcal{D}_2'}\ \Pi \vdash e_2 : \sigma_1 \quad \boxed{\mathcal{D}_2''}\ \Pi \vdash e_2 : \sigma_2}{\Pi \vdash e_2 : \sigma_1 \cap \sigma_2}}{\mathcal{D}^{\mathcal{S}} :: \Pi \vdash e_1.m(e_2) : \tau}$$

Formally, derivation substitution is defined as follows.

Definition 4.11 (Derivation Substitution)**.** *1. A* derivation substitution *is a partial function from derivations to derivations.*

2. *Let $\mathcal{D}_1 :: \Pi' \vdash e_1 : \phi_1, \ldots, \mathcal{D}_n :: \Pi' \vdash e_n : \phi_n$ be derivations, and $x_1, \ldots, x_n$ be distinct variables; then $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$ is a derivation substitution* based on Π'. *When each $\mathcal{D}_i$ is strong then we say that $\mathcal{S}$ is also strong. $\mathcal{S}$ is ω-safe when each $\mathcal{D}_i$ is either ω-safe or an instance of the (ω) rule.*

3. *If $\mathcal{D} :: \Pi \vdash e : \phi$ is a derivation such that $\Pi \subseteq \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$, then we say that $\mathcal{S}$ is* applicable *to $\mathcal{D}$, and the result of applying $\mathcal{S}$ to $\mathcal{D}$ (written $\mathcal{D}^{\mathcal{S}}$) is defined inductively as follows (where S is the term substitution induced by $\mathcal{S}$, i.e. $\mathsf{S} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$):*

   ($\mathcal{D} = \langle \textsc{var} \rangle :: \Pi \vdash x : \sigma$): *Then there are two cases to consider.*

   a) *Either $x{:}\sigma \in \Pi$ and so $x = x_i$ for some $i \in \overline{n}$ with $\mathcal{D}_i :: \Pi' \vdash e_i : \sigma$, then $\mathcal{D}^{\mathcal{S}} = \mathcal{D}_i$;*

   b) *or $x{:}\phi \in \Pi$ with $\phi = \sigma_1 \cap \ldots \cap \sigma_{n'}$ and $\sigma = \sigma_j$ for some $j \in \overline{n'}$. Also in this case $x = x_i$ for some $i \in \overline{n}$, so then $\mathcal{D}_i = \langle \mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}, \textsc{join} \rangle :: \Pi' \vdash e_i : \phi$ and $\mathcal{D}^{\mathcal{S}} = \mathcal{D}'_j :: \Pi' \vdash e_i : \sigma_j$.*

   ($\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}', \textsc{newM} \rangle :: \Pi \vdash \mathtt{new}\ C(\vec{e}) : \langle m : (\vec{\phi}) \to \sigma \rangle$):
   Then $\mathcal{D}^{\mathcal{S}} = \langle \mathcal{D}_b, \mathcal{D}'^{\mathcal{S}}, \textsc{newM} \rangle :: \Pi \vdash \mathtt{new}\ C(\vec{e})^{\mathsf{S}} : \langle m : (\vec{\phi}) \to \sigma \rangle$

   ($\mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: \Pi \vdash e : \phi, r \notin \{(\textsc{var}), (\textsc{newM})\}$):
   Then $\mathcal{D}^{\mathcal{S}} = \langle \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, r \rangle :: \Pi' \vdash e^{\mathsf{S}} : \phi$.

   Notice that the last case includes as a special case the base case of derivations of the form $\langle \omega \rangle :: \Pi \vdash e : \omega$.

4. *We extend the weakening operation to derivation substitutions as follows: for a derivation substitution $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1 :: \Pi \vdash e_1 : \phi_1, \ldots, x_n \mapsto \mathcal{D}_n :: \Pi \vdash e_n : \phi_n\}$, $\mathcal{S}[\Pi' \trianglelefteq \Pi]$ is the derivation substitution $\{x_1 \mapsto \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, x_n \mapsto \mathcal{D}_n[\Pi' \trianglelefteq \Pi]\}$.*

Lemma 4.12 (Soundness of Derivation Substitution)**.** *Let $\mathcal{D} :: \Pi \vdash e : \phi$ be a derivation and $\mathcal{S}$ be a derivation substitution based on Π' and applicable to $\mathcal{D}$; then $\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash e^{\mathsf{S}} : \phi$ where S is the term substitution induced by $\mathcal{S}$, is well-defined.*

Proof. By easy induction on the structure of derivations. Notice that when a substitution is applicable to a derivation then it is also applicable to its subderivations, and so when applying the inductive hypothesis we leave this to be noted implicitly.

$\langle \omega \rangle$: Then $\mathcal{D} :: \Pi \vdash e : \omega$. Notice that e^{S} is always well-defined and so and by the (ω) rule, so is the derivation $\langle \omega \rangle :: \Pi' \vdash e^{\mathsf{S}} : \omega$. By the definition of derivation substitution $\mathcal{D}^{\mathcal{S}} = \langle \omega \rangle :: \Pi' \vdash e^{\mathsf{S}} : \omega$ so it follows that $\mathcal{D}^{\mathcal{S}}$ is well-defined and $\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash e^{\mathsf{S}} : \omega$.

$\langle \textsc{var} \rangle$: Then $\mathcal{D} = \langle \textsc{var} \rangle :: \Pi \vdash x : \sigma$. Let $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$; notice, by definition, that each $\mathcal{D}_i$ is well-defined (and therefore so are its subderivations). By the definition of derivation substitution $\mathcal{D}^{\mathcal{S}}$ is (a subderivation of) some $\mathcal{D}_j$, and so therefore is a well-defined derivation. Also, since $\mathcal{S}$ is applicable to $\mathcal{D}$, it follows that $x = x_k$ for some $k \in \overline{n}$, thus $x^{\mathsf{S}} = x_k{}^{\mathsf{S}} = e_k$, and by the definition of derivation substitution $\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash e_k : \sigma$.

$\langle \mathcal{D}', \textsc{fld} \rangle$: Then $\mathcal{D} = \langle \mathcal{D}', \textsc{fld} \rangle :: \Pi \vdash e.f : \sigma$ and $\mathcal{D}' :: \Pi \vdash e : \langle f : \sigma \rangle$. By induction $\mathcal{D}'^{\mathcal{S}} :: \Pi' \vdash e^{\mathsf{S}} : \langle f : \sigma \rangle$ and is well-defined. Then by the $(\textsc{fld})$ rule $\langle \mathcal{D}'^{\mathcal{S}}, \textsc{fld} \rangle :: \Pi' \vdash e^{\mathsf{S}}.f : \sigma$ is also a well-defined derivation. Since $e^{\mathsf{S}}.f = e.f^{\mathsf{S}}$ it follows from the definition of derivation substitution that $\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash e.f^{\mathsf{S}} : \sigma$ and is well-defined.

(INVK), (OBJ), (NEWF), (NEWM), (JOIN): These cases are similar to (FLD) and follow straightforwardly by induction. □

Derivation substitution preserves strong and ω-safe derivations.

Lemma 4.13. *If $\mathcal{D}$ is strong (ω-safe) then, for any strong (ω-safe) derivation substitution $\mathcal{S}$ applicable to $\mathcal{D}$, $\mathcal{D}^{\mathcal{S}}$ is also strong (ω-safe).*

Proof. By straightforward induction on the structure of $\mathcal{D}$.

$\langle\omega\rangle$: Vacuously true since $\langle\omega\rangle$ derivations are neither strong nor ω-safe.

$\langle$VAR$\rangle$: Let $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$; then $\mathcal{D} = \langle\text{VAR}\rangle :: \Pi, x_j{:}\phi \vdash x : \sigma$ for some $j \in \overline{n}$ with $\mathcal{D}_j :: \Pi' \vdash e : \phi$ and $\phi \leqslant \sigma$. By Definition 4.11, $\mathcal{D}^{\mathcal{S}}$ is either $\mathcal{D}_j$ itself (if ϕ is strict), or one of its immediate subderivations (if ϕ is an intersection).

If $\mathcal{S}$ is strong, it follows by Definition 4.11 that each $\mathcal{D}_i$ is strong. In particular, this means that $\mathcal{D}_j$ is strong and, in the case that ϕ is an intersection, by Definition 4.8 it follows that the immediate subderivations of $\mathcal{D}_j$ are also strong. Thus, $\mathcal{D}^{\mathcal{S}}$ is strong.

If $\mathcal{S}$ is ω-safe, then each $\mathcal{D}_i$ is either ω-safe or an instance of the (ω) rule. We know that $\mathcal{D}_j$ cannot be an instance of the (ω) rule because if it were then, since $\mathcal{S}$ is applicable to $\mathcal{D}$, it would then follow that $\phi = \omega$ which cannot be the case since $\phi \leqslant \sigma$, which is strict. Thus, $\mathcal{D}_j$ is ω-safe and, in the case that ϕ is an intersection, by Definition 4.9 so are all of its immediate subderivations. Thus, $\mathcal{D}^{\mathcal{S}}$ is ω-safe.

$\langle\mathcal{D}', \text{FLD}\rangle$: Then $\mathcal{D} = \langle\mathcal{D}', \text{FLD}\rangle$ and by Definition 4.11 $\mathcal{D}^{\mathcal{S}} = \langle\mathcal{D}'^{\mathcal{S}}, \text{FLD}\rangle$. By induction $\mathcal{D}'^{\mathcal{S}}$ is strong (ω-safe), and so by Definition 4.8 (Definition 4.9) it follows that $\mathcal{D}^{\mathcal{S}}$ is also strong (ω-safe).

(INVK), (OBJ), (NEWF), (NEWM), (JOIN): These cases are similar to (FLD) and follow straightforwardly by induction. □

We also show that the operations of weakening and derivation substitution are commutative.

Lemma 4.14. *Let $\mathcal{D} :: \Pi'' \vdash e : \phi$ be a derivation and $\mathcal{S}$ be a derivation substitution based on Π and applicable to $\mathcal{D}$. Also let $[\Pi' \leqslant \Pi]$ be a weakening, then $\mathcal{D}^{\mathcal{S}}[\Pi' \leqslant \Pi] = \mathcal{D}^{\mathcal{S}[\Pi' \leqslant \Pi]}$.*

Proof. By induction on the structure of $\mathcal{D}$.

$\langle\omega\rangle$: Then $\mathcal{D} = \langle\omega\rangle :: \Pi'' \vdash e : \omega$. By Definition 4.11 $\mathcal{D}^{\mathcal{S}} = \langle\omega\rangle :: \Pi \vdash e^{\mathsf{S}} : \omega$ where S is the term substitution induced by $\mathcal{S}$. Then by Definition 4.5 $\mathcal{D}^{\mathcal{S}}[\Pi' \leqslant \Pi] = \langle\omega\rangle :: \Pi' \vdash e^{\mathsf{S}} : \omega$. Notice that by Definition 4.11 $\mathcal{S}[\Pi' \leqslant \Pi]$ is a derivation substitution still applicable to $\mathcal{D}$ but now based on Π'. Furthermore notice that S is also the term substitution induced by $\mathcal{S}[\Pi' \leqslant \Pi]$. Thus by Definition 4.11 again, $\mathcal{D}^{\mathcal{S}[\Pi' \leqslant \Pi]} = \langle\omega\rangle :: \Pi' \vdash e^{\mathsf{S}} : \omega = \mathcal{D}^{\mathcal{S}}[\Pi' \leqslant \Pi]$.

$\langle$VAR$\rangle$: Then $\mathcal{D} = \langle\text{VAR}\rangle :: \Pi'' \vdash x : \sigma$. $\mathcal{S}$ is based on $\Pi$and applicable to $\mathcal{D}$ so let $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1 :: \Pi \vdash e_1 : \phi_1, \ldots, x_1 \mapsto \mathcal{D}_n :: \Pi \vdash e_n : \phi_n\}$ with $\Pi'' \subseteq \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$. Then by Definition 4.11,

$$\mathcal{S}[\Pi' \leqslant \Pi] = \{x_1 \mapsto \mathcal{D}_1[\Pi' \leqslant \Pi] :: \Pi' \vdash e_1 : \phi_1, \ \ldots\ , x_n \mapsto \mathcal{D}_n[\Pi' \leqslant \Pi] :: \Pi' \vdash e_n : \phi_n\}$$

Now, there are two cases to consider:

1. $x{:}\sigma \in \Pi''$, then since $\Pi'' \subseteq \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$ it follows that $x = x_i$ for some $i \in \overline{n}$ and $\phi_i = \sigma$. By Definition 4.11 $\mathcal{D}^S = \mathcal{D}_i :: \Pi \vdash e_i : \sigma$ and then by Definition 4.5 $\mathcal{D}^S[\Pi' \trianglelefteqslant \Pi] = \mathcal{D}_i[\Pi' \trianglelefteqslant \Pi] :: \Pi' \vdash e_i : \sigma$. Furthermore, by Definition 4.11 $\mathcal{D}^{S[\Pi' \trianglelefteqslant \Pi]} = \mathcal{D}_i[\Pi' \trianglelefteqslant \Pi] :: \Pi' \vdash e_i : \sigma$. Thus $\mathcal{D}^S[\Pi' \trianglelefteqslant \Pi] = \mathcal{D}^{S[\Pi' \trianglelefteqslant \Pi]}$.

2. $x{:}\phi \in \Pi$ with $\phi = \sigma_1 \cap \ldots \cap \sigma_{n'}$ and $\sigma = \sigma_j$ for some $j \in \overline{n'}$. Since $\Pi'' \subseteq \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$ it follows that $x = x_i$ for some $i \in \overline{n}$ and $\phi_i = \phi$. So then $\mathcal{D}_i = \langle \mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}, \textsc{join} \rangle$ with $\mathcal{D}'_k :: \Pi \vdash e_i : \sigma_k$ for each $k \in \overline{n'}$. By Definition 4.11 $\mathcal{D}^S = \mathcal{D}'_j :: \Pi \vdash e_i : \sigma_j$ and by Definition 4.5 $\mathcal{D}^S[\Pi' \trianglelefteqslant \Pi] = \mathcal{D}'_j[\Pi' \trianglelefteqslant \Pi] :: \Pi' \vdash e_i : \sigma_j$. Furthermore, by Definition 4.5

$$\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi] = \langle \mathcal{D}'_1[\Pi' \trianglelefteqslant \Pi], \ldots, \mathcal{D}''_{n'}[\Pi' \trianglelefteqslant \Pi], \textsc{join} \rangle$$

So by Definition 4.11 $\mathcal{D}^{S[\Pi' \trianglelefteqslant \Pi]} = \mathcal{D}_i[\Pi' \trianglelefteqslant \Pi]$. Thus $\mathcal{D}^S[\Pi' \trianglelefteqslant \Pi] = \mathcal{D}^{S[\Pi' \trianglelefteqslant \Pi]}$.

$\langle \mathcal{D}', \textsc{fld} \rangle$:

$$\begin{array}{lll}
\mathcal{D} = \langle \mathcal{D}', \textsc{fld} \rangle & \Rightarrow & \text{(Def. 4.11)} \\
\mathcal{D}^S = \langle \mathcal{D}'^S, \textsc{fld} \rangle & \Rightarrow & \text{(Def. 4.5)} \\
\mathcal{D}^S[\Pi' \trianglelefteqslant \Pi] = \langle \mathcal{D}'^S[\Pi' \trianglelefteqslant \Pi], \textsc{fld} \rangle & \Rightarrow & \text{(Inductive Hypothesis)} \\
\mathcal{D}^S[\Pi' \trianglelefteqslant \Pi] = \langle \mathcal{D}'^{S[\Pi' \trianglelefteqslant \Pi]}, \textsc{fld} \rangle & \Rightarrow & \text{(Def. 4.11)} \\
\mathcal{D}^S[\Pi' \trianglelefteqslant \Pi] = \mathcal{D}^{S[\Pi' \trianglelefteqslant \Pi]} & &
\end{array}$$

(INVK), (OBJ), (NEWF), (NEWM), (JOIN): These cases are similar to (FLD) and follow straightforwardly by induction. □

Definition 4.15 (Identity Substitutions)**.** *Each environment* Π *induces a derivation substitution* Id_Π *which is called the* identity substitution *for* Π*. Let* $\Pi = \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$*; then* $Id_\Pi \triangleq \{x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$ *where for each* $i \in \overline{n}$*:*

- *If* $\phi_i = \omega$ *then* $\mathcal{D}_i = \langle \omega \rangle :: \Pi \vdash x_i : \omega$*;*
- *If* ϕ_i *is a strict type* σ *then* $\mathcal{D}_i = \langle \textsc{var} \rangle :: \Pi \vdash x_i : \sigma$*;*
- *If* $\phi_i = \sigma_1 \cap \ldots \cap \sigma_n$ *for some* $n \geq 2$ *then* $\mathcal{D}_i = \langle \overrightarrow{\mathcal{D}'}_n, \textsc{join} \rangle :: \Pi \vdash x : \sigma_1 \cap \ldots \cap \sigma_n$*, with* $\mathcal{D}'_j = \langle \textsc{var} \rangle :: \Pi \vdash x_i : \sigma_j$ *for each* $j \in \overline{n}$*.*

Notice that for every environment Π*, the identity substitution* Id_Π *is* based on Π.

It is easy to show that Id_Π is indeed the identity for the substitution operation on derivations using Π.

Lemma 4.16. *Let* $\mathcal{D} :: \Pi \vdash e : \phi$ *and* Id_Π *be the identity substitution for* Π*; then* $\mathcal{D}^{Id_\Pi} = \mathcal{D}$*.*

Proof. By straightforward induction on the structure of $\mathcal{D}$. □

Before defining the notion of derivation reduction itself, we first define the auxiliary notion of *advancing* a derivation. This is an operation which contracts redexes at some given position in expressions covered by ω in derivations. This operation will be used to reduce derivations which introduce intersections.

Definition 4.17 (Advancing)**.** *1. The* advance *operation* $\rightsquigarrow$ *on expressions contracts the redex at a given position* p *in* e *if it exists, and is undefined otherwise. It is defined as the smallest relation on tuples* (p, e) *and expressions satisfying the following properties (where we write* $e \overset{p}{\rightsquigarrow} e'$ *to*

mean $((p,e),e') \in \rightsquigarrow$*):*

$$\mathcal{F}(C) = \vec{f}_n \;\&\; e = \mathfrak{C}_p[\texttt{new } C(\vec{e}_n).f_i] \quad (i \in \overline{n}) \quad \Rightarrow \quad e \overset{p}{\rightsquigarrow} \mathfrak{C}_p[e_i]$$

$$\mathcal{M}b(C,m) = (\vec{x}_n, e_b) \;\&\; e = \mathfrak{C}_p[\texttt{new } C(\vec{e'}).m(\vec{e}_n)] \quad \Rightarrow \quad e \overset{p}{\rightsquigarrow} \mathfrak{C}_p[{e_b}^{\mathsf{S}}]$$
$$\textit{where } \mathsf{S} = \{\texttt{this} \mapsto \texttt{new } C(\vec{e'}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$$

2. *We extend* $\rightsquigarrow$ *to derivations via the following inductive definition (where we write* $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$ *to mean* $((p,\mathcal{D}),\mathcal{D}') \in \rightsquigarrow$*):*

   a) *If* $e \overset{p}{\rightsquigarrow} e'$*, then* $\mathcal{D} :: \Pi \vdash e : \omega \overset{p}{\rightsquigarrow} \langle\omega\rangle :: \Pi \vdash e' : \omega$*.*

   b) *If* $\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash e.f : \sigma$ *and* $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$*, then* $\langle \mathcal{D}, \textsc{fld}\rangle \overset{0\cdot p}{\rightsquigarrow} \langle \mathcal{D}', \textsc{fld}\rangle$*.*

   c) *If* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash e.m(\vec{e}_n) : \sigma$ *and* $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$*, then* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle \overset{0\cdot p}{\rightsquigarrow} \langle \mathcal{D}', \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle$*.*

   d) *If* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash e.m(\vec{e}_n) : \sigma$ *and* $\mathcal{D}_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j$ *for some* $j \in \overline{n}$*, then* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle \overset{j\cdot p}{\rightsquigarrow} \langle \mathcal{D}, \overrightarrow{\mathcal{D}'}_n, \textsc{invk}\rangle$ *where* $\mathcal{D}'_i = \mathcal{D}_i$ *for each* $i \in \overline{n}$ *such that* $i \neq j$*.*

   e) *If* $\langle \overrightarrow{\mathcal{D}}_n, \textsc{obj}\rangle :: \Pi \vdash \texttt{new } C(\vec{e}_n) : C$ *and* $\mathcal{D}_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j$ *for some* $j \in \overline{n}$*, then* $\langle \overrightarrow{\mathcal{D}}_n, \textsc{obj}\rangle \overset{j\cdot p}{\rightsquigarrow} \langle \overrightarrow{\mathcal{D}'}_n, \textsc{obj}\rangle$ *where* $\mathcal{D}'_i = \mathcal{D}_i$ *for each* $i \in \overline{n}$ *such that* $i \neq j$*.*

   f) *If* $\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle :: \Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f{:}\sigma\rangle$ *and* $\mathcal{D}_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j$ *for some* $j \in \overline{n}$*, then* $\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle \overset{j\cdot p}{\rightsquigarrow} \langle \overrightarrow{\mathcal{D}'}_n, \textsc{newF}\rangle$ *where* $\mathcal{D}'_i = \mathcal{D}_i$ *for each* $i \in \overline{n}$ *such that* $i \neq j$*.*

   g) *If* $\langle \mathcal{D}_b, \mathcal{D}, \textsc{newM}\rangle :: \Pi \vdash \texttt{new } C(\vec{e}) : \langle m{:}(\vec{\phi}) \to \sigma\rangle$ *and* $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$*, then* $\langle \mathcal{D}_b, \mathcal{D}, \textsc{newM}\rangle \overset{p}{\rightsquigarrow} \langle \mathcal{D}_b, \mathcal{D}', \textsc{newM}\rangle$*.*

   h) *If* $\langle \overrightarrow{\mathcal{D}}_n, \textsc{join}\rangle :: \Pi \vdash e : \phi$ *and* $\mathcal{D}_i \overset{p}{\rightsquigarrow} \mathcal{D}'_i$ *for each* $i \in \overline{n}$*, then* $\langle \overrightarrow{\mathcal{D}}_n, \textsc{join}\rangle \overset{p}{\rightsquigarrow} \langle \overrightarrow{\mathcal{D}'}_n, \textsc{join}\rangle$*.*

Notice that the advance operation does not change the structure *of derivations. Exactly the same rules are applied and the same types derived; only expressions which are typed with* ω *are altered.*

Lemma 4.18 (Soundness of Advancing)**.** *Let* $\mathcal{D} :: \Pi \vdash e : \phi$*; then* $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$ *for some* $\mathcal{D}'$ *if and only if a redex appears at position* p *in* e *and no derivation redex appears at* p *in* $\mathcal{D}$*, with* $e \overset{p}{\rightsquigarrow} e'$ *for some* e' *and* $\mathcal{D}' :: \Pi \vdash e' : \phi$*.*

Proof. By straightforward well-founded induction on $(p,\mathcal{D})$. □

The advance operation preserves strong (and ω-safe) typeability.

Lemma 4.19. *If* $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$ *is defined, and* $\mathcal{D}$ *is strong (*ω*-safe), then* $\mathcal{D}'$ *is also strong (*ω*-safe).*

Proof. Straightforward, by induction on the definition of the advance operation for derivations. □

The notion of derivation reduction is defined in two stages. First, the more specific notion of reduction at a certain position (i.e. within a given subderivation) is introduced. The full notion of derivation reduction is then a straightforward generalisation of this position-specific reduction over all positions.

Definition 4.20 (Derivation Reduction)**.** *1. The reduction of a derivation* $\mathcal{D}$ *at position* p *to* $\mathcal{D}'$ *is denoted by* $\mathcal{D} \xrightarrow{p} \mathcal{D}'$*, and is defined inductively on* $(p,\mathcal{D})$ *as follows:*

   a) *Let* $\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle :: \Pi \vdash \texttt{new } C(\vec{e}).f_i : \sigma$*; then* $\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle \xrightarrow{0} \mathcal{D}_i$ *for each* $i \in \overline{n}$*.*

   b) *Let* $\langle\langle \mathcal{D}_b, \mathcal{D}', \textsc{newM}\rangle, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash \texttt{new } C(\vec{e'}).m(\vec{e}_n) : \sigma$ *with* $\mathcal{M}b(C,m) = (\vec{x}_n, e_b)$*;*
   then $\langle\langle \mathcal{D}_b, \mathcal{D}', \textsc{newM}\rangle, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle \xrightarrow{0} {\mathcal{D}_b}^{\mathcal{S}}$*, where* $\mathcal{S} = \{\texttt{this} \mapsto \mathcal{D}', x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$*.*

   c) *If* $\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash e.f : \sigma$ *and* $\mathcal{D} \xrightarrow{p} \mathcal{D}'$*, then* $\langle \mathcal{D}, \textsc{fld}\rangle \xrightarrow{0\cdot p} \langle \mathcal{D}', \textsc{fld}\rangle$*.*

d) *If* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle :: \Pi \vdash \mathtt{e.m}(\vec{\mathtt{e}}_n) : \sigma$ *and* $\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}'$, *then* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle \xrightarrow{0 \cdot p} \langle \mathcal{D}', \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle$.

e) *If* $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle :: \Pi \vdash \mathtt{e.m}(\vec{\mathtt{e}}_n) : \sigma$ *and* $\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j$ *for some* $j \in \overline{n}$,
then $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle \xrightarrow{j \cdot p}_{\triangleright} \langle \mathcal{D}, \overrightarrow{\mathcal{D}'}_n, \text{INVK}\rangle$ *where* $\mathcal{D}'_i = \mathcal{D}_i$ *for each* $i \in \overline{n}$ *such that* $i \neq j$.

f) *If* $\langle \overrightarrow{\mathcal{D}}_n, \text{OBJ}\rangle :: \Pi \vdash \mathtt{new\ C}(\vec{\mathtt{e}}_n) : \mathtt{C}$ *and* $\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j$ *for some* $j \in \overline{n}$,
then $\langle \overrightarrow{\mathcal{D}}_n, \text{OBJ}\rangle \xrightarrow{j \cdot p}_{\triangleright} \langle \overrightarrow{\mathcal{D}'}_n, \text{OBJ}\rangle$ *where* $\mathcal{D}'_i = \mathcal{D}_i$ *for each* $i \in \overline{n}$ *such that* $i \neq j$.

g) *If* $\langle \overrightarrow{\mathcal{D}}_n, \text{NEWF}\rangle :: \Pi \vdash \mathtt{new\ C}(\vec{\mathtt{e}}_n) : \langle \mathtt{f} : \sigma\rangle$ *and* $\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j$ *for some* $j \in \overline{n}$,
then $\langle \overrightarrow{\mathcal{D}}_n, \text{NEWF}\rangle \xrightarrow{j \cdot p}_{\triangleright} \langle \overrightarrow{\mathcal{D}'}_n, \text{NEWF}\rangle$ *where* $\mathcal{D}'_i = \mathcal{D}_i$ *for each* $i \in \overline{n}$ *such that* $i \neq j$.

h) *If* $\langle \mathcal{D}_b, \mathcal{D}, \text{NEWM}\rangle :: \Pi \vdash \mathtt{new\ C}(\vec{\mathtt{e}}) : \langle m : (\vec{\phi}) \to \sigma\rangle$ *and* $\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}'$,
then $\langle \mathcal{D}_b, \mathcal{D}, \text{NEWM}\rangle \xrightarrow{p}_{\triangleright} \langle \mathcal{D}_b, \mathcal{D}', \text{NEWM}\rangle$.

i) *If* $\langle \overrightarrow{\mathcal{D}}_n, \text{JOIN}\rangle :: \Pi \vdash \mathtt{e} : \phi$, $\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j$ *for some* $j \in \overline{n}$ *and for each* $i \in \overline{n}$ *such that* $i \neq j$, *either* $\mathcal{D}_i \xrightarrow{p}_{\triangleright} \mathcal{D}'_i$ *or* $\mathcal{D}_i \rightsquigarrow_{\triangleright} \mathcal{D}'_i$, *then* $\langle \overrightarrow{\mathcal{D}}_n, \text{JOIN}\rangle \xrightarrow{p}_{\triangleright} \langle \overrightarrow{\mathcal{D}'}_n, \text{JOIN}\rangle$.

2. *The full reduction relation on derivations* $\to_{\mathfrak{D}}$ *is defined by:*

$$\mathcal{D} \to_{\mathfrak{D}} \mathcal{D}' \quad \triangleq \quad \exists\, p\, [\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}']$$

The reflexive and transitive closure of $\to_{\mathfrak{D}}$ *is denoted by* $\to^*_{\mathfrak{D}}$.

3. *We write* $SN(\mathcal{D})$ *whenever the derivation* $\mathcal{D}$ *is strongly normalising with respect to* $\to^*_{\mathfrak{D}}$.
Similarly to reduction for expressions, if $\mathcal{D} \to_{\mathfrak{D}} \mathcal{D}'$ *then we call* $\mathcal{D}$ *a* derivation redex *and* $\mathcal{D}'$ *its* derivation contractum.

The following properties hold of derivation reduction. They are used in the proofs of Theorem 4.27 and Lemma 4.30.

Lemma 4.21. 1. $SN(\langle \mathcal{D}, \text{FLD}\rangle :: \Pi \vdash \mathtt{e.f} : \sigma) \Leftrightarrow SN(\mathcal{D} :: \Pi \vdash \mathtt{e} : \langle \mathtt{f} : \sigma\rangle)$

2. $SN(\langle \mathcal{D}, \mathcal{D}_1, \ldots, \mathcal{D}_n, \text{INVK}\rangle :: \Pi \vdash \mathtt{e.m}(\vec{\mathtt{e}}_n) : \sigma) \Rightarrow SN(\mathcal{D})\ \&\ \forall i \in \overline{n}\ [SN(\mathcal{D}_i)]$

3. *For neutral contexts* $\mathfrak{C}$,
$SN(\mathcal{D}' :: \Pi \vdash \mathfrak{C}[x] : \langle m : (\vec{\phi}_n) \to \sigma\rangle)\ \&\ \forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi_i)] \Rightarrow$
$SN(\langle \mathcal{D}', \mathcal{D}_1, \ldots, \mathcal{D}_n, \text{INVK}\rangle :: \Pi \vdash \mathfrak{C}[x].\mathtt{m}(\vec{\mathtt{e}}_n) : \sigma)$

4. $SN(\langle \overrightarrow{\mathcal{D}}_n, \text{OBJ}\rangle :: \Pi \vdash \mathtt{new\ C}(\vec{\mathtt{e}}_n) : \mathtt{C}) \Leftrightarrow \exists\, \vec{\phi}_n\ [\forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi_i)]]$

5. $SN(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \text{JOIN}\rangle :: \Pi \vdash \mathtt{e} : \sigma_1 \cap \ldots \cap \sigma_n) \Leftrightarrow \forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash \mathtt{e} : \sigma_i)]$

6. $SN(\mathcal{D}[\Pi' \trianglelefteq \Pi]) \Leftrightarrow SN(\mathcal{D})$

7. *Let* $\mathtt{C}$ *be a class such that* $\mathcal{F}(\mathtt{C}) = \vec{\mathtt{f}}_n$, *then for all* $j \in \overline{n}$:
$SN(\langle \overrightarrow{\mathcal{D}}_n, \text{NEWF}\rangle :: \Pi \vdash \mathtt{new\ C}(\vec{\mathtt{e}}_n) : \langle \mathtt{f}_j : \sigma\rangle) \Leftrightarrow \exists\, \vec{\phi}_n\ [\sigma \trianglelefteq \phi_j\ \&\ \forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi_i)]]$

8. *Let* $\mathtt{C}$ *be a class such that* $\mathcal{F}(\mathtt{C}) = \vec{\mathtt{f}}_n$, *then for all* $j \in \overline{n}$:
$SN(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j] : \sigma)\ \&\ \forall i \in \overline{n}\ [i \neq j \Rightarrow \exists\, \phi\ [SN(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi)]]$
$\Rightarrow SN(\mathfrak{D}_{(p,\sigma')}[\langle \overrightarrow{\mathcal{D}}_n, \text{NEWF}\rangle, \text{FLD}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new\ C}(\vec{\mathtt{e}}_n).\mathtt{f}_j] : \sigma)$

9. *Let* $\mathtt{C}$ *be a class such that* $\mathcal{M}b(\mathtt{C},m) = (\vec{x}_n, e_b)$ *and* $\mathcal{D}_b :: \{\mathtt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash e_b : \sigma'$*, then for all derivation contexts* $\mathfrak{D}_{(p,\sigma')}$ *and expression contexts* $\mathfrak{C}$*:*

$SN(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \sigma)$ & $SN(\mathcal{D}_0 :: \Pi \vdash \mathtt{new\ C}(\overline{e}^{\star}) : \psi)$ &

$\forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)] \Rightarrow SN(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new\ C}(\overline{e}^{\star}).m(\overrightarrow{e}_n)] : \sigma)$

where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM}\rangle :: \Pi \vdash \mathtt{new\ C}(\overline{e}^{\star}) : \langle m{:}(\vec{\phi}_n) \to \sigma'\rangle$,

$\mathcal{S} = \{\mathtt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$,

$\mathsf{S} = \{\mathtt{this} \mapsto \mathtt{new\ C}(\overline{e}^{\star}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$

Proof. By Definition 4.20 □

Our notion of derivation reduction is not only *sound* (i.e. produces valid derivations) but, most importantly, we have that it corresponds to reduction on expressions.

Lemma 4.22. $\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}'$ *if and only if there is a derivation redex at position* p *in* $\mathcal{D}$*.*

Proof. (if): By easy induction on the structure of p.
(only if): By easy induction on definition of derivation reduction. □

Theorem 4.23 (Soundness of Derivation Reduction)**.** *If* $\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}'$*, then* $\mathcal{D}'$ *is a well-defined derivation, i.e. there exists some* e' *such that* $\mathcal{D}' :: \Pi \vdash e' : \phi$*; moreover, then* $e \overset{p}{\rightsquigarrow} e'$*.*

Proof. By induction on the definition of derivation reduction. The interesting cases are the two redex cases, and also the case for (JOIN), since in general there may be more than one redex to contract (i.e. corresponding reductions and advances must be made in *each* subderivation simultaneously). The other cases follow straightforwardly by induction: we demonstrate the case for field access.

$(\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle :: \Pi \vdash \mathtt{new\ C}(\overline{e}).f_i : \sigma \xrightarrow{0}_{\triangleright} \mathcal{D}_i, i \in \overline{n})$:

By Definition 4.20, $\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle :: \Pi \vdash \mathtt{new\ C}(\overline{e}).f_i : \sigma$ is a well-defined derivation, and so:

- by (FLD), $\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle :: \Pi \vdash \mathtt{new\ C}(\overline{e}) : \langle f_i : \sigma\rangle$ is a well-defined derivation;
- by (NEWF), $\mathcal{D}_j :: \Pi \vdash e_j : \phi_j$ is a well-defined derivation for each $j \in \overline{n}$, with $\phi_j = \sigma$.

In particular $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ is a well-defined derivation. Furthermore notice that by Definition 3.3, $\mathtt{new\ C}(\overline{e}).f_i \to e_i$. Also notice that by Definition 4.3, $\mathtt{new\ C}(\overline{e}).f_i = \mathfrak{C}_0[\mathtt{new\ C}(\overline{e}).f_i]$ and $e_i = \mathfrak{C}_0[e_i]$ where $\mathfrak{C}$ is the empty context $[\,]$. Thus by Definition 4.17, $\mathtt{new\ C}(\overline{e}).f_i \overset{0}{\rightsquigarrow} e_i$.

$(\langle\langle \mathcal{D}_b, \mathcal{D}', \textsc{newM}\rangle, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash \mathtt{new\ C}(\overline{e}^{\star}).m(\overrightarrow{e}_n) : \sigma \xrightarrow{0}_{\triangleright} \mathcal{D}_b{}^{\mathcal{S}})$:
with $\mathcal{M}b(\mathtt{C},m) = (\vec{x}_n, e_b)$, where $\mathcal{S} = \{\mathtt{this} \mapsto \mathcal{D}', x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$.

By Definition 4.20, $\langle\langle \mathcal{D}_b, \mathcal{D}', \textsc{newM}\rangle, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash \mathtt{new\ C}(\overline{e}^{\star}).m(\overrightarrow{e}_n) : \sigma$ is a well-defined derivation, and so:

by (INVK): 1. $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ is a well-defined derivation for each $i \in \overline{n}$; and
2. $\langle \mathcal{D}_b, \mathcal{D}', \textsc{newM}\rangle :: \Pi \vdash \mathtt{new\ C}(\overline{e}^{\star}) : \langle m{:}(\vec{\phi}_n) \to \sigma\rangle$ is a well-defined derivation.

by (NEWM): 1. $\mathcal{D}' :: \Pi \vdash \mathtt{new\ C}(\overline{e}^{\star}) : \psi$ is a well-defined derivation; and
2. by (NEWM), $\mathcal{D}_b :: \{\mathtt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash e_b : \sigma$ is a well-defined derivation.

Then by Definition 4.11, $\mathcal{S}$ is a well-defined derivation substitution based on Π, and applicable to $\mathcal{D}_b$. By Lemma 4.12, it follows that $\mathcal{D}_b{}^{\mathcal{S}} :: \Pi \vdash e_b{}^{\mathsf{S}} : \sigma$ is a well-defined derivation, where

$\mathsf{S} = \{\texttt{this} \mapsto \texttt{new } C(\overrightarrow{e}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the term substitution induced by $\mathcal{S}$. Furthermore, notice that by Definition 3.3, $\texttt{new } C(\overrightarrow{e}).m(\overrightarrow{e}_n) \to {e_b}^{\mathsf{S}}$. Also notice that by Definition 4.3, $\texttt{new } C(\overrightarrow{e}).m(\overrightarrow{e}_n) = \mathfrak{C}_0[\texttt{new } C(\overrightarrow{e}).m(\overrightarrow{e}_n)]$ and ${e_b}^{\mathsf{S}} = \mathfrak{C}_0[{e_b}^{\mathsf{S}}]$, where $\mathfrak{C}$ is the empty context $[\,]$. Thus by Definition 4.17, $\texttt{new } C(\overrightarrow{e}).m(\overrightarrow{e}_n) \overset{0}{\rightsquigarrow} {e_b}^{\mathsf{S}}$.

$(\langle \overrightarrow{\mathcal{D}}_n, \textsc{join} \rangle \overset{p}{\triangleright} \langle \overrightarrow{\mathcal{D}'}_n, \textsc{join} \rangle)$:

with $\mathcal{D}_j \overset{p}{\triangleright} \mathcal{D}'_j$ for some $j \in \overline{n}$, and for each $i \in \overline{n}$ such that $i \neq j$, either $\mathcal{D}_i \overset{p}{\triangleright} \mathcal{D}'_i$ or $\mathcal{D}_i \overset{p}{\rightsquigarrow} \mathcal{D}'_i$ as well as $\langle \overrightarrow{\mathcal{D}}_n, \textsc{join} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n$. Since $\mathcal{D}_j \overset{p}{\triangleright} \mathcal{D}'_j$ for some $j \in \overline{n}$, it follows by the inductive hypothesis that $\mathcal{D}'_j :: \Pi \vdash e' : \sigma_j$ is a well-defined derivation and $e \overset{p}{\rightsquigarrow} e'$ for some e'. Notice that by Definition 4.3, there is then an expression context $\mathfrak{C}_p$ such that $e = \mathfrak{C}_p[e_r]$ for some redex e_r with $e_r \to e_c$ and $e' = \mathfrak{C}_p[e_c]$. Now, we examine each $\mathcal{D}'_i$ for $i \in \overline{n}$ such that $i \neq j$. For each such i, there are two possibilities:

1. $\mathcal{D}_i \overset{p}{\triangleright} \mathcal{D}'_i$; then by the inductive hypothesis it follows that there is some expression e'' such that $\mathcal{D}'_i :: \Pi \vdash e'' : \sigma_i$ is a well-defined derivation and $e \overset{p}{\rightsquigarrow} e''$. Then, by Definition 4.3, there is then an expression context $\mathfrak{C}'_p$ such that $e = \mathfrak{C}'_p[e'_r]$ for some redex e'_r with $e'_r \to e'_c$ and $e'' = \mathfrak{C}'_p[e'_c]$. It follows that $\mathfrak{C}'_p[e'_r] = e\mathfrak{C}_p[e_r]$, and so $\mathfrak{C}'_p = \mathfrak{C}_p$ and $e'_r = e_r$. Thus $e'_c = e_c$ and $e'' = \mathfrak{C}'_p[e'_c] = \mathfrak{C}_p[e_c] = e'$.
2. $\mathcal{D}_i \overset{p}{\rightsquigarrow} \mathcal{D}'_i$, in which case it follows by Lemma 4.18 that $e \overset{p}{\rightsquigarrow} e''$ for some expression e'' with $\mathcal{D}'_i :: \Pi \vdash e'' : \sigma_i$. By the same reasoning as in the alternative case above, it follows that $e'' = e'$.

Thus $e \overset{p}{\rightsquigarrow} e'$ and, for each $i \in \overline{n}$, we have $\mathcal{D}'_i :: \Pi \vdash e' : \sigma_i$. So by (JOIN), it follows that $\langle \overrightarrow{\mathcal{D}'}_n, \textsc{join} \rangle :: \Pi \vdash e' : \sigma_1 \cap \ldots \cap \sigma_n$ is a well-defined derivation.

$(\langle \mathcal{D}, \textsc{fld} \rangle :: \Pi \vdash e.f : \sigma \;\&\; \mathcal{D} \overset{p}{\triangleright} \mathcal{D}' \Rightarrow \langle \mathcal{D}, \textsc{fld} \rangle \overset{0 \cdot p}{\triangleright} \langle \mathcal{D}', \textsc{fld} \rangle)$:

Since $\langle \mathcal{D}, \textsc{fld} \rangle :: \Pi \vdash e.f : \sigma$ it follows by rule (FLD) that $\mathcal{D} :: \Pi \vdash e : \langle f : \sigma \rangle$. Also, since $\mathcal{D} \overset{p}{\triangleright} \mathcal{D}'$ it follows from the inductive hypothesis that $\mathcal{D}'$ is a well-defined derivation and that $\mathcal{D}' :: \Pi \vdash e' : \langle f : \sigma \rangle$ for some e' with $e \overset{p}{\rightsquigarrow} e'$. Then, by rule (FLD), we have that $\langle \mathcal{D}', \textsc{fld} \rangle :: \Pi \vdash e'.f : \sigma$ is also a well-defined derivation. Furthermore, since $e \overset{p}{\rightsquigarrow} e'$, by Definition 4.3 it follows that there is some expression context $\mathfrak{C}_p$ such that $e = \mathfrak{C}_p[e_r]$ for some redex e_r with $e_r \to e_c$ and $e' = \mathfrak{C}_p[e_c]$. Take the expression context $\mathfrak{C}'_{0 \cdot p} = \mathfrak{C}_p.f$; then $e.f = \mathfrak{C}_p[e_r].f = \mathfrak{C}'_{0 \cdot p}[e_r]$ and $e'.f = \mathfrak{C}_p[e_c].f = \mathfrak{C}'_{0 \cdot p}[e_c]$. Then, by Definition 4.17, $e.f \overset{0 \cdot p}{\rightsquigarrow} e'.f$. □

We can also show that strong and ω-safe derivations are preserved by derivation reduction.

Lemma 4.24. *If $\mathcal{D}$ is strong (ω-safe) and $\mathcal{D} \to_{\mathfrak{D}} \mathcal{D}'$, then $\mathcal{D}'$ is strong (ω-safe).*

Proof. By induction on the definition of derivation reduction.

$(\langle \langle \overrightarrow{\mathcal{D}}_n, \textsc{newf} \rangle, \textsc{fld} \rangle :: \Pi \vdash \texttt{new } C(\overrightarrow{e}).f_i : \sigma \overset{0}{\triangleright} \mathcal{D}_j, j \in \overline{n})$:

If $\langle \langle \overrightarrow{\mathcal{D}}_n, \textsc{newf} \rangle, \textsc{fld} \rangle$ is a strong (ω-safe) derivation, then it follows from Definition 4.8 (Definition 4.9) that $\langle \overrightarrow{\mathcal{D}}_n, \textsc{newf} \rangle$ is also strong (ω-safe), and then also that each $\mathcal{D}_i$ is strong (ω-safe). So, in particular $\mathcal{D}_j$ is strong (ω-safe).

$(\langle \langle \mathcal{D}_b, \mathcal{D}', \textsc{newm} \rangle, \overrightarrow{\mathcal{D}}_n, \textsc{invk} \rangle :: \Pi \vdash \texttt{new } C(\overrightarrow{e}).m(\overrightarrow{e}_n) : \sigma \overset{0}{\triangleright} {\mathcal{D}_b}^{\mathcal{S}})$:

with $\mathcal{M}b(C, m) = (\overrightarrow{x}_n, e_b)$, where $\mathcal{S} = \{\texttt{this} \mapsto \mathcal{D}', x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$.

By rule (INVK) we have that $\langle \mathcal{D}_b, \mathcal{D}', \text{NEWM} \rangle :: \Pi \vdash \texttt{new } C(\overline{e}^{\curvearrowright}) : \langle m : (\vec{\phi}_n) \to \sigma \rangle$ and also that $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. Then also, by rule (NEWM) we have that $\mathcal{D}_b :: \{\texttt{this}:\psi, x_1:\phi_1, \ldots, x_n:\phi_n\} \vdash e_b : \sigma$ and $\mathcal{D}' :: \Pi \vdash \texttt{new } C(\overline{e}^{\curvearrowright}) : \psi$. Notice that this means that $\mathcal{S}$ is applicable to $\mathcal{D}_b$.

If $\langle \langle \mathcal{D}_b, \mathcal{D}', \text{NEWM} \rangle, \vec{\mathcal{D}}_n, \text{INVK} \rangle$ is a strong derivation then it follows from Definition 4.8 that each $\mathcal{D}_i$ ($i \in \overline{n}$) is strong, and also that $\langle \mathcal{D}_b, \mathcal{D}', \text{NEWM} \rangle$ is strong. Then it also follows that both $\mathcal{D}_b$ and $\mathcal{D}'$ are strong. Notice then that $\mathcal{S}$ is a strong derivation substitution, and so by Lemma 4.13 it follows that $\mathcal{D}_b{}^{\mathcal{S}}$ is also a strong derivation.

If $\langle \langle \mathcal{D}_b, \mathcal{D}', \text{NEWM} \rangle, \vec{\mathcal{D}}_n, \text{INVK} \rangle$ is an ω-safe derivation then it follows from Definition 4.9 that each $\mathcal{D}_i$ ($i \in \overline{n}$) is either ω-safe or an instance of the (ω) rule, and also that $\langle \mathcal{D}_b, \mathcal{D}', \text{NEWM} \rangle$ is ω-safe. Then it also follows that both $\mathcal{D}_b$ and $\mathcal{D}'$ are ω-safe. Notice then that $\mathcal{S}$ is an ω-safe derivation substitution, and so by Lemma 4.13 it follows that $\mathcal{D}_b{}^{\mathcal{S}}$ is also an ω-safe derivation.

($\langle \vec{\mathcal{D}}_n, \text{JOIN} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n$ & $\mathcal{D}_j \xrightarrow{p} \mathcal{D}'_j, j \in \overline{n} \Rightarrow \langle \vec{\mathcal{D}}_n, \text{JOIN} \rangle \xrightarrow{p} \langle \vec{\mathcal{D}'}_n, \text{JOIN} \rangle$):
where for each $i \in \overline{n}$ such that $i \neq j$, either $\mathcal{D}_i \xrightarrow{p} \mathcal{D}'_i$ or $\mathcal{D}_i \overset{p}{\rightsquigarrow} \mathcal{D}'_i$.

If $\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \text{JOIN} \rangle$ is a strong (ω-safe) derivation, then it follows from Definition 4.8 (Definition 4.9) that each $\mathcal{D}_i$ is also strong (ω-safe). Then, by induction it follows that $\mathcal{D}'_j$ is strong (ω-safe). Now, for each $i \in \overline{n}$ such that $i \neq j$, either $\mathcal{D}_i \xrightarrow{p} \mathcal{D}'_i$ in which case it again follows by induction that $\mathcal{D}'_i$ is a strong (ω-safe) derivation, or $\mathcal{D}_i \overset{p}{\rightsquigarrow} \mathcal{D}'_i$ in which case it follows by Lemma 4.19 that $\mathcal{D}'_i$ is strong (ω-safe). Thus, for each $i \in \overline{n}$ we have that $\mathcal{D}'_i$ is strong (ω-safe) and thus by Definition 4.8 (Definition 4.9) it follows that $\langle \vec{\mathcal{D}'}_n, \text{JOIN} \rangle$ is a strong (ω-safe) derivation.

($\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash e.f : \sigma$ & $\mathcal{D} \xrightarrow{p} \mathcal{D}' \Rightarrow \langle \mathcal{D}, \text{FLD} \rangle \xrightarrow{0 \cdot p} \langle \mathcal{D}', \text{FLD} \rangle$):
If $\langle \mathcal{D}, \text{FLD} \rangle$ is a strong (ω-safe) derivation then it follows from Definition 4.8 (Definition 4.9) that $\mathcal{D}$ is also strong (ω-safe). Then, since $\mathcal{D} \xrightarrow{p} \mathcal{D}'$ it follows by induction that $\mathcal{D}'$ is strong (ω-safe), and thus by Definition 4.8 (Definition 4.9) so too is $\langle \mathcal{D}', \text{FLD} \rangle$. □

Our aim is to prove that this notion of derivation reduction is *strongly normalising*, i.e. terminating. In other words, all derivations have a *normal form* with respect to $\to_{\mathfrak{D}}$. Our proof uses the well-known technique of *computability* [100]. As is standard, our notion is defined inductively over the structure of types, and is defined in such a way as to guarantee that computable derivations are strongly normalising.

Definition 4.25 (Computability)**.** *1. The set of* computable *derivations is defined as the smallest set satisfying the following conditions (where Comp*($\mathcal{D}$) *denotes that* $\mathcal{D}$ *is a member of the set of computable derivations):*

a) $Comp(\langle \omega \rangle :: \Pi \vdash e : \omega)$.

b) $Comp(\mathcal{D} :: \Pi \vdash e : \varphi) \Leftrightarrow SN(\mathcal{D} :: \Pi \vdash e : \varphi)$.

c) $Comp(\mathcal{D} :: \Pi \vdash e : C) \Leftrightarrow SN(\mathcal{D} :: \Pi \vdash e : C)$.

d) $Comp(\mathcal{D} :: \Pi \vdash e : \langle f : \sigma \rangle) \Leftrightarrow Comp(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash e.f : \sigma)$.

e) $Comp(\mathcal{D} :: \Pi \vdash e : \langle m : (\vec{\phi}_n) \to \sigma \rangle) \Leftrightarrow$

$$\forall \vec{\mathcal{D}}_n [\ \forall i \in \overline{n}\ [Comp(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)] \Rightarrow Comp(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \text{INVK} \rangle :: \Pi' \vdash e.m(\vec{e}_n) : \sigma)\ ]$$

where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteq \Pi]$ *and* $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ *for each* $i \in \overline{n}$ *with* $\Pi' = \bigcap \Pi \cdot \vec{\Pi}_n$.

f) $Comp(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n) \Leftrightarrow \forall i \in \overline{n}\ [\, Comp(\mathcal{D}_i)\,]$.

2. *A derivation substitution* $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$ *is* computable in an environment Π *if and only if for all* $x{:}\phi \in \Pi$ *there exists some* $i \in \overline{n}$ *such that* $x = x_i$ *and* $Comp(\mathcal{D}_i)$.

The weakening operation preserves computability:

Lemma 4.26. $Comp(\mathcal{D} :: \Pi \vdash e : \phi) \Leftrightarrow Comp(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \phi)$.

Proof. By straightforward induction on the structure of types.

(ω): Immediate since then $\mathcal{D} = \langle \omega \rangle :: \Pi \vdash e : \omega$ and $\mathcal{D}[\Pi' \trianglelefteq \Pi] = \langle \omega \rangle :: \Pi' \vdash e : \omega$, which are both computable by Definition 4.25.

(φ):
$$\begin{array}{lll} Comp(\mathcal{D} :: \Pi \vdash e : \varphi) & \Leftrightarrow & \text{(Def. 4.25)} \\ \mathrm{SN}(\mathcal{D} :: \Pi \vdash e : \varphi) & \Leftrightarrow & \text{(Lem. 4.21(6))} \\ \mathrm{SN}(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \varphi) & \Leftrightarrow & \text{(Def. 4.25)} \\ Comp(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \varphi) & & \end{array}$$

(C):
$$\begin{array}{lll} Comp(\mathcal{D} :: \Pi \vdash e : C) & \Leftrightarrow & \text{(Def. 4.25)} \\ \mathrm{SN}(\mathcal{D} :: \Pi \vdash e : C) & \Leftrightarrow & \text{(Lem. 4.21(6))} \\ \mathrm{SN}(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : C) & \Leftrightarrow & \text{(Def. 4.25)} \\ Comp(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : C) & & \end{array}$$

($\langle f{:}\sigma \rangle$):
$$\begin{array}{lll} Comp(\mathcal{D} :: \Pi \vdash e : \langle f{:}\sigma \rangle) & \Leftrightarrow & \text{(Def. 4.25)} \\ Comp(\langle \mathcal{D}, \textsc{fld} \rangle :: \Pi \vdash e.f : \sigma) & \Leftrightarrow & \text{(Inductive Hypothesis)} \\ Comp(\langle \mathcal{D}, \textsc{fld} \rangle[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e.f : \sigma) & \equiv & \text{(Def. 4.5)} \\ Comp(\langle \mathcal{D}[\Pi' \trianglelefteq \Pi], \textsc{fld} \rangle :: \Pi' \vdash e.f : \sigma) & \Leftrightarrow & \text{(Def. 4.25)} \\ Comp(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \langle f{:}\sigma \rangle) & & \end{array}$$

($\langle m{:}(\vec{\phi}_n) \to \sigma \rangle$):

$Comp(\mathcal{D} :: \Pi \vdash e : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle)$ $\Leftrightarrow$ (Def. 4.25)

$\forall \vec{\mathcal{D}}_n [\ \forall i \in \overline{n}\ [\, Comp(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)\,] \Rightarrow$
$Comp(\langle \mathcal{D}[\Pi_\alpha \trianglelefteq \Pi], \mathcal{D}_1[\Pi_\alpha \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi_\alpha \trianglelefteq \Pi_n], \textsc{invk} \rangle :: \Pi_\alpha \vdash e.m(\vec{e}_n) : \sigma)\ ]$
where $\Pi_\alpha = \bigcap \Pi \cdot \vec{\Pi}_n$
$\Leftrightarrow$ (Inductive Hypothesis)

$\forall \vec{\mathcal{D}}_n [\ \forall i \in \overline{n}\ [\, Comp(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)\,] \Rightarrow$
$Comp(\langle \mathcal{D}[\Pi_\alpha \trianglelefteq \Pi], \mathcal{D}_1[\Pi_\alpha \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi_\alpha \trianglelefteq \Pi_n], \textsc{invk} \rangle [\Pi_\beta \trianglelefteq \Pi_\alpha] :: \Pi_\beta \vdash e.m(\vec{e}_n) : \sigma)\ ]$
where $\Pi_\beta = \bigcap \Pi' \cdot \vec{\Pi}_n$
$\equiv$ (Def. 4.5)

$\forall \vec{\mathcal{D}}_n [\ \forall i \in \overline{n}\ [\, Comp(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)\,] \Rightarrow$
$Comp(\langle \mathcal{D}[\Pi_\alpha \trianglelefteq \Pi][\Pi_\beta \trianglelefteq \Pi_\alpha], \mathcal{D}_1[\Pi_\alpha \trianglelefteq \Pi_1][\Pi_\beta \trianglelefteq \Pi_\alpha], \ldots, \mathcal{D}_n[\Pi_\alpha \trianglelefteq \Pi_n][\Pi_\beta \trianglelefteq \Pi_\alpha], \textsc{invk} \rangle$
$:: \Pi_\beta \vdash e.m(\vec{e}_n) : \sigma)\ ]$ $\equiv$ (Lem. 4.6)

$\forall \vec{\mathcal{D}}_n [\ \forall i \in \overline{n}\ [\, Comp(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)\,] \Rightarrow$
$Comp(\langle \mathcal{D}[\Pi' \trianglelefteq \Pi][\Pi_\beta \trianglelefteq \Pi'], \mathcal{D}_1[\Pi_\beta \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi_\beta \trianglelefteq \Pi_n], \textsc{invk} \rangle :: \Pi_\beta \vdash e.m(\vec{e}_n) : \sigma)\ ]$
$\Leftrightarrow$ (Def. 4.25)

$Comp(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle)$

$(\sigma_1 \cap \ldots \cap \sigma_n)$: $Comp(\langle \overrightarrow{\mathcal{D}}_n, \text{JOIN} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n) \quad \Leftrightarrow$ (Def. 4.25)

$\forall i \in \overline{n}\ [\, Comp(\mathcal{D}_i :: \Pi \vdash e : \sigma_i)\,] \quad \Leftrightarrow$ (Inductive Hypothesis)

$\forall i \in \overline{n}\ [\, Comp(\mathcal{D}_i[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \sigma_i)\,] \quad \Leftrightarrow$ (Def. 4.25)

$Comp(\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi], \text{JOIN} \rangle :: \Pi' \vdash e : \sigma_1 \cap \ldots \cap \sigma_n)$

$\equiv$ (Def. 4.5)

$Comp(\langle \overrightarrow{\mathcal{D}}_n, \text{JOIN} \rangle [\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \sigma_1 \cap \ldots \cap \sigma_n)$

□

The key property of computable derivations, however, is that they are strongly normalising as shown in the first part of the following theorem.

Theorem 4.27. *1.* $Comp(\mathcal{D} :: \Pi \vdash e : \phi) \Rightarrow SN(\mathcal{D} :: \Pi \vdash e : \phi)$.

2. For neutral contexts $\mathfrak{C}$, $SN(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \phi) \Rightarrow Comp(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \phi)$.

Proof. By simultaneous induction on the structure of types.

(ω): The result follows immediately, by Definition 4.20 in the case of (1), and by Definition 4.25 in the case of (2).

(φ),(C): Immediate, by Definition 4.25.

$(\langle f : \sigma \rangle)$: 1. $Comp(\mathcal{D} :: \Pi \vdash e : \langle f : \sigma \rangle) \quad \Rightarrow$ (Def. 4.25)

$Comp(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash e.f : \sigma) \quad \Rightarrow$ (Inductive Hypothesis (1))

$SN(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash e.f : \sigma) \quad \Rightarrow$ (Lem. 4.21)

$SN(\mathcal{D} :: \Pi \vdash e : \langle f : \sigma \rangle)$

2. Assuming $SN(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle f : \sigma \rangle)$ with $\mathfrak{C}$ a neutral context, it follows by Lemma 4.21 that $SN(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash \mathfrak{C}[x].f : \sigma)$. Now, take the expression context $\mathfrak{C}' = \mathfrak{C}.f$; notice that by Definitions 4.2 and 4.3, $\mathfrak{C}'$ is a neutral context and $\mathfrak{C}[x].f = \mathfrak{C}'[x]$. Thus $SN(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash \mathfrak{C}'[x] : \sigma)$ and by induction it follows that $Comp(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash \mathfrak{C}'[x] : \sigma)$. Then from the definition of $\mathfrak{C}'$ we have $Comp(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash \mathfrak{C}[x].f : \sigma)$ and by Definition 4.25 that $Comp(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle f : \sigma \rangle)$.

$(\langle m : (\vec{\phi}_n) \to \sigma \rangle)$: 1. Assume $Comp(\mathcal{D} :: \Pi \vdash e : \langle m : (\vec{\phi}_n) \to \sigma \rangle)$. For each $i \in \overline{n}$, we take a fresh variable x_i and construct a derivation $\mathcal{D}_i$ as follows:

- If $\phi_i = \omega$ then $\mathcal{D}_i = \langle \omega \rangle :: \Pi_i \vdash x_i : \omega$, with $\Pi_i = \emptyset$;
- If ϕ_i is a strict type σ then $\mathcal{D}_i = \langle \text{VAR} \rangle :: \Pi_i \vdash x_i : \sigma$, with $\Pi_i = \{x_i : \sigma\}$;
- If $\phi_i = \sigma_1 \cap \ldots \cap \sigma_{n_i}$ with $n_i \geq 2$ then $\mathcal{D}_i = \langle \mathcal{D}'_{(i,1)}, \ldots, \mathcal{D}'_{(i,n_i)}, \text{JOIN} \rangle :: \Pi_i \vdash x : \sigma_1 \cap \ldots \cap \phi_{n_i}$ with $\Pi_i = \{x_i : \phi_i\}$ and $\mathcal{D}'_{(i,j)} = \langle \text{VAR} \rangle :: \Pi_i \vdash x_i : \sigma_j$ for each $j \in \overline{n_i}$.

Notice that each $\mathcal{D}_i$ is in normal form, so $SN(\mathcal{D}_i)$ for each $i \in \overline{n}$. Notice also that $\mathcal{D}_i :: \Pi_i \vdash \mathfrak{C}[x_i] : \phi_i$ for each $i \in \overline{n}$ where $\mathfrak{C}$ is the neutral context $[\,]$. So, by the second inductive hypothesis $Comp(\mathcal{D}_i)$ for each $i \in \overline{n}$. Then by Definition 4.25 it follows that $Comp(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_n, \text{INVK} \rangle :: \Pi' \vdash e.m(\vec{x}_n) : \sigma)$, where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteq \Pi]$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ for each $i \in \overline{n}$ with $\Pi' = \bigcap \Pi \cdot \overline{\Pi}_n$. So, by the first inductive hypothesis it then follows that $SN(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_n, \text{INVK} \rangle :: \Pi' \vdash e.m(\vec{x}_n) : \sigma)$. Lastly by Lemma 4.21(2) we have $SN(\mathcal{D}')$, and from Lemma 4.21(6) that $SN(\mathcal{D})$.

2. Assume $\mathrm{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle m : (\vec{\phi}_n) \to \sigma \rangle)$ with $\mathfrak{C}$ a neutral context. Also assume that there exist derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ such that $\mathit{Comp}(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)$ for each $i \in \overline{n}$. Then it follows from the first inductive hypothesis that $\mathrm{SN}(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)$ for each $i \in \overline{n}$. Let $\Pi' = \bigcap \Pi \cdot \vec{\Pi}_n$; notice that by Definition 3.6, $\Pi' \trianglelefteqslant \Pi$ and $\Pi' \trianglelefteqslant \Pi_i$ for each $i \in \overline{n}$. Then by Lemma 4.21(6), it follows that $\mathrm{SN}(\mathcal{D}[\Pi' \trianglelefteqslant \Pi])$ and $\mathrm{SN}(\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i])$ for each $i \in \overline{n}$. By Lemma 4.21(3), we then have $\mathrm{SN}(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{invk} \rangle :: \Pi' \vdash \mathfrak{C}[x].m(\vec{e}_n) : \sigma)$ where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteqslant \Pi]$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ for each $i \in \overline{n}$. Now, take the expression context $\mathfrak{C}' = \mathfrak{C}.m(\vec{e}_n)$; notice that, since $\mathfrak{C}$ is neutral, by Definitions 4.2 and 4.3, $\mathfrak{C}'$ is a neutral context and $\mathfrak{C}[x].m(\vec{e}_n) = \mathfrak{C}'[x]$. Thus by the second inductive hypothesis it follows that $\mathit{Comp}(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{invk} \rangle :: \Pi' \vdash \mathfrak{C}[x].m(\vec{e}_n) : \sigma)$. Since the derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ were arbitrary, the following implication holds

$$\forall \vec{\mathcal{D}}_n [\ \forall i \in \overline{n}\ [\mathit{Comp}(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)] \Rightarrow \mathit{Comp}(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{invk} \rangle :: \Pi' \vdash e.m(\vec{e}_n) : \sigma)\ ]$$

where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteqslant \Pi]$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ for each $i \in \overline{n}$ with $\Pi' = \bigcap \Pi \cdot \vec{\Pi}_n$. So by Definition 4.25 we have $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \langle m : (\vec{\phi}_n) \to \sigma \rangle)$.

($\sigma_1 \cap \ldots \cap \sigma_n, n \geq 2$): 1. Then $\mathit{Comp}(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n)$ and so by Definition 4.25 we have $\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash e : \sigma_i)$ for each $i \in \overline{n}$. From this it follows by induction that $\mathrm{SN}(\mathcal{D}_i)$ for each $i \in \overline{n}$ and so by Lemma 4.21 that $\mathrm{SN}(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join} \rangle)$.

2. Then $\mathrm{SN}(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join} \rangle :: \Pi \vdash \mathfrak{C}[x] : \sigma_1 \cap \ldots \cap \sigma_n)$ and so by Lemma 4.21 we have $\mathrm{SN}(\mathcal{D}_i :: \Pi \vdash \mathfrak{C}[x] : \sigma_i)$ for each $i \in \overline{n}$. From this it follows by induction that $\mathit{Comp}(\mathcal{D}_i)$ for each $i \in \overline{n}$ and so by Definition 4.25 that $\mathit{Comp}(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join} \rangle)$. □

From this, we can show that computability is closed for derivation expansion - that is, if a derivation contractum is computable then so is its redex. This property will be important when showing the *replacement* lemma below.

Lemma 4.28. *1. Let $\mathtt{C}$ be a class such that $\mathcal{F}(\mathtt{C}) = \vec{f}_n$, then for all $j \in \overline{n}$:*

$$\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \sigma)\ \&\ \forall i \in \overline{n}, i \neq j\ [\exists\, \phi\ [\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash e_i : \phi)]]$$
$$\Rightarrow \mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\langle \langle \vec{\mathcal{D}}_n, \textsc{newF} \rangle, \textsc{fld} \rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ \mathtt{C}(\vec{e}_n).f_j] : \sigma)$$

2. Let $\mathtt{C}$ be a class such that $\mathcal{M}b(\mathtt{C}, m) = (\vec{x}_n, e_b)$ and $\mathcal{D}_b :: \{\mathtt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash e_b : \sigma'$, then for derivation contexts $\mathfrak{D}_{(p,\sigma')}$ and expression contexts $\mathfrak{C}$:

$$\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \sigma)$$
$$\&\ \mathit{Comp}(\mathcal{D}_0 :: \Pi \vdash \mathtt{new}\ \mathtt{C}(\vec{e}^{\,\prime}) : \psi)\ \&\ \forall i \in \overline{n}\ [\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)]$$
$$\Rightarrow \mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \vec{\mathcal{D}}_n, \textsc{invk} \rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ \mathtt{C}(\vec{e}^{\,\prime}).m(\vec{e}_n)] : \sigma)$$

where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM} \rangle :: \Pi \vdash \mathtt{new}\ \mathtt{C}(\vec{e}^{\,\prime}) : \langle m : (\vec{\phi}_n) \to \sigma' \rangle$,

$\mathcal{S} = \{\mathtt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$,

$\mathsf{S} = \{\mathtt{this} \mapsto \mathtt{new}\ \mathtt{C}(\vec{e}^{\,\prime}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$

Proof. 1. By induction on the structure of strict types.

($\sigma = \varphi$): Assume $\mathit{Comp}(\mathfrak{D}_{(p,\sigma)}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j] : \varphi)$ and $\exists\, \phi\ [\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi)]$ for each $i \in \overline{n}$ such that $i \neq j$. By Theorem 4.27 it follows that $\mathrm{SN}(\mathfrak{D}_{(p,\sigma)}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j] : \varphi)$ and $\exists\, \phi\ [\mathrm{SN}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi)]$ for each $i \in \overline{n}$ such that $i \neq j$. Then by Lemma 4.21(8) we have that

$$\mathrm{SN}(\mathfrak{D}_{(p,\sigma)}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newf}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}_n).\mathtt{f}_j] : \varphi)$$

And the result follows by Definition 4.25

($\sigma = \mathtt{C}$): Similar to the case for type variables.

($\sigma = \langle \mathtt{f} : \sigma \rangle$): Assume $\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j] : \langle \mathtt{f} : \sigma \rangle)$ and $\exists\, \phi\ [\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi)]$ for each $i \in \overline{n}$ such that $i \neq j$. By Definition 4.25, $\mathit{Comp}(\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j].\mathtt{f} : \sigma)$. Now, take the expression context $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.\mathtt{f}$ and the derivation context $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma')}, \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p.\mathtt{f} : \sigma$. Notice that

$$\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j].\mathtt{f} : \sigma = \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[\mathtt{e}_j] : \sigma$$

Thus we have $\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[\mathtt{e}_j] : \sigma)$. Then by the inductive hypothesis it follows that

$$\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newf}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[\mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}_n).\mathtt{f}_j] : \sigma)$$

So by the definition of $\mathfrak{D}'$ we have

$$\mathit{Comp}(\langle \mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newf}\rangle, \textsc{fld}\rangle], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}_n).\mathtt{f}_j.\mathtt{f} : \sigma)$$

Then by Definition 4.25 we have

$$\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newf}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}_n).\mathtt{f}_j] : \langle \mathtt{f} : \sigma \rangle)$$

($\sigma = \langle \mathtt{m} : (\vec{\phi}_{n'}) \to \sigma \rangle$): Assume $\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j] : \langle \mathtt{m} : (\vec{\phi}_{n'}) \to \sigma \rangle)$ and, for each $i \in \overline{n}$ such that $i \neq j$, there is some ϕ such that $\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi)$. Now, take arbitrary derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ such that $\mathit{Comp}(\mathcal{D}'_k :: \Pi_k \vdash \mathtt{e}'_k : \phi_k)$ for each $k \in \overline{n'}$. By Definition 4.25, it then follows that $\mathit{Comp}(\langle \mathcal{D}', \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle) :: \Pi' \vdash \mathfrak{C}_p[\mathtt{e}_j].\mathtt{m}(\overrightarrow{\mathtt{e}''}_{n'}) : \sigma$ where $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi}_{n'}$ and also that $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j][\Pi' \trianglelefteq \Pi]$, with $\mathcal{D}''_k = \mathcal{D}'_k[\Pi' \trianglelefteq \Pi_k]$ for each $k \in \overline{n}$. By Lemma 4.7, we have

$$\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j][\Pi' \trianglelefteq \Pi] = \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\mathcal{D}_j[\Pi' \trianglelefteq \Pi]]$$

Now, take the expression context $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.\mathtt{m}(\overrightarrow{\mathtt{e}''}_{n'})$ and the derivation context $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma)}[\Pi' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p.\mathtt{m}(\overrightarrow{\mathtt{e}''}_{n'}) : \sigma$. Notice that

$$\langle \mathcal{D}', \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle = \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j[\Pi' \trianglelefteq \Pi]] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[\mathtt{e}_j] : \sigma$$

So we have

$$\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j[\Pi' \trianglelefteq \Pi]] :: \Pi' \vdash \mathfrak{C}'[\mathtt{e}_j] : \sigma)$$

Now, by Lemma 4.26, it follows that $\exists\, \phi\ [\mathit{Comp}(\mathcal{D}_i[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e_i : \phi)]$ for each $i \in \overline{n}$ such that $i \neq j$. Then by the inductive hypothesis it follows that

$$\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi], \textsc{newF}\rangle, \textsc{fld}\rangle] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new C}(\vec{e}_n)\texttt{.}f_j] : \sigma)$$

So by the definition of $\mathfrak{D}'$, this give us that

$$\mathit{Comp}(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\langle\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi], \textsc{newF}\rangle, \textsc{fld}\rangle], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n)\texttt{.}f_j\texttt{.}m(\overrightarrow{e'}_{n'})] : \sigma)$$

And then by Definition 4.5

$$\mathit{Comp}(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle[\Pi' \trianglelefteq \Pi]], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n)\texttt{.}f_j\texttt{.}m(\overrightarrow{e'}_{n'})] : \sigma)$$

And by Lemma 4.7

$$\mathit{Comp}(\langle \mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle][\Pi' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n)\texttt{.}f_j\texttt{.}m(\overrightarrow{e'}_{n'})] : \sigma)$$

Since the derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ were arbitrary, the following implication holds:

$$\forall \overrightarrow{\mathcal{D}''}_{n'}\ [\forall i \in \overline{n'}\ [\mathit{Comp}(\mathcal{D}'_i :: \Pi_i \vdash e'_i : \phi_i)] \Rightarrow \mathit{Comp}(\langle \mathcal{D}, \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n)\texttt{.}f_j\texttt{.}m(\overrightarrow{e'}_{n'})] : \sigma)]$$

where $\mathcal{D} = \mathfrak{D}_{(p,\sigma)}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle][\Pi' \trianglelefteq \Pi]$. Thus the result follows by Definition 4.25

$$\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n)\texttt{.}f_j] : \langle m : (\vec{\phi}_{n'}) \to \sigma\rangle)$$

2. By induction on the structure of strict types.

   ($\sigma = \varphi$): Assume $\mathit{Comp}(\mathfrak{D}_{(p,\sigma)}[{\mathcal{D}_b}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[{e_b}^{\mathsf{S}}] : \varphi)$ and $\mathit{Comp}(\mathcal{D}_0 :: \Pi \vdash \texttt{new C}(\overrightarrow{e'}) : \psi)$ with $\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for each $i \in \overline{n}$, where $\mathcal{S} = \{\texttt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$, and S is the term substitution induced by $\mathcal{S}$. Then by Theorem 4.27 it follows that $\mathrm{SN}(\mathfrak{D}_{(p,\sigma)}[{\mathcal{D}_b}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[{e_b}^{\mathsf{S}}] : \varphi)$, $\mathrm{SN}(\mathcal{D}_0 :: \Pi \vdash \texttt{new C}(\overrightarrow{e'}) : \psi)$ and $\mathrm{SN}(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for each $i \in \overline{n}$. By Lemma 4.21(9) we have that

   $$\mathrm{SN}(\mathfrak{D}_{(p,\sigma)}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new C}(\overrightarrow{e'})\texttt{.}m(\vec{e}_n)] : \varphi)$$

   where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM}\rangle :: \Pi \vdash \texttt{new C}(\overrightarrow{e'}) : \langle m : (\vec{\phi}_n) \to \sigma\rangle$, and the result follows by Definition 4.25

   ($\sigma = \mathtt{C}$): Similar to the case for type variables.

($\sigma = \langle f : \sigma \rangle$): Assume $\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \langle f : \sigma \rangle)$ and $\mathit{Comp}(\mathcal{D}_0 :: \Pi \vdash \texttt{new } C(\overline{e'}) : \psi)$ with $\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for all $i \in \overline{n}$, where $\mathcal{S} = \{\texttt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$, and S is the term substitution induced by $\mathcal{S}$. By Definition 4.25 it follows that

$$\mathit{Comp}(\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}], \textsc{fld} \rangle :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}].f : \sigma)$$

Take the expression context $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.f$ and the derivation context $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma')}, \textsc{fld} \rangle :: \Pi \vdash \mathfrak{C}_p.f : \sigma$. Notice that

$$\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}], \textsc{fld} \rangle :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}].f : \sigma = \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma$$

So we have

$$\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma)$$

Then by the inductive hypothesis it follows that

$$\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk} \rangle] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new } C(\overline{e'}).m(\overrightarrow{e}_n)] : \sigma)$$

where $\mathcal{D} = \langle \mathcal{D}_0, \mathcal{D}_0, \textsc{newM} \rangle :: \Pi \vdash \texttt{new } C(\overline{e'}) : \langle m : (\overrightarrow{\phi}_n) \to \sigma' \rangle$. So by the definition of $\mathfrak{D}'$ this gives us

$$\mathit{Comp}(\langle \mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk} \rangle], \textsc{fld} \rangle :: \Pi \vdash \mathfrak{C}_p[\texttt{new } C(\overline{e'}).m(\overrightarrow{e}_n)].f : \sigma)$$

and by Definition 4.25 it follows that

$$\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk} \rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new } C(\overline{e'}).m(\overrightarrow{e}_n)] : \langle f : \sigma \rangle)$$

($\sigma = \langle m' : (\overrightarrow{\phi'}_{n'}) \to \sigma \rangle$): Assume that $\mathit{Comp}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \langle m' : (\overrightarrow{\phi'}_{n'}) \to \sigma \rangle)$ with $\mathit{Comp}(\mathcal{D}_0 :: \Pi \vdash \texttt{new } C(\overline{e'}) : \psi)$ and $\mathit{Comp}(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for all $i \in \overline{n}$, where $\mathcal{S} = \{\texttt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$, and S is the term substitution induced by $\mathcal{S}$. Now, take arbitrary derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ such that $\mathit{Comp}(\mathcal{D}'_k :: \Pi_k \vdash e'_k : \phi'_k)$ for each $k \in \overline{n'}$. By Definition 4.25 it follows that $\mathit{Comp}(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_{n'}, \textsc{invk} \rangle :: \Pi' \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}].m'(\overline{e'}_{n'}) : \sigma)$ where $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi}_{n'}$, $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}][\Pi' \trianglelefteq \Pi]$ and $\mathcal{D}'_k = \mathcal{D}'_k[\Pi' \trianglelefteq \Pi_k]$ for each $k \in \overline{n'}$. By Lemma 4.7

$$\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}][\Pi' \trianglelefteq \Pi] = \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\mathcal{D}_b{}^{\mathcal{S}}[\Pi' \trianglelefteq \Pi]]$$

Now, take the expression context $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.m'(\overline{e'}_{n'})$ and the derivation context $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma)}[\Pi' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}'}_{n'}, \textsc{invk} \rangle :: \Pi' \vdash \mathfrak{C}_p.m'(\overline{e'}_{n'}) : \sigma$. Notice that

$$\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_{n'}, \textsc{invk} \rangle = \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}[\Pi' \trianglelefteq \Pi]] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma$$

So we have $\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}[\Pi' \trianglelefteq \Pi]] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma)$, and then by Lemma 4.14, $\mathit{Comp}(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}[\Pi' \trianglelefteq \Pi]}] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma)$. Now, by Lemma 4.26, $\mathit{Comp}(\mathcal{D}_0[\Pi' \trianglelefteq \Pi] :: \Pi \vdash \texttt{new } C(\overline{e'}) : \psi)$ and $\mathit{Comp}(\mathcal{D}_i[\Pi' \trianglelefteq \Pi] :: \Pi \vdash e_i : \phi_i)$ for all $i \in \overline{n}$. Thus, by the inductive

hypothesis

$$Comp(\mathfrak{D}'_{(0{\cdot}p,\sigma')}[\langle \mathcal{D}'', \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi], \textsc{invk}\rangle] \\ :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new } C(\overrightarrow{e'}).m(\vec{e}_n)] : \sigma)$$

where $\mathcal{D}'' = \langle \mathcal{D}_b, \mathcal{D}_0[\Pi' \trianglelefteqslant \Pi], \textsc{newM}\rangle :: \Pi' \vdash \texttt{new } C(\overrightarrow{e'}) : \langle m{:}(\vec{\phi}_n) \to \sigma'\rangle$. So, by the definition of $\mathfrak{D}'$, this gives us

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteqslant \Pi][\langle \mathcal{D}'', \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi], \textsc{invk}\rangle], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle \\ :: \Pi' \vdash \mathfrak{C}_p[\texttt{new } C(\overrightarrow{e'}).m(\vec{e}_n)].m'(\overrightarrow{e''}_{n'}) : \sigma)$$

Then by Definition 4.5 it follows that

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteqslant \Pi][\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle[\Pi' \trianglelefteqslant \Pi]], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle \\ :: \Pi' \vdash \mathfrak{C}_p[\texttt{new } C(\overrightarrow{e'}).m(\vec{e}_n)].m'(\overrightarrow{e''}_{n'}) : \sigma)$$

where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM}\rangle :: \Pi \vdash \texttt{new } C(\overrightarrow{e'}) : \langle m{:}(\vec{\phi}_n) \to \sigma'\rangle$, and by Lemma 4.7 we have

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle][\Pi' \trianglelefteqslant \Pi], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle \\ :: \Pi' \vdash \mathfrak{C}_p[\texttt{new } C(\overrightarrow{e'}).m(\vec{e}_n)].m'(\overrightarrow{e''}_{n'}) : \sigma)$$

Since the choice of the derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ was arbitrary, the following implication holds:

$$\forall \overrightarrow{\mathcal{D}'}_{n'} [\ \forall i \in \overline{n}\ [\ Comp(\mathcal{D}'_i :: \Pi_i \vdash e'_i : \phi'_i)\ ] \Rightarrow \\ Comp(\langle \mathcal{D}''', \mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}, \textsc{invk}\rangle :: \Pi' \vdash e.m(\vec{e}_n) : \sigma)\ ]$$

where$\mathcal{D}''' = \mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle][\Pi' \trianglelefteqslant \Pi]$ and $\mathcal{D}'_k = \mathcal{D}'_k[\Pi' \trianglelefteqslant \Pi_k]$ for each $k \in \overline{n'}$. Then, by Definition 4.25 we have

$$Comp(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new } C(\overrightarrow{e'}).m(\vec{e}_n)] : \langle m' : (\overrightarrow{\phi'}_{n'}) \to \sigma\rangle)$$

□

Another corollary of Theorem 4.27 is that identity (derivation) substitutions are computable in their own environments.

Lemma 4.29. *Let* Π *be a type environment; then* Id_Π *is computable in* Π.

Proof. Let $\Pi = \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$. So $Id_\Pi = \{x_1 \mapsto \mathcal{D}_1 :: \Pi \vdash x_1 : \phi_1, \ldots, x_n \mapsto \mathcal{D}_n :: \Pi \vdash x_1 : \phi_1\}$, by Definition 4.15. Notice that for each $i \in \overline{n}$ the derivation $\mathcal{D}_i$ contains no derivation redexes, i.e. it is in normal form and thus SN($\mathcal{D}_i$). Notice also that, since $x_i = \mathfrak{C}[x_i]$ where $\mathfrak{C}$ is the empty context [ ] (see Definition 4.3), SN($\mathcal{D}_i :: \Pi \vdash \mathfrak{C}[x] : \phi_i$) for each $i \in \overline{n}$. Then, by Theorem 4.27(2) it follows that $Comp(\mathcal{D}_i)$. Thus, for each $x{:}\phi \in \Pi$ there is some $i \in \overline{n}$ such that $x = x_i$ and $Comp(\mathcal{D}_i)$ and so by Definition 4.25, Id_Π is computable in Π. □

The final piece of the strong normalisation proof is the derivation replacement lemma, which shows that when we perform derivation substitution using computable derivations we obtain a derivation that is overall computable. In [10], where a proof of the strong normalisation of derivation reduction is given for λ-calculus, this part of the proof is achieved by a routine induction on the structure of derivations. In [15] however, where this result is shown for combinator systems, the replacement lemma was proved using an *encompassment* relation on terms. For that system, this was the only way to prove the lemma since the intersection type derivations in that system do not contain all the reduction information for the terms they type - some of the reduction behaviour is hidden because types for the combinators themselves are taken from an environment. Given the similarities between the reduction model of class-based programs and combinator systems, or TRS in general, one might think that a similar approach would be necessary for FJ$^{¢}$. This is not the case however, since our type system incorporates a novel feature: method bodies are typed for *each* individual invocation, and are part of the overall derivation. Thus, there will be subderivations for the constituents of each redex that will appear during reduction. The consequence of this is that, like for the λ-calculus, we are able to prove the replacement lemma by straightforward induction on derivations.

Lemma 4.30. *If $\mathcal{D} :: \Pi \vdash e : \phi$ and $\mathcal{S}$ is a derivation substitution computable in Π and applicable to $\mathcal{D}$, then Comp($\mathcal{D}^{\mathcal{S}}$).*

Proof. By induction on the structure of $\mathcal{D}$. The (NEWF) and (NEWM) cases are particularly tricky, and use Lemma 4.28. Let $\Pi = \{x_1{:}\phi'_1, \ldots, x_n{:}\phi'_{n'}\}$ and $\mathcal{S} = \{x'_1 \mapsto \mathcal{D}'_1 :: \Pi' \vdash e'_1 : \phi'_1, \ldots, x'_{n''} \mapsto \mathcal{D}'_{n''} :: \Pi' \vdash e'_{n''} : \phi'_{n''}\}$ with $\{x_1, \ldots, x_{n'}\} \subseteq \{x'_1, \ldots, x'_{n''}\}$. Also let S be the term substitution induced by $\mathcal{S}$. As for Lemma 4.12, when applying the inductive hypothesis we note implicitly that if $\mathcal{S}$ is applicable to $\mathcal{D}$ then it is also applicable to subderivations of $\mathcal{D}$.

(ω): Immediately by Definition 4.25 since $\mathcal{D}^{\mathcal{S}} = \langle\omega\rangle :: \Pi' \vdash e^{\mathsf{S}} : \omega$.

(VAR): Then $\mathcal{D} :: \Pi \vdash x : \sigma$. We examine the different possibilities for $\mathcal{D}^{\mathcal{S}}$:

- $x{:}\sigma \in \Pi$, so $x = x'_i$ for some $i \in \overline{n''}$ and $\mathcal{D}'_i :: \Pi' \vdash e'_i : \sigma$. Then $\mathcal{D}^{\mathcal{S}} = \mathcal{D}'_i$. Since $\mathcal{S}$ is computable in Π it follows that *Comp*($\mathcal{D}'_i$), and so *Comp*($\mathcal{D}^{\mathcal{S}}$).
- $x{:}\phi \in \Pi$ for some $\phi \leqslant \sigma$, so $\phi = \sigma_1 \cap \ldots \cap \sigma_n$ with $\sigma = \sigma_i$ for some $i \in \overline{n}$. Also, $x = x'_j$ for some $j \in \overline{n''}$ and $\mathcal{D}'_j :: \Pi' \vdash e'_j : \phi$, so $\mathcal{D}'_j = \langle\overline{\mathcal{D}'}_n, \text{JOIN}\rangle$ with $\mathcal{D}'_k :: \Pi' \vdash e'_j : \sigma_k$ for each $k \in \overline{n}$. Now, by Definition 4.11, $\mathcal{D}^{\mathcal{S}} = \mathcal{D}'_i :: \Pi' \vdash e'_j : \sigma_i$. Since $\mathcal{S}$ is computable in Π it follows that *Comp*($\mathcal{D}'_j$), and then, by Definition 4.25, that *Comp*($\mathcal{D}'_k$) for each $k \in \overline{n}$. Thus, in particular *Comp*($\mathcal{D}'_i$), and so *Comp*($\mathcal{D}^{\mathcal{S}}$).

(FLD): Then $\mathcal{D} = \langle\mathcal{D}', \text{FLD}\rangle :: \Pi \vdash e.f : \sigma$ and $\mathcal{D}' :: \Pi \vdash e : \langle f{:}\sigma\rangle$. By induction *Comp*($\mathcal{D}'^{\mathcal{S}} :: \Pi' \vdash e^{\mathsf{S}} : \langle f{:}\sigma\rangle$). Then by Definition 4.25, *Comp*($\langle\mathcal{D}'^{\mathcal{S}}, \text{FLD}\rangle :: \Pi' \vdash e^{\mathsf{S}}.f : \sigma$). Notice that $\langle\mathcal{D}'^{\mathcal{S}}, \text{FLD}\rangle = \mathcal{D}^{\mathcal{S}}$ and so *Comp*($\mathcal{D}^{\mathcal{S}}$).

(INVK): Then $\mathcal{D} = \langle\mathcal{D}_0, \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle :: \Pi \vdash e_0.m(\vec{e}_n) : \sigma$ with $\mathcal{D}_0 :: \Pi \vdash e_0 : \langle m{:}(\vec{\phi}_n) \to \sigma\rangle$ and $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. By induction we have that *Comp*($\mathcal{D}_0{}^{\mathcal{S}} :: \Pi' \vdash e_0{}^{\mathsf{S}} : \langle m{:}(\vec{\phi}_n) \to \sigma\rangle$) and also that *Comp*($\mathcal{D}_i{}^{\mathcal{S}} :: \Pi' \vdash e_i{}^{\mathsf{S}} : \phi_i$) for each $i \in \overline{n}$. Then, by Definition 4.25, it follows that

$$\mathit{Comp}(\langle\mathcal{D}_0{}^{\mathcal{S}}[\Pi'' \leqslant \Pi'], \mathcal{D}_1{}^{\mathcal{S}}[\Pi'' \leqslant \Pi'], \ldots, \mathcal{D}_n{}^{\mathcal{S}}[\Pi'' \leqslant \Pi'], \text{INVK}\rangle$$

$$:: \Pi'' \vdash e_0{}^{\mathsf{S}}.m(e_0{}^{\mathsf{S}},\ldots,e_n{}^{\mathsf{S}}) : \sigma)$$

where $\Pi'' = \bigcap \Pi' \cdot \vec{\Pi}_n$ and $\Pi_i = \Pi'$ for each $i \in \overline{n}$. Notice that $\Pi'' = \Pi'$ and that for all $\mathcal{D} :: \Pi \vdash e : \phi$, $\mathcal{D}[\Pi \trianglelefteq \Pi] = \mathcal{D}$, so it follows that

$$\mathit{Comp}(\langle \mathcal{D}_0{}^{\mathcal{S}}, \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{invk}\rangle :: \Pi' \vdash e_0{}^{\mathsf{S}}.m(e_0{}^{\mathsf{S}},\ldots,e_n{}^{\mathsf{S}}) : \sigma)$$

Notice that $\langle \mathcal{D}_0{}^{\mathcal{S}}, \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{invk}\rangle = \mathcal{D}^{\mathcal{S}}$ and so $\mathit{Comp}(\mathcal{D}^{\mathcal{S}})$.

(JOIN): Then $\mathcal{D} = \langle \vec{\mathcal{D}}_n, \textsc{join}\rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n$ and $\mathcal{D}_i :: \Pi \vdash e : \sigma_i$ for each $i \in \overline{n}$. By induction, $\mathit{Comp}(\mathcal{D}_i{}^{\mathcal{S}} :: \Pi' \vdash e^{\mathsf{S}} : \sigma_i)$ for each $i \in \overline{n}$ and so by Definition 4.25, $\mathit{Comp}(\langle \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{join}\rangle :: \Pi' \vdash e^{\mathsf{S}} : \sigma_1 \cap \ldots \cap \sigma_n)$. Notice that $\langle \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{join}\rangle = \mathcal{D}^{\mathcal{S}}$ and so $\mathit{Comp}(\mathcal{D}^{\mathcal{S}})$.

(OBJ): Then $\mathcal{D} = \langle \vec{\mathcal{D}}_n, \textsc{obj}\rangle :: \Pi \vdash \texttt{new } C(\vec{e}_n) : C$ and for each $i \in \overline{n}$ $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ for some ϕ_i. By induction it follows that $\mathit{Comp}(\mathcal{D}_i{}^{\mathcal{S}} :: \Pi' \vdash e_i{}^{\mathsf{S}} : \phi_i)$ for each $i \in \overline{n}$ and then by Theorem 4.27 we have that $\mathrm{SN}(\mathcal{D}_i{}^{\mathcal{S}} :: \Pi' \vdash e_i{}^{\mathsf{S}} : \phi_i)$ for each $i \in \overline{n}$. So by Lemma 4.21(4) we have that $\mathrm{SN}(\langle \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{obj}\rangle :: \Pi' \vdash \texttt{new } C(e_1{}^{\mathsf{S}},\ldots,e_n{}^{\mathsf{S}}) : C)$ and thus by Definition 4.25 that $\mathit{Comp}(\langle \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{obj}\rangle :: \Pi \vdash \texttt{new } C(e_1{}^{\mathsf{S}},\ldots,e_n{}^{\mathsf{S}}) : C)$. Notice that $\langle \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{obj}\rangle = \mathcal{D}^{\mathcal{S}}$ and so $\mathit{Comp}(\mathcal{D}^{\mathcal{S}})$.

(NEWF): Then $\mathcal{D} = \langle \vec{\mathcal{D}}_n, \textsc{newf}\rangle :: \Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f_j : \sigma\rangle$ with $\mathcal{F}(C) = \vec{f}_n$ and $j \in \overline{n}$, and there is some $\vec{\phi}_n$ such that $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$ with $\phi_j \trianglelefteq \sigma$ and $\phi_j \neq \omega$. By induction $\mathit{Comp}(\mathcal{D}_i{}^{\mathcal{S}} :: \Pi \vdash e_i : \phi_i)$ for each $i \in \overline{n}$. Now, take $\mathfrak{D}_{(0,\sigma)} = \langle [\,]\rangle$ and $\mathfrak{C} = [\,]$. Notice that $\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_j{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}[e_j{}^{\mathsf{S}}] : \sigma = \mathcal{D}_j{}^{\mathcal{S}} :: \Pi \vdash e_j{}^{\mathsf{S}} : \phi_j$ and so $\mathit{Comp}(\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_j{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}[e_j{}^{\mathsf{S}}] : \phi_j)$. Then by Lemma 4.28 it follows that $\mathit{Comp}(\mathfrak{D}_{(0,\sigma)}[\langle\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_j{}^{\mathcal{S}}, \textsc{newf}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}[\texttt{new } C(e_1{}^{\mathsf{S}},\ldots,e_n{}^{\mathsf{S}}).f_j] : \sigma)$, that is $\mathit{Comp}(\langle\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_j{}^{\mathcal{S}}, \textsc{newf}\rangle, \textsc{fld}\rangle :: \Pi \vdash \texttt{new } C(e_1{}^{\mathsf{S}},\ldots,e_n{}^{\mathsf{S}}).f_j : \sigma)$. Then by Definition 4.25 we have that $\mathit{Comp}(\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_j{}^{\mathcal{S}}, \textsc{newf}\rangle :: \Pi \vdash \texttt{new } C(e_1{}^{\mathsf{S}},\ldots,e_n{}^{\mathsf{S}}) : \langle f_j : \sigma\rangle)$. Notice that $\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_j{}^{\mathcal{S}}, \textsc{newf}\rangle = \mathcal{D}^{\mathcal{S}}$ and so $\mathit{Comp}(\mathcal{D}^{\mathcal{S}})$.

(NEWM): Then $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newm}\rangle :: \Pi \vdash \texttt{new } C(\vec{e}) : \langle m : (\vec{\phi}_n) \to \sigma\rangle$ with $\mathcal{M}b(C,m) = (\overline{x''}_n, e_b)$ such that $\mathcal{D}_b :: \Pi'' \vdash e_b : \sigma$ and $\mathcal{D}_0 :: \Pi \vdash \texttt{new } C(\vec{e}) : \psi$ where $\Pi'' = \{\texttt{this}{:}\psi, x''_1{:}\phi_1, \ldots, x''_n{:}\phi_n\}$. By induction we have $\mathit{Comp}(\mathcal{D}_0{}^{\mathcal{S}} :: \Pi' \vdash \texttt{new } C(\vec{e})^{\mathsf{S}} : \psi)$. Now, assume there exist derivations $\mathcal{D}_1 :: \Pi_1 \vdash e'_1 : \phi_1, \ldots, \mathcal{D}_1 :: \Pi_n \vdash e'_n : \phi_n$ such that $\mathit{Comp}(\mathcal{D}_i)$ for each $i \in \overline{n}$. Let $\Pi''' = \bigcap \Pi' \cdot \vec{\Pi}_n$; notice, by Lemma 3.7, that $\Pi''' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$ so from Lemma 4.6 it follows that $\mathit{Comp}(\mathcal{D}_i[\Pi''' \trianglelefteq \Pi_i] :: \Pi''' \vdash e'_i : \phi_i)$ for each. Also by Lemma 3.7, $\Pi''' \trianglelefteq \Pi'$ and so then too by Lemma 4.6 we have $\mathit{Comp}(\mathcal{D}_0{}^{\mathcal{S}}[\Pi''' \trianglelefteq \Pi'] :: \Pi''' \vdash \texttt{new } C(\vec{e})^{\mathsf{S}} : \psi)$. Now consider the derivation substitution $\mathcal{S}' = \{\texttt{this} \mapsto \mathcal{D}_0{}^{\mathcal{S}}[\Pi''' \trianglelefteq \Pi'], x'_1 \mapsto \mathcal{D}_1[\Pi''' \trianglelefteq \Pi_1], \ldots, x'_n \mapsto \mathcal{D}_n[\Pi''' \trianglelefteq \Pi_n]\}$. Notice that $\mathcal{S}'$ is computable in Π'' and applicable to $\mathcal{D}_b$. So by induction it follows that $\mathit{Comp}(\mathcal{D}_b{}^{\mathcal{S}'} :: \Pi''' \vdash e_b{}^{\mathsf{S}'} : \sigma)$ where S' is the term substitution induced by $\mathcal{S}'$. Taking the derivation context $\mathfrak{D}_{(0,\sigma)} = \langle [\,]\rangle$ and the expression context $\mathfrak{C} = [\,]$, notice that $\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_b{}^{\mathcal{S}'}] :: \Pi''' \vdash \mathfrak{C}[e_b{}^{\mathsf{S}'}] : \sigma = \mathcal{D}_b{}^{\mathcal{S}'} :: \Pi''' \vdash e_b{}^{\mathsf{S}'} : \sigma$, and so $\mathit{Comp}(\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_b{}^{\mathcal{S}'}] :: \Pi''' \vdash \mathfrak{C}[e_b{}^{\mathsf{S}'}] : \sigma)$. From Lemma 4.28 we then have

$$\mathit{Comp}(\mathfrak{D}_{(0,\sigma)}[\langle \mathcal{D}', \mathcal{D}_1[\Pi''' \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi''' \trianglelefteq \Pi_n], \textsc{invk}\rangle] :: \Pi''' \vdash \mathfrak{C}[\texttt{new } C(\vec{e})^{\mathsf{S}}.m(\vec{e'}_n)] : \sigma)$$

where $\mathcal{D}' = \langle \mathcal{D}_b, \mathcal{D}_0{}^{\mathcal{S}}[\Pi''' \trianglelefteqslant \Pi'], \text{NEWM} \rangle$, that is

$$Comp(\langle \mathcal{D}', \mathcal{D}_1[\Pi''' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi''' \trianglelefteqslant \Pi_n], \text{INVK} \rangle :: \Pi''' \vdash \texttt{new } C(\vec{e})^{\texttt{S}}.m(\overleftarrow{e}_n) : \sigma)$$

Notice that $\mathcal{D}' = \mathcal{D}^{\mathcal{S}}[\Pi' \trianglelefteqslant \Pi''']$. Since the existence of the derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ was assumed, the following implication holds:

$$\forall \vec{\mathcal{D}}_n \; [\, Comp(\mathcal{D}_i :: \Pi_i \vdash e'_i : \phi_i) \,] \Rightarrow Comp(\langle \mathcal{D}'', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \text{INVK} \rangle :: \Pi''' \vdash \texttt{new } C(\vec{e}).m(\overleftarrow{e}_n) : \sigma)$$

where $\mathcal{D}'' = \mathcal{D}^{\mathcal{S}}[\Pi''' \trianglelefteqslant \Pi']$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi''' \trianglelefteqslant \Pi_i]$ for each $i \in \overline{n}$, with $\Pi''' = \bigcap \Pi' \cdot \vec{\Pi}_n$. So, by Definition 4.25 it follows that $Comp(\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash \texttt{new } C(\vec{e})^{\texttt{S}} : \langle m : (\vec{\phi}_n) \to \sigma \rangle)$.

□

Using this, we can show that all valid derivations are computable.

Lemma 4.31. $\mathcal{D} :: \Pi \vdash e : \phi \Rightarrow Comp(\mathcal{D} :: \Pi \vdash e : \phi)$

Proof. Suppose $\Pi = \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$, then we take the identity substitution Id_Π which, by Lemma 4.29, is computable in Π. Notice also that, by Definition 4.11, Id_Π is applicable to $\mathcal{D}$. Then from Lemma 4.30 we have $Comp(\mathcal{D}^{Id_\Pi})$ and since, by Lemma 4.16, $\mathcal{D}^{Id_\Pi} = \mathcal{D}$ it follows that $Comp(\mathcal{D})$. □

Then the main result of this chapter follows directly.

Theorem 4.32 (Strong Normalisation for Derivations)**.** *If* $\mathcal{D} :: \Pi \vdash e : \phi$ *then* $SN(\mathcal{D})$.

Proof. By Lemma 4.31 and Theorem 4.27(1) □

5. The Approximation Result: Linking Types with Semantics

5.1. Approximation Semantics

In this section we will define an *approximation semantics* for FJ¢ by generalising the notion of approximant for the λ-calculus that was discussed in Section 3.2. The concept of approximants in the context of FJ¢ can be illustrated using the class table given on the following page in Figure 5.1. This program codes lists of integers and uses them to implement the Prime Sieve algorithm of Eratosthenes. It is not quite a proper FJ¢ program, since it uses some extensions to the language, namely pure integer values and arithmetic operations on them, and an `if-then-else` construct. Note that these features can be encoded in pure FJ¢ (see Section 6.4), and so these extensions serve merely as a syntactic convenience for the purposes of illustration.

Lists of integers are coded in this program as expressions of the following form:

```
new NonEmpty(n1, new NonEmpty(n2, ...
                                  new NonEmpty(nk, new IntList()) ...))
```

To denote such lists, we will use the shorthand notation `n1:n2:...:nk:[]`. To illustrate the concept of approximants we will first consider calling the `square` method on a list of integers, which returns a list containing the squares of all the numbers in the original list. The reduction behaviour of such a program is given below, where we also give the corresponding (direct) approximant for each stage of execution:

The expression:	has the approximant:
`(1:2:3:[]).square()`	$\bot$
$\rightarrow^*$ `1:(2:3:[]).square()`	`1:`$\bot$
$\rightarrow^*$ `1:4:(3:[]).square()`	`1:4:`$\bot$
$\rightarrow^*$ `1:4:9:([]).square()`	`1:4:9:`$\bot$
$\rightarrow^*$ `1:4:9:[]`	`1:4:9:[]`

In this case, the output is finite, and the final approximant is the end-result of the computation itself. Not all computations are terminating, however, but might still produce output. An example of such a program is the prime sieve algorithm, which is initiated in the program of Figure 5.1 by calling the `primes` method (note that in the following we have abbreviated the method name `removeMultiplesOf` to

```
class IntList extends Object {
    IntList square() { return new IntList(); }
    IntList removeMultiplesOf(int n) { return new IntList(); }
    IntList sieve() { return new IntList(); }
    IntList listFrom(int n) { return new NonEmpty(n, this.listFrom(n+1)); }
    IntList primes() { return this.listFrom(2).sieve(); }
}

class NonEmpty extends IntList {
    int val;
    IntList tail;
    IntList square() {
            return new NonEmpty(this.val * this.val, this.tail.square()); }
    IntList removeMultiplesOf(int n) {
        if (this.val % n == 0) return this.tail.removeMultiplesOf(n);
        else return new NonEmpty(this.val, this.tail.removeMultiplesOf(n));
    }
    IntList sieve() {
        return new NonEmpty(this.val,
                            this.tail.removeMultiplesOf(this.val).sieve());
    }
}
```

Figure 5.1.: The class table for the Sieve of Eratosthenes in FJ$^{¢}$

rMO):

The expression:	has the approximant:
`new IntList().primes()`	$\bot$
$\rightarrow^*$ `(2:3:4:5:6:7:8:9:10:11:...).sieve()`	$\bot$
$\rightarrow^*$ `2:(3:(4:5:6:7:8:9:10:11:...).rMO(2)).sieve()`	`2:`$\bot$
$\rightarrow^*$ `2:3:(((5:6:7:8:9:10:11:...)` `.rMO(2)).rMO(3)).sieve()`	`2:3:`$\bot$
$\rightarrow^*$ `2:3:5:((((7:8:9:10:11:...)` `.rMO(2)).rMO(3)).rMO(5)).sieve()`	`2:3:5:`$\bot$
$\vdots$	$\vdots$

The output keeps on ‘growing’ as the computation progresses, and thus it is infinite - there is no final approximant since the ‘result’ is never reached. Thus $\bot$ is in every approximant since, at every stage of the computation, reduction may still take place.

The approximation semantics is constructed by interpreting an expression as the set of all such approximations of its reduction sequence. We formalise this notion below and, as we will show shortly, such a semantics has a very direct and strong correspondence with the types that can be assigned to an expression.

Definition 5.1 (Approximate Expressions)**.** *1. The set of* approximate FJ$^{¢}$ *expressions is defined by the following grammar:*

$$a \;::=\; x \;\mid\; \bot \;\mid\; a.f \;\mid\; a.m(\vec{a}_n) \;\mid\; \texttt{new}\ C(\vec{a}_n) \quad (n \geq 0)$$

2. The set of normal *approximate expressions, $\mathbb{A}$, ranged over by A, is a strict subset of the set of*

approximate expressions and is defined by the following grammar:

$$\begin{array}{rcl} A & ::= & x \mid \bot \mid \texttt{new } C(\vec{A}_n) \quad (\mathcal{F}(C) = \vec{f}_n) \\ & \mid & A.f \mid A.m(\vec{A}) \quad (A \neq \bot, A \neq \texttt{new } C(\vec{A}_n)) \end{array}$$

The reason for naming *normal* approximate expressions becomes apparent when we consider the expressions that they approximate - namely expressions in (head) normal form. In addition, if we extend the notion of reduction so that field accesses and method calls on $\bot$ are themselves reduced to $\bot$, then we find that the normal approximate expressions are normal forms with respect to this extended reduction relation. Note that we enforce for normal approximate expressions of the form $\texttt{new } C(\vec{A})$ that the expression comprise the correct number of field values for the declared class C. We elaborate on this in Section 5.3 below.

Remark. *It is easy to show that all (normal approximate) expressions of form $A.f$ and $A.m(\vec{A})$ must necessarily be neutral (i.e. must have a variable in head position).*

The notion of approximation is formalised as follows.

Definition 5.2 (Approximation Relation)**.** *The* approximation relation $\sqsubseteq$ *is defined as the contextual closure of the smallest preorder on approximate expressions satisfying $\bot \sqsubseteq a$, for all a.*

The relationship between the approximation relation and reduction is characterised by the following result.

Lemma 5.3. *If $A \sqsubseteq e$ and $e \to^* e'$, then $A \sqsubseteq e'$.*

Proof. By induction on the definition of $\to^*$.

($e \to^* e$): $A \sqsubseteq e$ by assumption.

($e \to^* e'' \,\&\, e'' \to^* e'$): Double application of the inductive hypothesis.

($e \to e'$): By induction on the structure of normal approximate expressions.

- ($\bot$): Immediate, since $\bot \sqsubseteq e'$ by definition.
- (x): Trivial, since x does not reduce.
- ($A.f$): Then $e = e''.f$ with $A \sqsubseteq e''$. Also, since $A \neq \texttt{new } C(\vec{A}_n)$ it follows from Definition 5.2 that $e'' \neq \texttt{new } C(\vec{e}_n)$. Thus e is not a redex and the reduction must take place in e', that is $e' = e'''.f$ with $e'' \to e'''$. Then, by induction, $A \sqsubseteq e'''$ and so $A.f \sqsubseteq e'''.f$.
- ($A.m(\vec{A}_n)$): Then $e' = e'_0.m(\vec{e}_n)$ with $A \sqsubseteq e'_0$ and $A_i \sqsubseteq e_i$ for each $i \in \overline{n}$. Since $A \neq \texttt{new } C(\vec{A})$ it follows that $e'_0 \neq \texttt{new } C(\overleftarrow{e'})$. Since e is not a redex, there are only two possibilities for the reduction step:
  1. $e_0 \to e'_0$ and $e' = e'_0.m(\vec{e}_n)$. By induction $A \sqsubseteq e'_0$ and so also $A.m(\vec{A}_n) \sqsubseteq e'_0.m(\vec{e}_n)$.
  2. $e_j \to e'_j$ for some $j \in \overline{n}$ and $e' = e_0.m(\overleftarrow{e'}_n)$ with $e'_k = e_k$ for each $k \in \overline{n}$ such that $k \neq j$. Then, clearly $A_k \sqsubseteq e'_k$ for each $k \in \overline{n}$ such that $k \neq j$. Also, by induction $A_j \sqsubseteq e'_j$. Thus $A.m(\vec{A}_n) \sqsubseteq e_0.m(\overleftarrow{e'}_n)$.

(new $C(\vec{A}_n)$): Then $e = \texttt{new}\ C(\vec{e}_n)$ with $A_i \sqsubseteq e_i$ for each $i \in \overline{n}$. Also $e_j \rightarrow e'_j$ for some $j \in \overline{n}$ and $e' = \texttt{new}\ C(\vec{e'}_n)$ where $e'_k = e_k$ for each $k \in \overline{n}$ such that $k \neq j$. Then, clearly $A_k \sqsubseteq e'_k$ for each $k \in \overline{n}$ such that $k \neq j$ and by induction $A_j \sqsubseteq e'_j$. Thus, by Definition 5.2, $\texttt{new}\ C(\vec{A}_n) \sqsubseteq \texttt{new}\ C(\vec{e'}_n)$. □

Notice that this property expresses that the observable behaviour of a program can only increase (in terms of $\sqsubseteq$) through reduction.

We also define a *join* operation on approximate expressions.

Definition 5.4 (Join Operation)**.** *1. The* join *operation $\sqcup$ on approximate expressions is a partial mapping defined as the smallest reflexive and contextual closure of:*

$$\bot \sqcup a \quad = \quad a \sqcup \bot \quad = \quad a$$

2. We extend the join operation to sequences of approximate expressions as follows:

$$\sqcup \epsilon = \bot$$
$$\sqcup a \cdot \vec{a}_n = a \sqcup (\sqcup \vec{a}_n)$$

The following lemma shows that $\sqcup$ acts as an upper bound on approximate expressions, and that it is closed over the set of *normal* approximate expressions.

Lemma 5.5. *Let a_1, a_2 and a be approximate expressions such that $a_1 \sqsubseteq a$ and $a_2 \sqsubseteq a$; then $a_1 \sqcup a_2 \sqsubseteq a$, with both $a_1 \sqsubseteq a_1 \sqcup a_2$ and $a_2 \sqsubseteq a_1 \sqcup a_2$. Moreover, if a_1 and a_2 are* normal *approximate expressions, then so is $a_1 \sqcup a_2$.*

Proof. By induction on the structure of a.

($a = \bot$): Then by Definition 5.2, $a_1 = a_2 = \bot$ (so they are normal approximate expressions) and by Definition 5.4, $a_1 \sqcup a_2 = \bot$ (which is also normal). By Definition 5.2, $\bot \sqsubseteq \bot$, and so the result follows immediately.

($a = x$): Then we consider the different possibilities for a_1 and a_2 (notice in all cases both a_1 and a_2 are normal):

($a_1 = \bot, a_2 = \bot$): By Definition 5.4, $a_1 \sqcup a_2 = \bot \sqcup \bot = \bot$ (which is normal). By Definition 5.2, $\bot \sqsubseteq a$ and so $a_1 \sqcup a_2 \sqsubseteq a$, and also $\bot \sqsubseteq \bot$ so thus $a_1 \sqsubseteq a_1 \sqcup a_2$ and $a_2 \sqsubseteq a_1 \sqcup a_2$.

($a_1 = \bot, a_2 = x$): By Definition 5.4, $a_1 \sqcup a_2 = \bot \sqcup x = x$ (which is normal). By Definition 5.2, $x \sqsubseteq x$ and so $a_1 \sqcup a_2 \sqsubseteq a$ and $a_2 \sqsubseteq a_1 \sqcup a_2$. Also by Definition 5.2, $\bot \sqsubseteq x$ and so $a_1 \sqsubseteq a_1 \sqcup a_2$.

($a_1 = x, a_2 = \bot$): Symmetric to the case ($a_1 = \bot, a_2 = x$) above.

($a_1 = x, a_2 = x$): By Definition 5.4, $a_1 \sqcup a_2 = x \sqcup x = x$ (which is normal). The result follows from the fact that, by Definition 5.2, $x \sqsubseteq x$.

($a = a'.f$): Then again we consider the different possibilities for a_1 and a_2.

($a_1 = \bot, a_2 = \bot$): By Definition 5.4, $a_1 \sqcup a_2 = \bot \sqcup \bot = \bot$ (which is normal). By Definition 5.2, $\bot \sqsubseteq a$ and so $a_1 \sqcup a_2 \sqsubseteq a$, and also $\bot \sqsubseteq \bot$ so thus $a_1 \sqsubseteq a_1 \sqcup a_2$ and $a_2 \sqsubseteq a_1 \sqcup a_2$.

($a_1 = \bot, a_2 \neq \bot$): Notice $\bot$ is normal. By Definition 5.4, $a_1 \sqcup a_2 = \bot \sqcup a_2 = a_2$, and so $a_1 \sqcup a_2$ is trivially normal if a_2 is normal. By Definition 5.2, $\bot \sqsubseteq a_2$ and so $a_1 \sqsubseteq a_1 \sqcup a_2$. Also by Definition 5.2, $a_2 \sqsubseteq a_2$ and so $a_2 \sqsubseteq a_1 \sqcup a_2$. Finally, since $a_2 \sqsubseteq a$ by assumption, it follows that $a_1 \sqcup a_2 \sqsubseteq a$.

($a_1 \neq \bot, a_2 = \bot$): Symmetric to the case above.

($a_1 = a'_1.f, a_2 = a'_2.f, a'_1 \sqsubseteq a', a'_2 \sqsubseteq a'$): By induction it follows that $a'_1 \sqcup a'_2 \sqsubseteq a'$ with $a'_1 \sqsubseteq a'_1 \sqcup a'_2$ and $a'_2 \sqsubseteq a'_1 \sqcup a'_2$. Then by Definition 5.2 it immediately follows that $a'_1 \sqcup a'_2.f \sqsubseteq a'.f$ with $a'_1.f \sqsubseteq a'_1 \sqcup a'_2.f$ and $a'_2.f \sqsubseteq a'_1 \sqcup a'_2.f$. The result follows from the fact that, by Definition 5.4, $a_1 \sqcup a_2 = a'_1 \sqcup a'_2.f$.

Moreover, if a_1 and a_2 are normal, then by definition so are a'_1 and a'_2, with both a'_1 and a'_2 being neither $\bot$, nor of the form $\texttt{new}\ C(\overline{a}^{\prime}_n)$. Then by induction $a'_1 \sqcup a'_2$ is also normal, and by Definition 5.4 the join is neither equal to $\bot$ nor of the form $\texttt{new}\ C(\overline{a}^{\prime}_n)$. Thus, by Definition 5.2, $a'_1 \sqcup a'_2.f = a_1 \sqcup a_2$ is a normal approximate expression.

($a = a'.m(\overline{a}^{\prime}_n)$),($a = \texttt{new}\ C(\overline{a}^{\prime}_n)$): By straightforward induction similar to the case $a = a'.f$. □

Definition 5.6 (Approximants)**.** *The function $\mathcal{A}$ returns the set of* approximants *of an expression e and is defined by:*

$$\mathcal{A}(e) = \{\, A \mid \exists\, e' \ [\, e \to^* e' \ \&\ A \sqsubseteq e' \,] \,\}$$

Thus, an approximant is a normal approximate expression that approximates some (intermediate) stage of execution. This notion of approximant allows us to define an approximation model for FJ$^{¢}$.

Definition 5.7 (Approximation Semantics)**.** *The* approximation model *for an* FJ$^{¢}$ *program is a structure $\langle \wp(\mathbb{A}), [\![\cdot]\!] \rangle$, where the interpretation function $[\![\cdot]\!]$, mapping expressions to elements of the domain, $\wp(\mathbb{A})$, is defined by $[\![e]\!] = \mathcal{A}(e)$.*

As for models of LC, our approximation semantics equates pairs of expressions that are in the reduction relation, as shown by the following theorem.

Theorem 5.8. $e_1 \to^* e_2 \Rightarrow \mathcal{A}(e_1) = \mathcal{A}(e_2)$.

Proof. ($\supseteq$):

$$\begin{array}{lll} e_1 \to^* e_2 \ \&\ A \in \mathcal{A}(e_2) & \Rightarrow & \text{(Def. 5.6)} \\ e_1 \to^* e_2 \ \&\ \exists e_3 [\, e_2 \to^* e_3 \ \&\ A \sqsubseteq e_3 \,] & \Rightarrow & \text{(trans. } \to^*\text{)} \\ \exists e_3 [\, e_1 \to^* e_3 \ \&\ A \sqsubseteq e_3 \,] & \Rightarrow & \text{(Def. 5.6)} \\ A \in \mathcal{A}(e_1) & & \end{array}$$

($\subseteq$):

$$\begin{array}{lll} e_1 \to^* e_2 \ \&\ A \in \mathcal{A}(e_1) & \Rightarrow & \text{(Def. 5.6)} \\ e_1 \to^* e_2 \ \&\ \exists e_3 [\, e_1 \to^* e_3 \ \&\ A \sqsubseteq e_3 \,] & \Rightarrow & \text{(Church-Rosser)} \\ \exists e_3, e_4 [\, e_1 \to^* e_2 \ \&\ e_2 \to^* e_4 \ \&\ e_1 \to^* e_3 \ \&\ e_3 \to^* e_4 \ \&\ A \sqsubseteq e_3 \,] & \Rightarrow & \text{(Lem. 5.3)} \\ \exists e_4 [\, e_2 \to^* e_4 \ \&\ A \sqsubseteq e_4 \,] & \Rightarrow & \text{(Def. 5.6)} \\ A \in \mathcal{A}(e_2) & & \end{array}$$

□

5.2. The Approximation Result

We will now describe the relationship that our intersection type system from Chapter 3 has with the semantics that we defined in the previous section. This takes the form of an *Approximation Theorem*, which states that for every typeable approximant of an expression, the same type can be assigned to the expression itself:

$$\Pi \vdash e : \phi \Leftrightarrow \exists\, A \in \mathcal{A}(e)\ [\Pi \vdash A : \phi]$$

As in other systems [15, 10], this result is a direct consequence of the strong normalisability of derivation reduction, which was demonstrated in Chapter 4. In this section, we will show that the structure of the normal form of a given derivation exactly corresponds to the structure of the approximant which can be typed. This is a very strong property since, as we will demonstrate, it means that typeability provides a sufficient condition for the (head) normalisation of *expressions,* i.e. it leads to a *termination* analysis for $\text{FJ}^{\mathfrak{C}}$.

Definition 5.9 (Type Assignment for Approximate Expressions)**.** *Type assignment for approximate expressions is defined exactly as for expressions, using the rules given in Figure 3.1.*

Since we have not modified the type assignment rules in any way other than allowing them to operate over the (larger) set of *approximate* expressions, note that all the results from Chapters 3 and 4 hold of this extended type assignment. Furthermore, since there is no extra explicit rule for typing $\bot$, the only type which may be assigned to $\bot$ is ω. Indeed, this is the case for any expression of the form $\mathfrak{C}[\bot]$ where $\mathfrak{C}$ is a neutral context.

To use the result of Theorem. 4.32 to show the Approximation Result, we first need to show some intermediate properties. Firstly, we show that ω-safe derivations in normal form do not type expressions containing $\bot$; it is from this property that we can show the ω-safe typeability guarantees normalisation.

Lemma 5.10. *If $\mathcal{D} :: \Pi \vdash A : \phi$ with ω-safe $\mathcal{D}$ and Π, then A does not contain $\bot$; moreover, if A is neutral, then ϕ does not contain ω.*

Proof. By induction on the structure of $\mathcal{D}$.

$\langle\omega\rangle$: Vacuously true since $\langle\omega\rangle$ derivations are not ω-safe.

$\langle\text{VAR}\rangle$: Then $A = x$ and so does not contain $\bot$. Since x is neutral, we must also show that ϕ does not contain ω. Notice ϕ is strict and there is some $\psi \trianglelefteq \phi$ such that $x{:}\psi \in \Pi$. Since ϕ is strict, $\psi \neq \omega$ and since Π is ω-safe it follows that ψ does not contain ω; therefore, neither does ϕ.

$\langle\mathcal{D}', \overrightarrow{\mathcal{D}}_n, \text{INVK}\rangle$: Then $A = A'.m(\vec{A}_n)$ and ϕ is strict, hereafter called σ. Also $\mathcal{D}' :: \Pi \vdash A' : \langle m{:}(\vec{\phi}_n) \to \sigma\rangle$ with $\mathcal{D}'$ ω-safe, and $\mathcal{D}_i :: \Pi \vdash A_i : \phi_i$ for each $i \in \overline{n}$. By induction A' must not contain $\bot$. Also, notice that A must be neutral, and therefore so must A'. Then it also follows by induction that $\langle m{:}(\vec{\phi}_n) \to \sigma\rangle$ does not contain ω. This means that no $\phi_i = \omega$, and so it must be that each $\mathcal{D}_i$ is ω-safe; thus by induction it follows that no A_i contains $\bot$ either. Consequently, $A'.m(\vec{A}_n)$ does not contain $\bot$ and σ does not contain ω.

$\langle\mathcal{D}_b, \mathcal{D}', \text{NEWM}\rangle$: Then $\mathcal{D}_b :: \Pi' \vdash e_b : \sigma$ with $\texttt{this}{:}\psi \in \Pi'$ and $\mathcal{D}' :: \Pi \vdash A : \psi$. Since $\mathcal{D}$ is ω-safe so also is $\mathcal{D}'$ and by induction it then follows that A does not contain $\bot$.

(FLD), (OBJ), (NEWF), (JOIN): These cases follow straightforwardly by induction. □

The next lemma simply states the soundness of type assignment with respect to the approximation relation.

Lemma 5.11. *If $\mathcal{D} :: \Pi \vdash a : \phi$ (with $\mathcal{D}$ ω-safe) and $a \sqsubseteq a'$ then there exists a derivation $\mathcal{D}' :: \Pi \vdash a' : \phi$ (where $\mathcal{D}'$ is ω-safe).*

Proof. By induction on the structure of $\mathcal{D}$.

(ω): Immediate, taking $\mathcal{D}' = \langle\omega\rangle :: \Pi \vdash a' : \omega$. In the ω-safe version of the result, this case is vacuously true since $\mathcal{D} :: \Pi \vdash a : \omega$ is not an ω-safe derivation.

(VAR): Then $a = x$ and $\mathcal{D} = \langle\text{VAR}\rangle :: \Pi \vdash x : \sigma$. By Definition 5.2, it must be that $a' = x$, and so we take $\mathcal{D}' = \mathcal{D}$. Notice that $\mathcal{D}$ is an ω-safe derivation.

(FLD): Then $a = a_1.f$ and $\mathcal{D} = \langle\mathcal{D}', \text{FLD}\rangle :: \Pi \vdash a_1.f : \sigma$ with $\mathcal{D}' :: \Pi \vdash a_1 : \langle f : \sigma\rangle$ (notice that if $\mathcal{D}$ is ω-safe then by definition so is $\mathcal{D}'$). Since $a_1.f \sqsubseteq a'$, by Definition 5.2 it follows that $a' = a_2.f$ with $a_1 \sqsubseteq a_2$. By the inductive hypothesis there then exists a derivation $\mathcal{D}''$ such that $\mathcal{D}'' :: \Pi \vdash a_2 : \langle f : \sigma\rangle$ (with $\mathcal{D}''$ ω-safe) and by rule (FLD) it follows that $\langle\mathcal{D}'', \text{FLD}\rangle :: \Pi \vdash a_2.f : \sigma$ (which by definition is ω-safe if $\mathcal{D}''$ is).

(JOIN), (INVK), (OBJ), (NEWF), (NEWM): These cases follow straightforwardly by induction, similar to the case for (FLD) above. □

We can show that we can type the join of normal approximate expressions with the intersection of all the types which they can be individually assigned.

Lemma 5.12. *Let $A_1, \ldots, A_n$ be normal approximate expressions with $n \geq 2$ and e be an expression such that $A_i \sqsubseteq e$ for each $i \in \overline{n}$; if there are (ω-safe) derivations $\overrightarrow{\mathcal{D}_n}$ such that $\mathcal{D}_i :: \Pi \vdash A_i : \phi_i$ for each $i \in \overline{n}$, then $\sqcup\vec{A}_n \sqsubseteq e$ and there are (ω-safe) derivations $\overrightarrow{\mathcal{D}'_n}$ such that $\mathcal{D}'_i :: \Pi \vdash \sqcup\vec{A}_n : \phi_i$ for each $i \in \overline{n}$. Moreover, $\sqcup\vec{A}_n$ is also a normal approximate expression.*

Proof. By induction on n.

($n = 2$): Then there are A_1 and A_2 such that $A_1 \sqsubseteq e$ and $A_2 \sqsubseteq e$. By Lemma 5.5 it follows that $A_1 \sqcup A_2 \sqsubseteq e$ with $A_1 \sqcup A_2$ a normal approximate expression, and also that $A_1 \sqsubseteq A_1 \sqcup A_2$ and $A_1 \sqsubseteq A_2 \sqcup A_2$. Therefore, given that $\mathcal{D}_1 :: \Pi \vdash A_1 : \phi_1$ and $\mathcal{D}_2 :: \Pi \vdash A_2 : \phi_2$ (with ω-safe $\mathcal{D}_1$ and $\mathcal{D}_2$), it follows from Lemma 5.11 that there exist derivations $\mathcal{D}'_1$ and $\mathcal{D}'_2$ such that $\mathcal{D}'_1 :: \Pi \vdash A_1 \sqcup A_2 : \phi_1$ (with $\mathcal{D}'_1$ ω-safe) and $\mathcal{D}'_2 :: \Pi \vdash A_1 \sqcup A_2 : \phi_2$ (with $\mathcal{D}'_2$ ω-safe). The result then follows from the fact that, by Definition 5.4

$$\begin{aligned} \sqcup\vec{A}_2 &= A_1 \sqcup (\sqcup A_2 \cdot \epsilon) \\ &= A_1 \sqcup (A_2 \sqcup (\sqcup \epsilon)) \\ &= A_1 \sqcup (A_2 \sqcup \bot) \\ &= A_1 \sqcup A_2 \end{aligned}$$

($n > 2$): By assumption $A_i \sqsubseteq e$ and $\mathcal{D}_i :: \Pi \vdash A_i : \phi_i$ (with $\mathcal{D}_i$ ω-safe) for each $i \in \overline{n}$. Notice that $\vec{A}_n = A_1 \cdot \vec{A'}_{n'}$ where $n = n' + 1$ and $A'_i = A_{i+1}$ for each $i \in \overline{n'}$. Thus $A'_i \sqsubseteq e$ and $\mathcal{D}_{i+1} :: \Pi \vdash A'_i : \phi_{i+1}$ for

each $i \in \overline{n'}$. Therefore by the inductive hypothesis it follows that $\sqcup\overrightarrow{A}_{n'} \sqsubseteq e$ with $\sqcup\overrightarrow{A}_{n'}$ a normal approximate expression, and $\mathcal{D}'_i :: \Pi \vdash \sqcup\overrightarrow{A}_{n'} : \phi_{i+1}$ (with $\mathcal{D}'_i$ ω-safe) for each $i \in \overline{n'}$. Then we have by Lemma 5.5 that $A_1 \sqcup (\sqcup\overrightarrow{A}_{n'}) \sqsubseteq e$ with $A_1 \sqcup (\sqcup\overrightarrow{A}_{n'})$ a normal approximate expression, and also that $A_1 \sqsubseteq A_1 \sqcup (\sqcup\overrightarrow{A}_{n'})$ with $\sqcup\overrightarrow{A}_{n'} \sqsubseteq A_1 \sqcup (\sqcup\overrightarrow{A}_{n'})$. So by Lemma 5.11 there is a derivation $\mathcal{D}'''$ (with $\mathcal{D}'''$ ω-safe) such that $\mathcal{D}''' :: \Pi \vdash A_1 \sqcup (\sqcup\overrightarrow{A}_{n'}) : \phi_1$, and ($\omega$-safe) derivations $\overrightarrow{\mathcal{D}''}_{n'}$ such that $\mathcal{D}''_i :: \Pi \vdash A_1 \sqcup (\sqcup\overrightarrow{A}_{n'}) : \phi_{i+1}$ for each $i \in \overline{n'}$. The result then follows from the fact that, by Definition 5.4, $\sqcup\vec{A}_n = A_1 \sqcup (\sqcup\overrightarrow{A}_{n'})$. □

The next property is the most important, since it is this that expresses the relationship between the structure of a derivation and the typed approximant.

Lemma 5.13. *If $\mathcal{D} :: \Pi \vdash e : \phi$ (with $\mathcal{D}$ ω-safe) and $\mathcal{D}$ is in normal form with respect to $\rightarrow_{\mathfrak{D}}$, then there exists A and (ω-safe) $\mathcal{D}'$ such that $A \sqsubseteq e$ and $\mathcal{D}' :: \Pi \vdash A : \phi$.*

Proof. By induction on the structure of $\mathcal{D}$.

(ω): Take $A = \bot$. Notice that $\bot \sqsubseteq e$ by Definition 5.2, and by (ω) we can take $\mathcal{D}' = \langle\omega\rangle :: \Pi \vdash \bot : \omega$. In the ω-safe version of the result, this case is vacuously true since the derivation $\mathcal{D} = \langle\omega\rangle :: \Pi \vdash e : \omega$ is not ω-safe.

(VAR): Then $e = x$ and $\mathcal{D} = \langle\text{VAR}\rangle :: \Pi \vdash x : \sigma$ (notice that this is a derivation in normal form). By Definition 5.1, x is already an approximate normal form and $x \sqsubseteq x$ by Definition 5.2. So we take $A = x$ and $\mathcal{D}' = \mathcal{D}$. Moreover, notice that by Definition 4.9, $\mathcal{D}$ is an ω-safe derivation.

(JOIN): Then $\mathcal{D} = \langle\overrightarrow{\mathcal{D}}_n, \text{JOIN}\rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n$ with $n \geq 2$ and $\mathcal{D}_i :: \Pi \vdash e : \sigma_i$ for each $i \in \overline{n}$. Since $\mathcal{D}$ is in normal form it follows that each $\mathcal{D}_i (i \in \overline{n})$ is in normal form too (and also, if $\mathcal{D}$ is ω-safe then by Definition 4.9 each $\mathcal{D}_i$ is ω-safe too). By induction there then exist normal approximate expressions $\vec{A}_n$ and (ω-safe) derivations $\overrightarrow{\mathcal{D}'}_n$ such that, for each $i \in \overline{n}$, $A_i \sqsubseteq e$ and $\mathcal{D}'_i :: \Pi \vdash e : \sigma_i$. Now, by Lemma 5.12 it follows that $\sqcup\vec{A}_n \sqsubseteq e$ with $\sqcup\vec{A}_n$ normal and that there are (ω-safe) derivations $\overrightarrow{\mathcal{D}''}_n$ such that $\mathcal{D}''_i :: \Pi \vdash \sqcup\vec{A}_n : \sigma_i$ for each $i \in \overline{n}$. Finally, by the (JOIN) rule we can take (ω-safe) $\mathcal{D}' = \langle\overrightarrow{\mathcal{D}''}_n, \text{JOIN}\rangle :: \Pi \vdash \sqcup\vec{A}_n : \sigma_1 \cap \ldots \cap \sigma_n$.

(FLD): Then $e = e'.f$ and $\mathcal{D} = \langle\mathcal{D}', \text{FLD}\rangle :: \Pi \vdash e'.f : \sigma$ with $\mathcal{D}' :: \Pi \vdash e' : \langle f : \sigma\rangle$. Since $\mathcal{D}$ is in normal form, so too is $\mathcal{D}'$. Furthermore, if $\mathcal{D}$ is ω-safe then by Definition 4.9 so too is $\mathcal{D}'$. By the inductive hypothesis it follows that there is some A and (ω-safe) derivation $\mathcal{D}''$ such that $A \sqsubseteq e'$ and $\mathcal{D}'' :: \Pi \vdash A : \langle f : \sigma\rangle$. Then by rule (FLD), $\langle\mathcal{D}'', \text{FLD}\rangle :: \Pi \vdash A.f : \sigma$ and by Definition 5.2, $A.f \sqsubseteq e'.f$. Moreover, by Definition 4.9, when $\mathcal{D}''$ is ω-safe so too is $\langle\mathcal{D}'', \text{FLD}\rangle$.

(INVK), (OBJ), (NEWF), (NEWM): These cases follow straightforwardly by induction similar to (FLD). □

The above result shows that the derivation $\mathcal{D}'$ that types the approximant is constructed from the normal form $\mathcal{D}$ by replacing sub-derivations of the form $\langle\omega\rangle :: \Pi \vdash e : \omega$ by $\langle\omega\rangle :: \Pi \vdash \bot : \omega$ (thus covering any redexes appearing in e). Since $\mathcal{D}$ is in normal form, there are also no *typed* redexes, ensuring that the expression typed in the conclusion of $\mathcal{D}'$ is a normal approximate expression. The 'only if' part of the approximation result itself then follows easily from the fact that $\rightarrow_{\mathfrak{D}}$ corresponds to reduction of expressions, so $A$is also an *approximant* of e. The 'if' part follows from the first property above and subject expansion.

Theorem 5.14 (Approximation)**.** $\Pi \vdash e : \phi$ *if and only if there exists* $A \in \mathcal{A}(e)$ *such that* $\Pi \vdash A : \phi$.

Proof. (if): By assumption, there is an approximant A of e such that $\Pi \vdash A : \phi$, so $e \to^* e'$ with $A \sqsubseteq e'$. Then, by Lemma 5.11, $\Pi \vdash e' : \phi$ and by subject expansion (Theorem 3.11) also $\Pi \vdash e : \phi$.

(only if): Let $\mathcal{D} :: \Pi \vdash e : \phi$, then by Theorem 4.32, $\mathcal{D}$ is strongly normalising. Take the normal form $\mathcal{D}'$; by the soundness of derivation reduction (Theorem 4.23), $\mathcal{D}' :: \Pi \vdash e' : \phi$ and $e \to^* e'$. By Lemma 5.13, there is some normal approximate expression A such that $\Pi \vdash A : \phi$ and $A \sqsubseteq e'$. Thus by Definition 5.6, $A \in \mathcal{A}(e)$. □

5.3. Characterisation of Normalisation

As in other intersection type systems [15, 10], the approximation theorem underpins characterisation results for various forms of termination. Our intersection type system gives full characterisations of head normalising and strongly normalising expressions. As regards to normalisation however, our system only gives a guarantee rather than a full characterisation, since ω-safe derivations are not preserved by derivation expansion.

We will begin by defining (head) normal forms for FJ¢.

Definition 5.15 (FJ¢ Normal Forms)**.** *1. The set of (well-formed)* head-normal forms *(ranged over by* H*) is defined by:*

$$\begin{array}{rcl} H & ::= & x \mid \mathtt{new}\ C(\vec{e}_n) \quad (\mathcal{F}(C) = \vec{f}_n) \\ & \mid & H.f \mid H.m(\vec{e}) \quad (H \neq \mathtt{new}\ C(\vec{e})) \end{array}$$

2. The set of (well-formed) normal *forms (ranged over by* N*) is defined by:*

$$\begin{array}{rcl} N & ::= & x \mid \mathtt{new}\ C(\vec{N}_n) \quad (\mathcal{F}(C) = \vec{f}_n) \\ & \mid & N.f \mid N.m(\vec{N}) \quad (N \neq \mathtt{new}\ C(\vec{N})) \end{array}$$

Notice that the difference between normal and head-normal forms sits in the second and fourth alternatives, where head-normal forms allow arbitrary expressions to be used. Also note that we stipulate that a (head) normal expression of the form $\mathtt{new}\ C(\vec{e})$ must have the correct number of field values as defined in the declaration of class C. This ties in with our notion of normal approximate expressions (see Definition 5.6), and thus approximants, which also must have the correct number of field values. Expressions of this form with either less or more field values may *technically* constitute (head) normal forms in that they cannot be (head) reduced further, but we discount them as malformed since they do not 'morally' constitute valid objects according to the class table. This decision is motivated from a technical point of view, too. According to the typing rules (in particular, the (OBJ) and (NEWF) rules), object expressions can only be assigned non-trivial types if they have the correct number of field values. So in order to ensure that all head normal forms are non-trivially typeable, and thus obtain a full characterisation of head normalising expressions, we restrict (head) normal expressions to be 'well-formed'.

The following lemma shows that normal approximate expressions which are not $\bot$ are (head) normal forms.

Lemma 5.16. *1. If* $A \neq \bot$ *and* $A \sqsubseteq e$*, then* e *is a head-normal form.*

2. *If $A \sqsubseteq e$ and A does not contain $\bot$, then e is a normal form.*

Proof. By straightforward induction on the structure of A using Definition 5.2. □

Thus any type, or more accurately any type derivation other than those of the form $\langle\omega\rangle$ (corresponding to the approximant $\bot$), specifies the structure of a (head) normal form via the normal form of its derivation.

An important part of the characterisation of normalisation is that every (head) normal form is non-trivially typeable.

Lemma 5.17 (Typeability of (head) normal forms)**.** 1. *If e is a head-normal form then there exists a strict type σ and type environment Π such that $\Pi \vdash e : \sigma$; moreover, if e is not of the form $\mathtt{new}\ C(\vec{e}_n)$ then for any arbitrary strict type σ there is an environment such that $\Pi \vdash e : \sigma$.*

2. *If e is a normal form then there exist strong strict type σ, type environment Π and derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash e : \sigma$; moreover, if e is not of the form $\mathtt{new}\ C(\vec{e}_n)$ then for any arbitrary strong strict type there exist strong $\mathcal{D}$ and Π such that $\mathcal{D} :: \Pi \vdash e : \sigma$.*

Proof. 1. By induction on the structure of head normal forms.

(x): By the (VAR) rule, $\{x{:}\sigma\} \vdash x : \sigma$ for any arbitrary strict type.

($\mathtt{new}\ C(\vec{e}_n)$): Notice that $\mathcal{F}(C) = \vec{f}_n$, by definition of the head normal form. Let us take the empty type environment, $\emptyset$. Notice that by rule (ω) we can derive $\emptyset \vdash e_i : \omega$ for each $i \in \overline{n}$. Then, by rule (OBJ) we can derive $\emptyset \vdash \mathtt{new}\ C(\vec{e}_n) : C$ for any type environment.

($H.f$): Notice that, by definition, H is a head normal expression *not* of the form $\mathtt{new}\ C(\vec{e}_n)$, thus by induction for any arbitrary strict type σ there is an environment Π such that $\Pi \vdash H : \sigma$. Let us pick some (other) arbitrary strict type σ', then there is an environment Π such that $\Pi \vdash H : \langle f{:}\sigma'\rangle$. Thus, by rule (FLD) we can derive $\Pi \vdash H.f : \sigma'$ for any arbitrary strict type σ'.

($H.m(\vec{e}_n)$): This case is very similar to the previous one. Notice that, by definition, H is a head normal expression *not* of the form $\mathtt{new}\ C(\vec{e}_n)$, thus by induction for any arbitrary strict type σ there is an environment Π such that $\Pi \vdash H : \sigma$. Let us pick some (other) arbitrary strict type σ', then there is an environment Π such that $\Pi \vdash H : \langle m{:}(\vec{\omega}_n) \to \sigma'\rangle$. Notice that by rule ($\omega$) we can derive $\Pi \vdash e_i : \omega$ for each $i \in \overline{n}$. Thus, by rule (INVK) we can derive $\Pi \vdash H.m(\vec{e}_n) : \sigma'$ for any arbitrary strict type σ'.

2. By induction on the structure of normal forms.

(x): By the (VAR) rule, $\{x{:}\sigma\} \vdash x : \sigma$ for any arbitrary strict type, and in particular this holds for any arbitrary *strong* strict type. Also, notice that derivations of the form $\langle$VAR$\rangle$ are strong by Definition 4.8.

($\mathtt{new}\ C(\vec{N}_n)$): Notice that $\mathcal{F}(C) = \vec{f}_n$ by the definition of normal forms. Since each N_i is a normal form for $i \in \overline{n}$, it follows by induction that there are strong strict types $\vec{\sigma}_n$, environments $\vec{\Pi}_n$ and derivations $\vec{\mathcal{D}}_n$ such that $\mathcal{D}_i :: \Pi_i \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Let the environment $\Pi' = \bigcap \vec{\Pi}_n$; notice that, by Definition 3.6, $\Pi' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$, and also that since each Π_i is strong so is Π'. Thus, $[\Pi' \trianglelefteq \Pi_i]$ is a weakening for each $i \in \overline{n}$ and $\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i] :: \Pi' \vdash N_i : \sigma_i$ for each

$i \in \overline{n}$. Notice that, by Definition 4.5, weakening does not change the structure of derivations, therefore for each $i \in \overline{n}$, $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ is a strong derivation. Now, by rule (OBJ) we can derive

$$\langle \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \text{OBJ} \rangle :: \Pi' \vdash \texttt{new } C(\vec{N}_n) : C$$

Notice that C is a strong strict type, and that since each derivation $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ is strong then, by Definition 4.8, so is $\langle \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \text{OBJ} \rangle$.

($N.f$): Notice that, by definition, N is a normal expression *not* of the form $\texttt{new } C(\vec{N}_n)$, thus by induction for any arbitrary strong strict type σ there is a strong environment Π and derivation $\mathcal{D}$ such that $\Pi \vdash N : \sigma$. Let us pick some (other) arbitrary strong strict type σ', then there are strong Π and $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \langle f : \sigma' \rangle$. Thus, by rule (FLD) we can derive $\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash N.f : \sigma'$ for any arbitrary strict type σ'. Furthermore, notice that since $\mathcal{D}$ is strong it follows from Definition 4.8 that $\langle \mathcal{D}, \text{FLD} \rangle$ is also strong.

($N.m(\vec{N}_n)$): Since each N_i for $i \in \overline{n}$ is a normal form it follows by induction that there are strong strict types $\vec{\sigma}_n$, environments $\vec{\Pi}_n$ and derivations $\vec{\mathcal{D}}_n$ such that $\mathcal{D}_i :: \Pi_i \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Also notice that, by definition, N is a normal expression *not* of the form $\texttt{new } C(\vec{N}_n)$, thus by induction for any arbitrary strict type σ there is a strong environment Π and derivation $\mathcal{D}$ such that $\Pi \vdash N : \sigma$. Let us pick some (other) arbitrary strict type σ', then there are Π and $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \langle m : (\vec{\sigma}_n) \to \sigma' \rangle$. Let the environment $\Pi' = \bigcap \Pi \cdot \vec{\Pi}_n$ notice that, by Definition 3.6, $\Pi' \trianglelefteqslant \Pi$ and $\Pi' \trianglelefteqslant \Pi_i$ for each $i \in \overline{n}$, and also that since Π is strong and each Π_i is strong then so is Π'. Thus, $[\Pi' \trianglelefteqslant \Pi]$ is a weakening and $[\Pi' \trianglelefteqslant \Pi_i]$ is a weakening for each $i \in \overline{n}$. Then $\mathcal{D}[\Pi' \trianglelefteqslant \Pi] :: \Pi' \vdash N : \langle m : (\vec{\sigma}_n) \to \sigma' \rangle$ and $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i] :: \Pi' \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Notice that, by Definition 4.5, weakening does not change the structure of derivations, therefore $\mathcal{D}[\Pi' \trianglelefteqslant \Pi]$ is strong and for each $i \in \overline{n}$, $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ is also strong. Now, by rule (INVK)

$$\langle \mathcal{D}[\Pi' \trianglelefteqslant \Pi], \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \text{INVK} \rangle :: \Pi' \vdash N.m(\vec{N}_n) : \sigma'$$

for any arbitrary strong strict type σ'. Furthermore, by Definition 4.8, we have that

$$\langle \mathcal{D}[\Pi' \trianglelefteqslant \Pi], \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \text{INVK} \rangle$$

is a strong derivation. □

Now, using the approximation result and the above properties, the following characterisation of head-normalisation follows easily.

Theorem 5.18 (Head-normalisation)**.** $\Pi \vdash e : \sigma$ *if and only if* e *has a head-normal form.*

Proof. (if): Let e' be a head-normal of e. By Lemma 5.17(1) there exists a strict type σ and a type environment Π such that $\Pi \vdash e' : \sigma$. Then by subject expansion (Theorem 3.11) it follows that $\Pi \vdash e : \sigma$.

(only if): By the approximation theorem, there is an approximant A of e such that $\Pi \vdash A : \sigma$. Thus $e \to^* e'$ with $A \sqsubseteq e'$. Since σ is strict, it follows that $A \neq \bot$, so by Lemma 5.16 e' is a head-normal form. □

As we saw in Chapter 2 (Section 2.1), normalisability for the Lambda Calculus can be characterised in ITD as follows:

$$B \vdash M : \sigma \text{ with } B \text{ and } \sigma \text{ strong} \Leftrightarrow M \text{ has a normal form}$$

This result does not hold for FJ$^{\cent}$ (a counter-example can be found in one of the worked examples of the following chapter, namely the third expression in Example 6.11). In our system, in order to reason about the normalisation of expressions we must refer to properties of derivations as *whole*, and not just the environment and type used in the final judgement. In fact, we have already defined the conditions that derivations must satisfy in order to guarantee normalising since in FJ$^{\cent}$ expressions - namely, the conditions for ω-safe derivability.

In general, our type system only allows for a semi-characterisation result:

Theorem 5.19 (Normalisation)**.** *If* $\mathcal{D} :: \Pi \vdash e : \sigma$ *with* $\mathcal{D}$ *and* Π ω*-safe then* e *has a normal form.*

Proof. By the approximation theorem, there is an approximant A of e and derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi \vdash A : \sigma$ and $\mathcal{D} \rightarrow_{\mathfrak{D}}^{*} \mathcal{D}'$. Thus $e \rightarrow^{*} e'$ with $A \sqsubseteq e'$. Also, since derivation reduction preserves ω-safe derivations (Lemma 4.24), it follows that $\mathcal{D}'$ is ω-safe and thus by Lemma 5.10 that A does not contain $\bot$. Then by Lemma 5.16 we have that e' is a normal form. □

The reverse implication does not hold in general since our notion of ω-safe typeability is too fragile: it is not preserved by (derivation) expansion. Consider that while an ω-safe derivation may exist for $\Pi \vdash e_i : \sigma$, no ω-safe derivation may exist for $\Pi \vdash \texttt{new C}(\vec{e}_n)\texttt{.}f_i : \sigma$ (due to non-termination in the other expressions e_j) even though this expression too has a normal form, namely the same normal form as e_i. Such a completeness result *can* hold for certain particular programs, though. We will return to this in the following chapter, where we will give a class table and specify a set of expressions for which normalisation can be fully characterised by the FJ$^{\cent}$ intersection type system (see Section 6.5).

While we do not have a general characterisation of normalisation, we *can* show that the set of strongly normalising expressions are exactly those typeable using strong derivations. This follows from the fact that in such derivations, all redexes in the typed expression correspond to redexes in the derivation, and then any reduction step that can be made by the expression (via $\rightarrow$) is then matched by a corresponding reduction of the derivation (via $\rightarrow_{\mathfrak{D}}$).

Theorem 5.20 (Strong Normalisation for Expressions)**.** e *is strongly normalisable if and only if* $\mathcal{D} :: \Pi \vdash e : \sigma$ *with* $\mathcal{D}$ *strong.*

Proof. (if): Since $\mathcal{D}$ is strong, all redexes in e are typed and therefore each possible reduction of e is matched by a corresponding derivation reduction of $\mathcal{D}$. By Lemma 4.24 it follows that no reduction of $\mathcal{D}$ introduces subderivations of the form $\langle\omega\rangle$, and so since $\mathcal{D}$ is strongly normalising (Theorem 4.32) so too is e.

(only if): By induction on the maximum lengths of left-most outer-most reduction sequences for strongly normalising expressions, using the fact that all normal forms are typeable with strong derivations and that strong typeability is preserved under left-most outer-most redex expansion. □

6. Worked Examples

In this chapter, we will give several example programs and discuss how they are typed in the simple intersection type system. We will begin with some relatively simple examples, and then deal with some more complex programs. We will end the chapter by comparing the intersection type system with the nominal, class-based type system of Featherweight Java.

6.1. A Self-Returning Object

Perhaps the simplest example program that captures the essence of (the class-based approach to) object-orientation is that of an object that returns itself. This can be achieved using the following class:

```
class SR extends Object {
    SR self() { return this; }
}
```

Then, the expression `new SR().self()` reduces in a single step to `new SR()`. In fact, any arbitrary length sequence of calls to the `self` method on a `new SR()` object results, eventually, in an instance of the SR class:

$$\texttt{new SR().self()....self()} \rightarrow^{*} \texttt{new SR()}$$

This potentiality of behaviour is captured by the type analysis given to the expression `new SR()` by the intersection type system. We can assign it any of the infinite family of types:

$$\{\mathsf{SR}, \langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle, \langle \texttt{self:}() \rightarrow \langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle\rangle, \\ \langle \texttt{self:}() \rightarrow \langle \texttt{self:}() \rightarrow \langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle\rangle\rangle, \ldots\}$$

Derivations assigning these types to `new SR()` are given below.

$$\dfrac{}{\vdash \texttt{new SR()} : \mathsf{SR}}\,(\textsc{obj}) \qquad \dfrac{\dfrac{}{\texttt{this}:\mathsf{SR} \vdash \texttt{this} : \mathsf{SR}}\,(\textsc{var}) \qquad \dfrac{}{\vdash \texttt{new SR()} : \mathsf{SR}}\,(\textsc{obj})}{\vdash \texttt{new SR()} : \langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle}\,(\textsc{newM})$$

$$\dfrac{\dfrac{}{\{\texttt{this}:\langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle\} \vdash \texttt{this} : \langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle}\,(\textsc{var}) \qquad \vdots}{\vdash \texttt{new SR()} : \langle \texttt{self:}\langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle\rangle}\,(\textsc{newM}) \qquad \dfrac{\dfrac{}{\texttt{this}:\mathsf{SR} \vdash \texttt{this} : \mathsf{SR}}\,(\textsc{var}) \qquad \dfrac{}{\vdash \texttt{new SR()} : \mathsf{SR}}\,(\textsc{obj})}{\vdash \texttt{new SR()} : \langle \texttt{self:}() \rightarrow \mathsf{SR}\rangle}\,(\textsc{newM})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{this}{:}\texttt{SR} \vdash \texttt{this} : \texttt{SR}}\,(\textsc{var})}{\vdash \texttt{new SR()} : \langle \texttt{self}{:}() \to \texttt{SR}\rangle}\,(\textsc{newM}) \quad \dfrac{}{\vdash \texttt{new SR()} : \texttt{SR}}\,(\textsc{obj})}{\{\texttt{this}{:}\langle \texttt{self}{:}() \to \texttt{SR}\rangle\} \vdash \texttt{this} : \langle \texttt{self}{:}() \to \texttt{SR}\rangle}\,(\textsc{var}) \quad \vdots}{\vdash \texttt{new SR()} : \langle \texttt{self}{:}\langle \texttt{self}{:}() \to \texttt{SR}\rangle\rangle}\,(\textsc{newM}) \quad \vdots}{\vdash \texttt{new SR()} : \langle \texttt{self}{:}\langle \texttt{self}{:}\langle \texttt{self}{:}() \to \texttt{SR}\rangle\rangle\rangle}\,(\textsc{newM})$$

where $\sigma = \langle \texttt{self}{:}\langle \texttt{self}{:}() \to \texttt{SR}\rangle\rangle$

A variation on this is possible in the class-based paradigm, in which the object has a method that returns a new *instance* of the class of which itself is an instance:

```
class SR extends Object {
    SR newInst() { return new SR(); }
}
```

This program has the same behaviour as the previous one: invoking the `newInst` method on a `new SR()` object results in a `new SR()` object, and we can continue calling the `newInst` method as many times as we like. Thus, as expected, we can assign the types $\langle \texttt{newInst}{:}() \to \texttt{SR}\rangle$, $\langle \texttt{newInst}{:}() \to \langle \texttt{newInst}{:}() \to \texttt{SR}\rangle\rangle$, etc. For example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\texttt{this}{:}\texttt{SR}\} \vdash \texttt{new SR()} : \texttt{SR}}\,(\textsc{obj})}{\{\texttt{this}{:}\texttt{SR}\} \vdash \texttt{new SR()} : \langle \texttt{newInst}{:}() \to \texttt{SR}\rangle}\,(\textsc{newM}) \quad \dfrac{}{\{\texttt{this}{:}\texttt{SR}\} \vdash \texttt{new SR()} : \texttt{SR}}\,(\textsc{obj})}{\{\texttt{this}{:}\texttt{SR}\} \vdash \texttt{new SR()} : \langle \texttt{newInst}{:}() \to \langle \texttt{newInst}{:}() \to \texttt{SR}\rangle\rangle}\,(\textsc{newM}) \quad \vdots}{\vdash \texttt{new SR()} : \langle \texttt{newInst}{:}() \to \langle \texttt{newInst}{:}() \to \langle \texttt{newInst}{:}() \to \texttt{SR}\rangle\rangle\rangle}\,(\textsc{newM}) \quad \dfrac{}{\vdash \texttt{new SR()} : \texttt{SR}}\,(\textsc{obj})$$

Notice that there is a symmetry between this derivation for the `newInst` method, and the equivalent derivation for the `self` method. This is certainly to be expected since, operationally (in a pure functional setting at least), the use within method bodies of the self variable `this` and the new instance `new SR()` are interchangeable. In terms of the type analysis, the method types $\langle \texttt{newInst}{:}() \to \sigma\rangle$ are derived within the analysis for the method body whereas, on the other hand, each $\langle \texttt{self}{:}() \to \sigma\rangle$ is assumed for the self `this` when analysing the method body, and its derivation is deferred until the self types are checked for the receiver. Either way, however, there is always a subderivation assigning each type $\langle \texttt{self}{:}() \to \sigma\rangle$ to an instance of `new SR()`.

6.2. An Unsolvable Program

Let us now examine how the predicate system deals with programs that do not have a head-normal form. The approximation theorem states that any predicate which we can assign to an expression is also assignable to an approximant of that expression. As we mentioned in the previous chapter, approximants are snapshots of evaluation: they represent the information computed during evaluation. But by their very nature, programs which do not have a head-normal form do not compute any information. Formally, then, the characteristic property of unsolvable expressions (i.e. those without a head normal form) is that they do *not* have non-trivial approximants: their only approximant is $\bot$. From the approximation result

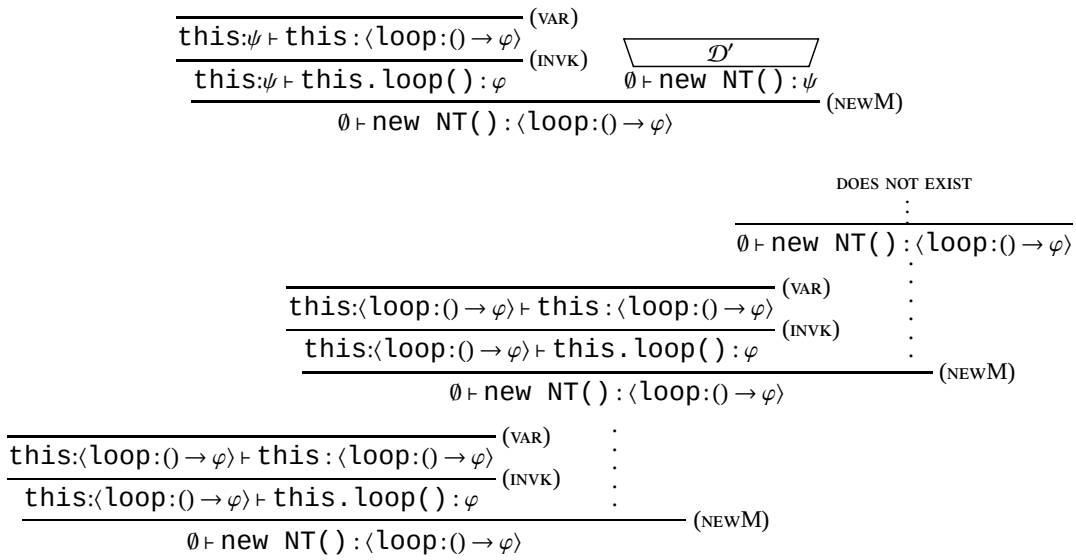


Figure 6.1.: Predicate Non-Derivations for a Non-Terminating Program

it therefore follows that we cannot build any derivation for these expressions that assigns a predicate other than ω (since that is the only predicate assignable to $\bot$).

To illustrate this, consider the following program which constitutes perhaps the simplest example of unsolvability in oo:

```
class NT extends Object {
    NT loop() { return this.loop(); }
}
```

The class `NT` contains a method `loop` which, when invoked (recursively) invokes itself on the receiver. Thus the expression `new NT().loop()`, executed using the above class table, will simply run to itself resulting in a non-terminating (and non-output producing) loop.

Figure 6.1 shows two candidate derivations assigning a non-trivial type to the non-terminating expression `new NT().loop()`, the first of which we can more accurately call a derivation *schema* since it specifies the form that any such derivation must take. When trying to assign a non-trivial type to the invocation of the method `loop` on `new NT()` we can proceed, without loss of generality, by building a derivation assigning a predicate variable φ, since we may then simply substitute any suitable (strict) predicate for φ in the derivation.

The derivation we need to build assigns the predicate φ to a method invocation so we must first build a derivation $\mathcal{D}$ that assigns the method predicate $\langle \texttt{loop:}() \to \varphi \rangle$ to the receiver `new NT()`. *This* derivation is constructed by examining the method body - `this.loop()` - and finding a derivation which assigns φ to it. This analysis reveals that the variable `this` must be assigned a predicate for the method `loop` which will be of the form $\langle \texttt{loop:}() \to \varphi \rangle$; `new NT()` (the receiver) must also satisfy the predicate ψ used for `this`. Finally, in order for the (VAR) leaf of the derivation to be valid the predicate ψ must satisfy the constraint that $\psi \trianglelefteq \langle \texttt{loop:}() \to \varphi \rangle$.

The second derivation of Figure 6.1 is an attempt at instantiating the schema that we have just constructed. In order to make the instantiation, we must pick a concrete predicate for ϕ satisfying the aforementioned constraint. Perhaps the simplest thing to do would be to pick $\phi = \langle \texttt{loop:}() \to \varphi \rangle$. Next,

we must instantiate the derivation $\mathcal{D}'$ assigning this predicate to the receiver `new NT()`. Here we run into trouble because, in order to achieve this, we must again type the body of method `m`, i.e. solve the same problem that we started with - we see that our instantiation of the derivation $\mathcal{D}'$ must be of exactly the same shape as our instantiation of the derivation $\mathcal{D}$; of course, this is impossible since $\mathcal{D}'$ is a proper subderivation of $\mathcal{D}$ and so no such derivation exists. Notice however, that the receiver `new NT()` itself is *not* unsolvable - indeed, it is a normal form - and so we can assign to it a non-trivial type. Namely, using the (OBJ) rule we can derive $\vdash$ `new NT() : NT`.

6.3. Lists

Recall that at the beginning of Chapter 5 we illustrated the concept of approximants using a program that manipulated lists of integers. In this section, we will return to the example of programming lists in FJ$^{¢}$ and briefly discuss two important features of the type analysis of the list construction.

The basic list construction in FJ$^{¢}$ consists of two classes - one to represent an empty list (`EL`), and the second to represent a non-empty list (`NEL`), i.e. a list with a head and a tail. In our FJ$^{¢}$ program, we will also define a `List` class, which will specify the basic interface for lists. These classes will also contain any methods that implement the operations that we would like to carry out on lists. We may write specialise lists in any way that we like, perhaps by writing subclasses that declare more methods implementing behaviour specific to different types of list (as in the program of Figure 5.1), but for now let us consider a basic list with one method to insert an element at the head of the list (`cons`) and another method to append one list onto the end of another:

```
class List extends Object {
      List cons(Object o) { return this; }
      List append(List l) { return this; }
}

class EL extends List {
      List cons(Object o) { return new NEL(o, this); }
      List append(List l) { return l; }
}

class NEL extends List {
      Object head;
      List tail;
      List cons(Object o) { return new NEL(o, this); }
      List append(List l) {
                return new NEL(this.head,
                               this.tail.append(l)); }
}
```

If we have some objects $o_1,\ldots,o_n$, then the list o_1`:`$\ldots$`:`o_n`:[]` (where `[]` denotes the empty list) is represented using the above program by the expression:

`new NEL(`o_1`, new NEL(`o_2`, ... new NEL(`o_n`, new EL()) ... ))`

The first key feature of the analysis of such a program provided by our intersection type system is that it is *generic*, in the sense that the type analysis reflects the capabilities of the actual objects in the list, no matter what kind of objects they are. For example, suppose we have some classes `Circle`, `Square`, `Triangle`, etc. representing different kinds of shapes, and each class contains a `draw` method. If we have a list containing instances of these classes then we can assign types to it that allow us to access these elements and invoke their `draw` method:

$$\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{new NEL(new Circle(...), new EL()):NEL}}\,(\textsc{new}O) \quad \vdots}{\Pi \vdash \texttt{new Square(...)} : \langle \texttt{draw}:(\vec{\sigma}) \to \tau \rangle}}{\Pi \vdash \texttt{new Square():new Circle():[]} : \langle \texttt{head}:\langle \texttt{draw}:(\vec{\sigma}) \to \tau \rangle\rangle}\,(\textsc{new}F)$$

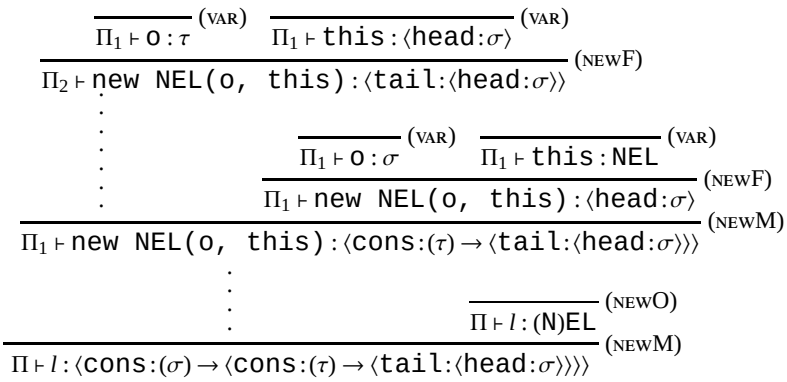
$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{new EL()} : \texttt{EL}}\,(\textsc{new}O) \quad \vdots}{\Pi \vdash \texttt{new Circle(...)} : \langle \texttt{draw}:(\vec{\sigma}) \to \tau \rangle}}{\Pi \vdash \texttt{new Circle(...):[]} : \langle \texttt{head}:\langle \texttt{draw}:(\vec{\sigma}) \to \tau \rangle\rangle}\,(\textsc{new}F) \quad \vdots}{\Pi \vdash \texttt{new Square()} : \texttt{Square}}\,(\textsc{new}O) \quad \vdots}{\Pi \vdash \texttt{new Square():new Circle():[]} : \langle \texttt{tail}:\langle \texttt{head}:\langle \texttt{draw}:(\vec{\sigma}) \to \tau \rangle\rangle\rangle}\,(\textsc{new}F)$$

If we had a different list containing objects implementing a different interface with some method `foo`, then the type system would provide an appropriate analysis, similar to the one described above, but assigning method types for `foo` instead. This is in contrast to the capabilities of Java (and FJ). If the above list construction were to be written and typed in FJ, while we would be allowed, via subsumption, to add any kind of object we chose to the list (since all classes are subtypes of `Object`), when retrieving elements from the list we would only be allowed to treat them as instances of `Object`, and thus not be able to invoke any of their methods. If we wanted to create lists of `Shape` objects and be able to invoke the `draw` method on those objects that we retrieve from it, we would either need to write new classes that code for lists of `Shape` objects *specifically*, or we would need to extend the type system with a mechanism for *generics*.

The second feature of the intersection type analysis for lists is that it allows for *heterogeneity*, or the ability to store objects of different kinds. There is nothing about the derivations above that forces the types derived for each element of the list to be the same. In general, for any type σ_i that can be derived for a list element o_i, the type

$$\underbrace{\langle \texttt{tail}:\langle \texttt{tail}:\ldots}_{i-1 \text{ times}}\langle \texttt{head}:\sigma_i\rangle\ldots\rangle\rangle$$

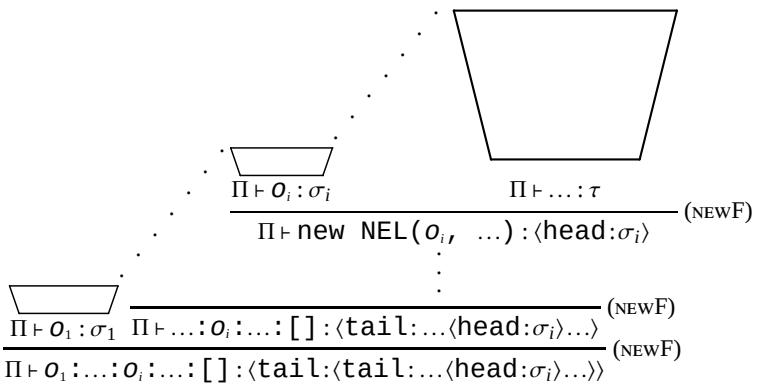
can be given to the list $o_1 \texttt{:}\ldots\texttt{:}o_i\texttt{:}\ldots\texttt{:[]}$ as illustrated by the diagram below:



where $\Pi_1 = \{\texttt{this}{:}(\texttt{N})\texttt{EL}, \texttt{o}{:}\sigma\}$

$\Pi_2 = \{\texttt{this}{:}\langle\texttt{head}{:}\sigma\rangle, \texttt{o}{:}\tau\}$

Figure 6.2.: Derivation for a heterogeneous `cons` method.



More important, perhaps, is that we can give types to the methods `cons` and `append` which allows us to create heterogeneous lists by invoking them. For example, for any types σ and τ, we can assign to a list l the type $\langle\texttt{cons}{:}(\sigma) \to \langle\texttt{cons}{:}(\tau) \to \langle\texttt{tail}{:}\langle\texttt{head}{:}\sigma\rangle\rangle\rangle\rangle$, as shown in the derivation in Figure 6.2. Types allowing the creation, via `cons`, of heterogeneous lists of any length can be derived however, obviously, the type derivations soon become monstrous! This fine-grained level of analysis is something which is not available via generics, which only allow for *homogeneous* lists.

6.4. Object-Oriented Arithmetic

We will now consider an encoding of natural numbers and some simple arithmetical operations on them. We remark that Abadi and Cardelli defined an object-oriented encoding of natural numbers in the ς-calculus. In their encoding, the successor of a number is obtained by calling a method on the encoding of that number, and due to the ability to override (i.e. replace) method bodies, only the encoding of zero need be defined explicitly. Since the class-based paradigm does not allow such an operation, our encoding must be slightly different.

The motivation for this example is twofold. Firstly, it serves as a simple, but effective illustration of the expressive power of intersection types. Secondly, and precisely because it is a program that admits of such expressive type analysis, it is a perfect program for mapping out the limits of type inference for

the intersection type system. Indeed, when we define a type inference procedure in the next chapter, we will consider the types that we may then infer for this program as an illustration of its limitations.

Our encoding is straightforward, and uses two classes - one to represent the number zero, and one to represent the successor of a(n encoded) number. As with the list example above, we will define a `Nat` class which simply serves to specify the interface of natural numbers. The full program is given below.

```
class Nat extends Object {
    Nat add(Nat x) { return this; }
    Nat mult(Nat x) { return this; }
}

class Zero extends Nat {
    Nat add(Nat x) { return x; }
    Nat mult(Nat x) { return this; }
}

class Suc extends Nat {
    Nat pred;
    Nat add(Nat x) { return new Suc(this.pred.add(x)); }
    Nat mult(Nat x) { return x.add(this.pred.mult(x)); }
}
```

The `Suc` class, representing the successor of a number uses a field to store its predecessor. The methods that implement addition and multiplication do so by translating the usual arithmetic identities for these operations into Featherweight Java syntax. Natural numbers are then encoded in the obvious fashion, as follows:

$$\llbracket 0 \rrbracket_{\mathbb{N}} = \texttt{new Zero()}$$
$$\llbracket i+1 \rrbracket_{\mathbb{N}} = \texttt{new Suc(}\llbracket i \rrbracket_{\mathbb{N}}\texttt{)}$$

Notice that each number n, then, has a *characteristic* type ν_n which can be assigned to that number and that number alone:

$$\nu_0 = \texttt{Zero}$$
$$\nu_{i+1} = \langle \texttt{pred:} \nu_i \rangle$$

This is already a powerful property for a type system, however in our intersection type system this has some very potent consequences. Because our system has the subject expansion property (Theorem 3.11), we can assign to any expression the characteristic type for its result. Thus, it is possible to prove statements like the following:

$$\forall n, m \in \mathbb{N} \,.\ \vdash \llbracket n \rrbracket_{\mathbb{N}}\texttt{.add(}\llbracket m \rrbracket_{\mathbb{N}}\texttt{)} : \nu_{n+m}$$
$$\forall n, m \in \mathbb{N} \,.\ \vdash \llbracket n \rrbracket_{\mathbb{N}}\texttt{.mult(}\llbracket m \rrbracket_{\mathbb{N}}\texttt{)} : \nu_{n*m}$$

For the simple operations of addition and multiplication this is more than straightforward. Notwithstanding, consider adding methods that implement more complex, indeed arbitrarily complex, arith-

metic functions. As a further example, we have included in the Appendix a type-based analysis of an implementation of Ackermann's function using our intersection type system.

The corollary to this is that we may also derive arbitrarily complex types describing the behaviour of the methods of `Zero` and `Suc` objects. The derivability of the typing statements that we gave above implies that we can also prove statements such as the following:

$$\forall n, m \in \mathbb{N} \,.\, \exists\, \sigma \,.\ \vdash [\![m]\!]_{\mathbb{N}} : \sigma \ \&\ \ \vdash [\![n]\!]_{\mathbb{N}} : \langle \texttt{mult:}(\sigma) \to \nu_{n*m} \rangle$$

Notice that we have not given the statement $\forall n, m \in \mathbb{N} \,.\ \ \vdash [\![n]\!]_{\mathbb{N}} : \langle \texttt{mult:}(\nu_m) \to \nu_{m*n} \rangle$ since it is not necessarily the case that ν_m is the type satisfying the requirements of the `mult` method on its argument. Indeed, it is *not* that simple - consider that the `mult` method (for positive numbers) needs to be able to call the `add` method on its argument.

To present another scenario, suppose for example that we were to combine our arithmetic program above with the list program of the previous section, and write a method `factors` that produces a list of the factors of a number (say, excluding one and itself) - a perfectly algorithmic process. The encodings of prime numbers then, would have the characteristic type $\langle \texttt{factors:}() \to \texttt{EL} \rangle$, expressing that the result of calling this method on them is the empty list, i.e. that they have no factors. It then becomes clear what the implications of a type inference procedure for this system are. If such a thing were to exist, we would need only to write a program implementing a function of interest, pass it to the type inference procedure, and run off a list of its number-theoretic properties.

As we have remarked previously, type assignment for a full intersection type system is undecidable, meaning there is no complete type inference algorithm. The challenge then becomes to restrict the intersection type system in such a way that type assignment becomes decidable (or simply to define an incomplete type inference algorithm) while still being able to assign useful types for programs. It is this last element of the problem which is the harder to achieve. In the next chapter, we will consider restricted notions of type assignment for our intersection type system, but observe that the conventional method of restricting intersection type assignment (based on rank) does not interact well with the object-oriented style of programming.

6.5. A Type-Preserving Encoding of Combinatory Logic

In this section, we show how Combinatory Logic can be encoded within FJ$^{¢}$. We also show that our encoding preserves Curry types, a result which could easily be generalised to intersection types. This is a very powerful result, since it proves that the intersection type system for FJ$^{¢}$ facilitates a functional analysis of all computable functions. Furthermore, using the results from the previous chapter, we can show that the type system also gives a *full* characterisation of the normalisation properties of the encoding.

Combinatory Logic (CL) is a Turing complete model of computation defined by H.B. Curry [44] independently of LC. It can be seen as a higher-order term rewriting system TRS consisting of the function symbols $\mathbf{S}, \mathbf{K}$ where terms are defined over the grammar

$$t \ ::= \ x \ \mid \ \mathbf{S} \ \mid \ \mathbf{K} \ \mid \ t_1\, t_2$$

```
class Combinator extends Object {
      Combinator app(Combinator x) { return this; }
}

class K extends Combinator {
      Combinator app(Combinator x) { return new K1(x); }
}

class K1 extends K {
      Combinator x;
      Combinator app(Combinator y) { return this.x; }
}

class S extends Combinator {
      Combinator app(Combinator x) { return new S1(x); }
}

class S1 extends S {
      Combinator x;
      Combinator app(Combinator y) { return new S2(this.x, y); }
}

class S2 extends S1 {
      Combinator y;
      Combinator app(Combinator z) {
          return this.x.app(z).app(this.y.app(z)); }
}
```

Figure 6.3.: The class table for Object-Oriented Combinatory Logic (oocl) programs

and the reduction is defined via following rewrite rules:

$$\begin{aligned} \mathbf{K}\, x\, y &\rightarrow x \\ \mathbf{S}\, x\, y\, z &\rightarrow x\, z\, (y\, z) \end{aligned}$$

Through our encoding, and the results we have shown in the previous chapter, we can achieve a type-based characterisation of all (terminating) computable functions in oo (see Theorem 6.10).

Our encoding of cl in fj$^{¢}$ is based on a Curryfied first-order version of the system above (see [14] for details), where the rules for **S** and **K** are expanded so that each new rewrite rule has a *single* operand, allowing for the partial application of function symbols. Application, the basic engine of reduction in trs, is modelled via the invocation of a method named `app`. The reduction rules of Curryfied cl each apply to (or are 'triggered' by) different 'versions' of the **S** and **K** combinators; in our encoding these rules are implemented by the bodies of five different versions of the `app` method which are each attached to different classes representing the different versions of the **S** and **K** combinators. In order to make our encoding a valid (typeable) program in full Java, we have defined a `Combinator` class containing an `app` method from which all the others inherit, essentially acting as an *interface* to which all encoded versions of **S** and **K** must adhere.

Definition 6.1. *The encoding of Combinatory Logic into the* fj$^{¢}$ *program* oocl *(Object-Oriented Combinatory Logic) is defined using the class table given in Figure 6.3 and the function* $⟦\cdot⟧$ *which translates*

terms of CL *into* FJ$^{¢}$ *expressions, and is defined as follows:*

$$\begin{array}{rcl} \ulcorner x \urcorner & = & x \\ \ulcorner t_1\ t_2 \urcorner & = & \ulcorner t_1 \urcorner \texttt{.app(} \ulcorner t_2 \urcorner \texttt{)} \\ \ulcorner \mathbf{K} \urcorner & = & \texttt{new K()} \\ \ulcorner \mathbf{S} \urcorner & = & \texttt{new S()} \end{array}$$

The reduction behaviour of OOCL mirrors that of CL.

Theorem 6.2. *If t_1, t_2 are terms of* CL *and $t_1 \to^* t_2$, then $\ulcorner t_1 \urcorner \to^* \ulcorner t_2 \urcorner$ in* OOCL.

Proof. By induction on the definition of reduction in CL; we only show the case for **S**:

$$\begin{array}{ll} \ulcorner \mathbf{S}\, t_1\, t_2\, t_3 \urcorner & \\ \triangleq & \texttt{new S().app(}\ulcorner t_1 \urcorner\texttt{).app(}\ulcorner t_2 \urcorner\texttt{).app(}\ulcorner t_3 \urcorner\texttt{)} \\ \to & \texttt{new S}_1\texttt{(}\ulcorner t_1 \urcorner\texttt{).app(}\ulcorner t_2 \urcorner\texttt{).app(}\ulcorner t_3 \urcorner\texttt{)} \\ \to & \texttt{new S}_2\texttt{(this.x,y).app(}\ulcorner t_3 \urcorner\texttt{)} \\ & \qquad\qquad \texttt{[this}\mapsto\texttt{new S}_1\texttt{(}\ulcorner t_1 \urcorner\texttt{),y}\mapsto\ulcorner t_2 \urcorner\texttt{]} \\ = & \texttt{new S}_2\texttt{(new S}_1\texttt{(}\ulcorner t_1 \urcorner\texttt{).x,}\ulcorner t_2 \urcorner\texttt{).app(}\ulcorner t_3 \urcorner\texttt{)} \\ \to & \texttt{new S}_2\texttt{(}\ulcorner t_1 \urcorner\texttt{,}\ulcorner t_2 \urcorner\texttt{).app(}\ulcorner t_3 \urcorner\texttt{)} \\ \to & \texttt{this.x.app(z).app(this.y.app(z))} \\ & \qquad\qquad \texttt{[this}\mapsto\texttt{new S}_2\texttt{(}\ulcorner t_1 \urcorner\texttt{,}\ulcorner t_2 \urcorner\texttt{),z}\mapsto\ulcorner t_3 \urcorner\texttt{]} \\ = & \texttt{new S}_2\texttt{(}\ulcorner t_1 \urcorner\texttt{,}\ulcorner t_2 \urcorner\texttt{).x.app(}\ulcorner t_3 \urcorner\texttt{)} \\ & \qquad \texttt{.app(new S}_2\texttt{(}\ulcorner t_1 \urcorner\texttt{.}\ulcorner t_2 \urcorner\texttt{).y.app(}\ulcorner t_3 \urcorner\texttt{))} \\ \to^* & \ulcorner t_1 \urcorner\texttt{.app(}\ulcorner t_3 \urcorner\texttt{).app(}\ulcorner t_2 \urcorner\texttt{.app(}\ulcorner t_3 \urcorner\texttt{))} \\ \triangleq & \ulcorner t_1\, t_3\, (t_2\, t_3) \urcorner \end{array}$$

The case for **K** is similar, and the rest is straightforward. □

Given the Turing completeness of CL, this result shows that FJ$^{¢}$ is also Turing complete. Although we are sure this does not come as a surprise, it is a nice formal property for our calculus to have. In addition, our type system can perform the same 'functional' analysis as ITD does CL, as well as LC since there are also type preserving translations from LC to CL [50]. We illustrate this by way of a *type preservation* result. Firstly, we describe Curry's type system for CL and then show we can give equivalent types to OOCL programs.

Definition 6.3 (Curry Type Assignment for CL)**.** *1. The set of* simple types *(also known as Curry types) is defined by the following grammar:*

$$A, B \quad ::= \quad \varphi \quad | \quad A \to B$$

2. A basis Γ *is a mapping from variables to Curry types, written as a set of statements of the form x:A in which each of the variables x is distinct.*

3. Simple types are assigned to CL*-terms using the following natural deduction system:*

$$(Ax): \ \frac{}{\Gamma \vdash_{\text{CL}} x{:}A}\,(x{:}A \in \Gamma) \qquad (\to E): \ \frac{\Gamma \vdash_{\text{CL}} t_1{:}A \to B \quad \Gamma \vdash_{\text{CL}} t_2{:}A}{\Gamma \vdash_{\text{CL}} t_1 t_2{:}B}$$

$$(\mathbf{K}): \ \frac{}{\Gamma \vdash_{\text{CL}} \mathbf{K}{:}A \to B \to A} \qquad (\mathbf{S}): \ \frac{}{\Gamma \vdash_{\text{CL}} \mathbf{S}{:}(A \to B \to C) \to (A \to B) \to A \to C}$$

The elegance of this approach is that we can now link types assigned to combinators to types assignable to object-oriented programs. To show this type preservation, we need to define what the equivalent of Curry's types are in terms of our FJ$^{¢}$ types. To this end, we define the following translation of Curry types.

Definition 6.4 (Type Translation)**.** *The function* $\lceil\cdot\rfloor$*, which transforms Curry types*[1]*, is defined as follows:*

$$\begin{aligned} \lceil\varphi\rfloor &= \varphi \\ \lceil A \to B\rfloor &= \langle \mathsf{app}{:}(\lceil A\rfloor) \to \lceil B\rfloor\rangle \end{aligned}$$

It is extended to contexts as follows: $\lceil\Gamma\rfloor = \{x{:}\lceil A\rfloor \mid x{:}A \in \Gamma\}$.

We can now show the type preservation result.

Theorem 6.5 (Preservation of Types)**.** *If* $\Gamma \vdash_{\text{CL}} t{:}A$ *then* $\lceil\Gamma\rfloor \vdash \lceil t\rfloor : \lceil A\rfloor$.

Proof. By induction on the derivation of $\Gamma \vdash_{\text{CL}} t{:}A$. The cases for (VAR) and ($\to E$) are trivial. For the rules (**K**) and (**S**), Figure 6.4 gives derivation schemas for assigning the translation of the respective Curry type schemes to the OOCL translations of **K** and **S**. □

Furthermore, since Curry's well-known translation of the simply typed LC into CL preserves typeability (see [50, 15]), we can also construct a type-preserving encoding of LC into FJ$^{¢}$; it is straightforward to extend this preservation result to full-blown strict intersection types. We stress that this result really demonstrates the validity of our approach. Indeed, our type system actually has more power than intersection type systems for CL as presented in [15], since there not all normal forms are typeable using strict types, whereas in our system they are. This is because our type system, in addition to giving a *functional* analysis, also gives a *structural* analysis through the class name type constants.

Example 6.6. *Let* δ *be the* CL*-term* **S** (**S K K**) (**S K K**)*. Notice that* $\delta\ \delta \to^* \delta\ \delta$*, i.e. it is unsolvable, and thus can only be given the type* ω *(this is also true for* $\lceil\delta\ \delta\rfloor$*). Now, consider the term* $t =$ **S** (**K** δ) (**K** δ)*. Notice that it is a normal form (*$\lceil t\rfloor$ *has a normal form also), but that for any term t',* **S** (**K** δ) (**K** δ) $t' \to^* \delta\ \delta$*. In a strict system, no functional analysis is possible for t since* $\phi \to \omega$ *is not a type and so the only way we can type this term is with* ω[2]*.*

In our type system however we may assign several different types to $\lceil t\rfloor$*. Most simply we can derive* $\vdash \lceil t\rfloor : \mathsf{S}_2$*, but even though a 'functional' analysis via the* app *method is impossible, it is still safe to access the fields of the value resulting from* $\lceil t\rfloor$ *– both* $\vdash \lceil t\rfloor : \langle \mathsf{x}{:}\mathsf{K}_2\rangle$ *and* $\vdash \lceil t\rfloor : \langle \mathsf{y}{:}\mathsf{K}_2\rangle$ *are also easily derivable statements. In fact, we can derive even more informative types: the expression* $\lceil\mathbf{K}\ \delta\rfloor$ *can be assigned types of the form* $\sigma_{\mathbf{K}\delta} = \langle \mathsf{app}{:}(\sigma_1) \to \langle \mathsf{app}{:}(\sigma_2 \cap \langle \mathsf{app}{:}(\sigma_2) \to \sigma_3\rangle) \to \sigma_3\rangle\rangle$*, and so we*

[1] Note we have *overloaded* the notation $\lceil\cdot\rfloor$, which we also use for the translation of CL terms to FJ$^{¢}$ expressions.

[2] In other intersection type systems (e.g. [20]) $\phi \to \omega$ is a permissible type, but is equivalent to ω (that is $\omega \le (\phi \to \omega) \le \omega$) and so semantics based on these type systems identify terms of type $\phi \to \omega$ with unsolvable terms.

can also assign $\langle \mathsf{x} : \sigma_{\mathbf{K}\delta} \rangle$ *and* $\langle \mathsf{y} : \sigma_{\mathbf{K}\delta} \rangle$ *to* $\llbracket t \rrbracket$*. Notice that the equivalent* λ*-term to* t *is* $\lambda y.(\lambda x.xx)(\lambda x.xx)$*, which is a* weak *head-normal form without a head-normal form. The 'functional' view is that such terms are observationally indistinguishable from unsolvable terms. When encoded in* FJ$^{\mathfrak{C}}$ *however, our type system shows that these terms become meaningful (head-normalisable). This is of course as expected, given that the notion of reduction in* FJ$^{\mathfrak{C}}$ *is weak.*

Our termination results from the previous chapter can be illustrated by applying them in the context of OOCL.

Definition 6.7 (OOCL normal forms)**.** *Let the set of* OOCL *normal forms be the set of expressions* n *such that* n *is the normal form of the image* $\llbracket t \rrbracket$ *of some* CL *term* t*. Notice that it can be defined by the following grammar:*

$$\begin{array}{rcl} n & ::= & x \mid \texttt{new K()} \mid \texttt{new K}_1\texttt{(}n\texttt{)} \mid \\ & & \texttt{new S()} \mid \texttt{new S}_1\texttt{(}n\texttt{)} \mid \texttt{new S}_2\texttt{(}n_1\texttt{,}n_2\texttt{)} \mid \\ & & n\texttt{.app(}n'\texttt{)} \quad (n \neq \texttt{new } C(\vec{e}_n)) \end{array}$$

Each OOCL normal form corresponds to a CL normal form, the translation of which can also be typed with an ω-safe derivation for each type assignable to the normal form.

Lemma 6.8. *If* e *is an* OOCL *normal form, then there exists a* CL *normal form* t *such that* $\llbracket t \rrbracket \to^* e$ *and for all* ω*-safe* $\mathcal{D}$ *and* Π *such that* $\mathcal{D} :: \Pi \vdash e : \sigma$*, there exists an* ω*-safe derivation* $\mathcal{D}'$ *such that* $\mathcal{D}' :: \Pi \vdash \llbracket t \rrbracket : \sigma$*.*

Proof. By induction on the structure of OOCL normal forms. □

We can also show that ω-safe typeability is preserved under expansion for the images of CL-terms in OOCL.

Lemma 6.9. *Let* t_1 *and* t_2 *be* CL*-terms such that* $t_1 \to t_2$*; if there is an* ω*-safe derivation* $\mathcal{D}$ *and environment* Π*, and a strict type* σ *such that* $\mathcal{D} :: \Pi \vdash \llbracket t_2 \rrbracket : \sigma$*, then there exists another* ω*-safe derivation* $\mathcal{D}'$ *such that* $\mathcal{D}' :: \Pi \vdash \llbracket t_1 \rrbracket : \sigma$*.*

Proof. By induction on the definition of reduction for CL. □

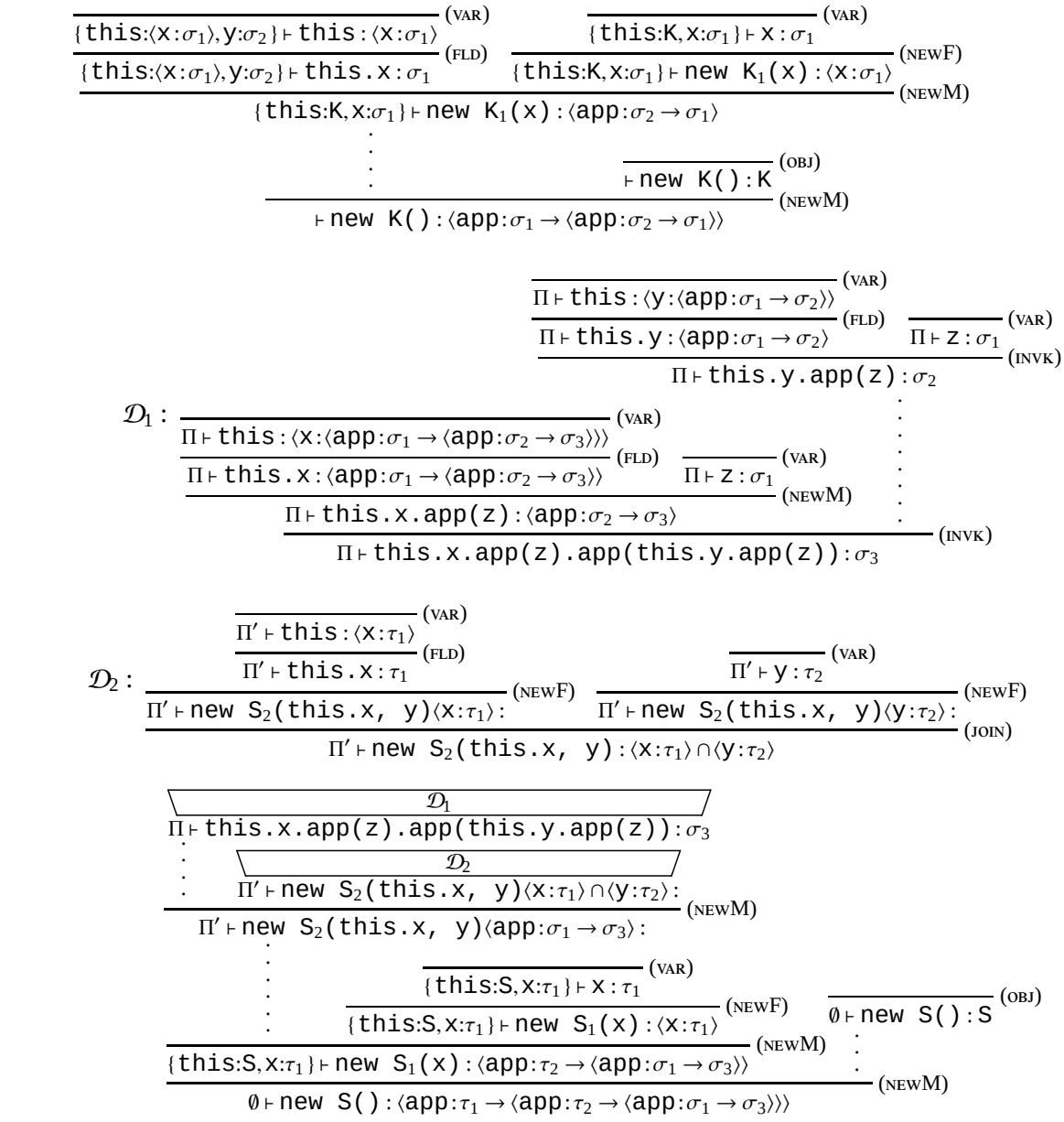
This property of course also extends to multi-step reduction.

Together with the lemma preceding it (and the fact that all normal forms can by typed with an ω-safe derivation), this leads to both a sound and *complete* characterisation of normalisability for the images of CL-terms in OOCL.

Theorem 6.10. *Let* t *be a* CL*-term: then* t *is normalisable if and only if there are* ω*-safe* $\mathcal{D}$ *and* Π*, and strict type* σ *such that* $\mathcal{D} :: \Pi \vdash \llbracket t \rrbracket : \sigma$*.*

Proof. (if): Directly by Theorem 5.19.

(only if): Let t' be the normal form of t; then, by Theorem 6.2, $\llbracket t \rrbracket \to^* \llbracket t' \rrbracket$. Since reduction in CL is confluent, $\llbracket t' \rrbracket$ is normalisable as well; let e be the normal form of $\llbracket t' \rrbracket$. Then by Lemma 5.17(2) there are strong strict type σ, environment Π and derivation $\mathcal{D}$ such that $\Pi \vdash e : \sigma$. Since $\mathcal{D}$ and Π are strong, they are also ω-safe. Then, by Lemma 6.8 and 6.9, there exists ω-safe $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi \vdash \llbracket t \rrbracket : \sigma$. □



where $\tau_1 = \langle \mathtt{app}{:}\sigma_1 \to \langle \mathtt{app}{:}\sigma_2 \to \sigma_3 \rangle \rangle, \tau_2 = \langle \mathtt{app}{:}\sigma_1 \to \sigma_2 \rangle,$

$\Pi = \{\mathtt{this}{:}\langle \mathtt{x}{:}\tau_1 \rangle \cap \langle \mathtt{y}{:}\tau_2 \rangle, \mathtt{z}{:}\sigma_1\}$, and

$\Pi' = \{\mathtt{this}{:}\langle \mathtt{x}{:}\tau_1 \rangle, \mathtt{y}{:}\tau_2\}$

Figure 6.4.: Derivation schemes for the translations of **S** and **K**

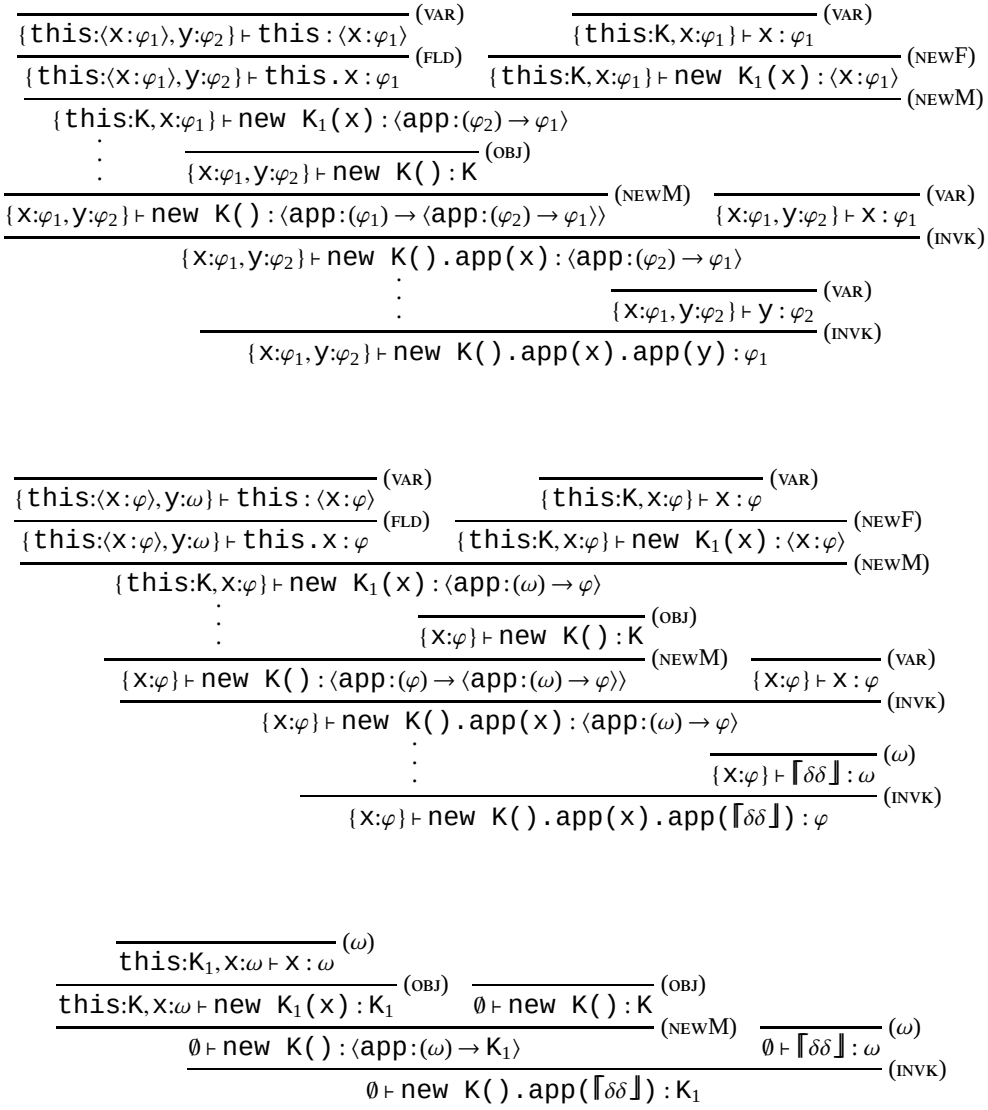


Figure 6.5.: Derivations for Example 6.11

The OOCL program very nicely illustrates the various characterisations of terminating behaviour that the intersection type assignment system gives.

Example 6.11. *Let δ be the* CL*-term* **S** (**S K K**) (**S K K**) *– i.e. $\delta\delta$ is an unsolvable term. Figure 6.5 shows, respectively,*

- *a strong derivation typing a strongly normalising expression of* OOCL*;*
- *an ω-safe derivation of a normalising (but not strongly normalising) expression of* OOCL*; and*
- *a derivation (*not *ω-safe) assigning a non-trivial type for a head-normalising (but not normalising)* OOCL *expression,*

The last of these examples was referred to in Section 5.3 as an illustration of the difference between the characterisation of normalising expression in ITD for LC and the corresponding characterisation in $\text{FJ}^{\text{¢}}$. It shows that we cannot look just at the derived type (and type environment) in order to know if some expression has a normal form - we must look at the whole typing derivation, as in the second example above.

The examples that we have discussed so far have not directly illustrated the Approximation Theorem (5.14). To finish this section, we will now look at an example which shows how the types we can assign in the intersection type system predict the approximants of an expression, and therefore provide information about runtime behaviour. The example that we will look at is that of a *fixed-point combinator*. The OOCL program only contains classes to encode the combinators **S** and **K** and, while it is possible to construct terms using only **S** and **K** which are fixed-point operators, there is no reason that we cannot extend our program and define new combinators directly.

A *fixed-point* of a function F is a value M such that $M = F(M)$; a fixed-point *combinator* (or *operator*) is a (higher-order) function that returns a fixed-point of its argument (another function). Thus, a fixed-point combinator G has the property that $GF = F(GF)$ for any function F. Turing's well-known fixed-point combinator in the λ-calculus is the following term:

$$Tur \quad = \quad \Theta\Theta \quad = \quad (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

That *Tur* provides a fixed-point constructor is easy to check:

$$Tur\ f \quad = \quad (\lambda xy.y(xxy))\Theta f \quad \rightarrow_\beta^* \quad f(\Theta\Theta f) \quad = \quad f(Tur\ f)$$

The term *Tur* itself has the reduction behaviour

$$\begin{array}{rll} Tur = (\lambda xy.y(xxy))\Theta & \rightarrow_\beta & \lambda y.y(\Theta\Theta y) \\ & \rightarrow_\beta & \lambda y.y((\lambda z.z(\Theta\Theta z))y) \\ & \rightarrow_\beta & \lambda y.y(y(\Theta\Theta y)) \\ & \vdots & \end{array}$$

which implies it has the following set of approximants:

$$\{\bot,\ \lambda y.y\bot,\ \lambda y.y(y\bot),\ \ldots\}$$

Thus, if z is a term variable, the approximants of *Tur z* are $\bot, z\bot, z(z\bot)$, etc. As well as satisfying the

$$\mathcal{D}_1 :: \dfrac{\dfrac{\dfrac{}{\Pi_2 \vdash \mathtt{x} : \langle \mathtt{app} : (\omega) \to \varphi_2 \rangle}\,(\textsc{var}) \quad \dfrac{}{\Pi_2 \vdash \mathtt{this.app(x)} : \omega}\,(\omega)}{\Pi_2 \vdash \mathtt{x.app(this.app(x))} : \varphi}\,(\textsc{invk}) \quad \dfrac{}{\Pi_1 \vdash \mathtt{new\ T()} : \omega}\,(\omega)}{\Pi_1 \vdash \mathtt{new\ T()} : \langle \mathtt{app} : (\langle \mathtt{app} : (\omega) \to \varphi \rangle) \to \varphi \rangle}\,(\textsc{newM}) \quad \dfrac{}{\Pi_1 \vdash \mathtt{z} : \langle \mathtt{app} : (\omega) \to \varphi \rangle}\,(\textsc{var})}{\Pi_1 \vdash \mathtt{new\ T().app(z)} : \varphi}\,(\textsc{invk})$$

$$\mathcal{D}_2 :: \dfrac{\dfrac{}{\Pi_1 \vdash \mathtt{z} : \langle \mathtt{app} : (\omega) \to \varphi \rangle}\,(\textsc{var}) \quad \dfrac{}{\Pi_1 \vdash \mathtt{new\ T().app(z)} : \omega}\,(\omega)}{\Pi_1 \vdash \mathtt{z.app(new\ T().app(z))} : \varphi}\,(\textsc{invk})$$

$$\mathcal{D}_3 :: \dfrac{\dfrac{}{\Pi_1 \vdash \mathtt{z} : \langle \mathtt{app} : (\omega) \to \varphi \rangle}\,(\textsc{var}) \quad \dfrac{}{\Pi_1 \vdash \bot : \omega}\,(\omega)}{\Pi_1 \vdash \mathtt{z.app}(\bot) : \varphi}\,(\textsc{invk})$$

where $\Pi_1 = \{\mathtt{z}{:}\langle \mathtt{app} : (\omega) \to \varphi \rangle\}$, $\Pi_2 = \{\mathtt{this}{:}\omega, \mathtt{x}{:}\langle \mathtt{app} : (\omega) \to \varphi \rangle\}$

Figure 6.6.: Type Derivations for the Fixed-Point Construction Example

characteristic property of fixed-point combinators mentioned above, the term *Tur* satisfies the stronger property that $\mathit{Tur}\ M \to_\beta^* M(\mathit{Tur}\ M)$ for any term M.

It is straightforward to define a new FJ¢ class that can be added to the OOCL program which mirrors this behaviour:

```
class T extends Combinator {
    combinator app(Combinator x) {
        return x.app(this.app(x));
    }
}
```

The body of the `app` method in the class `T` encodes the reduction behaviour we saw for *Tur* above. For any FJ¢ expression e:

$$\mathtt{new\ T().app}(e) \;\to\; e\mathtt{.app(new\ T().app}(e)\mathtt{)}$$

So, taking $M = \mathtt{new\ T().app}(e)$, we have

$$M \;\to\; e\mathtt{.app}(M)$$

Thus, by Theorem 5.8, the fixed point M of e (as returned by the fixed point combinator class `T`) is semantically equivalent to $e\mathtt{.app}(M)$, and so $\mathtt{new\ T().app}(\cdot)$ does indeed represent a fixed-point constructor.

The (executable) expression $e = \mathtt{new\ T().app(z)}$ has the reduction behaviour

$$\begin{aligned} \mathtt{new\ T().app(z)} &\to \mathtt{z.app(new\ T.app(z))} \\ &\to \mathtt{z.app(z.app(new\ T.app(z)))} \\ &\vdots \end{aligned}$$

so has the following (infinite) set of approximants:

$$\{\bot,\ \texttt{z.app(}\bot\texttt{)},\ \texttt{z.app(z.app(}\bot\texttt{))},\ \ldots\}$$

Notice that these exactly correspond to the set of the approximants for the λ-term *Tur z* that we considered above. The derivation $\mathcal{D}_1$ in Figure 6.6 shows a possible derivation assigning the type φ to `e`. In fact, the normal form of this derivation corresponds to the approximant `z.app(`$\bot$`)`, which we will now demonstrate.

The derivation $\mathcal{D}_1$ comprises a *typed redex*, in this case a derivation of the form $\langle\langle\cdot,\cdot,\text{NEWM}\rangle,^{*},\text{INVK}\rangle$, thus it will reduce. The derivation $\mathcal{D}_2$ shows the result of performing the reduction step. In this example, the type ω is assigned to the receiver `new T()`, since that is the type associated with `this` in the environment Π_2 used when typing the method body. It would have been possible to use a more specific type for `this` in Π_2 (consequently requiring a more structured subderivation for the receiver), but even had we done so the information contained in this subderivation would have been 'thrown away' by the derivation substitution operation during the reduction step, since the occurrence of the variable `this` in the method body is still covered by ω (i.e. any information about `this` in the environment Π_2 is not used).

The derivation $\mathcal{D}_2$ is now in *normal form* since although the expression that it types still contains a redex, that redex is covered by ω and so no further (derivation) reduction can take place there. The structure of this derivation therefore dictates the structure of an approximant of `e`: the approximant is formed by replacing all sub-expressions typed with ω by the element $\bot$. When we do this, we obtain the derivation $\mathcal{D}_3$ as given in the figure.

Although this example is relatively simple (we chose the derivation corresponding to the simplest non-trivial approximant), it does demonstrate the central concepts involved in the approximation theorem.

6.6. Comparison with Nominal Typing

To give a more intuitive understanding of both the differences and advantages of our approach over the conventional nominal approach to object-oriented static analysis (as exemplified in Featherweight Java), we will first define the nominal type system for FJ$^{¢}$, and then discuss some examples which illustrate the main issues.

Our nominal type system is almost exactly the same as the system presented in [66], except that it will exclude casts. It is defined as follows.

Definition 6.12 (Member type lookup)**.** *The lookup functions $\mathcal{FT}$ and $\mathcal{MT}$ return the class type declaration for a given field or methods of a given class. They are defined by:*

$$\mathcal{FT}(C,f) = \begin{cases} D & \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \ \{\overline{fd}\ \overline{md}\} \\ & \quad \& \ D\ f \in \overline{fd} \\ \mathcal{FT}(C',f) & \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \ \{\overline{fd}\ \overline{md}\} \\ & \quad \& \ D\ f \notin \overline{fd} \end{cases}$$

$$\mathcal{MT}(C,m) \;=\; \begin{cases} \vec{C}_n \to D & \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \ \{\overrightarrow{fd}\ \overrightarrow{md}\} \\ & \quad \& \ D\ m(\overrightarrow{C\ x})\{e\} \in \overrightarrow{md} \\ \mathcal{MT}(C',m) & \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \ \{\overrightarrow{fd}\ \overrightarrow{md}\} \\ & \quad \& \ D\ m(\overrightarrow{C\ x}_n)\{e\} \notin \overrightarrow{md} \end{cases}$$

Nominal type assignment in FJ$^{\cent}$ is a relatively easy affair, and more or less guided by the class hierarchy.

Definition 6.13 (Nominal Subtyping)**.** *The sub-typing relation* <: *on class types is generated by the* `extends` *construct in the language* FJ$^{\cent}$*, and is defined as the smallest pre-order satisfying:*

$$\texttt{class } C \texttt{ extends } D\ \{\overrightarrow{fd}\ \ \overrightarrow{md}\ \} \in \mathcal{CT} \Rightarrow C <: D$$

Notice that this relation depends on the class table, so the symbol $\lhd$ should be indexed by $\mathcal{CT}$; however, in keeping with the convention mentioned previously in Chapter 3, we leave this implicit.

Definition 6.14 (Nominal type assignment for FJ$^{\cent}$)**.**

1. *The* nominal *type assignment relation* $\vdash_{\mathcal{N}}$ *is defined by the following natural deduction system:*

$$(\texttt{var}):\ \frac{}{\Pi, x{:}C \vdash_{\mathcal{N}} x : C} \qquad (\texttt{fld}):\ \frac{\Pi \vdash_{\mathcal{N}} e : D}{\Pi \vdash_{\mathcal{N}} e.f : C}\ (\mathcal{FT}(D,f) = C)$$

$$(\texttt{sub}):\ \frac{\Pi \vdash_{\mathcal{N}} e : D}{\Pi \vdash_{\mathcal{N}} e : C}\ (D <: C) \qquad (\texttt{invk}):\ \frac{\Pi \vdash_{\mathcal{N}} e : E \quad \Pi \vdash_{\mathcal{N}} e_i : C_i\ (\forall i \in \overline{n})}{\Pi \vdash_{\mathcal{N}} e.m(\vec{e}_n) : D}\ (\mathcal{MT}(E,m) = \vec{C}_n \to D)$$

$$(\texttt{new}):\ \frac{\Pi \vdash_{\mathcal{N}} e_i : C_i \quad (\forall i \in \overline{n})}{\Pi \vdash_{\mathcal{N}} \texttt{new } D(\vec{e}_n) : D}\ (\mathcal{F}(D) = \vec{f}_n \ \&\ \mathcal{FT}(D,f_i) = C_i\ (\forall i \in \overline{n}))$$

2. *A declaration of method* `m` *in the class* `C` *is well typed when the type returned by* $\mathcal{MT}(m,C)$ *determines a type assignment for the method body.*

$$\frac{\overrightarrow{x{:}C}, \texttt{this}{:}D \vdash_{\mathcal{N}} e_b : G}{G\ m(\overrightarrow{C\ x})\ \{\ \texttt{return } e_b\texttt{;}\ \}\ \texttt{OK IN } D}\ (\mathcal{MT}(m,D) = \vec{C} \to G)$$

3. *Classes are well typed when so are all their methods, and a program is well typed when all the classes are themselves well typed, and the executable expression is typeable.*

$$\frac{md_i\ \texttt{OK IN } C\ (\forall i \in \overline{n})}{\texttt{class } C \texttt{ extends } D\ \{\ \overrightarrow{fd}\texttt{;}\ \overrightarrow{md}_n\ \}\ \texttt{OK}} \qquad \frac{\overrightarrow{cd}\ \texttt{OK} \quad \Gamma \vdash_{\mathcal{N}} e : C}{(\overrightarrow{cd}, e)\ \texttt{OK}}$$

Notice that in the nominal system, classes are typed once, and this type *checking* allows for a consistency check on the class type annotations that the programmer has given for each class declaration. Once the program has been verified consistent in this way, the declared types can then be used to type executable expressions. This is in contrast to the approach of our intersection type system which, rather than typing classes, has the two rules (NEWF) and (NEWM) that create a field or method type for an object on demand. In this approach, method bodies are checked *every time* we need that an object has a specific method type, and the various types for a method used throughout a program need not be the same, as is essentially the case for the nominal system.

There are immediate differences between the nominal type system and our intersection type system since the former allows for the typing of non-terminating (unsolvable) programs. Consider the unsolvable expression `new NT().loop()` from Section 6.2, for which $\vdash_\gamma$ `new NT().loop() : NT` can be derived.

Restricting our attention to (head) normalising terms, then, we can see that the intersection type system permits the typing of more programs. Consider the following two classes:

```
class A extends Object {
    A self() { return this; }
    A foo() { return this.self(); }
}

class B extends A {
    A f;
    A foo() { return this.self().f; }
}
```

The class `B` is not well typed according to the nominal type system, since its `foo` method is not well typed: it attempts to access the field `f` on the expression `this.self()` which, according to the declaration of the `self` method, has type `A` and the class `A` has no `f` field.

The intersection type system, on the other hand, can type the expression `new B(new A()).foo()` as shown by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \texttt{new A()} : \texttt{A}}\,(\textsc{obj})}{\vdash \texttt{new B(new A())} : \langle \texttt{f:A} \rangle}\,(\textsc{newF})}{\{\texttt{this}{:}\langle \texttt{f:A} \rangle\} \vdash \texttt{this} : \langle \texttt{f:A} \rangle}\,(\textsc{var})}{\vdash \texttt{new B(new A())} : \langle \texttt{self} : () \to \langle \texttt{f:A} \rangle \rangle}\,(\textsc{newM})}{\{\texttt{this}{:}\langle \texttt{self} : () \to \langle \texttt{f:A} \rangle \rangle\} \vdash \texttt{this} : \langle \texttt{self} : () \to \langle \texttt{f:A} \rangle \rangle}\,(\textsc{var})}{\{\texttt{this}{:}\langle \texttt{self} : () \to \langle \texttt{f:A} \rangle \rangle\} \vdash \texttt{this.self()} : \langle \texttt{f:A} \rangle}\,(\textsc{invk})}{\{\texttt{this}{:}\langle \texttt{self} : () \to \langle \texttt{f:A} \rangle \rangle\} \vdash \texttt{this.self().f} : \texttt{A}}\,(\textsc{fld})}{\vdash \texttt{new B(new A())} : \langle \texttt{foo} : () \to \texttt{A} \rangle}\,(\textsc{newM})}{\vdash \texttt{new B(new A()).foo()} : \texttt{A}}\,(\textsc{invk})$$

The example above might seem rather contrived, but the same essential situation occurs in the ubiquitous `ColourPoint` example which is used as a standard benchmark for object-oriented type systems. Assuming integers and strings, and boolean values and operators for FJ¢, this example can be expressed as follows:

```
class Point extends Object {
    int x;
    int y;
    bool equals(Point p) {
        return (this.x == p.x) && (this.y == p.y);
    }
}
```

```
class ColourPoint extends Point {
    string colour;
    bool equals(Point p) {
        return (this.x == p.x) && (this.y == p.y) &&
            (this.colour == p.colour);
    }
}
```

In this example we have a class `Point` which encodes a cartesian co-ordinate, with integer values for the `x` and `y` positions. The `Point` class also contains a method `equals`, which compares two `Point` instances and indicates if they represent the same co-ordinate. The `ColourPoint` class is an extension of the `Point` class which adds an extra dimension to `Point` objects - a colour. Now, to determine the equality of `ColourPoint` objects, we must check that their colours match in addition to their co-ordinate positions. The nominal system is unable to handle this since when the `equals` method is overridden in the `ColourPoint` class, it must maintain the same type signature as in the `Point` class, i.e. it is constrained to only accept `Point` objects (which do not contain a `colour` field), and not `ColourPoint` objects, as is required for the correct *functional* behaviour. Thus, the `ColourPoint` class is not well typed.

A solution to this problem comes in the form of casts. In order to make the `ColourPoint` class well typed (in the nominal type system), we *cast* the argument `p` of the `equals` method to be a `ColourPoint` object as follows:

```
class ColourPoint extends Point {
    string colour;
    bool equals(Point p) {
        return (this.x == p.x) && (this.y == p.y) &&
            (this.colour == ((ColourPoint) p).colour);
    }
}
```

The cast in the expression `((ColourPoint) p)` tells the type system that it should be considered to be of type `ColourPoint`, and so the access of the `colour` field can be considered well typed. Using a cast, therefore, is comparable to a promise by the programmer that the casted expression will at run time evaluate to an object having the specified class (or a subclass thereof). This is expressed in the type system by the following additional rule:

$$(\texttt{cast}): \frac{\Pi \vdash_{\mathcal{V}} e : C}{\Pi \vdash_{\mathcal{V}} \texttt{(}D\texttt{)}\ e : D}\ (D <: C)$$

For soundness reasons, this now requires doing a run-time check, which is expressed by the following extension to the reduction relation:

$$\texttt{(C) new D(...)} \ \rightarrow\ \texttt{new D(...)} \quad (\text{if } \texttt{D} <: \texttt{C})$$

Once this check has been carried out the cast disappears. As the `ColourPoint` example shows, in a nominal type system, (down) casts are essential for full programming convenience, and to be able to obtain the correct behaviour in overloaded methods.

This new `cast` rule now allows for the `ColourPoint` class above to be well typed, thus giving us that the following executable expressions are typeable:

```
new Point(1,2).equals(new Point(3,4))
new Point(1,2).equals(new ColourPoint(3,4,"red"))
new ColourPoint(1,2,"red")
                    .equals(new ColourPoint(3,4,"blue"))
```

The disadvantage to casts, however, is that they may result in a certain (albeit well-defined) form of 'stuck execution' - a ClassCastException - as happens when executing the following expression:

```
new ColourPoint(1,2,"red").equals(new Point(3,4))
```

Here, execution results in the cast `(ColourPoint) new Point(3,4)` which obviously fails, as `Point` is *not* a subclass of `ColourPoint` (rather, the other way around).

Our intersection type system could, with the appropriate extensions for booleans, integers and strings, perform a precise type analysis on the `ColourPoint` program *without* the need for casts, correctly typing the first three expressions above, and rejecting the fourth as ill-typed. Rather than add such extensions to support this claim we will now present another example which is, in a sense, equivalent to the `ColourPoint` example in that it suffers from the same typing issues, however it is formulated completely within FJ¢.

Our example models a situation involving cars and drivers. We can imagine that the scenario may be arbitrarily complex and that our classes implement all the functionality we need, however for our example we will focus on a single aspect: the action of a driver starting a car. For our purposes, we will assume that a car is started when its driver turns the ignition key and so the classes `Car` and `Driver` contain the following code:

```
class Car {
    Driver driver;
    Car start() { return this.driver.turnIgnition(this); }
}

class Driver {
    Car turnIgnition(Car c) { return c; }
}
```

Since we are working with a featherweight model of the language, we have had to abstract away some detail and are subject to certain restrictions. For instance, the operation of turning the ignition of the car may actually be modelled in a more detailed way, but for our illustration it is sufficient to assume that the act of calling the method itself models the action. Also, since in Featherweight Java we do not have a `void` return type, we return the `Car` object itself from the `start` and `turnIgnition` methods.

Now suppose that we are required to extend our model to include a special type of car - a *police*

car. In our model a police car naturally does all the things that an ordinary car does. In addition it may chase other cars, however in order to do so the police officer driving the car must first report to the headquarters. Thus, only police officers may initiate car chases.

Since we need police cars to behave as ordinary cars in all aspects other than being able to chase other cars, it makes sense to write a `PoliceCar` class that *extends* the `Car` class, and thus inherits all its methods and behaviour. Similarly, we will have to make the `PoliceOfficer` class extend the `Driver` class so that police officers are capable of driving cars (including police cars). Here we run into a problem, however, since the nominal approach to object-orientation imposes some restrictions: namely that when we override method definitions we must use the same type signature (i.e. we are not allowed to specialise the argument or return types), nor are we allowed to specialise the types of fields[3] that are inherited. Thus, we must define our new classes as follows, again as above modelling the extra functionality via methods that simply return the (police) car object involved:

```
class PoliceCar extends Car {
    PoliceCar chaseCar(Car c) {
        return this.driver.reportChase(this);
    }
}

class PoliceOfficer extends Driver {
    PoliceCar reportChase(PoliceCar c) { return c; }
}
```

Before considering typing our extra classes, let us examine their behaviour from a purely operational point of view. As desired, a police car driven by a police officer is able to chase another car (the method invocation results in a *value*, i.e. an object):

```
new PoliceCar(new PoliceOfficer())
                  .chaseCar(new Car(new Driver()))
              →   new PoliceCar(new PoliceOfficer()).driver
                      .reportChase(new PoliceCar(new PoliceOfficer()))
              →   new PoliceOfficer()
                      .reportChase(new PoliceCar(new PoliceOfficer()))
              →   new PoliceCar(new PoliceOfficer())
```

However, if a police car driven by an *ordinary* driver attempts to chase a car we run into trouble:

```
new PoliceCar(new Driver())
```

[3]The full Java language allows fields to be declared in a subclass with the same name as fields that exists in the superclasses, however the semantics of this construction is that a *new* field is created which *hides* the previously declared field; while this serves to mitigate the specific problem we are discussing here, it does introduce its own new problems.

```
        .chaseCar(new Car(new Driver()))
    →   new PoliceCar(new Driver()).driver
            .reportChase(new PoliceCar(new Driver()))
    →   new Driver()
            .reportChase(new PoliceCar(new Driver()))
```

Here, we get *stuck* trying to invoke the `reportChase` method on a `Driver` object since the `Driver` class does not contain such a method. This is the infamous '*message not understood*' error.

The nominal approach to static type analysis is twofold: firstly, to *ensure* that the values assigned to the fields of an object match their declared type; and then secondly, enforce within the bodies of the methods that the fields are used in a way consistent with their declared type. Thus, while it is type safe to allow the `driver` field of a `PoliceCar` object to contain a `PoliceOfficer` (since `PoliceOfficer` is a subtype of `Driver`), trying to invoke the `reportChase` method on the `driver` field in the body of the `chaseCar` method is *not* type safe since such an action is not consistent with the declared type (`Driver`) of the `driver` field. In such a situation, where a method body uses a field inconsistently, the nominal approach is to brand the entire class unsafe and prevent any instances being created. Thus, in Featherweight Java (as in full Java), the *subexpression* `new PoliceCar(new Driver())` is not well-typed, consequently entailing that the full expression

```
new PoliceCar(new Driver()).chaseCar(new Car(new Driver()))
```

is not well-typed.

This leaves us in an uncomfortable position, since we have seen that *some* instances of the `PoliceCar` class (namely, those that have `PoliceOfficer` drivers) are perfectly safe, and thus preventing us from creating any instances at all seems a little heavy-handed. There are two solutions to this problem. The first is to rewrite the `PoliceCar` and `PoliceOfficer` classes so that they do *not* extend the classes `Car` and `Driver`. That way, we are free to declare the `driver` field of the `PolieCar` class to be of type `PoliceOfficer`. However, this would mean having to *reimplement* all the functionality of `Car` and `Driver`. The other solution is to use *casts*: in the body of the `chaseCar` method we cast the `driver`, telling the type system that it is safe to consider the `driver` field to be of type `PoliceOfficer`:

```
class PoliceCar extends Car {
    PoliceCar chaseCar(Car c) {
        return ((PoliceOfficer) this.driver)
                    .reportChase(this);
    }
}
```

Now, the `PoliceCar` class is type safe: we can create instances of it and `PoliceCar` objects with `PoliceOfficer` drivers can chase cars:

```
new PoliceCar(new PoliceOfficer())
```

```
                    .chaseCar(new Car(new Driver()))
→   ((PoliceOfficer)
         new PoliceCar(new PoliceOfficer()).driver)
     .reportChase(new PoliceCar(new PoliceOfficer()))
→   ((PoliceOfficer) new PoliceOfficer())
     .reportChase(new PoliceCar(new PoliceOfficer()))
→   new PoliceOfficer()
     .reportChase(new PoliceCar(new PoliceOfficer()))
→   new PoliceCar(new PoliceOfficer())
```

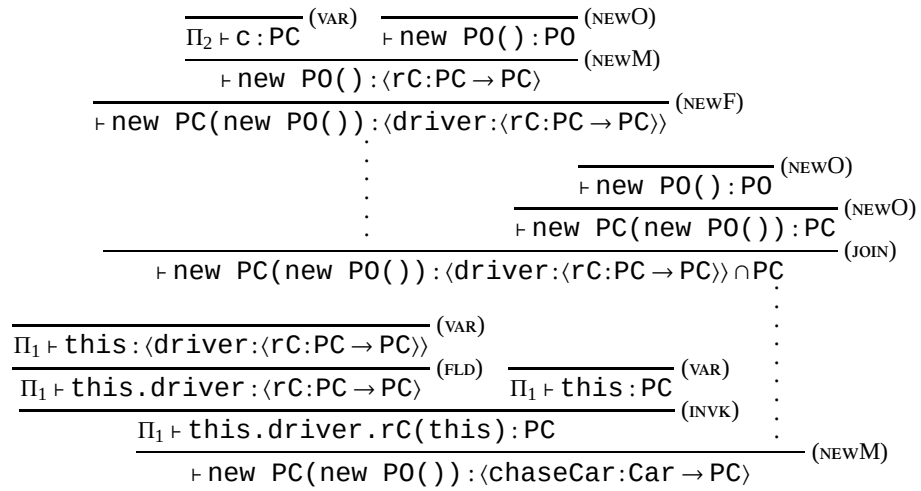
However we are not entirely home and dry, since to regain type soundness in the presence of casts we now have to check at runtime that the cast is valid:

```
new PoliceCar(new Driver()).chaseCar(new Car(new Driver()))
                →   ((PoliceOfficer) new PoliceCar(new Driver()).driver)
                       .reportChase(new PoliceCar(new Driver()))
                →   ((PoliceOfficer) new Driver())
                       .reportChase(new PoliceCar(new Driver()))
```

As the above reduction sequence shows, the 'message not understood' error from before has merely been *transformed* into a runtime 'cast exception' which occurs when we try to cast the `new Driver()` object to a `PoliceOfficer` object. Using the nominal approach to static typing, we are forced to choose the 'lesser of many evils', as it were: being unable to write typeable programs that implement what we desire; being unable to share implementations between classes; or having to allow some runtime exceptions (albeit only with the explicit permission of the programmer). We should point out here that some other solutions to this particular problem have been proposed in the literature (see, for example, the work on family polymorphism [55, 67]), but these solutions persist in the nominal typing approach and can thus only be achieved by extending the language itself.

The FJ$^{¢}$ intersection type system has two main characteristics that distinguish it from the traditional (nominal) type systems for object-orientation. Firstly, our types are structural and so provide a fully functional analysis of the behaviour of objects. We also keep the analysis of methods and fields *independent* from one another, allowing for a fine-grain analysis. This means that not all methods *need* be typeable - we do not reject instances of a class as ill-typed simply because they cannot satisfy *all* of the interface specified by the class (in terms of being able to *safely* - in a semantic sense - invoke all the methods). In other words, if we cannot assign a type to any particular method body from a given class, then this does not prevent us from creating instances of the class if other methods may be safely invoked and typed. In Figure 6.7 we can see a typing derivation in the intersection type system that assign a type for the `chaseCar` method to a `PoliceCar` object with `PoliceOfficer` driver (for space reasons, we have used some abbreviations: `PO` for `PoliceOfficer`, `PC` for `PoliceCar` and `rC` for `reportChase`).

Now consider replacing the `PoliceOfficer` object in this derivation with a `Driver` object, as we would have to do if we wanted to try and assign this type to a `PoliceCar` object with an 'ordinary'



where $\Pi_1 = \{\texttt{this} : \langle \texttt{driver}:\langle \texttt{rC}:\texttt{PC} \to \texttt{PC}\rangle\rangle \cap \texttt{PC}, \texttt{c} : \texttt{Car}\}$

$\Pi_2 = \{\texttt{this}:\texttt{PO}, \texttt{c}:\texttt{PC}\}$

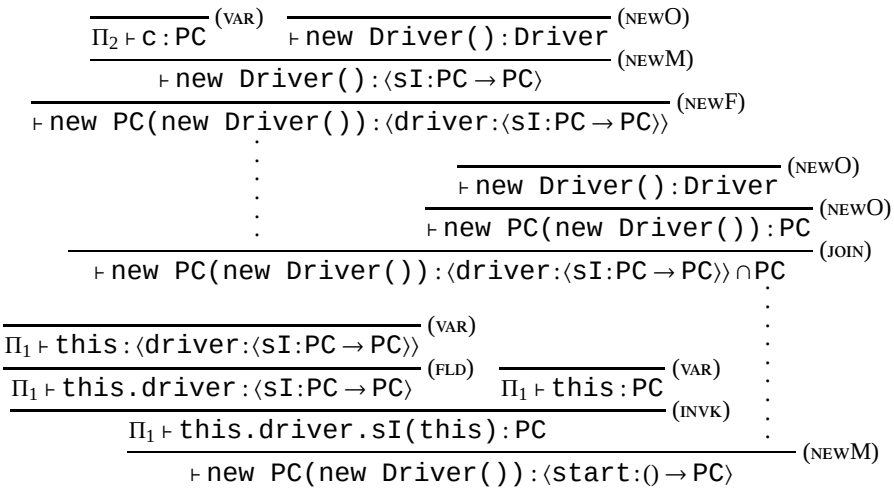
Figure 6.7.: Typing derivation for the `chaseCar` method of a `PoliceCar` object with a `PoliceOfficer` driver.

`Driver` driver. In doing so, we would run into problems since we would ultimately have to assign a type for the `reportChase` method to the driver (as has been done in the topmost subderivation in Figure 6.7) - obviously impossible seeing as no such method exists in the `Driver` class. This does not mean however that we should not be able to create such `PoliceCar` objects. After all, `PoliceCar`s are supposed to behave in all other respects as ordinary cars, so perhaps we might want ordinary `Driver`s to be able to use them as such. In Figure 6.8 we can see a typing derivation assigning a type for the `start` method to a `PoliceCar` object with a `Driver` driver, showing that this is indeed possible. Notice that this is also sound from an operational point of view too:

```
new PoliceCar(new Driver()).start()
  →  new PoliceCar(new Driver()).driver
        .turnIgnition(new PoliceCar(new Driver()))
  →  new Driver()
        .turnIgnition(new PoliceCar(new Driver()))
  →  new PoliceCar(new Driver())
```

The second characteristic is that our type system is a true type *inference* system. That is, no type annotations are required in the program itself in order for the type system to verify its correctness[4]. In the type *checking* approach, the programmer specifies the type that their program must satisfy. As our example shows, this can sometimes lead to inflexibility: in some cases, multiple types may exist for a given program (as in a system without finitely representable principal types) and then the programmer is forced to choose just one of them; in the worst case, a suitable type may not even be expressible in

[4]It is true that FJ¢ retains class type annotations, however this is a syntactic legacy due to the fact that we would like our calculus to be considered a true sibling of Featherweight Java, and nominal class type no longer constitute true types in our system.



where

$\Pi_1 = \{\texttt{this} : \langle \texttt{driver}:\langle \texttt{sI}:\texttt{PC} \to \texttt{PC}\rangle\rangle \cap \texttt{PC}\}$

$\Pi_2 = \{\texttt{this} : \texttt{Driver}, \texttt{c}:\texttt{PC}\}$

Figure 6.8.: Typing derivation for the `start` method of a `PoliceCar` object with a `Driver` driver.

the language. This is the case for our nominally typed cars example: the same `PoliceCar` class may give rise to objects which behave differently depending on the particular values assigned to their fields; this should be expressed through multiple different typings, however in the nominal system there is no way to express them. Our system does not force the programmer to choose a type for the program, thus retaining flexibility. Moreover, since our system is semantically complete, all safe behaviour is typeable and so it provides the *maximum* flexibility possible. Lastly, we have achieved this result without having to extend the programming language in any way.

The combination of the characteristics that we have described above constitutes a subtle shift in the philosophy of static analysis for class-based oo. In the traditional approach, the programmer specifies the class types that each input to the program (i.e. field values and method arguments) should have, on the understanding that the type *checking* system will guarantee that the inputs do indeed have these types. Since a class type represents the entire interface defined in the class declaration, the programmer acts on the assumption that they may safely call any method within this interface. Consequently, to keep up their end of the 'bargain', the programmer is under an obligation to ensure that the value returned by their program safely provides the *whole* interface of its declared type. In the approach suggested by our type system, by firstly removing the requirement to safely implement a full collection of methods regardless of the input values, the programmer is afforded a certain expressive freedom. Secondly, while they can no longer rely on the fact that all objects of a given class provide a particular interface, this apparent problem is obviated by type *inference*, which presents the programmer with an 'if-then' input-output analysis of class constructors and method calls. If a programmer wishes to create instances of some particular class (perhaps from a third party) and call its methods in order to utilise some given functionality, then it is up to them to ensure that they pass appropriate inputs (either field values or method arguments) that guarantee the behaviour they require.

7. Type Inference

In this chapter, we will consider a type inference procedure for the system that we defined in Chapter 3, or rather we will define a type inference algorithm for a restricted version of that system. Since the full intersection type system can characterise strongly normalising expressions it is, naturally, undecidable. Thus, to obtain a terminating type inference algorithm we must restrict the system in some way, accepting that not all (strongly) normalising expressions will be typeable. The key property that any such restriction should exhibit, however, is *soundness* with respect to the full system. In other words, if we assign some typing to an expression in the restricted system, then we can also assign that typing to the expression in the full system. Such a soundness property will allow the restricted system to inherit all the semantic results of the full system. Namely, typeability will still guarantee (strong) normalisation, and imply the existence of similarly typeable approximants meaning that restricted type assignment still describes the functional properties of expressions.

In the context of the λ-calculus type inference algorithms for intersection type assignment have mainly focused on restricting the full system using on a notion of *rank*, essentially placing a limit on how deeply intersections can be nested within any given type. Two notable exceptions are [94], which gives a semi-algorithm for type inference in the full system, and [43] which defines a restriction based on relevance rather than rank. Van Bakel gave a type inference algorithm for a rank-2 restriction [8], and later Kfoury and Wells showed that any *finite* rank restriction is decidable [74].

We can define a similar notion of rank for our intersection types. However, unlike for λ-calculus, every finite-rank restriction of our system is only *semi*-decidable. We will begin by defining the most restricted type assignment system in this family, the rank-0 system which essentially corresponds to Curry's type assignment system. We will then explain why the type inference algorithm for this system only terminates for *some* programs. Since all such systems will suffer from the same semi-decidability problem, we opt not to define further, more expressive, restrictions, but instead we decide to modify our system in a different way – by adding *recursive* types. This work forms the second part of this thesis, and we will motivate it further at the end of this chapter.

7.1. A Restricted Type Assignment System

Our first task will be to define a restricted version of our full intersection type assignment system. As mentioned in the introduction to this chapter, we will be defining a system that is essentially equivalent to Curry's system of simple types for the λ-calculus. Thus, while we retain the structural nature of types (i.e. we have class names, field and method types), we will not allow any intersections. As we will show later, even this very severe restriction of the system is only semi-decidable. More specifically, the algorithm that we will derive for this system only terminates when running on *non-recursive* programs, a property of programs that we will formally define later, but which intuitively expresses that no method creates a new instance of the class to which it belongs.

Definition 7.1 (Simple Types)**.** Simple *types are* FJ$^{¢}$ *types without intersections or* ω*. They are defined by the following grammar:*

$$\sigma,\tau \quad ::= \quad C \quad | \quad \varphi \quad | \quad \langle f{:}\sigma\rangle \quad | \quad \langle m{:}(\vec{\sigma}_n) \to \tau\rangle$$

Note that previously, we have used the metavariable σ referred to *strict* predicates (possibly containing intersections and ω), however in this chapter, we will use it to refer to simple types only. Notice that the set of simple types is a subset of the set of strict types. This fact will be used when showing the soundness of the restricted type assignment with respect to the full type assignment system.

Definition 7.2 (Simple Type Environments)**.** *1. A* simple type statement *is of the form* $\ell{:}\sigma$ *where* ℓ *is either a field name* f *or a variable* x *(and called the* subject *of the statement), and* σ *is a simple type.*

2. A simple type environment Γ *is a finite set of simple type statements in which the subjects are all* unique*. We may refer to simple type environments as just type environments.*

3. If there is a statement $\ell{:}\sigma \in \Gamma$ *then, in an abuse of notation, we write* $\ell \in \Gamma$*. In a further abuse of notation, we may write* $\Gamma(\ell) = \sigma$*.*

4. We relate simple type environments to intersection type environments by extending the subtyping relation $\leqslant$ *(Definition 3.5) as follows:*

$$\Pi \leqslant \Gamma \Leftrightarrow \forall x{:}\sigma \in \Gamma\ [\exists\, \phi \leqslant \sigma\ [x{:}\phi \in \Pi]] \,\&\, \forall f{:}\sigma \in \Gamma\ [\exists\, \phi \leqslant \sigma\ [\mathtt{this}{:}\phi \in \Pi]] \,\&\, \mathtt{this}{:}\sigma \in \Gamma \Rightarrow \exists\, \phi \leqslant \sigma\ [\mathtt{this}{:}\phi \in \Pi]$$

The following defines a function that returns the set of type variables used in a simple type or type environment.

Definition 7.3 (Type Variable Extraction)**.** *1. The function* TV *returns the set of type variables occurring in a simple type. It is defined as follows:*

$$\begin{aligned} \mathrm{TV}(C) &= \emptyset \\ \mathrm{TV}(\varphi) &= \{\varphi\} \\ \mathrm{TV}(\langle f{:}\sigma\rangle) &= \mathrm{TV}(\sigma) \\ \mathrm{TV}(\langle m{:}(\vec{\sigma}_n) \to \sigma\rangle) &= \mathrm{TV}(\sigma) \cup \mathrm{TV}(\sigma_1) \cup \ldots \cup \mathrm{TV}(\sigma_n) \end{aligned}$$

2. TV *is extended to simple type environments as follows:*

$$\mathrm{TV}(\Gamma) \quad = \quad (\bigcup_{x:\sigma\in\Gamma} \mathrm{TV}(\sigma)) \cup (\bigcup_{f:\sigma\in\Gamma} \mathrm{TV}(\sigma))$$

Definition 7.4 (Simple Type Assignment)**.** *Simple type assignment* $\vdash_{\mathrm{s}}$ *is a relation on simple type environments and simple type statements. It is defined by the natural deduction system given in Figure 7.1.*

As we mentioned in the introduction to this chapter, a crucial property of our restricted type assignment system is that it is sound with respect to the full intersection type assignment system.

$$(\textsc{var}):\ \frac{}{\Gamma, x{:}\sigma \vdash_{\mathsf{s}} x:\sigma}\ (x \neq \mathtt{this}) \qquad (\textsc{self-obj}):\ \frac{}{\Gamma, \mathtt{this}{:}C \vdash_{\mathsf{s}} \mathtt{this}:C}$$

$$(\textsc{self-fld}):\ \frac{}{\Gamma, \mathtt{this}{:}C, f{:}\sigma \vdash_{\mathsf{s}} \mathtt{this}:\langle f:\sigma\rangle}\ (f \in \mathcal{F}(C)) \qquad (\textsc{fld}):\ \frac{\Gamma \vdash_{\mathsf{s}} e:\langle f:\sigma\rangle}{\Gamma \vdash_{\mathsf{s}} e.f:\sigma}$$

$$(\textsc{invk}):\ \frac{\Gamma \vdash_{\mathsf{s}} e:\langle m:(\vec{\sigma}_n)\to\sigma\rangle \quad \Gamma \vdash_{\mathsf{s}} e_1:\sigma_1 \quad \ldots \quad \Gamma \vdash_{\mathsf{s}} e_n:\sigma_n}{\Gamma \vdash_{\mathsf{s}} e.m(\vec{e}_n):\sigma}$$

$$(\textsc{newObj}):\ \frac{\Gamma \vdash_{\mathsf{s}} e_1:\sigma_1 \quad \ldots \quad \Gamma \vdash_{\mathsf{s}} e_n:\sigma_n}{\Gamma \vdash_{\mathsf{s}} \mathtt{new}\ C(\vec{e}_n):C}\ (\mathcal{F}(C) = \vec{f}_n)$$

$$(\textsc{newF}):\ \frac{\Gamma \vdash_{\mathsf{s}} e_1:\sigma_1 \quad \ldots \quad \Gamma \vdash_{\mathsf{s}} e_n:\sigma_n}{\Gamma \vdash_{\mathsf{s}} \mathtt{new}\ C(\vec{e}_n):\langle f_i:\sigma_i\rangle}\ (\mathcal{F}(C) = \vec{f}_n, i \in \overline{n})$$

$$(\textsc{newM}):\ \frac{\{f_1{:}\sigma'_1, \ldots, f_{n'}{:}\sigma'_{n'}, \mathtt{this}{:}C, x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\} \vdash_{\mathsf{s}} e_{\mathsf{b}}:\sigma \quad \Gamma \vdash_{\mathsf{s}} e_i:\sigma'_i \quad (\forall\, i \in \overline{n'})}{\Gamma \vdash_{\mathsf{s}} \mathtt{new}\ C(\vec{e}_{n'}):\langle m:(\vec{\sigma}_n)\to\sigma\rangle}\ (\mathcal{F}(C) = \vec{f}_{n'}, \mathcal{M}b(C,m) = (\vec{x}_n, e_{\mathsf{b}}))$$

Figure 7.1.: Simple Type Assignment for FJ$^{\mathfrak{C}}$

Theorem 7.5 (Soundness of Simple Predicate Assignment)**.** *If $\Gamma \vdash_{\mathsf{s}} e:\sigma$, then there exists a strong derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash e:\sigma$, where Π is the smallest intersection type environment satisfying $\Pi \trianglelefteq \Gamma$.*

Proof. By induction on the structure of simple type assignment derivations. The only interesting case is for the (NEWM) rule. Then $\Gamma \vdash_{\mathsf{s}} \mathtt{new}\ D(\vec{e}_n):\langle m:(\vec{\tau'}_{n'})\to\tau\rangle$ and $\Gamma \vdash_{\mathsf{s}} e_i:\tau_i$ for each $i \in \overline{m}$ with $\mathcal{F}(D) = \vec{f'}_m$, $\mathcal{M}b(D,m) = (\vec{x'}_{m'}, e_0)$ and, moreover, $\{\mathtt{this}{:}D, f'_1{:}\tau_1, \ldots, f'_m{:}\tau'_m, x'_1{:}\tau'_1, \ldots, x'_{m'}{:}\tau'_{m'}\} \vdash_{\mathsf{s}} e_0:\tau$. Thus, by induction we have $\mathcal{D}_i :: \Pi \vdash e_i:\tau_i$ with $\mathcal{D}_i$ strong for each $i \in \overline{m}$, and we also have that $\mathcal{D}_0 :: \Pi' \vdash e_0:\tau$ with $\mathcal{D}_0$ strong where $\Pi' = \{\mathtt{this}{:}D \cap \langle f'_1:\tau_1\rangle \cap \ldots \cap \langle f'_m:\tau_m\rangle, x'_1{:}\tau'_1, \ldots, x'_{m'}{:}\tau'_{m'}\}$. Notice that then, by the (OBJ) rule of the full intersection type assignment system, it follows that $\langle \vec{\mathcal{D}}_m, \textsc{obj}\rangle :: \Pi \vdash \mathtt{new}\ D(\vec{e}_n):D$ is a strong derivation, and also by the (NEWF) rule of the full intersection type system we have that $\langle \vec{\mathcal{D}}_m, \textsc{newF}\rangle :: \Pi \vdash \mathtt{new}\ D(\vec{e}_n):\langle f'_i:\tau_i\rangle$ is a strong derivation for each $i \in \overline{m}$. Thus, by the (JOIN) rule it follows that there is a strong derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash \mathtt{new}\ D(\vec{e}_n):D \cap \langle f'_1:\tau_1\rangle \cap \ldots \cap \langle f'_m:\tau_m\rangle$. Then finally, by (NEWM) of the full intersection type system it follows that $\langle \mathcal{D}_0, \mathcal{D}, \textsc{newM}\rangle :: \Pi \vdash \mathtt{new}\ D(\vec{e}_n):\langle m:(\vec{\tau'}_{m'})\to\tau\rangle$ is a strong derivation. □

Because simple type assignment is sound with respect to the full intersection type assignment system, we obtain a strong normalisation guarantee 'for free'.

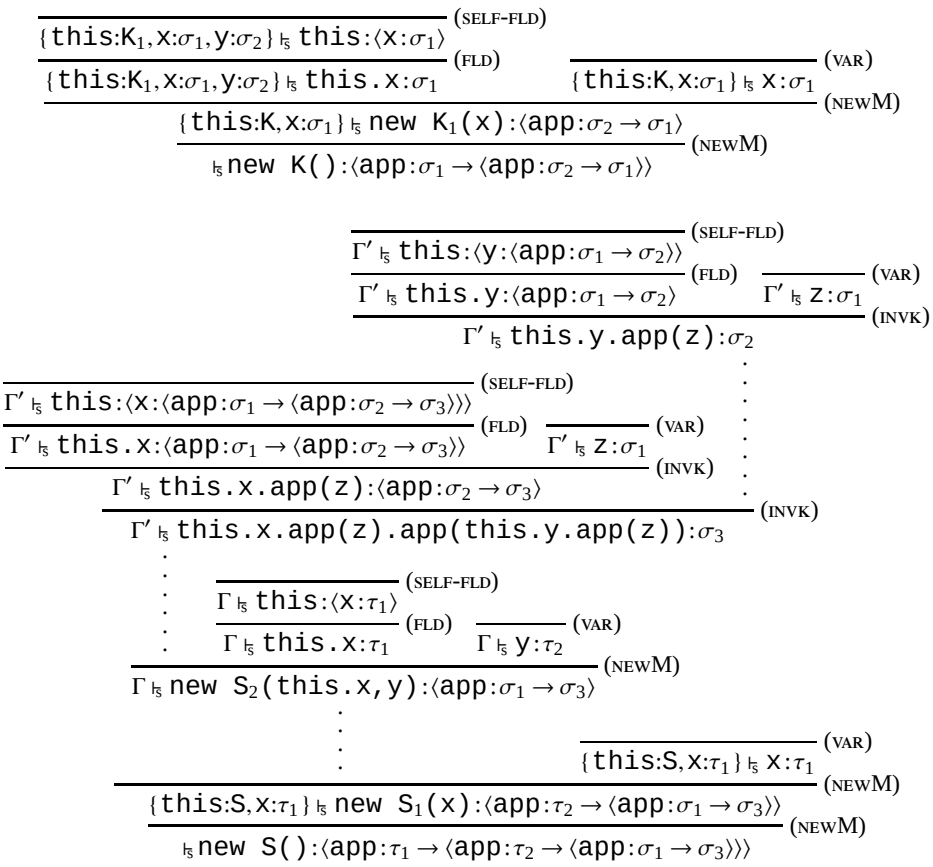
Corollary 7.6. *If $\Gamma \vdash_{\mathsf{s}} e:\sigma$ then e is strongly normalising.*

Proof. By Theorems 7.5 and 5.20. □

We can also prove a weakening lemma for this system, which we will need in order to show soundness of principal typings. Notice that we do not need a notion of subtyping for simple types, and so weakening in this context is simply widening.

Lemma 7.7 (Widening)**.** *Let Γ, Γ' be simple type environments such that $\Gamma \subseteq \Gamma'$; if $\Gamma \vdash_{\mathsf{s}} e:\sigma$, then $\Gamma' \vdash_{\mathsf{s}} e:\sigma$.*

Proof. By easy induction on the structure of simple type derivations. □



where $\tau_1 = \langle \mathtt{app}{:}\sigma_1 \to \langle \mathtt{app}{:}\sigma_2 \to \sigma_3\rangle\rangle, \tau_2 = \langle \mathtt{app}{:}\sigma_1 \to \sigma_2\rangle$,
$\Gamma = \{\mathtt{this}{:}\mathsf{S}_1, \mathtt{x}{:}\tau_1, \mathtt{y}{:}\tau_2\}$ and $\Gamma' = \{\mathtt{this}{:}\mathsf{S}_2, \mathtt{x}{:}\tau_1, \mathtt{y}{:}\tau_2, \mathtt{z}{:}\sigma_1\}$.

Figure 7.2.: Simple Type Assignment Derivation Schemes for the OOCL Translations of **S** and **K**

The simple type assignment system is expressive enough to type OOCL, the encoding of Combinatory Logic into FJ¢ that we gave in Section 6.5. Figure 7.2 gives simple type assignment derivation schemes assigning the principal Curry types of **S** and **K** to their OOCL translations.

7.2. Substitution and Unification

In this section we will define a notion of substitution on simple types, which is sound with respect to the type assignment system. We will also define an extension of Robinson's unification algorithm which we will use to unify simple types. These two operations will be central to showing the principal typings property for the system.

Definition 7.8 (Simple Type Substitutions)**.** *1. A* simple type substitution *s is a particular kind of operation on simple types, which replaces type variables by simple types. Formally, substitutions are mappings (total functions) from simple types to simple types satisfying the following criteria:*

a) the variable domain *(or simply the* domain*),* $\mathsf{dom}(s) \triangleq \{\varphi \mid s(\varphi) \neq \varphi\}$*, is finite;*

b) $s(\mathtt{C}) = \mathtt{C}$ *for all* $\mathtt{C}$*;*

c) $s(\langle \mathtt{f}{:}\sigma'\rangle) = \langle \mathtt{f}{:}s(\sigma')\rangle$*; and*

d) $s(\langle m : (\vec{\sigma}_n) \to \sigma' \rangle) = \langle m : (s(\sigma_1), \ldots, s(\sigma_n)) \to s(\sigma') \rangle$.

2. *The operation of substitution is extended to type environments by* $s(\Pi) = \{\ell : s(\sigma) \mid \ell : \sigma \in \Pi\}$.

3. *The notation* $[\varphi \mapsto \sigma]$ *stands for the substitution* s *with* $\mathsf{dom}(s) = \{\varphi\}$ *such that* $s(\varphi) = \sigma$.

4. Id *denotes the identity substitution, i.e.* $\mathsf{dom}(\mathsf{Id}) = \emptyset$.

5. *If* s_1 *and* s_2 *are simple type substitutions such that* $\mathsf{dom}(s_1) = \mathsf{dom}(s_2)$ *and* $s_1(\varphi) = s_2(\varphi)$ *for each* φ *in their shared domain, then we write* $s_1 = s_2$.

6. *When* $\mathsf{dom}(s) \cap \mathrm{TV}(\sigma) = \mathsf{dom}(s) \cap \mathrm{TV}(\Gamma) = \emptyset$, *then we say that* $\mathsf{dom}(s)$ *is* distinct *from* σ *and* Γ. *Notice that, in this case,* $s(\sigma) = \sigma$ *and* $s(\Gamma) = \Gamma$.

It is straightforward to show that the composition of two simple type substitutions is itself a simple type substitution.

Lemma 7.9 (Substitution Composition)**.** *If* s_1 *and* s_2 *are substitutions, then so is the composition* $s_2 \circ s_1$.

Proof. Using Definition 7.8 for each of s_1 and s_2.

1. The domain of $s_2 \circ s_1$ is finite, since $\mathsf{dom}(s_2 \circ s_1) \subseteq \mathsf{dom}(s_2) \cup \mathsf{dom}(s_1)$: take any type variable φ and suppose $\varphi \in \mathsf{dom}(s_2 \circ s_1)$, then either $\varphi \in \mathsf{dom}(s_1)$ or $\varphi \in \mathsf{dom}(s_2)$ otherwise $s_2 \circ s_1(\varphi) = s_2(s_1(\varphi)) = s_2(\varphi) = \varphi$ and then $\varphi \notin \mathsf{dom}(s_2 \circ s_1)$.

2. $s_2 \circ s_1(C) = s_2(s_1(C)) = s_2(C) = C$.

3. $s_2 \circ s_1(\langle f : \sigma \rangle) = s_2(s_1(\langle f : \sigma \rangle)) = s_2(\langle f : s_1(\sigma) \rangle) = \langle f : s_2(s_1(\sigma)) \rangle = \langle f : s_2 \circ s_1(\sigma) \rangle$.

4. $s_2 \circ s_1(\langle m : (\vec{\sigma}_n) \to \sigma' \rangle) = s_2(s_1(\langle m : (\vec{\sigma}_n) \to \sigma' \rangle))$ □
$$\begin{aligned} &= s_2(\langle m : (s_1(\sigma_1), \ldots, s_1(\sigma_n)) \to s_1(\sigma') \rangle) \\ &= \langle m : (s_2(s_1(\sigma_1)), \ldots, s_2(s_1(\sigma_n))) \to s_2(s_1(\sigma')) \rangle \\ &= \langle m : (s_2 \circ s_1(\sigma_1), \ldots, s_2 \circ s_1(\sigma_n)) \to s_2 \circ s_1(\sigma') \rangle \end{aligned}$$

A key result in the principal typings result is that substitution is a sound operation with respect to simple type assignment.

Theorem 7.10 (Soundness of Substitution)**.** *For all substitutions* s, *if* $\Gamma \vdash_{\mathsf{s}} e : \sigma$ *then* $s(\Gamma) \vdash_{\mathsf{s}} e : s(\sigma)$.

Proof. By straightforward induction on the structure of derivations. □

Using the notion of simple type substitution defined above, we will now define a procedure which will unify two simple types. This will be a central element of the definition of principal typings.

Definition 7.11 (Unification Problems)**.** 1. *A* unification problem *u is a (possibly empty)* sequence *of pairs of simple types* (σ, σ').

2. *Substitutions are extended to unification problems as follows:*

$$s(\epsilon) = \epsilon$$
$$s((\sigma, \sigma') \cdot u) = (s(\sigma), s(\sigma')) \cdot s(u)$$

Definition 7.12 (Unifiers)**.** *1. If s is a substitution such that $s(\sigma) = s(\sigma')$, then we say that s is a* unifier *of* (σ, σ').

2. If s is a unifier of each pair $(\sigma, \sigma') \in u$, then we say that s is a unifier of u.

3. A unifier s of u is most general *if and only if for every unifier s' of u, there exists a substitution s'' such that $s' = s'' \circ s$.*

Composition of substitutions preserves unifiers:

Property 7.13. *1. If s is a unifier of σ_1 and σ_2, then $s' \circ s$ is a unifier of σ_1 and σ_2, for all s'.*

2. If s is a unifier of u, then $s' \circ s$ is a unifier of u, for all u, s'.

Proof. 1. s is a unifier of σ_1 and σ_2, so $s(\sigma_1) = s(\sigma_2)$. Then $s' \circ s(\sigma_1) = s'(s(\sigma_1)) = s'(s(\sigma_2)) = s' \circ s(\sigma_2)$.

2. Take any pair $(\sigma_1, \sigma_2) \in u$. Since s is a unifier of u it follows by Definition 7.12 that s is a unifier of σ_1 and σ_2. Then, by the first property, so is $s' \circ s$. Since (σ_1, σ_2) was arbitrary, we have that $s' \circ s$ is a unifier of all pairs in u, and thus is a unifier of u. □

We will now define our unification procedure for simple types, which is a straightforward extension of Robinson's algorithm [93] to our system.

Definition 7.14 (Unification Procedure)**.** *1. The procedure* Unify *takes a pair of simple predicates (σ, σ') and returns a (unifying) substitution (when it exists, and is undefined otherwise). It is defined as follows:*

$$
\begin{aligned}
&\mathsf{Unify}(\varphi, \varphi') && = [\varphi \mapsto \varphi'] \\
&\left.\begin{array}{l}\mathsf{Unify}(\varphi, \sigma)\\ \mathsf{Unify}(\sigma, \varphi)\end{array}\right\} && = [\varphi \mapsto \sigma] \quad \textit{if } \sigma \textit{ not a type variable and } \varphi \textit{ does not occur in } \sigma \\
&\mathsf{Unify}(\mathsf{C}, \mathsf{C}) && = \mathsf{Id} \\
&\mathsf{Unify}(\langle \mathsf{f} : \sigma \rangle, \langle \mathsf{f}' : \sigma' \rangle) && = \mathsf{Unify}(\sigma, \sigma') \quad \textit{if } \mathsf{f} = \mathsf{f}' \\
&\mathsf{Unify}(\langle \mathsf{m} : (\vec{\sigma}_n) \to \sigma \rangle, \langle \mathsf{m}' : (\overrightarrow{\sigma'}_n) \to \sigma' \rangle) \\
&\qquad\qquad && = s' \circ s_n \circ \ldots \circ s_1 \quad \textit{if } \mathsf{m} = \mathsf{m}' \textit{ and} \\
&\qquad\qquad && \quad s_i = \mathsf{Unify}(s_{i-1} \circ \ldots \circ s_1(\sigma_i), \\
&\qquad\qquad && \qquad\qquad s_{i-1} \circ \ldots \circ s_1(\sigma'_i)) \textit{ for each } i \in \overline{n} \\
&\qquad\qquad && \quad s' = \mathsf{Unify}(s_n \circ \ldots \circ s_1(\sigma), \\
&\qquad\qquad && \qquad\qquad s_n \circ \ldots \circ s_1(\sigma'))
\end{aligned}
$$

2. The Unify *function is generalised to unification problems as follows:*

$$
\begin{aligned}
&\mathsf{Unify}(\epsilon) && = \mathsf{Id} \\
&\mathsf{Unify}((\sigma, \sigma') \cdot u) && = s_2 \circ s_1 \quad \textit{if} \quad s_1 = \mathsf{Unify}(\sigma, \sigma') \textit{ and} \\
&\qquad\qquad && \qquad\qquad s_2 = \mathsf{Unify}(s_1(u))
\end{aligned}
$$

3. when Unify *is undefined, we say that unification* fails*.*

Notice that the generalisation of unification to unification *problems* is merely a convenience, since for any unification problem $u = (\sigma_1, \sigma'_1), \ldots, (\sigma_n, \sigma'_n)$, we can obtain the same result as $\mathsf{Unify}(u)$ by calling the unification procedure for pairs $\mathsf{Unify}(\langle m{:}(\vec{\sigma}_{n-1}) \to \sigma_n\rangle, \langle m{:}(\vec{\sigma'}_{n-1}) \to \sigma'_n\rangle)$, for any method name m.

In addition to showing that unification is a termianting procedure, Robinson showed that it produces most general unifiers, and that if a unification problem has a unifier, then it has a most general unifier.

Property 7.15 (Robinson [93])**.** Unify *is a terminating procedure; furthermore, if* $\mathsf{Unify}(u) = s$ *then* s *is a most general unifier of* u*, and if* s *is a unifier of* u*, then there exists a substitution* s' *such that* $\mathsf{Unify}(u) = s'$.

Notice that Robinson's results trivially apply to our extended notion of unification given in Definition 7.14 since our simple types are merely function types *decorated* with labels. Removing these labels yields ordinary function types (augmented, of course, with type constants for each class). Thus, for any unification problem involving our simple types, an equivalent problem can be formulated for Robinson's algorithm as given in [93] by simply erasing the field and method annotations.

The following definition allows type environments to be unified.

Definition 7.16 (Type Environment Unification Problems)**.** *Let* $\vec{\Gamma}_n$ *be a sequence of simple type environments and let* u *be a unification problem satisfying the following:*

$$i, j \in \overline{n} \;\&\; i \neq j \;\&\; ((x{:}\sigma_1 \in \Gamma_i \text{ and } x{:}\sigma_2 \in \Gamma_j) \text{ or } (f{:}\sigma_1 \in \Gamma_i \text{ and } f{:}\sigma_2 \in \Gamma_j) \Rightarrow (\sigma_1, \sigma_2) \in u$$

Then u *is called* characteristic *for* $\vec{\Gamma}_n$*. If there is no smaller characteristic unification problem than* u*, then* u *is called* minimal*.*

Notice that minimal characteristic unification problems for any given environment Γ are unique up to reordering of their constituent pairs.

Lemma 7.17 (Unification of Type Environments)**.**

1. *If* u *is characteristic for* $\vec{\Gamma}$ *and* $u \subseteq u'$*, then* u' *is also characteristic for* $\vec{\Gamma}$*.*
2. *Let* u *be a characteristic unification problem for* $\vec{\Gamma}_n$*, and let* s *be a unifier of* u*; then* $\Gamma = s(\Gamma_1) \cup \ldots \cup s(\Gamma_n)$ *is a simple type environment.*
3. *Let* Γ *and* $\vec{\Gamma}_n$ *be a simple type environments and* s *a simple type substitution; if* $s(\Gamma_i) \subseteq \Gamma$ *for each* $i \in \overline{n}$*, then* s *is a unifier of any minimal characteristic unification problem for* $\vec{\Gamma}_n$*.*

Proof. 1. Take statements $x{:}\sigma_1$ and $x{:}\sigma_2$ such that $x{:}\sigma_1 \in \Gamma_i$ and $x{:}\sigma_1 \in \Gamma_j$ for some $i, j \in \overline{n}$ such that $i \neq j$, Then, since u is characteristic for $\vec{\Gamma}$ it follows that $(\sigma_1, \sigma_2) \in u$ and then since $u \subseteq u'$ it also follows that $(\sigma_1, \sigma_2) \in u'$. The case for statements with field name subjects is similar.

2. Γ is clearly finite since each Γ_i is. We must also show that the subject of each statement in Γ is unique. Suppose that this is not the case, and thus there are two statements $\ell{:}\sigma_1$ and $\ell{:}\sigma_2$ in Γ with $\sigma_1 \neq \sigma_2$. Then it must be that there are $i, j \in \overline{n}$ with $i \neq j$, and $\ell{:}\sigma'_1 \in \Gamma_i$ and $\ell{:}\sigma'_2 \in \Gamma_j$ for some σ'_1 and σ'_2, with $s(\sigma'_1) = \sigma_1$ and $s(\sigma'_2) = \sigma_2$. Thus $s(\sigma'_1) \neq s(\sigma'_2)$. However, it must be that $(\sigma'_1, \sigma'_2) \in u$

since u is characteristic for $\vec{\Gamma}_n$, and since s unifies u it follows that $s(\sigma'_1) = s(\sigma'_2)$. Since we have derived a contradiction, it must be that the subject of each statement in Γ is unique.

3. Let u be a minimal characteristic unification problem for $\vec{\Gamma}_n$ and take any $(\sigma, \tau) \in u$. Since u is minimal it must be that $\ell{:}\sigma \in \Gamma_i$ and $\ell{:}\tau \in \Gamma_j$ for some ℓ and $i, j \in \overline{n}$ such that $i \neq j$. By assumption, both $s(\Gamma_i) \subseteq \Gamma$ and $s(\Gamma_j) \subseteq \Gamma$, and so both $\ell{:}s(\sigma) \in \Gamma$ and $\ell{:}s(\tau) \in \Gamma$. Then, since Γ is a type environment the subject of each statement in Γ is unique, and therefore it must be that $s(\sigma) = s(\tau)$. Thus, s unifies all pairs in u. □

7.3. Principal Typings

We will now define a notion of principal typings for the simple type assignment system defined earlier. A typing is a pair consisting of a simple type environment and a simple type. As we will see, a principal typing for our system is actually a *set* of typings. We have chosen to formulate our principal typings in this way not because an object (or expression) may potentially contain many fields and methods (notice that traditional record types can handle this), but because even though the simple types assignable in this system do not contain intersections, a method may in general admit more than one analysis. This is due to the presence of class name constants in the type language. Take OOCL for example, our object-oriented encoding of Combinatory Logic, defined in Section 6.5. The `app` method of the object `new K()` has both of following (principal) type schemes:

$$\vdash_{\mathsf{s}} \texttt{new K()} : \langle \texttt{app} : (\varphi) \to \mathsf{K}_1 \rangle$$
$$\vdash_{\mathsf{s}} \texttt{new K()} : \langle \texttt{app} : (\varphi_1) \to \langle \texttt{app} : (\varphi_2) \to \varphi_1 \rangle \rangle$$

In general then, although we do not admit intersection in the type language, the principal type of an expression is in fact an intersection, even in the simple type assignment system. Thus, in the general case, no single record type is sufficient to capture all of the type information inferrable for an object in our system. Given this situation, the most efficient and straightforward way to deal with the 'principal' typing of an expression is simply to use the set of all the (strict) types in this intersection.

Definition 7.18 (Typings)**.** *1. A* typing *is a pair* $[\Gamma, \sigma]$ *of a simple type environment and a simple type.*

2. The function TV *is extended to operate on typings by:* $\mathrm{TV}([\Gamma, \sigma]) \triangleq \mathrm{TV}(\Gamma) \cup \mathrm{TV}(\sigma)$.

As we mentioned above, the principal type of an expression is a set of such typings. We will model this situation by defining a *relation* between expressions and typings. In this way, an expression may be related to any number of typings, and we will formulate our definition so that the typings that any given expression is related to are *principal* for it, in the sense that any other typing assignable to the expression having the same overall structure as the principal one can be generated from it using substitution and widening. After defining the principal typing relation, we will show that it does indeed capture the notion of 'principality' by proving it sound and complete.

Definition 7.19 (Principal Typings)**.** *$\mathcal{PTS}$ is a relation between expressions and typings. In an abuse of notation, we will write* $[\Gamma, \sigma] \in \mathcal{PTS}(e)$ *whenever* $(e, [\Gamma, \sigma]) \in \mathcal{PTS}$*, and thus* $\mathcal{PTS}(e)$ *denotes the*

set $\{[\Gamma, \sigma] \mid [\Gamma, \sigma] \in \mathcal{PTS}(e)\}$. *It is defined inductively as the smallest relation satisfying the following conditions:*

1. $[\{x{:}\varphi\}, \varphi] \in \mathcal{PTS}(x)$ *for all* $x \neq \texttt{this}$, φ.
2. $[\{\texttt{this}{:}C\}, C] \in \mathcal{PTS}(\texttt{this})$ *for all* C.
3. $[\{\texttt{this}{:}C\}, \langle f{:}\varphi\rangle] \in \mathcal{PTS}(\texttt{this})$ *for all* φ, C *and* $f \in \mathcal{F}(C)$.
4. *If* $[\Gamma, \sigma] \in \mathcal{PTS}(e)$ *and* $s = \mathsf{Unify}(\sigma, \langle f{:}\varphi\rangle)$ *with* $\varphi \notin \textsc{tv}([\Gamma, \sigma])$, *then* $[s(\Gamma), s(\varphi)] \in \mathcal{PTS}(e.f)$.
5. *If* $[\Gamma_i, \sigma_i] \in \mathcal{PTS}(e_i)$ *for all* $0 \leq i \leq n$, *and* $s = \mathsf{Unify}((\sigma_0, \langle m{:}(\vec{\sigma}_n\rangle \to \varphi)) \cdot u)$, *where* u *is a minimal characteristic unification problem for* $\Gamma_0, \ldots, \Gamma_n$ *and* $\varphi \notin \textsc{tv}([\Gamma_i, \sigma_i])$ *for each* $0 \leq i \leq n$, *then* $[s(\Gamma), s(\varphi)] \in \mathcal{PTS}(e_0.m(\vec{e}_n))$, *where* $\Gamma = \Gamma_0 \cup \ldots \cup \Gamma_n$.
6. *If* $[\Gamma_i, \sigma_i] \in \mathcal{PTS}(e_i)$ *for all* $i \in \overline{n}$ *and* $s = \mathsf{Unify}(u)$, *where* u *is a minimal characteristic unification problem for* $\vec{\Gamma}_n$, *then*
   a) $[s(\Gamma), C] \in \mathcal{PTS}(\texttt{new } C(\vec{e}_n))$ *for all* C *such that* $\mathcal{F}(C) = \vec{f}_n$; *and*
   b) $[s(\Gamma), s(\langle f_i{:}\sigma_i\rangle)] \in \mathcal{PTS}(\texttt{new } C(\vec{e}_n))$ *for all* C *such that* $\mathcal{F}(C) = \vec{f}_n$ *and* $i \in \overline{n}$

   where $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$.
7. *For all* C, m, $\vec{f}_n$, e_0, *and* $\vec{x}_{n'}$ *such that* $\mathcal{M}b(C, m) = (\vec{x}_{n'}, e_0)$ *and* $\mathcal{F}(C) = \vec{f}_n$, *if* $[\Gamma_i, \sigma_i] \in \mathcal{PTS}(e_i)$ *for each* $0 \leq i \leq n$ *and* $s = \mathsf{Unify}(u' \cdot u)$ *then:*

$$[s(\Gamma), s(\langle m{:}(\vec{\tau'}_{n'}) \to \sigma_0\rangle)] \in \mathcal{PTS}(\texttt{new } C(\vec{e}_n))$$

   where
   a) u *is a minimal characteristic unification problem for* $\vec{\Gamma}_n$;
   b) $u' = (C, \sigma) \cdot (\tau_1, \sigma_1) \cdot \ldots \cdot (\tau_n, \sigma_n)$ *if* $\texttt{this}{:}\sigma \in \Gamma_0$, $u' = (\tau_1, \sigma_1) \cdot \ldots \cdot (\tau_n, \sigma_n)$ *otherwise; and*
   c) $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$

   with
   a) $\vec{\varphi}$ *a sequence of* $n + n'$ distinct *type variables such that no* φ_j *occurs in any* $\textsc{tv}([\Gamma_i, \sigma_i])$ $(0 \leq i \leq n, 0 < j \leq n + n')$;
   b) $\vec{\tau}_n$ *a sequence of simple types satisfying:*
      i. $f_i{:}\sigma'_i \in \Gamma_0 \Rightarrow \tau_i = \sigma'_i$ *and ii.* $f_i \notin \Gamma_0 \Rightarrow \tau_i = \varphi_i$
      for each $i \in \overline{n}$; *and*
   c) $\vec{\tau'}_{n'}$ *a sequence of simple types satisfying:*
      i. $x_i{:}\sigma'_i \in \Gamma_0 \Rightarrow \tau'_i = \sigma'_i$ *and ii.* $x_i \notin \Gamma_0 \Rightarrow \tau'_i = \varphi_{n+i}$
      for each $i \in \overline{n'}$.

The following lemma states that the environment of a principal typing is 'minimal', that is it does not contain any more statements than is necessary to type the expression for which it is principal.

Lemma 7.20 (Minimality of Principal Environments)**.** *Let* $[\Gamma, \sigma] \in \mathcal{PTS}(e)$; *then* $x \in \Gamma$ *if and only if* $x \in \textsc{vars}(e)$, *and if* $\texttt{this}{:}C \in \Gamma$, *then* $f \in \mathcal{F}(C)$ *for all* $f \in \Gamma$.

Proof. By straightforward induction on the definition of principal typings. □

In order to show that our definition of principal typings above is adequate, we must show soundness and completeness. In other words, that any principal typing of an expression can be assigned to that expression (soundness), and that whenever $\Gamma \vdash_s e : \sigma$, e has a *principal* typing $[\Gamma', \sigma']$ such that there is a substitution s with $s(\sigma') = \sigma$ and $s(\Gamma') \subseteq \Gamma$.

Theorem 7.21 (Soundness of Principal Typings)**.** *If* $[\Gamma, \sigma] \in \mathcal{PTS}(e)$*, then* $\Gamma \vdash_s e : \sigma$.

Proof. By induction on the definition of principal typings.

$[\{x{:}\varphi\}, \varphi] \in \mathcal{PTS}(x)$: where $x \neq \texttt{this}$. Then $\{x{:}\varphi\} \vdash_s x : \varphi$ by rule (VAR).

$[\{\texttt{this}{:}C\}, C] \in \mathcal{PTS}(\texttt{this})$: for some C. Then $\{\texttt{this}{:}C\} \vdash_s \texttt{this} : C$ by rule (SELF-OBJ).

$[\{\texttt{this}{:}C, f{:}\varphi\}, \langle f : \varphi\rangle] \in \mathcal{PTS}(\texttt{this})$: for some C and $f \in \mathcal{F}(C)$. Then by rule (SELF-FLD), $\{\texttt{this}{:}C, f{:}\varphi\} \vdash_s \texttt{this} : \langle f : \varphi\rangle$.

$[s(\Gamma), s(\varphi)] \in \mathcal{PTS}(e.f)$: where $[\Gamma, \sigma] \in \mathcal{PTS}(e)$ and $s = \mathsf{Unify}(\sigma, \langle f : \varphi\rangle)$. By induction $\Gamma \vdash_s e : \sigma$. Then by Theorem 7.10, $s(\Gamma) \vdash_s e : \langle f : s(\varphi)\rangle$ and by rule (FLD), it follows that $s(\Gamma) \vdash_s e.f : s(\varphi)$.

$[s(\Gamma), s(\varphi)] \in \mathcal{PTS}(e_0.m(\vec{e}_n))$: where $[\Gamma_i, \sigma_i] \in \mathcal{PTS}(e_i)$ for each $0 \leq i \leq n$, $s = \mathsf{Unify}((\sigma_0, \langle m : (\vec{\sigma}_n) \to \varphi\rangle) \cdot u)$ with u a minimal characteristic unification problem for $\Gamma_0, \ldots, \Gamma_n$, and $\Gamma = \Gamma_0 \cup \ldots \cup \Gamma_n$. By induction $\Gamma_i \vdash_s e_i : \sigma_i$ for each $0 \leq i \leq n$. Then by Theorem 7.10, $s(\Gamma_i) \vdash_s e_i : s(\sigma_i)$ for each $0 \leq i \leq n$. Since u is characteristic for $\Gamma_0, \ldots, \Gamma_n$, it follows from Lemma 7.17 that so is $(\sigma_0, \langle m : (\vec{\sigma}_n) \to \varphi\rangle) \cdot u$ and thus that $s(\Gamma_0) \cup \ldots \cup s(\Gamma_n) = s(\Gamma)$ is a type environment. Notice that $s(\Gamma_i) \subseteq s(\Gamma)$ for each $0 \leq i \leq n$ and thus by Lemma 7.7, $s(\Gamma) \vdash_s e_i : s(\sigma_i)$ for each $0 \leq i \leq n$. Furthermore, for e_0 this gives $s(\Gamma) \vdash_s e_0 : \langle m : (s(\sigma_1), \ldots, s(\sigma_n)) \to s(\varphi)\rangle$. Then by rule (INVK) we have that $s(\Gamma) \vdash_s e_0.m(\vec{e}_n) : s(\varphi)$.

$[s(\Gamma), C] \in \mathcal{PTS}(\texttt{new } C(\vec{e}_n))$: where $\mathcal{F}(C) = \vec{f}_n$, $[\Gamma_i, \sigma_i] \in \mathcal{PTS}(e_i)$ for each $i \in \overline{n}$, $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$, and $s = \mathsf{Unify}(u)$ with u a minimal characteristic unification problem for $\vec{\Gamma}_n$. By induction we have $\Gamma_i \vdash_s e_i : \sigma_i$ for each $i \in \overline{n}$ and by Theorem 7.10, that $s(\Gamma_i) \vdash_s e_i : s(\sigma_i)$. Since u is characteristic for $\vec{\Gamma}_n$, it follows from Lemma 7.17 that $s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) = s(\Gamma)$ is a type environment. Notice that $s(\Gamma_i) \subseteq s(\Gamma)$ for each $i \in \overline{n}$ and so by Lemma 7.7, $s(\Gamma) \vdash_s e_i : s(\sigma_i)$ for each $i \in \overline{n}$. Then by rule (NEWOBJ), $s(\Gamma) \vdash_s \texttt{new } C(\vec{e}_n) : C$.

$[s(\Gamma), s(\langle f_j : \sigma_j\rangle)] \in \mathcal{PTS}(\texttt{new } C(\vec{e}_n))$: where $\mathcal{F}(C) = \vec{f}_n$, $[\Gamma_i, \sigma_i] \in \mathcal{PTS}(e_i)$ for each $i \in \overline{n}$, $j \in \overline{n}$, $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$, and $s = \mathsf{Unify}(u)$ with u a minimal characteristic unification problem for $\vec{\Gamma}_n$. By induction we have $\Gamma_i \vdash_s e_i : \sigma_i$ for each $i \in \overline{n}$ and by Theorem 7.10, that $s(\Gamma_i) \vdash_s e_i : s(\sigma_i)$. Since u is characteristic for $\vec{\Gamma}_n$, it follows from Lemma 7.17 that $s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) = s(\Gamma)$ is a type environment. Notice that $s(\Gamma_i) \subseteq s(\Gamma)$ for each $i \in \overline{n}$ and so by Lemma 7.7, $s(\Gamma) \vdash_s e_i : s(\sigma_i)$ for each $i \in \overline{n}$. Then by rule (NEWF), $s(\Gamma) \vdash_s \texttt{new } C(\vec{e}_n) : \langle f_j : s(\sigma_j)\rangle$.

$[s(\Gamma), s(\langle m : (\vec{\tau'}_{n'}) \to \sigma_0\rangle)] \in \mathcal{PTS}(\texttt{new } C(\vec{e}_n))$: with $\mathcal{F}(C) = \vec{f}_n$ and $\mathcal{MB}(C, m) = (\vec{x}_{n'}, e_0)$, and where $[\Gamma_i, \sigma_i] \in \mathcal{PTS}(e_i)$ for each $0 \leq i \leq n$, $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$ and $s = \mathsf{Unify}(u' \cdot u)$, with u a minimal characteristic unification problem for $\vec{\Gamma}_n$; see Definition 7.19(7) for the conditions holding of the unification problem u' and the types $\vec{\tau'}_{n'}$.

By induction $\Gamma_0 \vdash_s e_0 : \sigma_0$. Notice that $\mathrm{VARS}(e_0) \subseteq \vec{x}_{n'}$ since we assume all programs to be well

formed. By Lemma 7.20 it follows that $\Gamma_0 \subseteq \{x_1{:}\tau'_1, \ldots, x_{n'}{:}\tau'_{n'}\}$. Then since s is a unifier of u' it follows that if $\texttt{this} \in \Gamma_0$ then $\Gamma_0(\texttt{this}) = C$ and thus that $f \in \mathcal{F}(C)$ for all $f \in \Gamma_0$. Then, it must be that

$$s(\Gamma_0) \subseteq \Gamma' = \{\texttt{this}{:}C, f_1{:}s(\sigma_1), \ldots, f_1{:}s(\sigma_1), x_1{:}s(\tau'_1), \ldots, x_1{:}s(\tau'_1)\}$$

So, by Theorem 7.10 and Lemma 7.7, $\Gamma' \vdash_{\mathrm{s}} e_0 : s(\sigma_0)$.

By induction $\Gamma_i \vdash_{\mathrm{s}} e_i : \sigma_i$ for each $i \in \overline{n}$, and by Theorem 7.10 that $s(\Gamma_i) \vdash_{\mathrm{s}} e_i : s(\sigma_i)$. Since u is characteristic for $\vec{\Gamma}_n$, it follows from Lemma 7.17 that so too is $u' \cdot u$ and thus that $s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) = s(\Gamma)$ is a type environment. Notice that $s(\Gamma_i) \subseteq s(\Gamma)$ for each $i \in \overline{n}$ and thus by Lemma 7.7, $s(\Gamma) \vdash_{\mathrm{s}} e_i : s(\sigma_i)$ for each $i \in \overline{n}$. Then the result follows by rule (NEWM), that is we have $s(\Gamma) \vdash_{\mathrm{s}} \texttt{new}\ C(\vec{e}_n) : \langle m{:}(s(\tau'_1), \ldots, s(\tau'_{n'})) \to s(\sigma_0)\rangle$. □

We now consider the completeness of our definition of principal typings.

Theorem 7.22 (Completeness of Principal Typings)**.** *If $\Gamma \vdash_{\mathrm{s}} e : \sigma$ then there is a typing $[\Gamma', \sigma'] \in \mathcal{PTS}(e)$ and a simple type substitution s such that $s(\Gamma') \subseteq \Gamma$ and $s(\sigma') = \sigma$.*

Proof. By induction on the definition of simple type assignment.

(VAR): Then $\Gamma, x{:}\sigma \vdash_{\mathrm{s}} x : \sigma$ with $x \neq \texttt{this}$. By Definition 7.19, $[\{x{:}\varphi\}, \varphi] \in \mathcal{PTS}(x)$ for all φ, so take any such typing and let $s = [\varphi \mapsto \sigma]$. Then $s(\varphi) = \sigma$ and $s(\{x{:}\varphi\}) = \{x{:}\sigma\} \subseteq \Gamma, x{:}\sigma$.

(SELF-OBJ): Then $\Gamma, \texttt{this}{:}C \vdash_{\mathrm{s}} \texttt{this} : C$. $[\{\texttt{this}{:}C\}, C] \in \mathcal{PTS}(\texttt{this})$ by Definition 7.19, and $s(C) = C$ for any s, and so $s(\{\texttt{this}{:}C\}) = \{\texttt{this}{:}C\} \subseteq \Gamma, \texttt{this}{:}C$.

(SELF-FLD): Then $\Gamma, \texttt{this}{:}C, f{:}\sigma \vdash_{\mathrm{s}} \texttt{this} : \langle f{:}\sigma\rangle$ with $f \in \mathcal{F}(C)$. By Definition 7.19 it follows that, for all φ, $[\{\texttt{this}{:}C, f{:}\varphi\}, \langle f{:}\varphi\rangle] \in \mathcal{PTS}(\texttt{this})$, so take any such typing and let $s = [\varphi \mapsto \sigma]$. Then $s(\langle f{:}\varphi\rangle) = \langle f{:}\sigma\rangle$ and $s(\{\texttt{this}{:}C, f{:}\varphi\}) = \{\texttt{this}{:}C, f{:}\sigma\} \subseteq \Gamma, \texttt{this}{:}C, f{:}\sigma$.

(FLD): Then $\Gamma \vdash_{\mathrm{s}} e.f : \sigma$ with $\Gamma \vdash_{\mathrm{s}} e : \langle f{:}\sigma\rangle$. by induction there is a typing $[\Gamma', \sigma'] \in \mathcal{PTS}(e)$ and a substitution s such that $s(\sigma') = \langle f{:}\sigma\rangle$ and $s(\Gamma') \subseteq \Gamma$. Let φ be a type variable not occurring in σ, Γ or $\mathsf{dom}(s)$. Then $[\varphi \mapsto \sigma] \circ s(\Gamma') = s(\Gamma')$. Notice, also, that $[\varphi \mapsto \sigma] \circ s(\sigma') = \langle f{:}\sigma\rangle$ and $[\varphi \mapsto \sigma] \circ s(\langle f{:}\varphi\rangle) = \langle f{:}\sigma\rangle$. Thus, $[\varphi \mapsto \sigma] \circ s$ unifies $(\sigma', \langle f{:}\varphi\rangle)$. By Property 7.15, then, there are substitutions s' and s'' such that $s' = \mathsf{Unify}(\sigma', \langle f{:}\varphi\rangle)$ and $[\varphi \mapsto \sigma] \circ s = s'' \circ s'$. Now, by Definition 7.19, $[s'(\Gamma'), s'(\varphi)] \in \mathcal{PTS}(e.f)$. Then,

$$s''(s'(\varphi)) = s'' \circ s'(\varphi) = [\varphi \mapsto \sigma] \circ s(\varphi) = \sigma$$

and also

$$s''(s'(\Gamma')) = s'' \circ s'(\Gamma') = [\varphi \mapsto \sigma] \circ s(\Gamma') = s(\Gamma') \subseteq \Gamma$$

(INVK): Then $\Gamma \vdash_{\mathrm{s}} e_0.m(\vec{e}_n) : \sigma$ with $\Gamma \vdash_{\mathrm{s}} e_0 : \langle m{:}(\vec{\sigma}_n) \to \sigma\rangle$ and $\Gamma \vdash_{\mathrm{s}} e_i : \sigma_i$ for each $i \in \overline{n}$. By induction, there are typings $[\Gamma_0, \tau_0], \ldots, [\Gamma_n, \tau_n]$ and substitutions $s_0, \ldots, s_n$ such that $[\Gamma_i, \tau_i] \in \mathcal{PTS}(e_i)$ with $s(\Gamma_i) \subseteq \Gamma$ and $s(\tau_i) = \sigma_i$ for each $0 \leq i \leq n$.

Without loss of generality, we can assume that $[\Gamma, \langle m{:}(\vec{\sigma}_n) \to \sigma\rangle]$ and each $[\Gamma_i, \tau_i]$ $(0 \leq i \leq n)$ do not have any type variables in common with each other; that is, their sets of type variables are all pairwise distinct. We can also assume that, for each $0 \leq i \leq n$, $\mathsf{dom}(s_i) \subseteq \mathrm{TV}([\Gamma_i, \sigma_i])$, since we are

always able to construct substitutions s'_i such that $\mathsf{dom}(s'_i) = \mathsf{dom}(s_i) \cap \mathrm{TV}([\Gamma_i, \tau_i])$ and $s'_i(\varphi) = s_i(\varphi)$ for all $\varphi \in \mathsf{dom}(s'_i)$, and thus $s'_i([\Gamma_i, \tau_i]) = s_i([\Gamma_i, \tau_i])$.

Now, take some type variable φ not occurring in any $[\Gamma_i, \tau_i]$ $(0 \leq i \leq n)$, or in $[\Gamma, \langle m : (\vec{\sigma}_n) \to \sigma \rangle,]$; then let $s = [\varphi \mapsto \sigma] \circ s_n \circ \ldots \circ s_0$. Notice that:

1. $s(\Gamma_i) \subseteq \Gamma$ for each $0 \leq i \leq n$.

   Take any i; remember that $s_i(\Gamma_i) \subseteq \Gamma$. Then since φ does not occur in Γ, it follows that $[\varphi \mapsto \sigma](s_i(\Gamma_i)) = s(\Gamma_i)$. Furthermore, since $\mathsf{dom}(s_j)$ is distinct from Γ for all $0 \leq j \leq n$, it follows that $s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i)) = s_i(\Gamma)$. Thirdly, since $\mathsf{dom}(s_j)$ is distinct from Γ_i for each $0 \leq j \leq n$ such that $i \neq j$, $s_{i-1} \circ \ldots \circ s_0(\Gamma_i) = \Gamma_i$. Thus:

$$\begin{aligned}[\varphi \mapsto \sigma](s_i(\Gamma_i)) &= [\varphi \mapsto \sigma](s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i))) \\ &= [\varphi \mapsto \sigma](s_n \circ \ldots \circ s_{i+1}(s_i(s_{i-1} \circ \ldots \circ s_0(\Gamma_i)))) \\ &= s(\Gamma_i) \subseteq \Gamma\end{aligned}$$

2. s unifies $(\tau_0, \langle m : (\vec{\tau}_n) \to \varphi \rangle)$:

   a) Since $\mathsf{dom}(s_i)$ is distinct from $\langle m : (\vec{\sigma}_n) \to \sigma \rangle$ for all $0 \leq i \leq n$, $s_n \circ \ldots \circ s_1(\langle m : (\vec{\sigma}_n) \to \sigma \rangle) = \langle m : (\vec{\sigma}_n) \to \sigma \rangle$. Also, since φ does not occur in $\langle m : (\vec{\sigma}_n) \to \sigma \rangle$, it follows that

$$\begin{aligned}[\varphi \mapsto \sigma](\langle m : (\vec{\sigma}_n) \to \sigma \rangle) &= [\varphi \mapsto \sigma](s_n \circ \ldots \circ s_1(\langle m : (\vec{\sigma}_n) \to \sigma \rangle)) \\ &= \langle m : (\vec{\sigma}_n) \to \sigma \rangle\end{aligned}$$

   Thus: $\begin{aligned}[t] s(\tau_0) &= [\varphi \mapsto \sigma] \circ s_n \circ \ldots \circ s_0(\tau_0) \\ &= [\varphi \mapsto \sigma] \circ s_n \circ \ldots \circ s_1(\langle m : (\vec{\sigma}_n) \to \sigma \rangle) \\ &= \langle m : (\vec{\sigma}_n) \to \sigma \rangle\end{aligned}$

   b) i. Since φ does not occur in any $[\Gamma_i, \tau_i]$, $s_i(\varphi) = \varphi$ for all $0 \leq i \leq n$ and thus $s_n \circ \ldots \circ s_0(\varphi) = \varphi$; then $s(\varphi) = [\varphi \mapsto \sigma](s_n \circ \ldots \circ s_0(\varphi)) = [\varphi \mapsto \sigma](\varphi) = \sigma$.

   ii. We show that $s(\tau_i) = \sigma_i$ for all $0 \leq i \leq n$. Take any i; since $\mathsf{dom}(s_j)$ is distinct from τ_i for each $0 \leq j \leq n$ such that $i \neq j$, it follows that $s_{i-1} \circ \ldots \circ s_0(\tau_i) = \tau_i$. Also, since for all $0 \leq j \leq n$ $\mathsf{dom}(s_j)$ is distinct from $\langle m : (\vec{\sigma}_n) \to \sigma \rangle$, it is therefore distinct from σ_i; thus $s_n \circ \ldots \circ s_{i+1}(\sigma_i) = \sigma_i$. Lastly, since φ does not occur in any σ_j $(j \in \overline{n})$, $[\varphi \mapsto \sigma](\sigma_i) = [\varphi \mapsto \sigma](s_n \circ \ldots \circ s_{i+1}(\sigma_i)) = \sigma_i$. Then, since $s_i(\tau_i) = \sigma_i$, we have:

$$\begin{aligned}[\varphi \mapsto \sigma](s_n \circ \ldots \circ s_{i+1}(\sigma_i)) &= [\varphi \mapsto \sigma](s_n \circ \ldots \circ s_{i+1}(s_i(\tau_i))) \\ &= [\varphi \mapsto \sigma](s_n \circ \ldots \circ s_{i+1}(s_i(s_{i-1} \circ \ldots \circ s_0(\tau_i)))) \\ &= s(\tau_i) \\ &= \sigma_i\end{aligned}$$

   Thus $\begin{aligned}[t] s(\langle m : (\vec{\tau}_n) \to \varphi \rangle) &= \langle m : (s(\tau_1), \ldots, s(\tau_n)) \to s(\varphi) \rangle \\ &= \langle m : (\vec{\sigma}_n) \to \sigma \rangle\end{aligned}$

Let u be a minimal characteristic unification problem for $\Gamma_0, \ldots, \Gamma_n$. By point 1 above and Lemma

7.17(3), s is a unifier of u. We therefore have that s unifies $((\tau_0, \langle m : (\vec{\tau}_n) \to \varphi \rangle) \cdot u)$ and thus by Property 7.15 there are s' and s'' such that $s' = \mathsf{Unify}((\tau_0, \langle m : (\vec{\tau}_n) \to \varphi \rangle) \cdot u)$ and $s = s'' \circ s'$. By Definition 7.19, $[s'(\Gamma'), s(\varphi)] \in \mathcal{PTS}(e_0.m(\vec{e}_n))$ where $\Gamma' = \Gamma_0 \cup \ldots \cup \Gamma_n$. By point (1) above, it follows that $s(\Gamma_0) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$. Notice then that $s''(s'(\Gamma')) = s'' \circ s'(\Gamma') = s(\Gamma') = s(\Gamma_0) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$ and also by point 2(b)i above that $s''(s'(\varphi)) = s'' \circ s'(\varphi) = s(\varphi) = \sigma$.

(NEWOBJ): Then $\Gamma \vdash_s \mathtt{new}\ C(\vec{e}_n) : C$ for some C such that $\mathcal{F}(C) = \vec{f}_n$ with $\Gamma \vdash_s e_i : \sigma_i$ for each $i \in \overline{n}$. By induction, there are typings $[\Gamma_1, \tau_1], \ldots, [\Gamma_n, \tau_n]$ and simple type substitutions $\vec{s}_n$ such that $[\Gamma_i, \tau_i] \in \mathcal{PTS}(e_i)$ with $s_i(\Gamma_i) \subseteq \Gamma$ and $s_i(\tau_i) = \sigma_i$ for each $i \in \overline{n}$. Without loss of generality, we can assume that $[\Gamma, C]$ and each $[\Gamma_i, \tau_i]$ ($i \in \overline{n}$) do not have any type variables in common with one another, and that $\mathsf{dom}(s_i) \subseteq \mathrm{TV}([\Gamma_i, \tau_i])$.

Let $s = s_n \circ \ldots \circ s_1$. Now, take any Γ_i. Remember that $s_i(\Gamma_i) \subseteq \Gamma$. Since $\mathsf{dom}(s_j)$ is distinct from Γ for each $j \in \overline{n}$, we have $s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i)) = s_i(\Gamma_i)$. Also, since $\mathsf{dom}(s_j)$ is distinct from Γ_i for each $j \in \overline{n}$ such that $i \neq j$, it follows that $s_{i-1} \circ \ldots \circ s_1(\Gamma_i) = \Gamma_i$. Therefore

$$\begin{aligned} s_i(\Gamma_i) &= s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i)) \\ &= s_n \circ \ldots \circ s_{i+1}(s_i(s_{i-1} \circ \ldots \circ s_1(\Gamma_i))) \\ &= s(\Gamma_i) \subseteq \Gamma \end{aligned}$$

And thus $s(\Gamma_i) \subseteq \Gamma$ for each $i \in \overline{n}$. Let u be a minimal characteristic unification problem for $\vec{\Gamma}_n$. By Lemma 7.17(3) it follows that s is a unifier of u and so by Property 7.15 there are s' and s'' such that $s' = \mathsf{Unify}(u)$ and $s = s'' \circ s'$. By Definition 7.19, $[s'(\Gamma'), C] \in \mathcal{PTS}(\mathtt{new}\ C(\vec{e}_n))$ where $\Gamma' = \Gamma_1 \cup \ldots \cup \Gamma_n$. Since $s(\Gamma_i) \subseteq \Gamma$ for each $i \in \overline{n}$, it follows that $s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$, and so $s''(s'(\Gamma')) = s'' \circ s'(\Gamma') = s(\Gamma') = s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$. Also we have that $s''(C) = C$ by definition.

(NEWF): Then $\Gamma \vdash_s \mathtt{new}\ C(\vec{e}_n) : \langle f_j : \sigma_j \rangle$ for some C such that $\mathcal{F}(C) = \vec{f}_n$ with $\Gamma \vdash_s e_i : \sigma_i$ for each $i \in \overline{n}$, and where $j \in \overline{n}$. By induction, there are typings $[\Gamma_1, \tau_1], \ldots, [\Gamma_n, \tau_n]$ and simple type substitutions $\vec{s}_n$ such that $[\Gamma_i, \tau_i] \in \mathcal{PTS}(e_i)$ with $s_i(\Gamma_i) \subseteq \Gamma$ and $s_i(\tau_i) = \sigma_i$ for each $i \in \overline{n}$. Without loss of generality, we can assume that each $[\Gamma_i, \tau_i]$ ($i \in \overline{n}$) do not have any type variables in common with one another, and that each $[\Gamma_i, \tau_i]$ does not share any type variables with Γ and $\vec{\sigma}_n$. We can also assume that $\mathsf{dom}(s_i) \subseteq \mathrm{TV}([\Gamma_i, \tau_i])$.

Let $s = s_n \circ \ldots \circ s_1$. Since $\mathsf{dom}(s_i)$ is distinct from τ_j for each $i \in \overline{n}$ such that $i \neq j$, it follows that $s_{j-1} \circ \ldots \circ s_1(\tau_j) = \tau_j$. Also, since $\mathsf{dom}(s_i)$ is distinct from σ_j for each $i \in \overline{n}$, we have $s_n \circ \ldots \circ s_{j+1}(\sigma_j) = \sigma_j$. Then, since $s_j(\tau_j) = \sigma_j$, it follows that:

$$\begin{aligned} s_j(\tau_j) &= s_n \circ \ldots \circ s_{j+1}(s_j(\tau_j)) \\ &= s_n \circ \ldots \circ s_{j+1}(s_j(s_{j-1} \circ \ldots \circ s_1(\tau_j))) \\ &= s(\tau_j) \\ &= \sigma_j \end{aligned}$$

Next, take any Γ_i. Remember that $s_i(\Gamma_i) \subseteq \Gamma$. Since $\mathsf{dom}(s_j)$ is distinct from Γ for each $j \in \overline{n}$, we have $s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i)) = s_i(\Gamma_i)$. Also, since $\mathsf{dom}(s_j)$ is distinct from Γ_i for each $j \in \overline{n}$ such that

$i \neq j$, it follows that $s_{i-1} \circ \ldots \circ s_1(\Gamma_i) = \Gamma_i$. Therefore

$$\begin{aligned} s_i(\Gamma_i) &= s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i)) \\ &= s_n \circ \ldots \circ s_{i+1}(s_i(s_{i-1} \circ \ldots \circ s_1(\Gamma_i))) \\ &= s(\Gamma_i) \subseteq \Gamma \end{aligned}$$

And thus $s(\Gamma_i) \subseteq \Gamma$ for each $i \in \overline{n}$. Let u be a minimal characteristic unification problem for $\vec{\Gamma}_n$. By Lemma 7.17(3) it follows that s is a unifier of u and so by Property 7.15 there are s' and s'' such that $s' = \mathsf{Unify}(u)$ and $s = s'' \circ s'$. By Definition 7.19, $[s'(\Gamma'), s'(\langle f_j : \tau_j \rangle)] \in \mathcal{PTS}(\texttt{new}\ C(\vec{e}_n))$ where $\Gamma' = \Gamma_1 \cup \ldots \cup \Gamma_n$. Since $s(\Gamma_i) \subseteq \Gamma$ for each $i \in \overline{n}$, it follows that $s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$, and so $s''(s'(\Gamma')) = s'' \circ s'(\Gamma') = s(\Gamma') = s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$. We also have that $s''(s'(\langle f_j : \tau_j \rangle)) = s'' \circ s'(\langle f_j : \tau_j \rangle) = s(\langle f_j : \tau_j \rangle) = \langle f_j : \sigma_j \rangle$.

(NEWM): Then $\Gamma \vdash_s \texttt{new}\ C(\vec{e}_n) : \langle m : (\overrightarrow{\sigma'}_{n'}) \to \sigma_0 \rangle$ with $\Gamma' \vdash_s e_0 : \sigma_0$ and $\Gamma \vdash_s e_i : \sigma_i$ for each $i \in \overline{n}$, where $\mathcal{F}(C) = \vec{f}_n$, $\mathcal{M}b(C, m) = (\vec{x}_{n'}, e_0)$ and $\Gamma' = \{\texttt{this}{:}C, f_1{:}\sigma_1, \ldots, f_n{:}\sigma_n, x_1{:}\sigma'_1, \ldots, x_{n'}{:}\sigma'_{n'}\}$.

By induction there are typings $[\Gamma_1, \tau''_1], \ldots, [\Gamma_n, \tau''_n]$ and simple type substitutions $\vec{s}_n$ such that $[\Gamma_i, \tau''_i] \in \mathcal{PTS}(e_i)$ with $s_i(\Gamma_i) \subseteq \Gamma$ and $s_i(\tau''_i) = \sigma_i$ for each $i \in \overline{n}$, and there is a typing $[\Gamma_0, \tau''_0] \in \mathcal{PTS}(e_0)$ and a substitution s_0 such that $s_0(\Gamma_0) \subseteq \Gamma'$ and $s_0(\tau''_0) = \sigma_0$). Without loss of generality, we can assume that each $[\Gamma_i, \tau''_i]$ $(0 \leq i \leq n)$ do not have any type variables in common with one another, and that each $[\Gamma_i, \tau''_i]$ $(0 \leq i \leq n)$ does not share any type variables with Γ, $\langle m : (\overrightarrow{\sigma'}_{n'}) \to \sigma_0 \rangle$ and $\vec{\sigma}_n$. We can also assume that $\mathsf{dom}(s_i) \subseteq \mathrm{TV}([\Gamma_i, \tau_i])$ for each $0 \leq i \leq n$.

Let $\vec{\varphi}_n$ and $\overrightarrow{\varphi'}_{n'}$ be distinct type variables not occurring in any $[\Gamma_i, \tau''_i]$ $(0 \leq i \leq n)$, or Γ, $\langle m : (\overrightarrow{\sigma'}_{n'}) \to \sigma_0 \rangle$ and $\vec{\sigma}_n$, and let s' be the type substitution such that: i) $\mathsf{dom}(s') = \{\varphi_1, \ldots, \varphi_n, \varphi'_1, \ldots, \varphi'_{n'}\}$; ii) $s'(\varphi_i) = \sigma_i$ for all $i \in \overline{n}$; and iii) $s'(\varphi'_i) = \sigma'_i$ for all $i \in \overline{n'}$. Then let $s = s' \circ s_n \circ \ldots \circ s_0$.

Take any Γ_i $(i \in \overline{n})$; remember that $s_i(\Gamma_i) \subseteq \Gamma$. Since $\mathsf{dom}(s_j)$ is distinct from Γ_i for each $0 \leq j \leq n$ such that $i \neq j$, it follows that $s_{i-1} \circ \ldots s_0(\Gamma_i) = \Gamma_i$. Also, since $\mathsf{dom}(s_j)$ is distinct from Γ for each $0 \leq j \leq n$, and no φ_k $(k \in \overline{n})$ or φ'_k $(k \in \overline{n'})$ occurs in Γ, it then follows that $s' \circ s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i)) = s_i(\Gamma_i)$. Thus:

$$\begin{aligned} s_i(\Gamma_i) &= s' \circ s_n \circ \ldots \circ s_{i+1}(s_i(\Gamma_i)) \\ &= s' \circ s_n \circ \ldots \circ s_{i+1}(s_i(s_{i-1} \circ \ldots s_0(\Gamma_i))) \\ &= s(\Gamma_i) \subseteq \Gamma \end{aligned}$$

And so $s(\Gamma_i) \subseteq \Gamma$ for all $i \in \overline{n}$. Let u be a minimal characteristic unification problem for $\vec{\Gamma}_n$, then by Lemma 7.17(3), s unifies u.

Let $\vec{\tau}_n$ be a sequence of simple types satisfying

i) $f_i{:}\sigma''_i \in \Gamma_0 \Rightarrow \tau_i = \sigma''_i$; and ii) $f_i \notin \Gamma_0 \Rightarrow \tau_i = \varphi_i$

for each $i \in \overline{n}$, and $\overrightarrow{\tau'}_{n'}$ be a sequence of types satisfying

i) $x_i{:}\sigma''_i \in \Gamma_0 \Rightarrow \tau'_i = \sigma''_i$; and ii) $x_i \notin \Gamma_0 \Rightarrow \tau'_i = \varphi'_i$

for each $i \in \overline{n'}$. If $\texttt{this} \in \Gamma_0$ then let $u' = (C, \Gamma_0(\texttt{this})) \cdot (\tau_1, \tau''_1) \cdot \ldots \cdot (\tau_n, \tau''_n)$, otherwise let $u' = (\tau_1, \tau''_1) \cdot \ldots \cdot (\tau_n, \tau''_n)$. If $\texttt{this} \in \Gamma_0$ then, since $s_0(\Gamma_0) \subseteq \Gamma'$, $s_0(\Gamma_0(\texttt{this})) = C$ and so by Definition 7.8, $\Gamma_0(\texttt{this}) = C$. Thus $s(C) = s(\Gamma_0(\texttt{this})) = C$. Now, take any (τ_i, τ''_i) $(i \in \overline{n})$; we will show that $s(\tau_i) = s(\tau''_i)$.

a) There are two cases for τ_i:

($f_i \in \Gamma_0$): then $\tau_i = \Gamma_0(f_i)$; notice that since $s_0(\Gamma_0) \subseteq \Gamma$ and $\Gamma'(f_i) = \sigma_i$, it must be that $s_0(\tau_i) = \sigma_i$. Then since $\mathsf{dom}(s_j)$ is distinct from σ_i for all $0 \leq j \leq n$, and no φ_k ($k \in \overline{n}$) or φ'_k ($k \in \overline{n'}$) occurs in σ_i, we have $s' \circ s_n \circ \ldots \circ s_1(\sigma_i) = \sigma_i$. Thus $s(\tau_i) = s' \circ s_n \circ \ldots \circ s_1 \circ s_0(\tau_i) = s' \circ s_n \circ \ldots \circ s_1(\sigma_i) = \sigma_i$.

($f_i \notin \Gamma_0$): then $\tau_i = \varphi_i$ and since φ_i does not occur in any typing $[\Gamma_i, \tau''_i]$ for $(0 \leq i \leq n)$, we have that $\varphi_i \notin \mathsf{dom}(s_j)$ for any $0 \leq j \leq n$ and so $s_n \circ \ldots \circ s_0(\varphi_i) = \varphi_i$. Notice that $s'(\varphi_i) = \sigma_i$ and thus $s(\tau_i) = s(\varphi_i) = s' \circ s_n \circ \ldots \circ s_0(\varphi_i) = s'(\varphi_i) = \sigma_i$.

So, $s(\tau_i) = \sigma_i$.

b) Since $\mathsf{dom}(s_j)$ is distinct from τ''_i for all $0 \leq j \leq n$ such that $i \neq j$, it follows that $s_{i-1} \circ \ldots \circ s_0(\tau''_i) = \tau''_i$. Also, since $\mathsf{dom}(s_j)$ is distinct from σ_i for all $j \in \overline{n}$, and no φ_k ($k \in \overline{n}$) or φ'_k ($k \in \overline{n'}$) occurs in σ_i, we have that $s' \circ s_n \circ \ldots \circ s_{i+1}(\sigma_i) = \sigma_i$. Thus, since $s_i(\tau''_i) = \sigma_i$, it follows that

$$\begin{aligned} s_i(\tau''_i) &= s' \circ s_n \circ \ldots \circ s_{i+1}(s_i(\tau''_i)) \\ &= s' \circ s_n \circ \ldots \circ s_{i+1}(s_i(s_{i-1} \circ \ldots \circ s_0(\tau''_i))) \\ &= s(\tau''_i) = \sigma_i \end{aligned}$$

Thus, we can conclude s unifies u'. We will now show that $s(\langle m : (\overrightarrow{\tau'}_{n'}) \to \tau''_0 \rangle) = \langle m : (\overrightarrow{\sigma'}_{n'}) \to \sigma_0 \rangle$.

a) Take any τ'_i ($i \in \overline{n'}$; there are two cases:

($x_i \in \Gamma_0$): then $\tau'_i = \Gamma_0(x_i)$; notice that since $s_0(\Gamma_0) \subseteq \Gamma$ and $\Gamma'(x_i) = \sigma'_i$, it must be that $s_0(\tau'_i) = \sigma'_i$. Then since $\mathsf{dom}(s_j)$ is distinct from σ'_i for all $0 \leq j \leq n$, and no φ_k ($k \in \overline{n}$) or φ'_k ($k \in \overline{n'}$) occurs in σ'_i, it follows that $s' \circ s_n \circ \ldots \circ s_1(\sigma'_i) = \sigma'_i$. Thus $s(\tau'_i) = s' \circ s_n \circ \ldots \circ s_1 \circ s_0(\tau'_i) = s' \circ s_n \circ \ldots \circ s_1(\sigma'_i) = \sigma'_i$.

($x_i \notin \Gamma_0$): then $\tau'_i = \varphi'_i$ and since φ'_i does not occur in any typing $[\Gamma_i, \tau''_i]$ for $(0 \leq i \leq n)$, we have that $\varphi'_i \notin \mathsf{dom}(s_j)$ for any $0 \leq j \leq n$ and so $s_n \circ \ldots \circ s_0(\varphi'_i) = \varphi'_i$. Notice that $s'(\varphi'_i) = \sigma'_i$ and thus $s(\tau'_i) = s(\varphi'_i) = s' \circ s_n \circ \ldots \circ s_0(\varphi'_i) = s'(\varphi'_i) = \sigma'_i$.

So, $s(\tau'_i) = \sigma'_i$ for all $i \in \overline{n'}$.

b) We have that $s_0(\tau''_0) = \sigma_0$. Also, since $\mathsf{dom}(s_i)$ is distinct from σ_0 for all $0 \leq i \leq n$ and no φ_k ($k \in \overline{n}$) or φ'_k ($k \in \overline{n'}$) occurs in σ_0, it follows that $s' \circ s_n \circ \ldots \circ s_1(\sigma_0) = \sigma_0$. So $s(\tau''_0) = s' \circ s_n \circ \ldots \circ s_1 \circ s_0(\tau''_0) = s' \circ s_n \circ \ldots \circ s_1(\sigma_0) = \sigma_0$.

Thus we can conclude that

$$\begin{aligned} s(\langle m : (\overrightarrow{\tau'}_{n'}) \to \tau''_0 \rangle) &= \langle m : (s(\tau'_1), \ldots, s(\tau'_{n'})) \to s(\tau''_0) \rangle \\ &= \langle m : (\overrightarrow{\sigma'}_{n'}) \to \sigma_0 \rangle \end{aligned}$$

Now, since s unifies both u and u', it follows that s unifies $u' \cdot u$ and so by Property 7.15 there are s''' and s'' such that $s'' = \mathsf{Unify}(u' \cdot u)$ and $s = s''' \circ s''$. By Definition 7.19, $[s''(\Gamma''), s''(\langle m : (\overrightarrow{\tau'}_{n'}) \to \tau''_0 \rangle)] \in \mathcal{PTS}(\mathsf{new}\ \ C(\vec{e}_n))$ where $\Gamma'' = \Gamma_1 \cup \ldots \cup \Gamma_n$. Since $s(\Gamma_i) \subseteq \Gamma$ for each $i \in \overline{n}$, it follows that $s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$, and so $s'''(s''(\Gamma'')) = s''' \circ s''(\Gamma'') = s(\Gamma'') = s(\Gamma_1) \cup \ldots \cup s(\Gamma_n) \subseteq \Gamma$. Finally, $s'''(s''(\langle m : (\overrightarrow{\tau'}_{n'}) \to \tau''_0 \rangle)) = s''' \circ s''(\langle m : (\overrightarrow{\tau'}_{n'}) \to \tau''_0 \rangle) = s(\langle m : (\overrightarrow{\tau'}_{n'}) \to \tau''_0 \rangle) = \langle m : (\overrightarrow{\sigma'}_{n'}) \to \sigma_0 \rangle$. □

To address the question of the decidability of simple type assignment, we turn the above definition of principal typings into the following *algorithm*. We can then reason about its termination properties.

Definition 7.23 (Principal Typings Algorithm)**.** *The semi-algorithm* **PTS** *computes the set of principal typings for an expression. It is defined, in the context of a program P, by cases as follows:*

PTS(x) = *if* (x = `this`) *then*
let $T = \emptyset$
for each class `C` *in the program*
add [{`this`:`C`}, `C`] *to* T
for each field `f` *in* $\mathcal{F}$(`C`)
let φ *be fresh*
add [{`this`:`C`, `f`:φ}, ⟨`f`:φ⟩] *to* T
return T
else
let φ *be fresh*
return {[{x:φ}, φ]}

PTS(e`.`f) = *let* $T = \emptyset$
T' = **PTS**(e)
for each $[\Gamma, \sigma] \in T'$
let φ *be fresh*
$s = \mathsf{Unify}(\sigma, \langle f : \varphi \rangle)$
if unification did not fail, then add [$s(\Gamma)$, $s(\varphi)$] *to* T
return T

PTS(e_0`.`m(`(`$\vec{e}_n$`)`)) =
let $T = \emptyset$
T_i = **PTS**(e_i) *for each* $0 \le i \le n$
for each combination of $[\Gamma_0, \sigma_0], \ldots, [\Gamma_n, \sigma_n]$ *such that*
$[\Gamma_i, \sigma_i] \in T_i$ *for* $0 \le i \le n$
let u be a minimal characteristic unification problem for $\Gamma_0, \ldots, \Gamma_n$
$\Gamma = \Gamma_0 \cup \ldots \cup \Gamma_n$
φ *be fresh*
$s = \mathsf{Unify}((\sigma_0, \langle m : (\vec{\sigma}_n) \to \varphi \rangle) \cdot u)$
if unification did not fail, add [$s(\Gamma)$, $s(\varphi)$] *to* T
return T

PTS(`new` `C`(`(`$\vec{e}_n$`)`)) =
let $\vec{f}_m = \mathcal{F}$(`C`)
if ($n = m$) *then*
let $T = \emptyset$
T_i = **PTS**(e_i) *for each* $i \in \overline{n}$
for each combination of $[\Gamma_1, \sigma_1], \ldots, [\Gamma_n, \sigma_n]$ *such that*
$[\Gamma_i, \sigma_i] \in T_i$ *for all* $i \in \overline{n}$
let $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$
u be a minimal characteristic unification problem for $\vec{\Gamma}_n$
$s = \mathsf{Unify}(u)$

if unification did not fail, then

add $[s(\Gamma), \mathtt{C}]$ *to* T

for each $i \in \overline{n}$

add $[s(\Gamma), s(\langle \mathtt{f}_i : \sigma_i \rangle)]$ *to* T

for each method $\mathtt{m}$ *in* $\mathtt{C}$

let $\mathcal{M}\mathrm{b}(\mathtt{C}, \mathtt{m}) = (\vec{x}_{n'}, e_0)$

$T_0 = \mathbf{PTS}(e_0)$

for each combination of $[\Gamma_0, \sigma_0], \ldots, [\Gamma_n, \sigma_n]$ *such that*

$[\Gamma_i, \sigma_i] \in T_i$ *for* $0 \leq i \leq n$

let $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$

u *be a minimal characteristic unification problem for* $\vec{\Gamma}_n$

$u' = (\tau_1, \sigma_1) \cdot \ldots \cdot (\tau_n, \sigma_n)$

where for each $i \in \overline{n}$

$\tau_i = \Gamma_0(\mathtt{f}_i)$ *if* $\mathtt{f}_i \in \sigma_0$ *and*

$\tau'_i = \varphi$ *otherwise, with* φ *fresh*

$s = \mathsf{Unify}(u' \cdot u)$

if unification did not fail, add $[s(\Gamma), s(\langle \mathtt{m} : (\vec{\tau'}_{n'}) \to \sigma_0 \rangle)]$ *to* T

where for each $i \in \overline{n'}$

$\tau'_i = \Gamma_0(x_i)$ *if* $x_i \in \sigma_0$ *and*

$\tau'_i = \varphi$ *otherwise, with* φ *fresh*

return T

PTS is indeed a semi-algorithm because for certain programs it will loop forever, never returning any output. Consider the self-returning objects that we considered in Section 6.1, which are instances of the following class:

```
class SR extends Object {
    SR self() { return this; }
    SR newInst() { return new SR(); }
}
```

If we run the algorithm on the expression `new SR()`, it will successfully add the typing $[\emptyset, \langle \mathtt{self} : () \to \mathtt{SR} \rangle]$ to its set T of principal typings. However, when it comes to analyse the `newInst` method, it will recursively call itself on the body of the method, which is again the expression `new SR()`, thus entering a non-terminating loop.

The program we have just considered is an example of *recursive* program - it contains a class that has a method which, when invoked, results in the creation of a new instance of the class itself. For any such program, the algorithm **PTS** will not terminate. However, for *non*-recursive programs, it correctly computes principal typing sets.

We formalise the notion of recursive (and non-recursive) programs by defining a *dependency* relation on the classes in a program. This notion of dependency first relies on a notion of subexpression:

Definition 7.24 (Subexpression Relation)**.** *The* subexpression *relation* $<$ *is defined as the smallest tran-*

sitive relation on expressions satisfying the following:

$$e < \begin{cases} e.f \\ e.m(\vec{e}) \end{cases}$$

$$\textit{(for all } i \in \overline{n}) \quad e_i < \begin{cases} e.m(\vec{e}_n) \\ \texttt{new } C(\vec{e}_n) \end{cases}$$

Notice that the subexpression relation we have just defined is a *strict* order, i.e. it is irreflexive. This fact will be an important component in the proof of Theorem 7.28.

When the body of a method in class C refers to a class D, then we say that C *depends* on D.

Definition 7.25 (Class Dependency Relation)**.** *The class dependency relation $\prec$ is defined as the smallest transitive relation on classes satisfying the following:*

$$\mathcal{M}b(C,m) = (\vec{x}, \texttt{new } D(\vec{e})) \Rightarrow D \prec C$$
$$\mathcal{M}b(C,m) = (\vec{x}, e) \;\&\; \texttt{new } D(\vec{e}) < e \Rightarrow D \prec C$$

Definition 7.26 (Recursive Programs)**.** 1. *We say that a class C is* recursive *if $C \prec C$.*

2. *We say that a program is recursive if it contains at least one recursive class.*

3. *We say that a program is* non-recursive *if it is not recursive.*

To show that the algorithm **PTS** terminates for non-recursive programs, we will define an encompassment relation on expressions. For non-recursive programs, this encompassment relation turns out to be well-founded.

Definition 7.27 (Encompassment)**.** *The* encompassment *relation $\lhd$ on expressions (for a program P), is the smallest relation on expressions satisfying the following two conditions:*

$$e < e' \Rightarrow e \lhd e'$$
$$\mathcal{M}b(C,m) = (\vec{x}, e) \Rightarrow e \lhd \texttt{new } C(\vec{e}) \quad \textit{(for all } \vec{e})$$

Theorem 7.28 (Well-foundedness of Encompassment)**.** *If P is a non-recusive program, then its encompassment relation is well-founded.*

Proof. We prove the contrapositive: i.e. if the encompassment relation of P is *not* well-founded, then P is recursive. Take any program P and assume its encompassment relation is not well-founded. Then there exists some infinite descending chain

$$e_1 \rhd e_2 \rhd e_3 \ldots$$

By Definition 7.27, for each e_i and e_{i+1} in the chain, either $e_{i+1} < e_i$ or e_i is of the form $\texttt{new } C(\vec{e})$ and there is some m such that $\mathcal{M}b(C,m) = (\vec{x}, e_{i+1})$. Notice that these two possibilities are mutually exclusive. Since the chain is infinite and for every $e_{i+1} < e_i$, e_{i+1} is strictly smaller than e_i, there must therefore be

an infinite number of pairs of expressions e_j, e_{j+1} in the chain such that e_j is of the form $\texttt{new } C(\vec{e})$ and $\mathcal{M}b(C,m) = (\vec{x}, e_{j+1})$. Thus, the chain is as follows:

$$e_1^1 > \ldots > e_{n_1}^1 > \texttt{new } C_1(\vec{e}) \rhd e_1^2 > \ldots > e_{n_2}^2 > \texttt{new } C_2(\vec{e}) \rhd e_1^3 > \ldots$$

where, for each $i \geq 1$, either

1. $n_{i+1} = 0$, so $\mathcal{M}b(C_i,m) = (\vec{x}, \texttt{new } C_{i+1}(\vec{e}))$ for some m, and thus by Definition 7.25 we have $C_i > C_{i+1}$; or
2. $n_{i+1} > 0$, so $\mathcal{M}b(C_i,m) = (\vec{x}, e_1^{i+1})$ for some m; then since $<$ is a transitive relation, it follows that $e_1^{i+1} > \texttt{new } C_{i+1}(\vec{e})$ and thus by Definition 7.25 we have $C_i > C_{i+1}$.

Therefore, there is an infinite chain $C_1 > C_2 > C_3 > \ldots$ and by transitivity of the class dependency relation, $C_i > C_j$ for all $i, j \geq 1$ such that $i < j$. Now, since the program must be finite (i.e. contain a finite number of classes), there must be $i, j \geq 1$ such that $i < j$ and $C_i = C_j$, and so there is a class that depends on itself. Thus, the program is recursive. □

Now, using the fact that the encompassment relation for non-recursive programs is well-founded, we can show a termination result for **PTS**.

Theorem 7.29 (Termination of **PTS**)**.** *For non-recursive programs,* **PTS**(e) *terminates on all expressions.*

Proof. By Noetherian induction on $\lhd$, which is well-founded for non-recursive programs. We do a case analysis on e:

(x): If $x \neq \texttt{this}$ then we simply have to construct a single typing and return it; if $x = \texttt{this}$, then we have to do this for each class in the program and each of their fields. Since there are a finite number of these, this will terminate.

($e.f$): First of all, we recursively call the algorithm on e; since $e \lhd e.f$, by induction we know this will terminate, and if it does not fail it must necessarily return a finite set of typings. For each of these typings we must unify a pair of types and apply the resulting substitution, all of which are terminating procedures.

($e_0.m(\vec{e}_n)$): Firstly, we recursively call the algorithm on each expression e_i. Since for each i, $e_i \lhd e_0.m(\vec{e}_n)$, by induction each of these calls will terminate. If none of them fail, they must each necessarily return a finite set of typings. Thus, the number of all possible combinations for choosing a typing from each set is finite. For each of these combinations, we must build a unification problem, call the Unify procedure on it, generate a typing and apply a substitution to it. Since the type environment of each typing is finite, we can compute the minimal characteristic unification problem. The procedure Unify always terminates (Property 7.15). As remarked in the previous case, generating typings and applying substitutions are also terminating procedures.

($\texttt{new } C(\vec{e}_n)$): The number of fields in a class is finite and (for well-formed programs), the lookup procedure for fields is terminating. If the number of expressions in $\vec{e}_n$ matches the number of fields, we recursively call the type inference algorithm on each one. Since $e_i \lhd \texttt{new } C(\vec{e}_n)$ each of these calls will terminate. If none of them fail, they must each necessarily return a finite set of typings. In this case, the algorithm has two main tasks:

1. For each combination of choosing a typing from each set $\mathbf{PTS}(e_i)$, the algorithm must construct a (minimal) unification problem for the type environments which, as remarked above, is a terminating procedure. The algorithm then applies the Unify procedure, which is terminating (Property 7.15), and adds a typing for the class type C, and one for each field of C, of which there are a finite number.
2. For each method m in C, we lookup the method's formal parameters and body, $\mathcal{M}b(C,m) = (\vec{x}, e_0)$. As for field lookup, this is a terminating procedure for well-formed programs, and there are a finite number of methods. The algorithm then recursively calls itself on the method body e_0. Since $e_0 \lhd \texttt{new } C(\vec{e}_n)$, by the inductive hypothesis this is terminating, and necessarily returns a finite set of typings. Since each set is finite, the number of combinations of typings chosen from the principal typing set of each $e_0, \ldots, e_n$ is finite. For each combination, the algorithm builds a (minimal) characteristic unification problem for the type environments, and also constructs a second unification problem of size n. These both take finite time. It then combines the two and applies the Unify procedure, which is terminating. If unification succeeds, it builds a typing and applies a substitution, as remarked, both terminating procedures.

□

Notice that since a program is a finite entity, and the number of classes it contains is finite, it is decidable whether any given program is recursive or not. Thus, we can always insert a pre-processing step prior to type inference which checks if the input program is non-recursive.

This restricted form of type assignment and its type inference algorithm could straightforwardly be extended to incorporate intersections of finite rank. This is not much help, though, in a typical object-oriented setting, since the 'natural' way to program in such a context is with recursive classes. Consider the oo arithmetic program of Section 6.4 - there the Suc class depends (in the sense of Def. 7.25) upon itself. If this example seems too 'esoteric', consider instead the program of Section 6.3 defining lists, an integral component of any serious programmer's collection of tools.

A slightly different approach to type inference that we could take is to keep track, as we recurse through the program, of all the classes that we have already 'looked inside' - i.e. all those classes for which we have already looked up method bodies. Then, whenever we encounter a $\texttt{new } C(\vec{e})$ expression, if the class C is in the list of previously examined classes, we only allow the algorithm to infer typings of the form $[\Gamma, C]$ or $[\Gamma, \langle f{:}\sigma \rangle]$. That is, we do not allow it to look inside the method definitions a second time.

We could also modify the definition of simple type assignment to reflect this, by defining the type assignment judgement to refer to a second environment Σ containing class names. This second environment would allow the system to keep track of which class definitions it has already 'unfolded'. The only type assignment rule that would need modifying is the (NEWM) rule, which would be redefined as follows:

$$\frac{\Sigma \cup \{C\}; \{f_1{:}\sigma'_1, \ldots, f_{n'}{:}\sigma'_{n'}, \texttt{this}{:}C, x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\} \vdash_{\mathrm{s}} e_{\mathrm{b}} : \sigma \quad \Sigma; \Gamma \vdash_{\mathrm{s}} e_i : \sigma'_i \quad (\forall\, i \in \overline{n'})}{\Sigma; \Gamma \vdash_{\mathrm{s}} \texttt{new } C(\vec{e}_{n'}) : \langle m : (\vec{\sigma}_n) \to \sigma \rangle}$$

$$(\mathcal{F}(C) = \vec{f}_{n'},\ \mathcal{M}b(C,m) = (\vec{x}_n, e_{\mathrm{b}}),\ C \notin \Sigma)$$

The modified type inference algorithm would then be complete with respect to this modified type assignment system. It would also be terminating for all programs. From a practical point of view, however, this does not constitute a great improvement in the object-oriented setting - the types inferred for recursive programs are quite limiting. Take, for example, the oo arithmetic program: the set of principal typings for `new Suc(`$⟦n⟧_{\mathbb{N}}$`)` objects in our decidable type inference system (for any finite rank of intersection) only contains typings of the following general forms:

$$[\Gamma, \texttt{Suc}] \qquad [\Gamma, \langle \texttt{pred:}\sigma \rangle]$$
$$[\Gamma, \langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle] \qquad [\Gamma, \langle \texttt{add:}(\varphi) \to \langle \texttt{pred:}\sigma \rangle \rangle]$$

The set of principal typings for `new Zero()` consists of the following two typings:

$$[\emptyset, \texttt{Zero}] \qquad [\emptyset, \langle \texttt{add:}(\varphi) \to \varphi \rangle]$$

Thus, while we can infer the 'characteristic' type for each object-oriented natural number (as discussed in Section 6.4), the types we can infer for the methods `add` and `mult` are the limiting factor. For example, these types do allow us to add an arbitrary sequence of numbers together by writing an expression of the form $⟦n_1⟧_{\mathbb{N}}$`.add(`$⟦n_2⟧_{\mathbb{N}}$`.add(....add(`$⟦n_m⟧_{\mathbb{N}}$`)))`. However, 'equivalent' expressions of the form $⟦n_1⟧_{\mathbb{N}}$`.add(`$⟦n_2⟧_{\mathbb{N}}$`).....add(`$⟦n_m⟧_{\mathbb{N}}$`)` are rejected as ill-typed (unless each n, ..., n_{m-1} is zero) since the only type we can derive for the expression $⟦n_1⟧_{\mathbb{N}}$`.add(`$⟦n_2⟧_{\mathbb{N}}$`)` is `Suc`, preventing us from invoking the remaining `add` methods.

The situation is even worse if we consider the `mult` method. For `new Zero()`, we can derive types of the form $\langle$`mult:`$(\varphi) \to$ `Zero`$\rangle$, leaving us in pretty much the same situation as with the `add` method. For `new Suc(new Zero())`, the encoding of one, we are slightly more restricted: we can assign types of the form $\langle$`mult:`$\langle$`add:Zero` $\to \varphi\rangle \to \varphi\rangle$. Since, as we have seen, $\langle$`add:Zero` $\to \varphi\rangle$ is not a type we can infer for any number, we must substitute the type variable φ for something in order to make this into a type we can use for an invocation of the `mult` method. There are two candidates: $\langle$`add:Zero` $\to$ `Zero`$\rangle$, which we can infer for `new Zero()`, or $\langle$`add:Zero` $\to$ `Suc`$\rangle$ which we can infer for encodings of positive numbers. Thus, we may only type the multiplication of 1 by a single number. For the encoding of any number greater than one, we can only infer the single type $\langle$`mult:`$\langle$`add:Zero` $\to$ `Zero`$\rangle \to$ `Zero`$\rangle$, meaning that for $n \geq 2$ we may only type the expressions $⟦n⟧_{\mathbb{N}}$`.mult(new Zero())`. From this discussion, it should be obvious that the utility of our type inference procedure is limited - it types too few programs.

To consider a final example, we turn our attention to the list program of Section 6.3. This is quite similar to the case for the `add` method in the arithmetic program. Indeed, the `append` method functions in an almost identical manner. This means that our type inference algorithm can only infer types of the form $\langle$`append:`$(\varphi) \to \varphi\rangle$ for empty lists, and the types $\langle$`append:`$(\varphi) \to \langle \underbrace{\texttt{tail:}\ldots\langle\texttt{tail}}_{n \text{ times}}\texttt{:}\varphi\rangle\ldots\rangle\rangle$ for lists of size n. As for the `cons` method, we obtain the type schemes $\langle$`cons:`$(\varphi) \to$ `NEL`$\rangle$, $\langle$`cons:`$(\varphi) \to \langle$`head:`$\varphi\rangle\rangle$, and $\langle$`cons:`$(\varphi) \to \langle$`tail:NEL`$\rangle\rangle$ for non-empty lists, and for empty lists the additional type scheme $\langle$`cons:`$(\varphi') \to \sigma\rangle$, where σ is one the three type schemes for non-empty lists.

At this point, it is natural to ask the question whether there is any way to modify the system so that we can infer more useful types for recursively defined programs. An answer to this question can be found

if we go back a step and consider, not the types that we can algorithmically infer for say, the arithmetic program, but the (infinite) set of principal types it has according to Definition 7.19. Let us not be too ambitious, and restrict ourselves to considering just those types which pertain to the `add` method. What we find is that, even though this set of types is infinite, it is *regular*. Namely, for each encoded number, we can assign the following sequence of types:

$$\begin{aligned}
&\langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle \\
&\langle \texttt{add:}(\langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle) \to \langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle \rangle \\
&\langle \texttt{add:}(\langle \texttt{add:}(\langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle) \to \langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle \rangle) \\
&\qquad \to \langle \texttt{add:}(\langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle) \to \langle \texttt{add:}(\varphi) \to \texttt{Suc} \rangle \rangle \rangle \quad \ldots
\end{aligned}$$

As can be seen, each successive type for the `add` method forms both the argument and the result type of the subsequent type. In the limit, if we were to allow types to be of infinite size, we would obtain a type σ which is characterised by the following equation:

$$\sigma = \langle \texttt{add:}\sigma \to \sigma \rangle$$

In a certain sense, this type is the *most specific*, or principal one because it contains the most information. The type in the above equation is defined, or expressed in terms of itself, and as such can be described by *recursive* type $\mu X.\langle \texttt{add:}X \to X \rangle$ which denotes the type which is the solution to the above equation. This type also nicely illustrates the object-oriented concept of a *binary method*, which is a method that takes as an argument an object of the *same* kind as the receiver. This is expressed in the nominal typing system (see Section 6.6) by specifying in the type annotation for the formal parameter the same class as the method is declared in. For the arithmetic program, this can be seen in the specification of the `add` method in the `Nat` class (interface), which specifies that the argument should be of class `Nat`. The recursive type that we have given above expresses this relationship via the use of the *recursively bound* type variable X.

We do not have to look at a program as relatively complex as the arithmetic program to make this observation regarding recursive types. We remarked in Section 6.1 that the self-returning object program defines a class whose instances can be given the infinite, but regular family of types $\langle \texttt{self:}() \to \texttt{SR} \rangle$, $\langle \texttt{self:}() \to \langle \texttt{self:}() \to \texttt{SR} \rangle \rangle$, …, etc. As for the `add` method, the (infinite) type which is the limit of this sequence can be denoted by the recursive type $\mu X.\langle \texttt{self:}() \to X \rangle$.

The use of recursive types to describe object-oriented programs is not new. We have already seen in Chapter 2, for example, that Abadi and Cardelli consider recursive types for the ς-calculus. The problem with such recursive types is that, traditionally, they do not capture the termination properties of programs, which is one of the key advantages of the intersection type discipline. In the second part of this thesis, we will consider a particular variation on the theme of recursive types that we claim will allow us to do just that, and so obtain a system with similar expressive power to ITD, but which also admits the inference of useful types for recursively defined classes.

Part II.

Logical Recursive Types

8. Logical vs. Non-Logical Recursive Types

At the end of the first part of this thesis, we remarked that recursive types very naturally and effectively capture the behaviour of object-oriented programs, since they are finite representations of (regular) infinite types. As we also mentioned, this is well known. In this second part of the thesis, we will investigate the potential for semantically-based, decidable type inference for oo provided by a particular flavour of so-called 'logical' recursive types.

In this chapter, we will review the relevant background and current research in this area. We start by presenting a basic extension of the simple type theory for the λ-calculus which incorporates recursive types. This very simple extension of the type theory shows that a naive treatment of recursive types leads to logical inconsistency, and therefore does not provide a sound *semantic* basis for type analysis. At heart, this is a very old result, the essence of which was first formulated mathematically by Bertrand Russell, but analogous logical paradoxes involving self-reference have been known to philosophers since antiquity.

The situation is not a hopeless one, however. The logical inconsistency we describe stems from using *unrestricted* self-reference, the operative term here begin 'unrestricted'. By placing restrictions on the form that self-reference may take, logical consistency can be regained. A well-known result of Mendler [78] in the theory of recursive types is that by disallowing *negative* self-reference (i.e. occurrences of recursively bound type variables on the left-hand sides of arrow, or function, types), typeable terms once again become strongly normalising as for Simply Typed λ-calculus. In the setting of oo however, this is not an altogether viable solution, since there are quintessentially object-oriented features such as binary methods (discussed in the previous chapter) which require negative self-reference.

An alternative approach to restricting self-reference has been described by Nakano, who has developed a family of type systems with recursive types which do not suffer from the aforementioned logical paradox, and which also do not forbid negative occurrences of recursively bound variables. As such, these type systems allow a form of characterisation of normalisation. They are not as powerful as systems in the intersection type discipline, since they do not characterise normalising or strong normalising terms, however they do give head normalisation and weak normalisation guarantees.

We believe that Nakano's variant of recursive type assignment is therefore a good starting point for building semantic, decidable type systems which are well-suited to the object-oriented programming paradigm. This observation is made by Nakano himself, however he does not describe explicitly how his type systems might be applied in the context of oo, nor does he discuss a type inference procedure. This is where we take up the baton: the answering of these questions is that which shall concern us in the remainder of this thesis, and wherein the contribution of our work lies.

8.1. Non-Logical Recursive Types

While recursive types very naturally capture the behaviour of recursively defined constructions, if we are not careful we can introduce logical inconsistency into the type analysis of such entities. As we will later point out, this kind of logical inconsistency is not preclusive to the functional analysis of programs, but limits the analysis to an expression of *partial* correctness only. That is, it does capture the termination properties of programs, and therefore cannot be called fully semantic.

This can be illustrated by using a straightforward extension of the simply typed λ-calculus to recursive types. In [34] Cardone and Coppo present a comprehensive description of recursive type systems for λ-calculus, and in [35] they review the results on the decidability of equality of recursive types. Here we present one of the type systems described in [34], in which the logical inconsistency can be illustrated. We shall call the system that we describe below λ_μ (a name given by Nakano, which we borrow since it is unnamed in [34]).

Definition 8.1 (Types)**.** *The types of λ_μ are defined by the following grammar, where $X, Y, Z \ldots$ range over a denumerable set of type variables:*

$$A, B, C \quad ::= \quad X \quad | \quad A \to B \quad | \quad \mu X . A$$

We say that the type variable X is bound *in the type $\mu X . A$, and defined the usual notion of free and bound type variables. The notation $A[B/X]$ denotes the type formed by replacing all free occurrences of X in A by the type B.*

The type $\mu X . A$ is a recursive type, which can be 'unfolded' to $A[\mu X . A/X]$. This process of unfolding and folding of recursive types induces a notion of equivalence.

Definition 8.2 (Equivalence of Types)**.** *The equivalence relation $\sim$ is defined as the smallest such relation on λ_μ types satisfying the following conditions:*

$$\begin{aligned} \mu X . A &\sim A[\mu X . A/X] \\ A \sim B &\Rightarrow \mu X . A \sim \mu X . B \\ A \sim C \;\&\; B \sim D &\Rightarrow A \to B \sim C \to D \end{aligned}$$

This notion of equivalence is the weaker of the two equivalence relations described by Cardone and Coppo in [34]. The stronger notion is derived by allowing type expressions to be infinite, and considering two (recursive) types to be equivalent when their infinite unfoldings are equal to one another.

This equivalence relation plays a crucial role in type assignment, since we allow types to be replaced 'like-for-like' during assignment. This means that, because a recursive type is equivalent to its unfolding, types can be folded and unfolded as desired during type assignment. It is this capability that will lead to logical inconsistency, as we will explain shortly.

Definition 8.3 (Type Assignment)**.**

1. *A type* statement *is of the form $M : A$ where M is a λ-term, and A is a λ_μ type; the term M is called the* subject *of the statement.*

2. *A type* environment Γ *is a finite set of type statements in which the subject of each statement is a* unique *term variable. The notation $\Gamma, x : A$ stands for the type environment $\Gamma \cup \{x : A\}$ where x does not appear as the subject of any statement in Γ.*

3. *Type* assignment $\Gamma \vdash M : A$ *is a relation between type environments and type statements. It is defined by the following natural deduction system:*

$$(Var): \frac{}{\Gamma, x : A \vdash x : A} \qquad (\sim): \frac{\Gamma \vdash M : A}{\Gamma \vdash M : B}\,(A \sim B)$$

$$(\to I): \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad (\to E): \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

The system enjoys the usual property that is desired in a type system, namely subject reduction [34, Lemma 2.5]. It does not have a principal typings property [34, Remark 2.13], although its sibling system based on the stronger notion of equivalence that we mentioned above does have this property [34, Theorem 2.9].

The logical inconsistency permitted by this type assignment system is manifested in the fact that to some terms, we can assign any and all types. An example of such a term is $(\lambda x.xx)(\lambda x.xx)$. Let A be any type of λ_μ, and let $B = \mu X\,.\,X \to A$. Then we can derive $\vdash (\lambda x.xx)(\lambda x.xx) : A$ as witnessed by the following derivation schema:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : B \vdash x : B}\,(\text{Var})}{x : B \vdash x : B \to A}\,(\sim) \qquad \dfrac{}{x : B \vdash x : B}\,(\text{Var})}{x : B \vdash xx : A}\,(\to E)}{\vdash \lambda x.xx : B \to A}\,(\to I) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : B \vdash x : B}\,(\text{Var})}{x : B \vdash x : B \to A}\,(\sim) \qquad \dfrac{}{x : B \vdash x : B}\,(\text{Var})}{x : B \vdash xx : A}\,(\to E)}{\vdash \lambda x.xx : B \to A}\,(\to I)}{\Gamma \vdash \lambda x.xx : B}\,(\sim)}{\Gamma \vdash (\lambda x.xx)(\lambda x.xx) : A}\,(\to E)$$

The reason for calling this a logical inconsistency becomes apparent when considering a Curry-Howard correspondence [64] between the type system and a formal logic. In this correspondence, types are seen as logical formulae, and the type assignment rules are viewed as inference rules for a formal logical system, obtained by erasing the all λ-terms in the type statements. Then, derivations of the type assignment system become derivations of formulas in the logical system, i.e. proofs. A formal logical system is said to inconsistent if every formula is derivable (i.e. has a proof). Thus, the derivation above constitutes a proof for every formula, and the corresponding logic is therefore inconsistent. The connection with self-reference comes from noticing that recursive types, when viewed as logical formulae, are logical statements that refer to themselves.

The significance of this result in the context of our research is that for such logically inconsistent type systems, type assignment is no longer semantically grounded. That is, it no longer expresses the termination properties of typeable terms. This can be seen to derive from the fact that we can no longer show an approximation result for such systems - types no longer correspond to approximants. Consider, again, the term that we have just typed above: it is an unsolvable (non-terminating) term and so has only the approximant $\bot$. The only type assignable to $\bot$ is the top type ω, however we are able to assign *any* type to the original term.

Even though these non-logical systems no longer capture the termination properties of programs, they do still constitute a functional analysis. Since for typeable terms it must be that all the subterms are typeable, and since the system has the subject reduction property, we are guaranteed that all applications that appear during reduction are well-typed, and thus will not go awry. A semantic basis for this result is also given in [77]. Therefore, we can describe these non-logical systems as providing a *partial* cor-

rectness analysis, as opposed to the fully correct analysis given by intersection type assignment which guarantees termination as well as functional correctness.

While we have formulated and demonstrated the illogical character of the (unrestricted) recursive type assignment within the context of λ-calculus, this result is by no means limited to that system. The inconsistency is inherent to the recursive types themselves. As an example, we will consider a typeable term in the ς-calculus of objects of Abadi and Cardelli that displays the same logical inconsistency. We refer the reader back to Section 2.2 for the details of the calculus and the type system.

Consider the (untyped) object:

$$o = [m = \varsigma(z).\lambda x.z.m(x)]$$

We will give a derivation schema that assigns any arbitrary type A to the term $o.m(o)$ - i.e. the *self-application* of the object o. We will use the recursive object type $O = \mu X.[m : X \to A]$. Notice that we can assign the type $[m : O \to A]$ to the object o itself, using the following derivation $\mathcal{D}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{z : [m : O \to A], x : O\} \vdash z : [m : O \to A]}\,(\text{Val x})}{\{z : [m : O \to A], x : O\} \vdash z.m : O \to A}\,(\text{Val Select}) \quad \dfrac{}{\{z : [m : O \to A], x : O\} \vdash x : O}\,(\text{Val x})}{\{z : [m : O \to A], x : O\} \vdash z.m(x) : A}\,(\text{Val App})}{\{z : [m : O \to A]\} \vdash \lambda x.z.m(x) : O \to A}\,(\text{Val Fun})}{\vdash [m = \varsigma(z : [m : O \to A]).\lambda x.z.m(x)] : [m : O \to A]}\,(\text{Val Object})$$

Then, we can fold this type up into the recursive type O and type the self application:

$$\dfrac{\dfrac{\dfrac{\boxed{\mathcal{D}}}{\vdash o : [m : O \to A]}}{\vdash o.m : O \to A}\,(\text{Val Select}) \quad \dfrac{\dfrac{\boxed{\mathcal{D}}}{\vdash o : [m : O \to A]}}{\vdash \mathsf{fold}(O, o) : O}\,(\text{Val Fold})}{\vdash o.m(\mathsf{fold}(O, o)) : A}\,(\text{Val App})$$

In fact since the ς binder represents an *implicit* form of recursion (similar to that represented by the class mechanism itself, which we shall discuss later in Section 10.3.4), we do not even need recursive types to derive this logical inconsistency in the ς-calculus.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{z : [m : A]\} \vdash z : [m : A]}\,(\text{Val x})}{\{z : [m : A]\} \vdash z.m : A}\,(\text{Val Select})}{\vdash [m = \varsigma(z : [m : A]).z.m] : [m : A]}\,(\text{Val Object})}{\vdash [m = \varsigma(z : [m : A]).z.m].m : A}\,(\text{val Select})$$

As a last example, we can also do the same thing in (nominally typed) FJ (and FJ$^{\mathfrak{C}}$) and Java. Recall the non-terminating program from Section 6.2. There, the class `NT` declared a `loop` method which called itself recursively on the receiver. Remember also that the method was declared to return a value of (class) type `NT`. In fact, we can declare this method to return *any* class type (as long as the class is declared in the class table), and the method will be well-typed.

8.2. Nakano's Logical Systems

Nakano defines a family of four related systems of recursive types for the λ-calculus [84], and introduces an *approximation* modality which essentially controls the folding of these recursive types. In this section, we will give a presentation of Nakano's family of type systems and discuss their main properties. The

family of systems can collectively be called $\lambda\bullet\mu$, and is characterised by a core set of type assignment rules. The four variants are named S-$\lambda\bullet\mu$, S-$\lambda\bullet\mu^+$, F-$\lambda\bullet\mu$ and F-$\lambda\bullet\mu^+$, and are defined by different subtyping relations.

8.2.1. The Type Systems

The type language of Nakano's systems is essentially that of Simply Typed Lambda Calculus, extended with recursive types and the $\bullet$ approximation modality (called "bullet"), which is a unary type constructor. Intuitively, this operator ensures that recursive references are 'well-behaved', and its ability to do so derives from the requirement that every recursive reference must occur within the scope of the approximation modality. Since this syntactic property is non-local, we must first define a set of *pretypes* (or pseudo type expressions, as Nakano calls them).

Definition 8.4 ($\lambda\bullet\mu$ Pretypes)**.** *1. The set of* $\lambda\bullet\mu$ pretypes *are defined by the following grammar:*

$$P, Q, T \quad ::= \quad X \quad | \quad \bullet P \quad | \quad P \to Q \quad | \quad \mu X\,.\,(P \to Q)$$

where X, Y, Z range over a denumerable set of type variables*.*

2. The notation $\bullet^n P$ *denotes the pretype* $\underbrace{\bullet \ldots \bullet}_{n \text{ times}} P$*, where* $n \geq 0$*.*

The type constructor μ is a *binder* and we can define the usual notion of free and bound occurrences of type variables. Also, for a pretype $\mu X\,.\,P$ we will call all bound occurrences of X in P *recursive* variables.

Certain types in $\lambda\bullet\mu$ are equivalent to the type ω of the intersection type discipline, and can be assigned to all terms. These types are called $\top$-variants.

Definition 8.5 ($\top$-Variants)**.** *1. A pretype P is an F-*$\top$*-variant if and only if P is of the form*

$$\bullet^{m_0} \mu X_1\,.\,\bullet^{m_1} \mu X_2 \ldots . \mu X_n\,.\,\bullet^{m_n} X_i$$

for some $n > 0$ *and* $1 \leq i \leq n$ *with* $m_i + \ldots + m_n > 0$*.*

2. Let $(\cdot)^*$ *be the following transformation on pretypes*[1]*:*

$$X^* = X \qquad (P \to Q)^* = Q^*$$
$$(\bullet X)^* = \bullet(X^*) \qquad (\mu X\,.\,P)^* = \mu X\,.\,P^*$$

Then a pretype P is an S-$\top$*-variant if and only if* P^* *is an F-*$\top$*-variant.*

3. We will use the constant $\top$ *to denote any F-*$\top$*-variant or S-*$\top$*-variant.*

The well-behavedness property on recursive references that we mentioned above is expressed formally through the notion of properness:

[1] Nakano uses the notation $\overline{P}$ to denote this transformation, however since we use this notation for another purpose, we have defined an alternative.

Definition 8.6 (Properness)**.** *A pretype P is called* F-proper *(respectively* S-proper*) in a type variable X whenever X occurs freely in P only (a) within the scope of the $\bullet$ type constructor; or (b) in a subexpression $Q \to T$ where T is an F-$\top$-variant (resp. S-$\top$-variant). We may simply write that a pretype is* proper *in X when it is clear from the context whether we mean F-proper or S-proper.*

The types of $\lambda\bullet\mu$ are those pretypes which are proper in all their recursive type variables.

Definition 8.7 ($\lambda\bullet\mu$ Types)**.** *The set of F- (respectively S-)* types *consists of those pretypes P such that P is F-proper (resp. S-proper) in X for all of its subexpressions of the form $\mu X\,.\,Q$. The metavariables A, B, C, D will be used to range over types* only.

Types are considered modulo α-equivalence (renaming of type variables respecting μ-bindings), and the notation $A[B/X]$, as usual, stands for the type A in which all the (free) occurrences of X have been replaced by the type B.

An equivalence relation is given for each set of $\lambda\bullet\mu$ types.

Definition 8.8 ($\lambda\bullet\mu$ Type Equivalence)**.** *The equivalence relation $\simeq$ on F-types (respectively, S-types) is defined as the smallest such equivalence relation (i.e. reflexive, transitive and symmetric) satisfying the following conditions:*

($\simeq$-$\bullet$) *If $A \simeq B$ then $\bullet A \simeq \bullet B$.*
($\simeq$-$\to$) *If $A \simeq B$ and $C \simeq D$ then $A \to C \simeq B \to D$.*
($\simeq$-fix) $\mu X\,.\,A \simeq A[\mu X\,.\,A/X]$.
($\simeq$-uniq) *If $A \simeq B[A/X]$ and B is (F/S-)proper in X, then $A \simeq \mu X\,.\,B$.*

where the equivalence relation on F-types satisfies the additional condition:

($\simeq$-$\top$) $A \to \top \simeq B \to \top$ *(for all F-$\lambda\bullet\mu$ types A and B).*

and the equivalence relation on S-types satisfies the additional condition:

($\simeq$-$\top$) $A \to \top \simeq \top$ *(for all S-$\lambda\bullet\mu$ types A).*

Nakano remarks that two types are equivalent according to this relation whenever their possibly infinite unfolding (according to the ($\simeq$ -fix) rule above) is the same. He does not explicitly define types to be infinite expressions which is what would be required for his remark to hold true. However, it seems obvious from his remark that this is the implicit intention in the definition. As we mentioned in the previous section when considering the system λ_μ of [34], we may define types to be either finite or infinite expressions. If one only allows type expressions to be finite, then the notion of equality given by $\simeq$ is called *weak* and, conversely, if one allows type expressions to be infinite then $\simeq$ is called *strong*. In the following chapter, when we define a type inference procedure for Nakano's systems, we use a notion of weak equivalence.

The approximation modality induces a subtyping relation $\preceq$ for each of the four systems, which Nakano defines in the style of Amadio and Cardelli [5] using a derivability relation on subtyping *judgements*.

Definition 8.9 (Subtyping Relation)**.** *1. a subtyping* statement *is of the form $A \preceq B$.*

2. *A subtyping* assumption *γ is a set of subtyping statements $X \preceq Y$ (that is the types in the statement are* variables*, and for each such statement in γ, X and Y do not appear in any other statement in γ. We write $\gamma_1 \cup \gamma_2$ only when γ_1 and γ_2 are subtyping assumptions and their union is also a (valid) subtyping assumption.*

3. *A subtyping* judgement *is of the form $\gamma \vdash A \preceq B$. Valid subtyping judgements are derived by the following derivation rules:*

$$(\preceq\text{-}assump): \frac{}{\gamma \cup \{X \preceq Y\} \vdash X \preceq Y} \qquad (\preceq\text{-}\top): \frac{}{\gamma \vdash A \preceq \top}$$

$$(\preceq\text{-}approx): \frac{}{\gamma \vdash A \preceq \bullet A} \qquad (\preceq\text{-}reflex): \frac{}{\gamma \vdash A \preceq B}\,(A \simeq B)$$

$$(\preceq\text{-}trans): \frac{\gamma_1 \vdash A \preceq B \quad \gamma_2 \vdash B \preceq C}{\gamma_1 \cup \gamma_2 \vdash A \preceq C}$$

$$(\preceq\text{-}\bullet): \frac{\gamma \vdash A \preceq B}{\gamma \vdash \bullet A \preceq \bullet B} \qquad (\preceq\text{-}\to): \frac{\gamma_1 \vdash C \preceq A \quad \gamma_2 \vdash B \preceq D}{\gamma_1 \cup \gamma_2 \vdash A \to B \preceq C \to D}$$

$$(\preceq\text{-}\mu): \frac{\gamma \cup \{X \preceq Y\} \vdash A \preceq B}{\gamma \vdash \mu X.A \preceq \mu Y.B}\begin{pmatrix} X, Y \text{ do not occur free in } A, B \text{ resp.} \\ A \text{ and } B \text{ proper in } X, Y \text{ resp.}\end{pmatrix}$$

where, for the systems F-$\lambda\bullet\mu$ and F-$\lambda\bullet\mu^+$ (respectively S-$\lambda\bullet\mu$ and S-$\lambda\bullet\mu^+$), $\top$ ranges over F-$\top$ variants (respectively S-$\top$-variants) and $\simeq$ is the equivalence relation on F-types (respectively S-types); and additionally:

a) the subtyping relation for the systems F-$\lambda\bullet\mu$ and F-$\lambda\bullet\mu^+$ satisfies the rule:

$$(\preceq\text{-}\to\bullet): \frac{}{\gamma \vdash A \to B \preceq \bullet A \to \bullet B}$$

b) the subtyping relation for the systems S-$\lambda\bullet\mu$ and S-$\lambda\bullet\mu^+$ satisfies the rule:

$$(\preceq\text{-}\to\bullet): \frac{}{\gamma \vdash \bullet(A \to B) \preceq \bullet A \to \bullet B}$$

c) the subtyping relation for the systems F-$\lambda\bullet\mu^+$ and S-$\lambda\bullet\mu^+$ satisfies the rule:

$$(\preceq\text{-}\to\bullet): \frac{}{\gamma \vdash \bullet A \to \bullet B \preceq \bullet(A \to B)}$$

4. *We write $A \preceq B$ whenever $\vdash A \preceq B$ is a valid subtyping judgement.*

F- and S-types are assigned to λ-terms as follows.

Definition 8.10 ($\lambda\bullet\mu$ Type Assignment)**.** *1. An F-type (respectively S-type) statement is of the form $M : A$ where M is a λ-term and A is an F-type (resp. S-type). The λ-term M is called the* subject *of the statement.*

2. *An F-type (respectively S-type) environment Γ is a set of F-type (resp, S-type) statements in which the subject of each statement is a term variable, and is also* unique*. We write $\Gamma, x : A$ for the F-type (resp. S-type) environment $\Gamma \cup \{x : A\}$ where x does not appear as the subject of any statement in Γ. If $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$, then $\bullet\Gamma$ denotes the type environment $\{x_1 : \bullet A_1, \ldots, x_n : \bullet A_n\}$.*

3. *Type assignment $\vdash$ in the systems F-$\lambda\bullet\mu$ and F-$\lambda\bullet\mu^+$ (respectively S-$\lambda\bullet\mu$ and S-$\lambda\bullet\mu^+$) is a relation*

between F-type (resp. S-type) environments and F-type (resp. S-type) statements. It is defined by the following natural deduction rules:

$$(var): \frac{}{\Gamma, x:A \vdash x:A} \qquad (\top): \frac{}{\Gamma \vdash M:\top}$$

$$(nec): \frac{\Gamma \vdash M:A}{\bullet\Gamma \vdash M:\bullet A} \qquad (\preceq): \frac{\Gamma \vdash M:A}{\Gamma \vdash M:B}\,(A \preceq B)$$

$$(\to I): \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M:A \to B} \qquad (\to E): \frac{\Gamma \vdash M:\bullet^n(A \to B) \quad \Gamma \vdash N:\bullet^n A}{\Gamma \vdash MN:\bullet^n B}$$

where $\top$ ranges over F-$\top$-variants (resp. S-$\top$-variants) and the subtyping relation in the (*sub*) *rule is appropriate to the system being defined. Furthermore, the system F-$\lambda\bullet\mu^+$ (resp. S-$\lambda\bullet\mu^+$) has the following additional rule:*

$$(\bullet): \frac{\bullet\Gamma \vdash M:\bullet A}{\Gamma \vdash M:A}$$

Notice that in the system S-$\lambda\bullet\mu$ and its extension S-$\lambda\bullet\mu^+$, since the subtyping relation gives us $\bullet(A \to B) \preceq \bullet A \to \bullet B$, the rule for application can be simplified to its standard form:

$$\frac{\Gamma \vdash M:A \to B \quad \Gamma \vdash N:A}{\Gamma \vdash MN:B}$$

Also, in the systems F-$\lambda\bullet\mu^+$ and S-$\lambda\bullet\mu^+$ we can show that the (nec) rule is redundant.

Nakano motivates these different systems by giving a realizability interpretation of types over various classes of Kripke frames, into models of the untyped λ-calculus. The reason for calling the systems F-$\lambda\bullet\mu$ and S-$\lambda\bullet\mu$ then becomes clear, since the semantics of these systems corresponds, respectively, to the F-semantics and the Simple semantics of types (cf. [62]). The precise details of these semantics are not immediately relevant to the research in this thesis, and so we will not discuss them here. The interested reader is referred to [82, 84]. The important feature of the semantics, however, is that they allow to show a number of convergence results for typeable terms, which we describe next.

8.2.2. Convergence Properties

Definition 8.11 (Tail Finite Types)**.** *A type A is* tail finite *if and only if*

$$A \simeq \bullet^{m_1}(B_1 \to \bullet^{m_1}(B_2 \to \ldots \bullet^{m_n} B_n \to X))$$

for some $n, m_1, \ldots, m_n \geq 0$ and types $B_1, \ldots, B_n$ and type variable X.

Using this notion of tail finiteness, we can state some convergence properties of typeable terms in Nakano's systems.

Theorem 8.12 (Convergence [84, Theorem 2])**.** *Let $\Gamma \vdash M : A$ be derivable in any of the systems F-$\lambda\bullet\mu$, F-$\lambda\bullet\mu^+$, S-$\lambda\bullet\mu$ or S-$\lambda\bullet\mu^+$, and let $\Gamma \vdash N : B$ be derivable in either F-$\lambda\bullet\mu$ or F-$\lambda\bullet\mu^+$; then*

1. *if A is tail finite, then M is head normalisable.*

2. *if $B \neq \top$ then N is weakly head normalisable (i.e. reduces to a λ-abstraction).*

To provide some intuition as to why typeability in Nakano's systems entails these convergence properties, let us consider how we might try and modify the derivation of the unsolvable term $(\lambda x.xx)(\lambda x.xx)$ given in Section 8.1 to be a valid derivation in Nakano's type assignment systems. The crucial element is that the type $\mu X.(X \to A)$ is now no longer well-formed since the recursive variable X does not occur under the scope of the $\bullet$ type constructor. Let us modify it, then, as follows, and let $B' = \mu X.(\bullet X \to A)$. Now notice that we may only assign the type $B' \to A$ to the term $\lambda x.xx$:

$$\dfrac{\dfrac{\dfrac{}{x : B' \vdash x : B'}\,(\text{var})}{x : B' \vdash x : \bullet B' \to A}\,(\preceq) \qquad \dfrac{\dfrac{}{x : B' \vdash x : B'}\,(\text{var})}{x : B' \vdash x : \bullet B'}\,(\preceq)}{\dfrac{x : B' \vdash xx : A}{\vdash \lambda x.xx : B' \to A}\,(\to I)}\,(\to E)$$

The unfolding of the type B' is $\bullet B' \to A$; notice that we have $\bullet B' \to A \preceq B' \to A$ but *not* the converse. Therefore, we cannot 'fold' the type $B' \to A$ back up into the type B' in order to type the application of $\lambda x.xx$ to itself. We could try adding a bullet to the type assumption for x, but this does not get us very far, as then we will have to derive the type statement $\lambda x.xx : \bullet B' \to \bullet A$:

$$\dfrac{\dfrac{\dfrac{}{x : \bullet B' \vdash x : \bullet B'}\,(\text{var})}{x : B' \vdash x : \bullet(\bullet B' \to A)}\,(\preceq) \qquad \dfrac{\dfrac{}{x : \bullet B' \vdash x : \bullet B'}\,(\text{var})}{x : \bullet B' \vdash x : \bullet\bullet B'}\,(\preceq)}{\dfrac{x : \bullet B' \vdash xx : \bullet A}{\vdash \lambda x.xx : \bullet B' \to \bullet A}\,(\to I)}\,(\to E)$$

and again, the subtyping relation gives us $\bullet B' \to A \preceq \bullet B' \to \bullet A$, but not the converse. Notice also that $\bullet B' \to \bullet A \preceq \bullet(B' \to A)$, thus we may only derive *supertypes* of $\bullet B' \to A$, and so we will never be able to fold up the type we derive into the type B' itself. It is for this reason that we describe the approximation modality $\bullet$ as controlling the *folding* of recursive types.

This also shows why we call Nakano's systems 'logical'. Since we cannot assign types (other than $\top$) to terms such as $(\lambda x.xx)(\lambda x.xx)$, there are now no longer terms for which any type A can be derived. In other words, viewing the type system as a logic, it is not possible to derive all formulas. In [84], Nakano explores the notion of his type systems as modal logics and makes the observation that, viewed as such, they are extensions of the intuitionistic logic of provability GL [23].

8.2.3. A Type for Fixed-Point Operators

After its logical character and convergence properties, the most important feature of the $\lambda\bullet\mu$ type systems for our work is that terms which are *fixed-point combinators* (cf. Section 6.5) have the characteristic type scheme $(\bullet A \to A) \to A$. This can be illustrated using Curry's fixed-point operator $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ and the following derivation, which is valid in each of the four systems we have described above, Let $\mathcal{D}$ be the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{f:\bullet A\to A, x:\bullet B'\}\vdash x:\bullet B'}(\text{var})}{\{f:\bullet A\to A, x:\bullet B'\}\vdash x:\bullet(\bullet B'\to A)}(\preceq) \quad \dfrac{\dfrac{}{\{f:\bullet A\to A, x:\bullet B'\}\vdash x:\bullet B'}(\text{var})}{\{f:\bullet A\to A, x:\bullet B'\}\vdash x:\bullet\bullet B'}(\preceq)}{\{f:\bullet A\to A, x:\bullet B'\}\vdash xx:\bullet A}(\to E) \quad \vdots}{\{f:\bullet A\to A, x:\bullet B'\}\vdash f(xx):A}(\to E)}{\{f:\bullet A\to A\}\vdash \lambda x.f(xx):\bullet B'\to A}(\to I)$$

where $B' = \mu X.(\bullet X \to A)$ is the type that we considered above. Then we can derive:

$$\dfrac{\dfrac{\boxed{\mathcal{D}}}{\{f:(\bullet A\to A)\}\vdash \lambda x.f(xx):\bullet B'\to A} \quad \dfrac{\boxed{\mathcal{D}}}{\{f:(\bullet A\to A)\}\vdash \lambda x.f(xx):\bullet B'}(\preceq)}{\{f:(\bullet A\to A)\}\vdash (\lambda x.f(xx))(\lambda x.f(xx)):A}(\to E)}{\vdash \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)):(\bullet A\to A)\to A}(\to I)$$

The powerful corollary to this result is that this allows us to give a logical, type-based treatment to recursion, and more specifically, to recursively defined *classes*. However, before describing how Nakano's approach can be applied in the object-oriented setting, in the following chapter we will consider a type inference procedure for Nakano's systems.

One final remark that we will make first, though, concerns Nakano's definition of $\top$-variants in the different systems. We point out that Nakano's definition distinguishes each of the type schemes $\mu X.(A\to\bullet X)$, $A\to\top$ and $\top$ in the F-$\lambda\bullet\mu$ systems but *not* in the S-$\lambda\bullet\mu$ systems. It is for this reason, essentially, that the F-systems can give weak head normalisation guarantees whereas the S-systems cannot, as the first two of these types can be assigned to weakly head normalisable terms that do not have head normal forms:

$$\dfrac{\boxed{\vdash \mathbf{Y}:(\bullet\mu X.(A\to\bullet X)\to\mu X.(A\to\bullet X))\to\mu X.(A\to\bullet X)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\{x:\bullet\mu X.(A\to\bullet X), y:A\}\vdash x:\bullet\mu X.(A\to\bullet X)}(\text{var})}{\{x:\bullet\mu X.(A\to\bullet X)\}\vdash \lambda y.x:A\to\bullet\mu X.(A\to\bullet X)}(\to I)}{\vdash \lambda xy.x:\bullet\mu X.(A\to\bullet X)\to A\to\bullet\mu X.(A\to\bullet X)}(\to I)}{\vdash \lambda xy.x:\bullet\mu X.(A\to\bullet X)\to\mu X.(A\to\bullet X)}(\preceq)}{\vdash \mathbf{Y}(\lambda xy.x):\mu X.(A\to\bullet X)}(\to E)$$

$$\dfrac{\dfrac{}{\{y:A\}\vdash(\lambda x.xx)(\lambda x.xx):\top}(\top)}{\vdash\lambda y.(\lambda x.xx)(\lambda x.xx):A\to\top}(\to I)$$

We do not see the necessity of making this distinction for the two systems, from a semantic point of view. We believe that by adopting a uniform definition for $\top$-variants across all the systems, the S-$\lambda\bullet\mu$ systems could also enjoy weak head normalisation. In the following chapter, we will use such a system when formulating a type inference procedure, since we would like to distinguish the type $\mu X.(A \to \bullet X)$ from $\top$, while being able to rely on the equivalence $\bullet(A \to B) \simeq \bullet A \to \bullet B$.

Indeed, the first term we have typed above is a crucial example in demonstrating the application of this approach to oo, since it corresponds to the self-returning object that we considered in Section 6.1. Notice that we may assign to this term the more particular type $\mu X.(\top \to \bullet X)$, and this in turn allows us to type, with that *same* type, any application of the form $\mathbf{Y}(\lambda xy.x)M_1 \dots M_n$, for *arbitrarily large* values of n. This type analysis reflects the fact that the term has the reduction behaviour $\mathbf{Y}(\lambda xy.x)M_1 \dots M_n \to^* \mathbf{Y}(\lambda xy.x)$ for any n. Compare this with the behaviour of the self-returning object which has the reduction behaviour `new SR().self() ... .self()` $\to^*$ `new SR()` for any number of consecutive invocations of the `self` method. That we can draw this parallel between a (conventionally) 'meaningless' term in λ-calculus and a meaningful term in an object-oriented model should not come as a great surprise since, as we remarked in Section 6.5, when we interpret λ-calculus in systems with weak reduction, such terms become meaningful.

9. Type Inference for Nakano's System

In this chapter, we will present an algorithm which we claim decides if a term is typeable in Nakano's type system (or rather, the type system S-$\lambda \bullet \mu^+$ strengthened by assuming the definition for F-$\top$-variants rather than S-$\top$-variants). Our algorithm is actually based on a *variation* of Nakano's system, the main feature of which is the introduction of a new set of (type) variables, which we name *insertion* variables. These variables actually act us unary type constructors, and are designed to allow extra bullets to be inserted into types during unification. To support this intended functionality for insertion variables, we define an operation called *insertion*. Insertions can be viewed as an analogue, or parallel, to the operation of *substitution* which replaces ordinary type variables. Similarities can also be drawn with the *expansion* variables of Kfoury and Wells [74, 75]. It is this operation of insertion (mediated via insertion variables) which makes the type inference possible, thus insertion variables really play a key role. This is discussed more fully with examples towards the end of the chapter.

We also make some other minor modifications to Nakano's system. The most obvious one is that we define recursive types using de Bruijn indices instead of explicitly naming the (recursive) type variables which are bound by the μ type constructor; we do this in order to avoid having to deal with α-conversion during unification. Lastly, to simplify the formalism at this early stage of development, we do not consider a 'top' type. Reincorporating the top type is an objective for future research.

An important remark to make regarding our type inference procedure is that it is *unification*-based: typings are first inferred for subterms and the algorithm then searches for operations on the types they contain such that applying the operations to the typings makes them equal. This leads to type inference since the operations are sound with respect to the type assignment system - in other words, the operations on the types actually correspond to operations on the typing derivations themselves. This approach contrasts with the *constraint*-based approach to type inference in which sets of typing constraints are constructed for each subterm and then combined. Thus the algorithm infers constraint sets rather than typings, the solution of which implies and provides a (principal) typing for the term. It is this latter approach that is employed by Kfoury and Wells [75], for example, as well as Boudol [24], in their type inference algorithms for rr_D, by Palsberg and others [86, 71] in their system of (non-logical) recursive types for λ-calculus, and also for many type inference algorithms for object-oriented type systems [90, 51, 52, 85, 106, 29, 6].

The two approaches to type inference are, in effect, equivalent in the sense that two types are unifiable if and only if an appropriate set of constraints is solvable. One can view the unification-based approach as solving the constraints 'on the fly', as they are generated, while the constraint-based approach collects all the constraints together first and then solves them all at the end. One might have a better understanding of one over the other, or find one or the other more intuitive - it is largely a matter of personal taste. We find the unification-based approach the more intuitive, which is the primary (or perhaps the sole) reason for this research taking that direction.

The aim in defining the following type system, and associated inference procedures, is to show that

type inference for Nakano's system is decidable. Our work is at an early stage and, as such, we do not give proofs for many propositions in this chapter. Therefore, we do not claim a formal result, but instead present our work in this chapter as a proof *sketch* of the intended results.

9.1. Types

We define a set of pretypes, constructed from two set of variables (ordinary type variables, and insertion variables) and Nakano's approximation type operator, as well as the familiar arrow, or function, type constructor. We also have recursive types, which we formulate in an α-independent fashion using de Bruijn indices.

Definition 9.1 (Pretypes)**.** *1. The set of* pretypes *(ranged over by π), and its (strict) subset of* functional *pretypes (ranged over by ϕ) are defined by the following grammar, where de Bruijn indices* $\mathbf{n}$ *range over the set of natural numbers, φ ranges over a denumerable set of type variables, and ι ranges over a denumerable set of insertion variables:*

$$\begin{array}{lllllllll} \pi & ::= & \varphi & | & \mathbf{n} & | & \bullet\pi & | & \iota\pi & | & \phi \\ \phi & ::= & & \pi_1 \to \pi_2 & & | & \bullet\phi & | & \iota\phi & | & \mu.\phi \end{array}$$

2. *We use the shorthand notation $\bullet^n\pi$ (where $n \geq 0$) to denote the pretype π prefixed by n occurrences of the $\bullet$ operator, i.e. $\underbrace{\bullet \ldots \bullet}_{n \text{ times}}\pi$.*
3. *We use the shorthand notation $\iota_n\pi$ (where $n \geq 0$) to denote the pretype π prefixed by each ι_k in turn, i.e. $\iota_1 \ldots \iota_n\pi$.*

We also define the following functions which return various different sets of variables that occur in a pretype.

Definition 9.2 (Type Variable Set)**.** *The function* TV *takes a pretype π and returns the set of type variables occurring in it. It is defined inductively on the structure of pretypes as follows:*

$$\begin{array}{rlrl} \text{TV}(\varphi) &= \{\varphi\} & \text{TV}(\iota\pi) &= \text{TV}(\pi) \\ \text{TV}(\mathbf{n}) &= \emptyset & \text{TV}(\pi_1 \to \pi_2) &= \text{TV}(\pi_1) \cup \text{TV}(\pi_2) \\ \text{TV}(\bullet\pi) &= \text{TV}(\pi) & \text{TV}(\mu.\phi) &= \text{TV}(\phi) \end{array}$$

Definition 9.3 (Decrement Operation)**.** *If X is a set of de Bruijn indices (i.e. natural numbers) then the set $X\downarrow$ is defined by $X\downarrow = \{\mathbf{n} \mid \mathbf{n}+1 \in X\}$. That is, all the de Bruijn indices have been decremented by* 1*.*

Definition 9.4 (Free Variable Set)**.** *The function* FV *takes a pretype π and returns the set of de Bruijn indices representing the free recursive 'variables' of π. It is defined inductively on the structure of pretypes as follows:*

$$\begin{array}{rlrl} \text{FV}(\varphi) &= \emptyset & \text{FV}(\iota\pi) &= \text{FV}(\pi) \\ \text{FV}(\mathbf{n}) &= \{\mathbf{n}\} & \text{FV}(\pi_1 \to \pi_2) &= \text{FV}(\pi_1) \cup \text{FV}(\pi_2) \\ \text{FV}(\bullet\pi) &= \text{FV}(\pi) & \text{FV}(\mu.\phi) &= \text{FV}(\phi)\downarrow \end{array}$$

We say that a pretype π is closed *when it contains no free recursive variables, i.e.* $\text{FV}(\pi) = \emptyset$.

Definition 9.5 (Raw Variable Set)**.** *1. The function* RAW_μ *takes a pretype π and returns the set of its* raw *recursive variables - those recursive variables (i.e. de Bruijn indices) occurring in π which do not occur within the scope of a $\bullet$. It is defined inductively on the structure of pretypes as follows:*

$$\begin{aligned} \text{RAW}_\mu(\varphi) &= \emptyset & \text{RAW}_\mu(\iota\,\pi) &= \text{RAW}_\mu(\pi) \\ \text{RAW}_\mu(\mathbf{n}) &= \{\mathbf{n}\} & \text{RAW}_\mu(\pi_1 \to \pi_2) &= \text{RAW}_\mu(\pi_1) \cup \text{RAW}_\mu(\pi_2) \\ \text{RAW}_\mu(\bullet\,\pi) &= \emptyset & \text{RAW}_\mu(\mu.\phi) &= \text{RAW}_\mu(\phi)\downarrow \end{aligned}$$

2. The function RAW_φ *takes a pretype π and returns the set of its raw* type *variables - the set of type variables occurring in π which do not occur within the scope of* either *a bullet* or *an insertion variable. It is defined inductively on the structure of pretypes as follows:*

$$\begin{aligned} \text{RAW}_\varphi(\varphi) &= \{\varphi\} & \text{RAW}_\varphi(\iota\,\pi) &= \emptyset \\ \text{RAW}_\varphi(\mathbf{n}) &= \emptyset & \text{RAW}_\varphi(\pi_1 \to \pi_2) &= \text{RAW}_\varphi(\pi_1) \cup \text{RAW}_\varphi(\pi_2) \\ \text{RAW}_\varphi(\bullet\,\pi) &= \emptyset & \text{RAW}_\varphi(\mu.\phi) &= \text{RAW}_\varphi(\phi) \end{aligned}$$

We will now use this concept of 'raw' (recursive) variables to impose an extra property, called *adequacy*, on pretypes which will be a necessary condition for considering a pretype to be a true type. We have also extended the concept of rawness to ordinary type variables, although we have relaxed the notion slightly - a type variable is only considered raw when it does not fall under the scope of *either* a bullet *or* an insertion variable. This is because later, when we come to define a unification procedure for types, we will want to ensure that certain type variables always fall under the scope of a bullet. Because we will also define an operation that converts insertion variables into bullets, it will be sufficient for those given type variables to fall under the scope of either a bullet or an insertion variable.

Our notion of adequacy is equivalent to Nakano's notion of properness (see previous chapter).

Definition 9.6 (Adequacy)**.** *The set of* adequate *pretypes are those pretypes for which every μ binder binds at least one occurrence of its associated recursive variable, and every* bound *recursive variable occurs within the scope of a $\bullet$. It is defined as the smallest set of pretypes satisfying the following conditions:*

1. *φ is adequate, for all φ;*
2. *$\mathbf{n}$ is adequate, for all $\mathbf{n}$;*
3. *if π is adequate, then so are $\bullet\,\pi$ and $\iota\,\pi$;*
4. *if π_1 and π_2 are both adequate, then so is $\pi_1 \to \pi_2$;*
5. *if ϕ is adequate and $\mathbf{0} \in \text{FV}(\phi) \setminus \text{RAW}_\mu(\phi)$, then $\mu.\phi$ is adequate.*

Definition 9.7 (Types)**.** *We call a pretype π a* type *whenever it is* both *adequate* and *closed. The set of types is thus a (strict) subset of the set of pretypes.*

The following substitution operation allows us to formally describe how recursive types are folded and unfolded, and thus also plays a role in the definition of the subtyping relation.

Definition 9.8 (μ-substitution)**.** *A μ-substitution is a function from pretypes to pretypes. Let ϕ be a functional pretype, then the μ-substitution $[\mathbf{n} \mapsto \mu.\phi]$ is defined by induction on the structure of pretypes*

simultaneously for every $\mathbf{n} \in \mathbb{N}$ *as follows:*

$$[\mathbf{n} \mapsto \mu.\phi](\varphi) = \varphi$$

$$[\mathbf{n} \mapsto \mu.\phi](\mathbf{n}') = \begin{cases} \mu.\phi & \text{if } \mathbf{n} = \mathbf{n}' \\ \mathbf{n}' & \text{otherwise} \end{cases}$$

$$[\mathbf{n} \mapsto \mu.\phi](\bullet\pi) = \bullet([\mathbf{n} \mapsto \mu.\phi](\pi))$$

$$[\mathbf{n} \mapsto \mu.\phi](\iota\pi) = \iota([\mathbf{n} \mapsto \mu.\phi](\pi))$$

$$[\mathbf{n} \mapsto \mu.\phi](\pi_1 \to \pi_2) = ([\mathbf{n} \mapsto \mu.\phi](\pi_1)) \to ([\mathbf{n} \mapsto \mu.\phi](\pi_2))$$

$$[\mathbf{n} \mapsto \mu.\phi](\mu.\phi') = \mu.([\mathbf{n}+1 \mapsto \mu.\phi](\phi'))$$

Notice that μ-substitution has no effect on *types* since they are *closed*.

Lemma 9.9. *Let* $[\mathbf{n} \mapsto \mu.\phi]$ *be a* μ*-substitution and* π *be a pretype such that* $\mathbf{n} \notin \text{FV}(\sigma)$*, then* $[\mathbf{n} \mapsto \mu.\phi](\pi) = \pi$.

Proof. By straightforward induction on the structure of pretypes. □

Corollary 9.10. *Let* $[\mathbf{n} \mapsto \mu.\phi]$ *be any* μ*-substitution and* σ *be any type, then the following equation holds:* $[\mathbf{n} \mapsto \mu.\phi](\sigma) = \sigma$.

Proof. Since σ is a type, it follows from Definition 9.7 that $\text{FV}(\sigma) = \emptyset$, thus trivially $\mathbf{n} \notin \text{FV}(\sigma)$. Then the result follows immediately by Lemma 9.9. □

We now define a *subtyping* relation on pretypes. As we mentioned at the end of the previous chapter and in the introduction to the current one, our subtyping relation is based on the subtyping relation for the system S-$\lambda\bullet\mu^+$, so we have the equivalence $\bullet(\sigma \to \tau) \simeq \bullet\sigma \to \bullet\tau$. The rules defining our subtyping relation are thus a simple extension of Nakano's to apply to insertion variables as well as the $\bullet$ operator.

Definition 9.11 (Subtyping)**.** *The subtype relation* $\leq$ *on pretypes is defined as the smallest preorder on pretypes satisfying the following conditions:*

$$\pi \leq \bullet\pi \qquad \pi \leq \iota\pi \qquad\qquad \pi_1 \leq \pi_2 \Rightarrow \begin{cases} \bullet\pi_1 \leq \bullet\pi_2 \\ \iota\pi_1 \leq \iota\pi_2 \end{cases}$$

$$\bullet\iota\pi \leq \iota\bullet\pi \qquad \iota\bullet\pi \leq \bullet\iota\pi \qquad\qquad \iota_1\iota_2\pi \leq \iota_2\iota_1\pi$$

$$\bullet(\pi_1 \to \pi_2) \leq \bullet\pi_1 \to \bullet\pi_2 \qquad \bullet\pi_1 \to \bullet\pi_2 \leq \bullet(\pi_1 \to \pi_2)$$

$$\iota(\pi_1 \to \pi_2) \leq \iota\pi_1 \to \iota\pi_2 \qquad \iota\pi_1 \to \iota\pi_2 \leq \iota(\pi_1 \to \pi_2)$$

$$\mu.\phi \leq [\mathbf{0} \mapsto \mu.\phi](\phi) \qquad [\mathbf{0} \mapsto \mu.\phi](\phi) \leq \mu.\phi$$

$$\phi_1 \leq \phi_2 \Rightarrow \mu.\phi_1 \leq \mu.\phi_2$$

$$\pi_1' \leq \pi_1 \;\&\; \pi_2 \leq \pi_2' \Rightarrow \pi_1 \to \pi_2 \leq \pi_1' \to \pi_2'$$

We write $\pi_1 \simeq \pi_2$ whenever both $\pi_1 \leq \pi_2$ and $\pi_2 \leq \pi_1$.

The following properties hold of the subtype relation.

Lemma 9.12. *1. If $\pi \leq \pi'$ then $\pi \leq \vec{\iota}\pi'$ and $\vec{\iota}\pi \leq \vec{\iota}\pi'$ for all sequences $\vec{\iota}$.*

2. If $\vec{\iota'}$ is a permutation of $\vec{\iota}$, then $\vec{\iota}\pi \simeq \vec{\iota}\pi$ for all pretypes π.

Proof. By Defintion 9.11. □

We now define a subset of pretypes by specifying a *canonical* form. This canonical form will play a central role in our type inference algorithm by allowing us to separate the *strucutral* content of a type from its *logical* content, as encoded in the bullets and insertion variables. If pretypes are seen as trees, then canonical pretypes are the trees in which all the bullets and insertion variables have been collected at the leaves (the type variables and de Bruijn indices), or at the head of μ-recursive types. As we will see in sections 9.4 and 9.5, this allows for a clean separation of the two orthogonal subproblems involved in unification and type inference.

Definition 9.13 (Canonical Types)**.** *1. The set of* canonical *pretypes (ranged over by κ), and its (strict) subsets of* exact *canonical pretypes (ranged over by ξ),* approximative *canonical pretypes (ranged over by α) and* partially *approximative canonical pretypes (ranged over by β) are defined by the following grammar:*

$$\begin{array}{lllll} \kappa & ::= & \beta & | & \kappa_1 \to \kappa_2 \\ \beta & ::= & \alpha & | & \iota\beta \\ \alpha & ::= & \xi & | & \bullet\alpha \\ \xi & ::= & \varphi & | & \mathbf{n} \qquad | \quad \mu.(\kappa_1 \to \kappa_2) \end{array}$$

2. Canonical types are canonical pretypes which are both adequate and closed.

The following lemma shows that our grammatical definition of canonicity defined above is adequate.

Lemma 9.14. *For every pretype π there exists a canonical pretype κ such that $\pi \simeq \kappa$.*

Proof. By straightforward induction on the structure of pretypes. □

9.2. Type Assignment

We will now define our variant of Nakano's type assignment. The type assignment rules are almost identical to those of Nakano's original system - the difference lies almost entirely in the type language and the subtyping relation. Nakano's original typing rules themselves are almost identical to the familiar type assignment rules for the λ-calculus: there is just one additional rule that deals with the approximation $\bullet$ type constructor. Similarly, our system, having added insertion variables, includes one extra rule which is simply the analogue of Nakano's rule, but for insertion variables.

Definition 9.15 (Type Environments)**.** *1. A type* statement *is of the form $M{:}\sigma$, where M is a λ-term and σ is a type. We call M the* subject *of the statement.*

2. *A type* environment Π *is a finite set of type statements such that the subject of each statement in* Π *is a* variable, *and is also* unique.

3. *We write* $x \in \Pi$ *if and only if there is a statement* $x{:}\sigma \in \Pi$. *Similarly, we write* $x \notin \Pi$ *if and only if there is* no *statement* $x{:}\sigma \in \Pi$.

4. *The notation* $\Pi, x{:}\sigma$ *denotes the type environment* $\Pi \cup \{x{:}\sigma\}$ *where* x *does not appear as the subject of any statement in* Π.

5. *The notation* $\bullet\Pi$ *denotes the type environment* $\{x{:}\bullet\sigma \mid x{:}\sigma \in \Pi\}$ *and similarly the environment* $\iota\Pi$ *denotes the type environment* $\{x{:}\iota\sigma \mid x{:}\sigma \in \Pi\}$.

6. *The subtyping relation is extended to type environments as follows:*

$$\Pi_2 \leq \Pi_1 \text{ if and only if } \forall x{:}\sigma \in \Pi_1 \,.\, \exists \tau \leq \sigma \,.\, x{:}\tau \in \Pi_2$$

Definition 9.16 (Type Assignment)**.** *Type* assignment $\Pi \vdash M{:}\sigma$ *is a relation between type environments and type statements. It is defined by the following natural deduction system:*

$$(\textsc{var}) : \frac{}{\Pi, x{:}\sigma \vdash x{:}\sigma} \qquad (\textsc{sub}) : \frac{\Pi \vdash M{:}\sigma}{\Pi \vdash M{:}\tau}\,(\sigma \leq \tau)$$

$$(\bullet) : \frac{\bullet\Pi \vdash M{:}\bullet\sigma}{\Pi \vdash M{:}\sigma} \qquad (\iota) : \frac{\iota\Pi \vdash M{:}\iota\sigma}{\Pi \vdash M{:}\sigma}$$

$$(\to I) : \frac{\Pi, x{:}\sigma \vdash M{:}\tau}{\Pi \vdash \lambda x.M{:}\sigma \to \tau} \qquad (\to E) : \frac{\Pi \vdash M{:}\sigma \to \tau \quad \Pi \vdash N{:}\sigma}{\Pi \vdash M\,N{:}\tau}$$

If $\Pi \vdash M{:}\sigma$ *holds, then we say that the term* M *can be* assigned *the type* σ *using the type environment* Π.

Lemma 9.17 (Weakening)**.** *Let* $\Pi_2 \leq \Pi_1$*; if* $\Pi_1 \vdash M{:}\sigma$ *then* $\Pi_2 \vdash M{:}\sigma$.

Proof. By straightforward induction on the structure of typing derivations. □

The following holds of type assignment in our system (notice that the result as stated for the $\bullet$ type constructor is shown in Nakano's paper, and its extension to insertion variables for our system also holds).

Lemma 9.18. *Let* Π_1 *and* Π_2 *be* disjoint *type environments (i.e. the set of subjects used in the statements of* Π_1 *is disjoint from the set of subjects used in the statements of* Π_2*); if* $\Pi_1 \cup \Pi_2 \vdash M{:}\sigma$ *is derivable, then so are* $\bullet\Pi_1 \cup \Pi_2 \vdash M{:}\bullet\sigma$ *and* $\iota\Pi_1 \cup \Pi_2 \vdash M{:}\iota\sigma$.

Proof. By induction on the structure of typing derivations. □

We claim the completeness of our system with respect to Nakano's original system S-$\lambda\bullet\mu^+$. We do not give a rigorous proof, which would include defining a translation from our types based on de Bruijn indices to Nakano's types using μ-bound type variables and also showing that subtyping is preserved via this translation. However, we appeal to the reader's intuition to see that this result holds: one can imagine defining a one-to-one mapping between de Bruijn indices and type variables, and using this mapping to define a translation of types. It should be easy to see that under such a translation, subtyping in the one system mirrors subtyping in the other. Nakano types do not, of course, include insertion variables, and

thus neither would their translation, however any type without insertion variables is also a type in our system. The result then follows since all the rules of Nakano's type system are contained in our system.

Proposition 9.19 (Completeness of Type Assignment)**.** *If a term M is typeable in Nakano's system S-$\lambda\bullet\mu^+$ without using $\top$-variants, then it is also typeable in our type assignment system of Definition 9.16.*

We will also claim the *soundness* of our system with respect to Nakano's, however in order to do this we will need to define some operations on types, which we will do in the following section.

9.3. Operations on Types

We are almost ready to define our unification and type inference procedures. However, in order to do so we will need to define a set of *operations* that transform (pre)types. We do so in this section. The operations include the familiar one of *substitution*, although we define a slight variant of the traditional notion which ensures (and, more importantly for our algorithm, preserves) the canonical structure of pretypes. We also define the new operation of *insertion*, which allows us to place bullets (and other insertion variables) in types by replacing insertion variables.

We begin by defining operations which push bullets innermost and insertion variables to the outermost occurrence along each path of a bullet or insertion variable.

Definition 9.20 (Push)**.** *1. The bullet pushing operation* bPush *is defined inductively on the structure of pretypes as follows:*

$$\begin{aligned}
\mathsf{bPush}(\varphi) &= \bullet\varphi \\
\mathsf{bPush}(\mathbf{n}) &= \bullet\mathbf{n} \\
\mathsf{bPush}(\bullet\pi) &= \bullet(\mathsf{bPush}(\pi)) \\
\mathsf{bPush}(\iota\pi) &= \iota(\mathsf{bPush}(\pi)) \\
\mathsf{bPush}(\pi_1 \to \pi_2) &= (\mathsf{bPush}(\pi_1)) \to (\mathsf{bPush}(\pi_2)) \\
\mathsf{bPush}(\mu.\phi) &= \bullet\mu.\phi
\end{aligned}$$

We use the shorthand notation $\mathsf{bPush}[n]$ *to denote the composition of* bPush *n times: formally, we define inductively over n:*

$$\begin{aligned}
\mathsf{bPush}[1] &= \mathsf{bPush} \\
\mathsf{bPush}[n+1] &= \mathsf{bPush} \circ \mathsf{bPush}[n]
\end{aligned}$$

with $\mathsf{bPush}[0]$ *denoting the identity function.*

2. For each insertion variable ι, the insertion variable pushing operation $\mathsf{iPush}[\iota]$ *is defined inductively over the structure of pretypes as follows:*

$$\begin{aligned}
\mathsf{iPush}[\iota](\varphi) &= \iota\varphi \\
\mathsf{iPush}[\iota](\mathbf{n}) &= \iota\mathbf{n}
\end{aligned}$$

$$\begin{aligned}
\mathsf{iPush}[\iota](\bullet\,\pi) &= \iota \bullet \pi \\
\mathsf{iPush}[\iota](\iota'\,\pi) &= \iota\,\iota'\,\pi \\
\mathsf{iPush}[\iota](\pi_1 \to \pi_2) &= (\mathsf{iPush}[\iota](\pi_1)) \to (\mathsf{iPush}[\iota](\pi_2)) \\
\mathsf{iPush}[\iota](\mu.\phi) &= \iota\,\mu.\phi
\end{aligned}$$

We use the notation $\mathsf{iPush}[\vec{\iota}_r]$ *(where* $r > 0$*) to denote the composition of each* $\mathsf{iPush}[\iota_k]$*, that is* $\mathsf{iPush}[\iota_1] \circ \ldots \circ \mathsf{iPush}[\iota_r]$*. The notation* $\mathsf{iPush}[\epsilon]$ *denotes the identity function on pretypes.*

We use this operation to define our *canonicalising* substitution operation.

Definition 9.21 (Canonicalising Type Substitution)**.** *A* canonicalising type substitution *is an operation on pretypes that replaces type variables by (canonical) pretypes, while at the same time converting the resulting type to a canonical form. Let* φ *be a type variable and* κ *be a canonical pretype; then the canonicalising type substitution* $[\varphi \mapsto \kappa]$ *is defined inductively on the structure of pretypes as follows:*

$$\begin{aligned}
[\varphi \mapsto \kappa](\varphi') &= \begin{cases} \kappa & \text{if } \varphi = \varphi' \\ \varphi' & \text{otherwise} \end{cases} \\
[\varphi \mapsto \kappa](\mathbf{n}) &= \mathbf{n} \\
[\varphi \mapsto \kappa](\bullet\,\pi) &= \mathsf{bPush}([\varphi \mapsto \kappa](\pi)) \\
[\varphi \mapsto \kappa](\iota\,\pi) &= \mathsf{iPush}[\iota]([\varphi \mapsto \kappa](\pi)) \\
[\varphi \mapsto \kappa](\pi_1 \to \pi_2) &= ([\varphi \mapsto \kappa](\pi_1)) \to ([\varphi \mapsto \kappa](\pi_2)) \\
[\varphi \mapsto \kappa](\mu.\phi) &= \mu.([\varphi \mapsto \kappa](\phi))
\end{aligned}$$

It is straightforward to show that the result of apply a canonicalising substitution is a canonical type.

Lemma 9.22. *1. Let* κ *be a canonical type; then* $\mathsf{bPush}(\kappa)$ *and* $\mathsf{iPush}(\kappa)$ *are both canonical types.*

2. Let π *be a type and* $[\varphi \mapsto \kappa]$ *be a canonicalising substitution; then* $[\varphi \mapsto \kappa](\pi)$ *is a canonical type.*

Proof. 1. By straightforward induction on the structure of canonical pretypes.

2. By straightforward induction on the structure of pretypes, using the first part for the cases where $\pi = \bullet\,\pi'$ and $\pi = \iota\,\pi'$. □

As we have already mentioned, the insertion operation replaces insertion variables by sequences of insertion variables and bullets. Insertions are needed for type *inference*, and in Section 9.6.1 we will discuss in detail why this is.

Definition 9.23 (Insertion)**.** *An* insertion I *is a function from pretypes to pretypes which inserts a number of insertion variables and/or bullets in to a pretype at specific locations by replacing insertion variables, and then canonicalises the resulting type. If* $\vec{\iota}$ *is a sequence of insertion variables, then the insertion* $[\iota \mapsto \vec{\iota} \bullet^r]$ *(where* $r \geq 0$*) is defined inductively over the structure of pretypes as follows:*

$$\begin{aligned}
[\iota \mapsto \vec{\iota} \bullet^r](\varphi) &= \varphi \\
[\iota \mapsto \vec{\iota} \bullet^r](\mathbf{n}) &= \mathbf{n} \\
[\iota \mapsto \vec{\iota} \bullet^r](\bullet\,\pi) &= \bullet([\iota \mapsto \vec{\iota} \bullet^r](\pi))
\end{aligned}$$

$$[\iota \mapsto \check{\iota}\bullet^r](\iota'\,\pi) = \begin{cases} \check{\iota}(\mathsf{bPush}[r]([\iota \mapsto \check{\iota}\bullet^r](\pi))) & \text{if } \iota = \iota' \\ \iota'([\iota \mapsto \check{\iota}\bullet^r](\pi)) & \text{otherwise} \end{cases}$$

$$[\iota \mapsto \check{\iota}\bullet^r](\pi_1 \to \pi_2) = ([\iota \mapsto \check{\iota}\bullet^r](\pi_1)) \to ([\iota \mapsto \check{\iota}\bullet^r](\pi_2))$$

$$[\iota \mapsto \check{\iota}\bullet^r](\mu.\phi) = \mu.([\iota \mapsto \check{\iota}\bullet^r](\phi))$$

We may write $[\iota \mapsto \check{\iota}]$ *for* $[\iota \mapsto \check{\iota}\bullet^r]$ *where* $r = 0$.

We now abstract each of the specific operations into a single concept.

Definition 9.24 (Operations)**.** *We define* operations O *as follows:*

1. *The identity function* Id *on pretypes is an operation;*
2. *Canonicalising type substitutions are operations;*
3. *Insertions are operations;*
4. *if* O_1 *and* O_2 *are operations, then so is their composition* $\mathsf{O}_2 \circ \mathsf{O}_1$*, where* $\mathsf{O}_2 \circ \mathsf{O}_1(\pi) = \mathsf{O}_2(\mathsf{O}_1(\pi))$ *for all pretypes* π.

The operations we have defined above should exhibit a number of soundness properties of these operations with respect to subtyping and type assignment. These soundness properties will be necessary in order to show the soundness of our unification and type inference procedures.

Proposition 9.25. *Let* O *be an operation; if* σ *is a type, then so is* $\mathsf{O}(\sigma)$.

Proof technique. The proof is by induction on the structure of pretypes. We must first show this holds for the operations bPush and iPush, and then we use this to show that it holds for each different kind of operation.

Proposition 9.26. *Let* O *be an operation, and* π_1, π_2 *be pretypes such that* $\pi_1 \leq \pi_2$*; then* $\mathsf{O}(\pi_1) \leq \mathsf{O}(\pi_2)$ *also holds.*

Proof technique. By induction on the definition of subtyping. Again, we must prove for the operations bPush and iPush first, and then for each kind of operation.

Most importantly, using these previous results, we would be able to show that operations are sound with respect to type assignment.

Proposition 9.27. *If* $\Pi \vdash M{:}\sigma$ *then* $\mathsf{O}(\Pi) \vdash M{:}\mathsf{O}(\sigma)$ *for all operations* O.

Proof technique. By induction on the structure of typing derivations. As before, we must show the result for bPush, iPush and each kind of operation in turn. The case for the subtyping rule (SUB) would the soundness result we formulated previously, Proposition 9.26.

We claim as a corollary of this, that our system is sound with respect to Nakano's system.

Proposition 9.28 (Soundness of Type Assignment)**.** *If the term M is typeable in system of Definition 9.16, then it is typeable in Nakano's system S-*$\lambda \bullet \mu^+$.

Proof technique. For any typing derivation, we can construct an operation which removes all the insertion variables from the types it contains - if $\{\iota_1, \ldots, \iota_n\}$ is the set of all insertion variables mentioned in the derivation, we simply construct the operation $\mathsf{O} = [\iota_1 \mapsto \epsilon] \circ \ldots \circ [\iota_n \mapsto \epsilon]$. Applying this operation to any type in the derivation would result in a type not containing any insertion variables, i.e. a straightforward Nakano type (modulo the translation between de Bruijn indices and μ-bound type variables discussed in the previous section). It is then unproblematic to show by induction on the structure of derivations in our type system that a typing derivation for the term exists in Nakano's system, as the structure of the rules in our variant of type assignment are identical to the rules of Nakano's system, apart from the (ι) rule, which is in any case obviated by the operation O since it removes all insertion variables.

9.4. A Decision Procedure for Subtyping

In this section we will give a procedure for deciding whether one type is a subtype of another. It will be defined on *canonical* types, which implies a decision procedure for all types since it is straightforward to find, for any given type, the canonical type to which it is equivalent. The procedure we will define is sound, but *incomplete*, so it returns either the answer "yes", or "unknown".

Our approach to deciding subtyping is to split the question into two orthogonal sub-questions: a *structural* one, and a *logical* one. The logical information of a type is encoded by the bullet constructor, while the structural information is captured using the function ($\to$) and recursive (μ) type constructors. The use of *canonical* types (in which bullets – and insertion variables – are pushed innermost) allows us to collect all the logical constraints into one place where they can be checked independently of the structural constraints. The structural part of the problem then turns out to be the same as that of for non-logical recursive types, which is shown to be decidable in [35]. The logical constraints boil down, in the end, to simple (in)equalities on natural numbers and sequences of insertion variables.

As in [35], we will define an inference system whose judgements assert that one pretype is a subtype of another which we will then show to be decidable. However, before we do this we will need to define a notion that allows us to check the logical constraints expressed by the insertion variables in a type.

Definition 9.29 (Permutation Suffix)**.** *Let $\vec{\iota}$ and $\vec{\iota'}$ be two sequences of insertion variables; if $\vec{\iota''}$ and $\vec{\iota'''}$ are permutations of $\vec{\iota}$ and $\vec{\iota'}$ respectively, such that $\vec{\iota'''}$ is a* suffix *of $\vec{\iota''}$ (i.e. $\vec{\iota''} = \vec{\iota''''} \cdot \vec{\iota'''}$ for some $\vec{\iota''''}$) then we say that $\vec{\iota'}$ is a* permutation suffix *of $\vec{\iota}$ and write $\vec{\iota} \sqsubseteq \vec{\iota'}$.*

Notice that the permutation suffix property is decidable since it can be computed by the following procedure. First, count the number of occurrences of each insertion variable in the sequences $\vec{\iota}$ and $\vec{\iota'}$. Secondly, check that each insertion variable occurs at least as often in $\vec{\iota}$ as it does in $\vec{\iota'}$. If this is the case, then $\vec{\iota} \sqsubseteq \vec{\iota'}$, otherwise not.

We can now define our subtyping inference system.

Definition 9.30 (Subtype Inference)**.**

1. *A subtyping judgement asserts that one (canonical) pretype is a subtype of another, and is of the form $\vdash \kappa_1 \leq \kappa_2$.*

2. *Valid subtyping judgements are derived using the following natural deduction inference system:*

$$(\textsc{st-var}): \quad \frac{}{\vdash \vec{\iota} \bullet^r \varphi \leq \vec{\iota'} \bullet^s \varphi} (r \leq s \;\&\; \vec{\iota'} \sqsubseteq \vec{\iota})$$

$$(\textsc{st-recvar}): \quad \frac{}{\vdash \vec{\iota}\bullet^r \mathbf{n} \leq \vec{\iota'}\bullet^s \mathbf{n}}\,(r \leq s \;\&\; \vec{\iota'} \sqsubseteq \vec{\iota})$$

$$(\textsc{st-fun}): \quad \frac{\vdash \kappa_1' \leq \kappa_1 \qquad \vdash \kappa_2 \leq \kappa_2'}{\vdash \kappa_1 \to \kappa_2 \leq \kappa_1' \to \kappa_2'}$$

$$(\textsc{st-recfun}): \quad \frac{\vdash \kappa_1 \to \kappa_2 \leq \kappa_1' \to \kappa_2'}{\vdash \vec{\iota}\bullet^r \mu.(\kappa_1 \to \kappa_2) \leq \vec{\iota'}\bullet^s \mu.(\kappa_1' \to \kappa_2')}\,(r \leq s \;\&\; \vec{\iota'} \sqsubseteq \vec{\iota})$$

$$(\textsc{st-unfold}\mathrm{L}): \quad \frac{\vdash \mathsf{iPush}[\vec{\iota}](\mathsf{bPush}[r]([\mathbf{0} \mapsto \mu.(\kappa_1 \to \kappa_2)](\kappa_1 \to \kappa_2))) \leq \kappa_1' \to \kappa_2'}{\vdash \vec{\iota}\bullet^r \mu.(\kappa_1 \to \kappa_2) \leq \kappa_1' \to \kappa_2'}$$

$$(\textsc{st-unfold}\mathrm{R}): \quad \frac{\vdash \kappa_1 \to \kappa_2 \leq \mathsf{iPush}[\vec{\iota}](\mathsf{bPush}[s]([\mathbf{0} \mapsto \mu.(\kappa_1' \to \kappa_2')](\kappa_1' \to \kappa_2')))}{\vdash \kappa_1 \to \kappa_2 \leq \vec{\iota}\bullet^s \mu.(\kappa_1' \to \kappa_2')}$$

3. *We will write* $\vdash \pi_1 \simeq \pi_2$ *whenever both* $\vdash \pi_1 \leq \pi_2$ *and* $\vdash \pi_2 \leq \pi_1$ *are valid subtyping judgements; we will also write* $\nvdash \pi_1 \leq \pi_2$ *whenever the judgement* $\vdash \pi_1 \leq \pi_2$ *is* not *derivable.*

Derivability in this inference system implies subtyping.

Lemma 9.31. *If* $\vdash \pi_1 \leq \pi_2$ *is derivable then* $\pi_1 \leq \pi_2$.

Proof. By straightforward induction on the structure of derivations. Each rule corresponds to a case in Definition 9.11. □

We have remarked that our decision procedure is not complete with respect to the subtyping relation. Thus, there exist types σ and τ such that $\sigma \leq \tau$ but $\nvdash \sigma \leq \tau$ is *not* derivable. This stems from the fact that the subtyping relation is defined through an *interplay* of structural and logical rules, but the inference system deals first with the structure of a pretype, and only secondly with the logical aspect.

Example 9.32 (Counter-example to completeness)**.** *The pair of canonical pretypes* $(\varphi \to \varphi, \bullet\varphi \to \bullet\varphi)$ *is in the subtype relation, but the corresponding subtype inference judgement* $\vdash \varphi \to \varphi \leq \bullet\varphi \to \bullet\varphi$ *is not derivable.*

1. $\varphi \to \varphi \leq \bullet(\varphi \to \varphi) \leq \bullet\varphi \to \bullet\varphi$

2. *Suppose a derivation exists for the judgement* $\vdash \varphi \to \varphi \leq \bullet\varphi \to \bullet\varphi$. *The last rule applied must be* (ST-FUN)*, and thus both the judgements* $\vdash \bullet\varphi \leq \varphi$ *and* $\vdash \varphi \leq \bullet\varphi$ *must also be derivable. The latter of these follows immediately from the* (ST-VAR) *rule, but the former (which could only be derived using the* (ST-VAR) *rule again) is not valid since the side condition does not hold: the left hand type in the judgement has one more bullet than the right hand type. Thus, the original judgement* $\vdash \varphi \to \varphi \leq \bullet\varphi \to \bullet\varphi$ *is not derivable.*

We now aim to show that derivability in the subtyping inference system is decidable. To this end we define a mapping which identifies a *structural representative* for each pretype. These structural representatives are themselves pretypes, but ones that do not contain any bullets or insertion variables (indeed, they are ordinary, 'non-logical' recursive types); thus, they contain only the *structural* information of a pretype. We will use these structural representatives to argue that the amount of structural information in a pretype is a calculable, finite quantity. We will also use them to argue that the structure of any derivation depends only on the structure of the types in the judgement, and thus that the structure of

derivations in the subtyping inference system have a well-defined bound - implying the decidability of derivability.

Definition 9.33 (Structural Representatives)**.** *The* structural representative *of a pretype π is defined inductively in the structure of pretypes as follows:*

$$\begin{aligned}
\mathsf{struct}(\varphi) &= \varphi \\
\mathsf{struct}(\mathbf{n}) &= \mathbf{n} \\
\left.\begin{array}{r}\mathsf{struct}(\bullet\pi)\\ \mathsf{struct}(\iota\pi)\end{array}\right\} &= \mathsf{struct}(\pi) \\
\mathsf{struct}(\pi_1 \to \pi_2) &= (\mathsf{struct}(\pi_1)) \to (\mathsf{struct}(\pi_2)) \\
\mathsf{struct}(\mu.\phi) &= \mu.(\mathsf{struct}(\phi))
\end{aligned}$$

We now define a notion, called the *structural closure*, that allows us to calculate how much structural information a pretype contains. It is inspired by the *subterm closure* construction given in [26, 35], however we have chosen to give our definition a slightly different name since it does not include *all* syntactic subterms of a type, instead abstracting away bullets and insertion variables.

Definition 9.34 (Structural Closure)**.** *1. The* structural closure *of a pretype π is defined by cases as follows:*

$$\begin{aligned}
\mathcal{SC}(\varphi) &= \{\varphi\} \\
\mathcal{SC}(\mathbf{n}) &= \{\mathbf{n}\} \\
\mathcal{SC}(\bullet\pi) &= \mathcal{SC}(\pi) \\
\mathcal{SC}(\iota\pi) &= \mathcal{SC}(\pi) \\
\mathcal{SC}(\pi_1 \to \pi_2) &= \{\mathsf{struct}(\pi_1 \to \pi_2)\} \cup \mathcal{SC}(\pi_1) \cup \mathcal{SC}(\pi_2) \\
\mathcal{SC}(\mu.\phi) &= \{\mathsf{struct}(\mu.\phi)\} \cup \mathcal{SC}(\phi) \cup \mathcal{SC}([\mathbf{0} \mapsto \mu.\phi](\phi))
\end{aligned}$$

2. We extend the notion of structural closure to sets of pretypes P as follows:

$$\mathcal{SC}(P) = \bigcup_{\pi \in P} \mathcal{SC}(\pi)$$

The following result was stated in [35], and proven in [26], and implies that we can easily compute the structural closure.

Proposition 9.35. *For any pretype π, the set $\mathcal{SC}(\pi)$ is finite.*

We admit that the system presented here is slightly different from the systems in those papers, in that our treatment uses de Bruijn indices instead of μ-bound variables, and so the proof given by Brandt and Henglein does not automatically justify the result as formulated for our system. However, we point to recent work by Endrullis *et al* [53] which presents a much fuller treatment of the question of the decidability of weak μ-equality and the subterm closure construction, including α-independent representations of μ-terms (i.e. de Bruijn indices). For now, given that our system is clearly a variant in this family, we

conjecture that the result holds for our formulation. Proving this result holds for our system specifically is left for future work.

This result immediately implies the following corollary.

Lemma 9.36. *Let P be a set of pretypes; if P is finite, then so is $\mathcal{SC}(P)$.*

Proof. Immediate, by Proposition 9.35 since $\mathcal{SC}(P)$ is simply the union of the structural closures of each $\pi \in P$, which given that P is finite, is thus a finite union of finite sets. □

The following properties hold of the structural closure construction. They are needed to show Lemma 9.39 below.

Lemma 9.37 (Properties of Structural Closures)**.**
1. $\mathsf{struct}(\pi) \in \mathcal{SC}(\pi)$.
2. $\mathcal{SC}(\mathsf{bPush}[n](\pi)) = \mathcal{SC}(\pi)$.
3. $\mathcal{SC}(\mathsf{iPush}[i](\pi)) = \mathcal{SC}(\pi)$.

Proof. By straightforward induction on the structure of pretypes, using Definition 9.34. □

Returning to the question at hand, we note that the inference system possesses two properties which result in the decidability of derivability. The first is that it is entirely *structure directed*: each rule matches a structural feature of types (with the logical constraints checked as side conditions). In addition, it is entirely *deterministic*: for each structural combination there is exactly one rule and so the structure of a pair of pretypes in the subtype relation *uniquely* determines the derivation that witnesses the validity of subtyping.

Proposition 9.38. *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the derivations for $\vdash \kappa_1 \leq \kappa_2$ and $\vdash \kappa_1' \leq \kappa_2'$ respectively; if $\mathsf{struct}(\kappa_1) = \mathsf{struct}(\kappa_1')$ and $\mathsf{struct}(\kappa_2) = \mathsf{struct}(\kappa_2')$, then $\mathcal{D}_1$ and $\mathcal{D}_2$ have the same structure (i.e. the same rules are applied in the same order).*

Proof technique. By induction on the structure of subtype inference derivations.

Secondly, for any derivation the structural representatives of the types in the statements it contains are all themselves members of a well-defined and, most importantly, *finite* set - the union of the subterm closures of the structural representatives of the pretypes in the derived judgement.

Proposition 9.39. *Let $\mathcal{D}$ be a derivation of $\vdash \kappa_1 \leq \kappa_2$, then all the statements $\kappa_1' \leq \kappa_2'$ occurring in it are such that both $\mathsf{struct}(\kappa_1')$ and $\mathsf{struct}(\kappa_2')$ are in the set $\mathcal{SC}(\{\kappa_1, \kappa_2\})$.*

Proof technique. By induction on the structure of subtype inference derivations.

This means that the height of any derivation in the subtyping inference system is finitely bounded. Consequently, to decide if any given subtyping judgement is derivable, we need only check the validity (i.e. derivability) of a finite number of statements.

Corollary 9.40. *Let $\mathcal{D}$ be a derivation for $\vdash \kappa \leq \kappa'$; then the height of $\mathcal{D}$ is no greater than $|\mathcal{SC}(\{\kappa, \kappa'\})|^2$.*

Proof. By contradiction.

Let D be the set $\mathcal{SC}(\mathsf{struct}(\kappa)) \cup \mathcal{SC}(\mathsf{struct}(\kappa'))$ and let $\mathcal{D}$ be the derivation for $\vdash \kappa \leq \kappa'$. Assume $\mathcal{D}$ has a height $h > |D|^2$, then there are derivations $\mathcal{D}_1, \ldots, \mathcal{D}_h$ such that $\mathcal{D} = \mathcal{D}_1$ and for each $i \in \overline{h}$

the derivations $\mathcal{D}_{i+1}, \ldots, \mathcal{D}_h$ are (proper) subderivations of $\mathcal{D}_i$. Thus there is a set of pairs of pretypes $\{(\kappa_1, \kappa_1'), \ldots, (\kappa_h, \kappa_h')\}$ which are the pretypes in the final judgements of each of the derivations $\mathcal{D}_1, \ldots, \mathcal{D}_h$. By Proposition 9.39 we know that for each pair (κ_i, κ_i'), both $\mathsf{struct}(\kappa_i)$ and $\mathsf{struct}(\kappa_i')$ are in D.

Since the number of unique pairs (π, π') such that both π and π' are in D is $|D|^2 < h$, it must be that there are two distinct $j, k \leq h$ such that $\mathsf{struct}(\kappa_j) = \mathsf{struct}(\kappa_k)$ and $\mathsf{struct}(\kappa_j') = \mathsf{struct}(\kappa_k')$. Then we know by Proposition 9.38 that $\mathcal{D}_j$ and $\mathcal{D}_k$ have the same structure and must therefore have the same height. However, since j and k are distinct, it must be that either $j < k$ or $k < j$, and so either one of $\mathcal{D}_j$ or $\mathcal{D}_k$ is a proper subderivation of the other. This is impossible however, since the two derivations must have the same structure. Therefore, the height of $\mathcal{D}$ cannot exceed $|D|^2$. □

The subtyping inference system defined above can thus very straightforwardly be turned into a *terminating algorithm* which decides if any given subtyping judgement is derivable.

Definition 9.41 (Subtyping Decision Algorithm)**.** *The algorithm* $\mathsf{Inf}_{\leq}$ *takes in two (canonical) pretypes and an integer parameter as input and returns either* ***true*** *or* ***false****. It is defined as follows (where in case the input does not match any of the clauses, the algorithm returns* ***false****):*

$$\mathsf{Inf}_{\leq}(d, \tilde{\iota} \bullet^r \varphi, \vec{\iota'} \bullet^s \varphi) = \textbf{\textit{true}} \qquad \textit{(if } d > 0 \textit{ with } r \leq s \textit{ and } \vec{\iota'} \sqsubseteq \tilde{\iota}\textit{)}$$

$$\mathsf{Inf}_{\leq}(d, \tilde{\iota} \bullet^r \mathbf{n}, \vec{\iota'} \bullet^s \mathbf{n}) = \textbf{\textit{true}} \qquad \textit{(if } d > 0 \textit{ with } r \leq s \textit{ and } \vec{\iota'} \sqsubseteq \tilde{\iota}\textit{)}$$

$$\mathsf{Inf}_{\leq}(d, \kappa_1 \to \kappa_2, \kappa_1' \to \kappa_2') = \qquad \textit{(if } d > 0\textit{)}$$
$$\mathsf{Inf}_{\leq}(d-1, \kappa_1', \kappa_1) \wedge \mathsf{Inf}_{\leq}(d-1, \kappa_2, \kappa_2')$$

$$\mathsf{Inf}_{\leq}(d, \tilde{\iota} \bullet^r \mu.(\kappa_1 \to \kappa_2), \vec{\iota'} \bullet^s \mu.(\kappa_1' \to \kappa_2')) = \qquad \textit{(if } d > 0 \textit{ with } r \leq s \textit{ and } \vec{\iota'} \sqsubseteq \tilde{\iota}\textit{)}$$
$$\mathsf{Inf}_{\leq}(d-1, \kappa_1 \to \kappa_2, \kappa_1' \to \kappa_2')$$

$$\mathsf{Inf}_{\leq}(d, \tilde{\iota} \bullet^r \mu.(\kappa_1 \to \kappa_2), \kappa_1' \to \kappa_2') = \qquad \textit{(if } d > 0\textit{)}$$
$$\mathsf{Inf}_{\leq}(d-1, \mathsf{iPush}[\tilde{\iota}](\mathsf{bPush}[r]([\mathbf{0} \mapsto \mu.(\kappa_1 \to \kappa_2)](\kappa_1 \to \kappa_2))), \kappa_1' \to \kappa_2')$$

$$\mathsf{Inf}_{\leq}(d, \kappa_1 \to \kappa_2, \tilde{\iota} \bullet^s \mu.(\kappa_1' \to \kappa_2')) = \qquad \textit{(if } d > 0\textit{)}$$
$$\mathsf{Inf}_{\leq}(d-1, \kappa_1 \to \kappa_2, \mathsf{iPush}[\tilde{\iota}](\mathsf{bPush}[s]([\mathbf{0} \mapsto \mu.(\kappa_1' \to \kappa_2')](\kappa_1' \to \kappa_2'))))$$

Proposition 9.42 (Soundness and Completeness for $\mathsf{Inf}_{\leq}$)**.** *1.* $\exists\, d\ [\,\mathsf{Inf}_{\leq}(d, \pi_1, \pi_2) = \textbf{\textit{true}}\,] \Rightarrow \vdash \pi_1 \leq \pi_2$.

2. If $\mathcal{D}$ *is the derivation for* $\vdash \pi_1 \leq \pi_2$ *and* $\mathcal{D}$ *has height* h*, then for all* $d \geq h$*,* $\mathsf{Inf}_{\leq}(d, \pi_1, \pi_2) = \textbf{\textit{true}}$.

Proof technique. 1. By induction on the definition of $\mathsf{Inf}_{\leq}$.

2. By induction on the structure of subtype inference derivations.

This immediately gives us a partial correctness result for the subtyping decision algorithm.

Conjecture 9.43 (Partial Correctness for $\mathsf{Inf}_{\leq}$)**.** *Let* $d = |\mathcal{SC}(\{\pi_1, \pi_2\})|^2$*, then* $\vdash \pi_1 \leq \pi_2 \Leftrightarrow \mathsf{Inf}_{\leq}(d, \pi_1, \pi_2) =$ ***true***

Proof technique. By Proposition 9.42.

Lastly, we must show that the algorithm $\mathsf{Inf}_{\leq}$ terminates.

Theorem 9.44 (Termination of $\mathsf{Inf}_{\leq}$)**.** *The algorithm* $\mathsf{Inf}_{\leq}$ *terminates on all inputs* (d, π_1, π_2).

Proof. By easy induction on d. In the base case ($d = 0$), Definition 9.41 gives that the algorithm terminates returning false, since no cases apply. For the inductive case, we do a case analysis on π_1 and π_2. If they are both either type or recursive variables (prefixed by some number of bullets and insertion variables), then the algorithm terminates returning either true or false depending on the relative number of bullets prefixing each type and whether the insertion variables prefixing the one type are a permutation suffix of those prefixing the other. In the other defined cases, the termination of the recursive calls, and thus the outer call, follows by the inductive hypothesis. In all other undefined cases, Definition 9.41 gives that the algorithm returns false. □

9.5. Unification

In this section we will define a procedure to unify two canonical types modulo the subtype relation. That is, our procedure, when given two types σ and τ, will return an operation O such that $\mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$. In fact, when defining such a procedure we must be very careful, since the presence of recursive types in our system may cause it to loop indefinitely, just as when trying to decide the subtyping relation itself.

In formulating our unification algorithm, we will take the same approach as in the previous section. We will first define an inference system whose derivable judgements entail the unification of two pretypes modulo subtyping by some operation O. Then, we will again argue that the size of any derivation of the inference system is bounded by some well-defined (decidable) limit. As with our subtyping decision procedure, the inference system that we define can be straightforwardly converted into an algorithm whose recursion is bounded by an input parameter.

One of the key aspects to the unification procedure is the generation of recursive types. Whenever we try to unify a type variable with another type containing that variable, instead of failing, as Robinson's unification procedure does, we instead produce a substitution which replaces the type variable with a recursive type such that the application of the substitution to the original type we were trying to unify against is the *unfolding* of the recursive type that we substitute.

Take, for example, the two (pre)types φ and $\varphi \to \varphi'$. Robinson's approach to unification would treat these two types as non-unifiable since the second type contains the variable that we are trying to unify against. However, we can unify these types using a *recursive* type σ that satisfies the following equation:

$$\sigma \quad = \quad \sigma \to \varphi'$$

This equation can be seen as giving a *definition* (or specification) of the type σ, thus such a recursive type can be systematically constructed for any σ and any definition by simply replacing the type in the

definition with a recursive type variable, and then forming a recursive type using the μ type constructor:

$$\sigma \quad = \quad \mu X.(X \to \varphi')$$

Or, using de Bruijn indices:

$$\sigma \quad = \quad \mu.(\mathbf{0} \to \varphi')$$

The subtlety of doing this in the Nakano setting is that, in order to construct a valid type, we must make sure that there are bullets in appropriate places, i.e. when we introduce a recursive type variable, it must fall within the scope of a $\bullet$ operator, thus satisfying the *adequacy* property of types (see Definition 9.6).

Notice that this procedure bears a strong resemblance to that of constructing recursively defined functions in the λ-calculus, where we abstract over the function identifier (i.e. the name we give to the function), and then apply a fixed point combinator. This is not a coincidence and, in fact, it is directly analogous since in our case we are constructing a recursively defined *type*: we abstract over the identifier of the type in its definition using a recursive type variable (instead of a term variable), and the recursive type constructor μ plays the same role as a fixed point combinator term.

To facilitate the constructing of recursive types in this way, we define a further substitution operation that replaces type variables with recursive type variables (i.e. de Bruijn indices).

Definition 9.45 (Variable Promotion)**.** *A* variable promotion P *is an operation on pretypes that promotes type variables to recursive type variables (de Bruijn indices). If φ is a type variable and $\mathbf{n}$ is a de Bruijn index, then the variable promotion $[\mathbf{n}/\varphi]$ is defined inductively on the structure of pretypes simultaneously for each $\mathbf{n} \in \mathbb{N}$ as follows:*

$$[\mathbf{n}/\varphi](\varphi') = \begin{cases} \mathbf{n} & \text{if } \varphi = \varphi' \\ \varphi' & \text{otherwise} \end{cases}$$

$$[\mathbf{n}/\varphi](\mathbf{n}') = \mathbf{n}'$$

$$[\mathbf{n}/\varphi](\bullet\pi) = \bullet([\mathbf{n}/\varphi](\pi))$$

$$[\mathbf{n}/\varphi](\iota\pi) = \iota([\mathbf{n}/\varphi](\pi))$$

$$[\mathbf{n}/\varphi](\pi_1 \to \pi_2) = ([\mathbf{n}/\varphi](\pi_1)) \to ([\mathbf{n}/\varphi](\pi_2))$$

$$[\mathbf{n}/\varphi](\mu.\phi) = \mu.([\mathbf{n}+1/\varphi](\phi))$$

We must show that the composition of a μ-substitution and a variable promotion acts as kind of (canonicalising) type substitution (modulo the equivalence relation $\simeq$). The corollary to this result is that if we construct a recursive type out of some function type by promoting one its type variables, then the type we obtain by substituting the newly created recursive type for the type variable instead of promoting it, is equivalent to the recursive type itself - in fact, this is because it is equivalent to the *unfolding* of the recursive type. This result will be needed to show the soundness of our unification procedure.

Proposition 9.46. *Let $\mu.\phi$ be a type and π be a pretype such that $\mathbf{n} \notin \mathrm{FV}(\pi)$, then*

$$[\mathbf{n} \mapsto \mu.\phi]([\mathbf{n}/\varphi](\pi)) \simeq [\varphi \mapsto \mu.\phi](\pi)$$

Proof technique. By induction on the structure of pretypes.

Corollary 9.47. *Let ϕ be a type, then $\mu.([\mathbf{0}/\varphi](\phi)) \simeq [\varphi \mapsto \mu.([\mathbf{0}/\varphi](\phi))](\phi)$.*

Proof. By Definition 9.11 and Proposition 9.46. □

We mentioned above that when we construct a recursive type, we must make sure that all the occurrences of the bound recursive variable that we introduce (via variable promotion) must be under the scope of a bullet (•) type constructor. If the type variable that we are promoting is not in the set of raw type variables, then we can make sure that this is the case. If the type variable occurs in the type, but is not raw, then by definition (see Def. 9.5) every occurrence of the type variable will be within the scope of either a • or some insertion variable. We will now define a function that will return the (smallest) set of insertion variables that capture the occurrences of a given type variable within their scope that do not also fall within the scope of the • type constructor. We will call this set the *cover* set of the type variable. If we then insert a bullet under each of these insertion variables (which can be done by composing all insertions of the form $[\iota_i \mapsto \iota_i \bullet]$ where ι_i is in the cover set), we ensure that each occurrence of the type variable now falls within the scope of a bullet. Thus, when the type variable is promoted, each occurrence of the newly introduced recursive type variable will also fall within the scope of a bullet, and the recursive type can be safely closed (i.e. the recursively closing the type produces an adequate pretype).

Definition 9.48 (Cover Set)**.** *The* cover set $\mathsf{Cov}[\varphi](\pi)$ *of the pretype π with respect to the type variable φ is the (minimal) set of insertion variables under whose scope the type variable φ occurs raw. For each type variable φ it is defined inductively on the structure of pretypes as follows:*

$$\begin{aligned}
\mathsf{Cov}[\varphi](\varphi') &= \emptyset & \mathsf{Cov}[\varphi](\iota\,\pi) &= \begin{cases} \{\iota\} & \text{if } \varphi \in \textsc{raw}_\varphi(\pi) \\ \mathsf{Cov}[\varphi](\pi) & \text{otherwise} \end{cases} \\
\mathsf{Cov}[\varphi](\mathbf{n}) &= \emptyset & & \\
\mathsf{Cov}[\varphi](\bullet\pi) &= \emptyset & \mathsf{Cov}[\varphi](\pi_1 \to \pi_2) &= \mathsf{Cov}[\varphi](\pi_1) \cup \mathsf{Cov}[\varphi](\pi_2) \\
& & \mathsf{Cov}[\varphi](\mu.\phi) &= \mathsf{Cov}[\varphi](\phi)
\end{aligned}$$

The following results will be needed to show that we construct recursive *types* (i.e. adequate, closed pretypes) during unification, and thence that the unification procedure returns an operation.

Lemma 9.49. *1. If $\varphi \in \textsc{tv}(\pi)$, then $\mathbf{n} \in \textsc{fv}([\mathbf{n}/\varphi](\pi))$.*

2. If $\mathsf{O} = \mathsf{I}_n \circ \ldots \circ \mathsf{I}_1$, then $\textsc{tv}(\pi) = \textsc{tv}(\mathsf{O}(\pi))$.

3. $\textsc{raw}_\varphi(\mathsf{bPush}(\pi)) = \emptyset$, and $\mathsf{Cov}[\varphi](\mathsf{bPush}(\pi)) = \emptyset$.

4. Let π be a type and φ be a type variable such that $\varphi \in \textsc{tv}(\pi)$ with $\mathsf{Cov}[\varphi](\pi) = \{\iota_1, \ldots, \iota_n\}$; if $\varphi \notin \textsc{raw}_\varphi(\pi)$, then $\varphi \notin \textsc{raw}_\varphi(\mathsf{O}(\pi))$ and $\mathsf{Cov}[\varphi](\mathsf{O}(\pi)) = \emptyset$, where $\mathsf{O} = [\iota_n \mapsto \iota_n \bullet] \circ \ldots \circ [\iota_1 \mapsto \iota_1 \bullet]$.

5. Let π be a pretype such that $\mathbf{n} \notin \textsc{raw}_\mu(\pi)$ and $\varphi \in \textsc{tv}(\pi)$; if $\varphi \notin \textsc{raw}_\varphi(\pi)$ and $\mathsf{Cov}[\varphi](\pi) = \emptyset$, then $\mathbf{n} \notin \textsc{fv}([\mathbf{n}/\varphi](\pi)) \setminus \textsc{raw}_\mu([\mathbf{n}/\varphi](\pi))$.

Proof. The proof of (2) is by induction on n, with the case for $n = 1$ being proved by induction on π. The other lemmas are proved by straightforward induction on the structure of π. □

We now come to define the notion of *unification inference*, similar to the notion of subtype inference in Definition 9.30. The inference system will derive unification *judgements*, which assert the unifiability

of two pretypes using some operation O. We define unification in this way so that we can reason about the termination of our unification inference procedure. The inference rules can be seen as a 'bottom-up' approach to the problem of unification, as opposed to the more common 'top-down' algorithmic view. The two approaches, however, are dual and we will later convert the unification inference system into an equivalent algorithm (Definition 9.61 below).

Definition 9.50 (Unification Inference)**.** *1. A* unification judgement *is a statement of the form* $\mathsf{O} \vdash \pi_1 \leq \pi_2$ *and asserts that the operation* O unifies *the pretypes* π_1 *and* π_2 *modulo the subtyping relation; that is* $\mathsf{O}(\pi_1) \leq \mathsf{O}(\pi_2)$*.*

2. Valid unification judgements are derived using the following natural deduction inference system, in which we classify rules as either structural *or* logical*:*

Unifying Type Variables (Structural Rules)

$$\frac{}{[\iota \mapsto \check{\iota} \bullet^{s-r}] \vdash \iota \bullet^r \varphi \leq \check{\iota} \bullet^s \varphi} \ (\iota \notin \check{\iota} \text{ and } r \leq s)$$

$$\frac{}{[\iota \mapsto \check{\iota} \bullet^{r-s}] \vdash \check{\iota} \bullet^r \varphi \leq \iota \bullet^s \varphi} \ (\iota \notin \check{\iota} \text{ and } s < r)$$

$$\frac{}{\mathsf{Id} \vdash \bullet^r \varphi \leq \bullet^s \varphi} \ (r \leq s)$$

$$\frac{}{[\varphi \mapsto \bullet^{s-r} \varphi'] \vdash \bullet^r \varphi \leq \bullet^s \varphi'} \ (\varphi \neq \varphi' \text{ and } r \leq s)$$

$$\frac{}{[\varphi' \mapsto \bullet^{r-s} \varphi] \vdash \bullet^r \varphi \leq \bullet^s \varphi'} \ (\varphi \neq \varphi' \text{ and } s < r)$$

Unifying Type Variables (Logical Rules)

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\check{\iota}_n \bullet^r \varphi) \leq \mathsf{O}_1(\vec{\iota'}_m \bullet^s \varphi')}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \cdot \check{\iota}_n \bullet^r \varphi \leq \iota' \cdot \vec{\iota'}_m \bullet^s \varphi'} \ (\iota \neq \iota' \text{ and } n, m > 0)$$

where $\mathsf{O}_1 = [\iota \mapsto \iota']$

$$\frac{\mathsf{O} \vdash \bullet^r \varphi \leq \bullet^s \varphi'}{\mathsf{O} \circ [\iota \mapsto \check{\iota}] \vdash \iota \bullet^r \varphi \leq \check{\iota} \bullet^s \varphi'} \ (\iota \notin \check{\iota} \text{ and } \varphi \neq \varphi')$$

$$\frac{\mathsf{O}_2 \vdash \bullet^r \varphi \leq \mathsf{O}_1(\check{\iota} \bullet^s \varphi')}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \bullet^r \varphi \leq \check{\iota} \bullet^s \varphi'} \ (\iota \in \check{\iota} \text{ or } (\varphi = \varphi' \text{ and } s < r))$$

where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

$$\frac{\mathsf{O} \vdash \bullet^r \varphi \leq \bullet^s \varphi'}{\mathsf{O} \circ [\iota \mapsto \check{\iota}] \vdash \check{\iota} \bullet^r \varphi \leq \iota \bullet^s \varphi'} \ (\iota \notin \check{\iota} \text{ and } \varphi \neq \varphi')$$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\check{\iota} \bullet^r \varphi) \leq \bullet^s \varphi'}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \check{\iota} \bullet^r \varphi \leq \iota \bullet^s \varphi'} \ (\iota \in \check{\iota} \text{ or } (\varphi = \varphi' \text{ and } r \leq s))$$

where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

$$\frac{\mathsf{O}_2 \vdash \bullet^r \varphi \leq \mathsf{O}_1(\check{\iota}_m \bullet^s \varphi')}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \bullet^r \varphi \leq \iota \cdot \check{\iota}_m \bullet^s \varphi'} \; (m > 0)$$

where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\check{\iota}_n \bullet^r \varphi) \leq \bullet^s \varphi'}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \cdot \check{\iota}_n \bullet^r \varphi \leq \bullet^s \varphi'} \; (n > 0)$$

where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

Unifying Type Variables and Function Types (Structural Rules)

$$\frac{}{[\varphi \mapsto \kappa_1 \to \kappa_2] \vdash \varphi \leq \kappa_1 \to \kappa_2}$$

$(\varphi \notin \mathrm{TV}(\kappa_1 \to \kappa_2)$ *and* $\kappa_1 \to \kappa_2$ *a type*$)$

$$\frac{}{[\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\kappa_1 \to \kappa_2)))] \circ \mathsf{O} \vdash \varphi \leq \kappa_1 \to \kappa_2}$$

$(\varphi \in \mathrm{TV}(\kappa_1 \to \kappa_2) \setminus \mathrm{RAW}_\varphi(\kappa_1 \to \kappa_2)$ *and* $\kappa_1 \to \kappa_2$ *a type*$)$

where $\mathsf{Cov}[\varphi](\kappa_1 \to \kappa_2) = \{\iota_1, \ldots, \iota_n\}$
$\mathsf{O} = [\iota_n \mapsto \iota_n \bullet] \circ \ldots \circ [\iota_1 \mapsto \iota_1 \bullet]$

$$\frac{}{[\varphi \mapsto \kappa_1 \to \kappa_2] \vdash \kappa_1 \to \kappa_2 \leq \check{\iota} \bullet^s \varphi}$$

$(\varphi \notin \mathrm{TV}(\kappa_1 \to \kappa_2)$ *and* $\kappa_1 \to \kappa_2$ *a type)*

$$\frac{}{[\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\kappa_1 \to \kappa_2)))] \circ \mathsf{O} \vdash \kappa_1 \to \kappa_2 \leq \check{\iota} \bullet^s \varphi}$$

$(\varphi \in \mathrm{TV}(\kappa_1 \to \kappa_2) \setminus \mathrm{RAW}_\varphi(\kappa_1 \to \kappa_2)$ *and* $\kappa_1 \to \kappa_2$ *a type*$)$

where $\mathsf{Cov}[\varphi](\kappa_1 \to \kappa_2) = \{\iota_1, \ldots, \iota_n\}$
$\mathsf{O} = [\iota_n \mapsto \iota_n \bullet] \circ \ldots \circ [\iota_1 \mapsto \iota_1 \bullet]$

Unifying Type Variables and Function Types (Logical Rules)

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\check{\iota}_n \bullet^r \varphi) \leq \mathsf{O}_1(\kappa_1 \to \kappa_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \cdot \check{\iota}_n \bullet^r \varphi \leq \kappa_1 \to \kappa_2}$$

where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

Unifying Type Variables and Head-Recursive Types (Structural Rules)

$$\frac{}{[\varphi \mapsto \bullet^{s-r} \mu.(\kappa_1 \to \kappa_2)] \vdash \bullet^r \varphi \leq \bullet^s \mu.(\kappa_1 \to \kappa_2)}$$

$$(\varphi \notin \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2)),\ \mu.(\kappa_1 \to \kappa_2)\ \textit{a type and}\ r \le s)$$

$$\frac{\mathsf{O} \vdash \varphi \le \mathsf{bPush}[s-r]([\mathbf{0} \mapsto \mu.(\kappa_1 \to \kappa_2)](\kappa_1 \to \kappa_2))}{\mathsf{O} \vdash \bullet^r \varphi \le \bullet^s \mu.(\kappa_1 \to \kappa_2)}$$

$$(\varphi \in \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2))\ \textit{and}\ r \le s)$$

$$\frac{}{[\varphi \mapsto \bullet^{r-s} \mu.(\kappa_1 \to \kappa_2)] \vdash \bullet^r \mu.(\kappa_1 \to \kappa_2) \le \bullet^s \varphi}$$

$$(\varphi \notin \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2)),\ \mu.(\kappa_1 \to \kappa_2)\ \textit{a type and}\ s \le r)$$

$$\frac{}{[\varphi \mapsto \mu.(\kappa_1 \to \kappa_2)] \vdash \bullet^r \mu.(\kappa_1 \to \kappa_2) \le \bullet^s \varphi}$$

$$(\varphi \notin \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2)),\ \mu.(\kappa_1 \to \kappa_2)\ \textit{a type and}\ r < s)$$

$$\frac{\mathsf{O} \vdash \mathsf{bPush}[r]([\mathbf{0} \mapsto \mu.(\kappa_1 \to \kappa_2)](\kappa_1 \to \kappa_2)) \le \bullet^s \varphi}{\mathsf{O} \vdash \bullet^r \mu.(\kappa_1 \to \kappa_2) \le \bullet^s \varphi}$$

$$(\varphi \in \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2)))$$

Unifying Recursive Type Variables/Head-Recursive Types (Structural Rules)

$$\frac{}{\mathsf{Id} \vdash \bullet^r \mathbf{n} \le \bullet^s \mathbf{n}}\ (r \le s)$$

$$\frac{\mathsf{O} \vdash \kappa_1 \to \kappa_2 \le \kappa_1' \to \kappa_2'}{\mathsf{O} \vdash \bullet^r \mu.(\kappa_1 \to \kappa_2) \le \bullet^s \mu.(\kappa_1' \to \kappa_2')}\ (r \le s)$$

Unifying Function Types (Structural Rule)

$$\frac{\mathsf{O}_1 \vdash \kappa_1' \le \kappa_1 \quad \mathsf{O}_2 \vdash \mathsf{O}_1(\kappa_2) \le \mathsf{O}_1(\kappa_2')}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \kappa_1 \to \kappa_2 \le \kappa_1' \to \kappa_2'}$$

Unifying Function Types and Head-Recursive Types (Structural Rules)

$$\frac{\mathsf{O} \vdash \kappa_1 \to \kappa_2 \le \mathsf{iPush}[\check{\iota}](\mathsf{bPush}[s]([\mathbf{0} \mapsto \mu.(\kappa_1' \to \kappa_2')](\kappa_1' \to \kappa_2')))}{\mathsf{O} \vdash \kappa_1 \to \kappa_2 \le \check{\iota} \bullet^s \mu.(\kappa_1' \to \kappa_2')}$$

$$\frac{\mathsf{O} \vdash \mathsf{iPush}[\check{\iota}](\mathsf{bPush}[r]([\mathbf{0} \mapsto \mu.(\kappa_1 \to \kappa_2)](\kappa_1 \to \kappa_2))) \le \kappa_1' \to \kappa_2'}{\mathsf{O} \vdash \check{\iota} \bullet^r \mu.(\kappa_1 \to \kappa_2) \le \kappa_1' \to \kappa_2'}$$

Generic Logical Rules

$$\frac{\mathsf{O} \vdash \check{\iota}_n \alpha_1 \le \vec{\iota'}_m \alpha_2}{\mathsf{O} \vdash \iota \cdot \check{\iota}_n \alpha_1 \le \iota \cdot \vec{\iota'}_m \alpha_2}\ (n, m > 0)$$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\check{\iota}_n \bullet^r \xi_1) \le \mathsf{O}_1(\vec{\iota'}_m \bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \cdot \check{\iota}_n \bullet^r \xi_1 \le \iota' \cdot \vec{\iota'}_m \bullet^s \xi_2}$$

($\iota \neq \iota'$ *and either* ($r \leq s \,\&\, n > 0$) *or* ($s < r \,\&\, m > 0$)
and either ξ_1 *or* ξ_2 *not a type variable*)
where $\mathsf{O}_1 = [\iota \mapsto \iota']$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\xi_1) \leq \mathsf{O}_1(\xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \bullet^r \xi_1 \leq \check{\iota} \bullet^s \xi_2}$$

($\iota \notin \check{\iota}$ *and* $r \leq s$ *and either* ξ_1 *or* ξ_2 *not a type variable*)
where $\mathsf{O}_1 = [\iota \mapsto \check{\iota} \bullet^{s-r}]$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\bullet^r \xi_1) \leq \mathsf{O}_1(\check{\iota} \bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \bullet^r \xi_1 \leq \check{\iota} \bullet^s \xi_2}$$

($\iota \in \check{\iota}$ *and* $r \leq s$ *and either* ξ_1 *or* ξ_2 *not a type variable*)
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\xi_1) \leq \mathsf{O}_1(\xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \check{\iota} \bullet^r \xi_1 \leq \iota \bullet^s \xi_2}$$

($\iota \notin \check{\iota}$ *and* $s < r$ *and either* ξ_1 *or* ξ_2 *not a type variable*)
where $\mathsf{O}_1 = [\iota \mapsto \check{\iota} \bullet^{r-s}]$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\check{\iota} \bullet^r \xi_1) \leq \mathsf{O}_1(\bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \check{\iota} \bullet^r \xi_1 \leq \iota \bullet^s \xi_2}$$

($\iota \in \check{\iota}$ *and* $s < r$ *and either* ξ_1 *or* ξ_2 *not a type variable*)
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\check{\iota}_n \bullet^r \xi_1) \leq \mathsf{O}_2(\bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \cdot \check{\iota}_n \bullet^r \xi_1 \leq \bullet^s \xi_2}$$

($n > 0$ *or* $s < r$ *and either* ξ_1 *or* ξ_2 *not a type variable*)
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\bullet^r \xi_1) \leq \mathsf{O}_1(\check{\iota}_m \bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \bullet^r \xi_1 \leq \iota \cdot \check{\iota}_m \bullet^s \xi_2}$$

($m > 0$ *or* $r \leq s$ *and either* ξ_1 *or* ξ_2 *not a type variable*)
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$

We claim that the inference system defined above is sound with respect to the subtyping relation; in other words, valid unification judgements *correctly* assert that there is a unifying operation for two pretypes.

Proposition 9.51 (Soundness of Unification Inference)**.** *If* $\mathsf{O} \vdash \pi_1 \leq \pi_2$*, then* O *is an operation and* $\mathsf{O}(\pi_1) \leq \mathsf{O}(\pi_2)$.

Proof technique. By induction on the structure of the unification inference derivations using Definition 9.11 and the soundness of operations with respect to subtyping (Proposition 9.26). In the base cases where a substitution of type variable for a new recursive type is generated, we use Corollary 9.47.

However, like subtype inference, unification is *incomplete* - that is, there are pairs of pretypes which *are* unifiable but not *inferrably* so. For example, the unification judgement $\mathsf{O} \vdash \bullet\varphi \leq \bullet\varphi' \to \bullet\varphi'$ is *not* derivable for any operation O, even though the canonicalising type substitution $[\varphi \mapsto (\varphi' \to \varphi')]$ unifies the two types.

As well as soundness, we also claim that the unification inference procedure is *deterministic*. This means that if a derivation exists that witnesses the validity of a unification judgement, then it is *unique*.

Property 9.52 (Determinism of Unification Inference)**.** *For any pair of (canonical) pretypes in a unification judgement, there is at most one inference rule which applies.*

We will now define a measure of the *height* of a unification inference derivation. This concept will be a key element in proving the decidability of unification inference.

Definition 9.53 (Unification Inference Derivation Height)**.** *Let $\mathcal{D}$ be a derivation in the unification inference system; then the* height *of $\mathcal{D}$ is defined inductively on the structure of derivations as follows:*

1. *If the last rule applied in $\mathcal{D}$ is a structural one and it has no immediate subderivations, then the height of $\mathcal{D}$ is 1.*

2. *If the last rule applied in $\mathcal{D}$ is a structural one, and h is the maximum of the heights of its immediate subderivations, then the height of $\mathcal{D}$ is $h+1$.*

3. *If the last rule applied in $\mathcal{D}$ is a logical one, and h is the maximum of the heights of its immediate subderivations, then the height of $\mathcal{D}$ is h.*

In general, we can relate the height of a derivation to the heights of its subderivations in the following way:

Lemma 9.54. *Let $\mathcal{D}$ be a derivation in the unification inference system, and $\mathcal{D}'$ be a (proper) subderivation of $\mathcal{D}$ in which the last rule applied is a structural one. Then:*

1. *if the last rule applied in $\mathcal{D}$ is a logical one, then the height of $\mathcal{D}$ is greater than or equal to the height of $\mathcal{D}'$;*

2. *if the last rule applied in $\mathcal{D}$ is a structural one, then the height of $\mathcal{D}$ is greater than the height of $\mathcal{D}'$.*

Proof. By straightforward induction on the structure of unification inference derivations. □

Furthermore, for pairs of (inferrably) unifiable pretypes that have the same structural representatives, the heights of their unification derivations are the same. This shows that, as for subtype inference, the inference system is *structurally* driven, and this again will form a key part in the proof of its decidability.

Proposition 9.55. *Let κ_1 and κ_1', and κ_2 and κ_2' be structurally equivalent pairs of canonical pretypes, i.e.* $\mathsf{struct}(\kappa_1) = \mathsf{struct}(\kappa_1')$ *and* $\mathsf{struct}(\kappa_2) = \mathsf{struct}(\kappa_2')$*, and let $\mathcal{D}$ and $\mathcal{D}'$ be the derivations of $\mathsf{O} \vdash \kappa_1 \leq \kappa_2$ and $\mathsf{O}' \vdash \kappa_1' \leq \kappa_2'$ respectively; then heights of $\mathcal{D}$ and $\mathcal{D}'$ are the same.*

Proof technique. By induction on the structure of unification inference derivations.

To demonstrate the decidability of the unification inference system, we will argue that the height of any derivation has a well-defined (and computable) bound. As for subtype inference, and following [35], our approach to calculating such a bound is to consider all the possible pairs of pretypes (or rather, structurally representative pairs) that might be compared within any given derivation. This is slightly more complicated than the situation for subtyping, or type equality. Since the unification inference procedure involves constructing and applying *operations* to pretypes, we cannot generate all such pairs simply by breaking apart the pretypes to be unified into their subcomponents, as we did for subtype inference. We must also consider the *substitutions* that might take place on these subcomponents. For example, when unifying two function types $\kappa_1 \to \kappa_2$ and $\kappa_1' \to \kappa_2'$ we first attempt to unify the left-hand sides κ_1' and κ_1. If this succeeds, it produces an operation O (consisting of substitutions and insertions) which we must apply to the right hand sides *before* unifying them, that is we must unify $\mathsf{O}(\kappa_2)$ with $\mathsf{O}(\kappa_2')$, and *not* κ_2 with κ_2'. Thus, the derivation may contain judgements $\mathsf{O} \vdash \pi_1 \leq \pi_2$ where π_1 and π_2 are not simply subcomponents of the two top-level pretypes $\kappa_1 \to \kappa_2$ and $\kappa_1' \to \kappa_2'$.

Despite this increased complexity, it is still possible to calculate the set of pretypes that can be generated in this way because the unification procedure is 'well-behaved' in a particular sense. Again, as for subtype inference, we can abstract away from the logical component of the types meaning that we can ignore the insertion operations that are generated during unification, leaving us only to consider the substitutions that may be generated. The key observation here is, firstly that these substitutions only replace the type variables occurring within the types that we are trying to unify, and secondly the types that they are replaced with do *not* contain the type variable itself. This means that when recursively unifying subcomponents of a pretype after applying an operation (as happens when unifying two function pretypes), there is a strictly smaller set of type variables from which to build the unifying operation.

The result is that, for a given pair of (inferrably unifiable) pretypes, the unification procedure generates a composition of substitutions $[\varphi_1 \mapsto \sigma_1] \circ \ldots \circ [\varphi_n \mapsto \sigma_n]$ (of course interspersed with insertions) where each φ_i is distinct, and each σ_i is a subcomponent of a type (or a recursive type generated from such a type) resulting from applying a (smaller) composition of substitutions to the original pretypes π and π' themselves. Since the number of type variables (and the number of structural subcomponents) occurring in the pretypes π and π' is finite, we can calculate all possible such compositions of substitutions, and thus build the set of all structural representatives of pretypes that might occur in the derivation of $\mathsf{O} \vdash \pi \leq \pi'$.

Of course, when considering the types that might get substituted during unification, in addition to subcomponents of the types being unified, we must take into account *recursive* types that might be constructed when we unify a type variable with another type in which that variable occurs. To this end, we define a a further closure set construction that accounts for types generated in this way.

Definition 9.56 (Recursion Complete Structural Closure)**.** *1. The* recursion complete *structural closure of a pretype π is defined as follows:*

$$\mathcal{SC}_{+\mu}(\pi) = \mathcal{SC}(\pi) \cup \bigcup_{\substack{\pi_1 \to \pi_2 \in \mathcal{SC}(\pi) \\ \mathrm{FV}(\pi_1 \to \pi_2) = \emptyset}} \left(\bigcup_{\varphi \in \mathrm{TV}(\pi_1 \to \pi_2)} \mathcal{SC}_{+\mu}(\mu.([\mathbf{0}/\varphi](\pi_1 \to \pi_2))) \right)$$

2. This notion is extended to sets of pretypes P as follows:

$$\mathcal{SC}_{+\mu}(P) = \bigcup_{\pi \in P} \mathcal{SC}_{+\mu}(\pi)$$

Using this enhanced structural closure, we are now able to define a construction which can represent all of the pretypes that might be compared during the unification procedure.

Definition 9.57 (Unification Closure)**.** *Let P be a set of pretypes. The* unification closure $\mathcal{UC}(P)$ *of P is defined by:*

$$\mathcal{UC}(P) \quad = \quad \mathcal{SC}_{+\mu}(P) \cup \bigcup_{\varphi \in \mathrm{TV}(P)} \left(\bigcup_{\substack{\pi \in \mathcal{SC}_{+\mu}(P) \\ \varphi \notin \mathrm{TV}(\pi)}} \mathcal{UC}([\varphi \mapsto \pi](P)) \right)$$

Since the structural closure of a type is a finite set, it follows that the recursion complete structural closure and the unification closure of a type are also finite sets. Thus, the definitions above give an effective way to compute these sets.

Lemma 9.58 (Finiteness of Closures)**.** *Let π be a pretype, and P be a finite set of pretypes; then the following sets are finite: 1. $\mathcal{SC}_{+\mu}(\pi)$, 2. $\mathcal{SC}_{+\mu}(P)$, and 3. $\mathcal{UC}(P)$.*

Proof. 1. By mathematical induction on number of distinct type variables in π, using the fact that $\mathcal{SC}(\pi)$ is finite (Lemma 9.36), and that if $\varphi \in \mathrm{TV}(\pi)$ then the number of type variables in $[\mathbf{n}/\varphi](\pi)$ is one less than the number of type variables in π.

2. By the first part of the lemma, $\mathcal{SC}_{+\mu}(\pi)$ is finite for all $\pi \in P$. Thus, if P is finite, then $\mathcal{SC}_{+\mu}(P)$ is simply a finite union of finite sets, and therefore is itself finite.

3. By mathematical induction on the number of distinct type variables in P, using the fact that $\mathcal{SC}_{+\mu}(P)$ is finite (first part of the lemma), and that if $\mathrm{TV}(\pi) \subseteq \mathrm{TV}(P)$ and $\varphi \notin \mathrm{TV}(\pi)$ then the number of type variables in $[\varphi \mapsto \pi](P)$ is one less than the number of type variables in P. □

As for deciding subtype inference, to show decidability of unification inference we are required to show that the structural representative of each pretype in the statements of any unification inference derivation belongs to a well-defined, finite and computable set – the unification closure.

Proposition 9.59. *Let $\mathcal{D}$ be the derivation of $\mathsf{O} \vdash \kappa_1 \leq \kappa_2$, then all the statements $\kappa_1' \leq \kappa_2'$ occurring in it are such that both $\mathsf{struct}(\kappa_1')$ and $\mathsf{struct}(\kappa_2')$ are in the set $\mathcal{UC}(\{\kappa_1, \kappa_2\})$.*

Proof technique. By induction on the structure of unification inference derivations.

This again means that the height of any unification inference derivation is bounded by the size of the unification closure.

Corollary 9.60. *Let $\mathcal{D}$ be the derivation for $\vdash \kappa \leq \kappa'$; then the height of $\mathcal{D}$ is no greater than $|\mathcal{UC}(\{\kappa_1, \kappa_1\})|^2$.*

Proof. Similar to the proof of Corollary 9.40, using Proposition 9.59 instead of Proposition 9.39. □

The unification inference system leads straightforwardly to an algorithm that decides whether any given unification statement is valid. As for the algorithm to decide subtype inference, this algorithm has a decreasing input parameter, which is decremented every time a recursive call is made that corresponds to the application of a structural rule in the unification inference system. Thus, the algorithm is guaranteed to terminate.

Definition 9.61 (Unification Algorithm)**.** *The unification algorithm* $\mathsf{Unify}^{\mu}_{\leq}$ *takes two canonical pretypes and an integer as input, and either returns an operation or fails. It is defined as follows, where for any input types that do not match the cases given below the algorithm terminates in a 'fail' state:*

Unifying Type Variables (Structural Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \bullet^{r} \varphi, \vec{\iota}_m \bullet^{s} \varphi') = [\iota \mapsto \vec{\iota}_m \bullet^{s-r}]$$
if $\iota \notin \vec{\iota}_m$ *and* $\varphi = \varphi'$ *with* $d > 0$ *and* $r \leq s$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \vec{\iota}_n \bullet^{r} \varphi, \iota \bullet^{s} \varphi') = [\iota \mapsto \vec{\iota}_n \bullet^{r-s}]$$
if $\iota \notin \vec{\iota}_n$ *and* $\varphi = \varphi'$ *with* $d > 0$ *and* $s \leq r$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \varphi, \bullet^{s} \varphi') = \mathsf{Id}$$
if $\varphi = \varphi'$ *with* $d > 0$ *and* $r \leq s$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \varphi, \bullet^{s} \varphi') = [\varphi \mapsto \bullet^{s-r} \varphi']$$
if $\varphi \neq \varphi'$ *with* $d > 0$ *and* $r \leq s$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \varphi, \bullet^{s} \varphi') = [\varphi \mapsto \bullet^{r-s} \varphi']$$
if $\varphi \neq \varphi'$ *with* $d > 0$ *and* $s < r$

Unifying Type Variables (Logical Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \cdot \vec{\iota}_n \bullet^{r} \varphi, \iota' \cdot \vec{\iota'}_m \bullet^{s} \varphi') = \mathsf{O}_2 \circ \mathsf{O}_1$$
if $\iota \neq \iota'$ *and* $d, n, m > 0$
where $\mathsf{O}_1 = [\iota \mapsto \iota']$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\vec{\iota}_n \bullet^{r} \varphi), \mathsf{O}_1(\vec{\iota'}_m \bullet^{s} \varphi'))$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \bullet^{r} \varphi, \vec{\iota} \bullet^{s} \varphi') = \mathsf{O} \circ [\iota \mapsto \vec{\iota}]$$
if $\iota \notin \vec{\iota}$ *and* $\varphi \neq \varphi'$ *with* $d > 0$
where $\mathsf{O} = \mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \varphi, \bullet^{s} \varphi')$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \bullet^{r} \varphi, \vec{\iota} \bullet^{s} \varphi') = \mathsf{O}_2 \circ \mathsf{O}_1$$
if $d > 0$ *and either* $\iota \in \vec{\iota}$ *or* $(\varphi = \varphi'$ *and* $s < r)$
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \varphi, \mathsf{O}_1(\vec{\iota} \bullet^{s} \varphi'))$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \vec{\iota} \bullet^{r} \varphi, \iota \bullet^{s} \varphi') = \mathsf{O} \circ [\iota \mapsto \vec{\iota}]$$

$$\text{if } \iota \notin \vec{\iota} \text{ and } \varphi \neq \varphi' \text{ with } d > 0$$
$$\text{where } \mathsf{O} = \mathsf{Unify}^{\mu}_{\leq}(d, \bullet^r \varphi, \bullet^s \varphi')$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \vec{\iota} \bullet^r \varphi, \iota \bullet^s \varphi') = \mathsf{O}_2 \circ \mathsf{O}_1$$
$$\text{if } d > 0 \text{ and either } \iota \in \vec{\iota} \text{ or } (\varphi = \varphi' \text{ and } r \leq s)$$
$$\text{where } \mathsf{O}_1 = [\iota \mapsto \epsilon]$$
$$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\vec{\iota} \bullet^r \varphi), \bullet^s \varphi')$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^r \varphi, \iota \cdot \vec{\iota}_m \bullet^s \varphi') = \mathsf{O}_2 \circ \mathsf{O}_1$$
$$\text{if } d, m > 0$$
$$\text{where } \mathsf{O}_1 = [\iota \mapsto \epsilon]$$
$$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \bullet^r \varphi, \mathsf{O}_1(\vec{\iota}_m \bullet^s \varphi'))$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \cdot \vec{\iota}_n \bullet^r \varphi, \bullet^s \varphi') = \mathsf{O}_2 \circ \mathsf{O}_1$$
$$\text{if } d, n > 0$$
$$\text{where } \mathsf{O}_1 = [\iota \mapsto \epsilon]$$
$$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\vec{\iota}_n \bullet^r \varphi), \bullet^s \varphi')$$

Unifying Type Variables and Function Types (Structural Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(d, \varphi, \kappa_1 \to \kappa_2) = [\varphi \mapsto (\kappa_1 \to \kappa_2)]$$
$$\text{if } \varphi \notin \mathrm{TV}(\kappa_1 \to \kappa_2) \text{ and } d > 0 \text{ with } \kappa_1 \to \kappa_2 \text{ a type}$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \varphi, \kappa_1 \to \kappa_2) = [\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\kappa_1 \to \kappa_2)))] \circ \mathsf{O}$$
$$\text{if } \varphi \in \mathrm{TV}(\kappa_1 \to \kappa_2) \setminus \mathrm{RAW}_{\varphi}(\kappa_1 \to \kappa_2) \text{ and } d > 0$$
$$\text{with } \kappa_1 \to \kappa_2 \text{ a type}$$
$$\text{where } \mathsf{Cov}[\varphi](\kappa_1 \to \kappa_2) = \{\iota_1, \ldots, \iota_n\}$$
$$\mathsf{O} = [\iota_n \mapsto \iota_n \bullet] \circ [\iota_1 \mapsto \iota_1 \bullet]$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \kappa_1 \to \kappa_2, \vec{\iota} \bullet^s \varphi) = [\varphi \mapsto (\kappa_1 \to \kappa_2)]$$
$$\text{if } \varphi \notin \mathrm{TV}(\kappa_1 \to \kappa_2) \text{ and } d > 0 \text{ with } \kappa_1 \to \kappa_2 \text{ a type}$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \kappa_1 \to \kappa_2, \vec{\iota} \bullet^s \varphi) = [\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\kappa_1 \to \kappa_2)))] \circ \mathsf{O}$$
$$\text{if } \varphi \in \mathrm{TV}(\kappa_1 \to \kappa_2) \setminus \mathrm{RAW}_{\varphi}(\kappa_1 \to \kappa_2) \text{ and } d > 0$$
$$\text{with } \kappa_1 \to \kappa_2 \text{ a type}$$
$$\text{where } \mathsf{Cov}[\varphi](\kappa_1 \to \kappa_2) = \{\iota_1, \ldots, \iota_n\}$$
$$\mathsf{O} = [\iota_n \mapsto \iota_n \bullet] \circ [\iota_1 \mapsto \iota_1 \bullet]$$

Unifying Type Variables and Function Types (Logical Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \cdot \check{\iota} \bullet^{r} \varphi, \kappa_1 \to \kappa_2) = \mathsf{O}_2 \circ \mathsf{O}_1$$

if $d > 0$
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\check{\iota} \bullet^{r} \varphi), \mathsf{O}_1(\kappa_1 \to \kappa_2))$

Unifying Type Variables with Head-Recursive Types (Structural Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \varphi, \bullet^{s} \mu.(\kappa_1 \to \kappa_2)) = [\varphi \mapsto \bullet^{s-r} \mu.(\kappa_1 \to \kappa_2)]$$

if $\varphi \notin \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2))$ *and* $r \leq s$
with $d > 0$ *and* $\mu.(\kappa_1 \to \kappa_2)$ *a type*

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \varphi, \bullet^{s} \mu.(\kappa_1 \to \kappa_2)) = \mathsf{Unify}^{\mu}_{\leq}(d-1, \varphi, \mathsf{bPush}[s-r]([\mathbf{0} \mapsto \mu.(\kappa_1 \to \kappa_2)](\kappa_1 \to \kappa_2)))$$

if $\varphi \in \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2))$ *and* $r \leq s$ *with* $d > 0$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \mu.(\kappa_1 \to \kappa_2), \bullet^{s} \varphi) = [\varphi \mapsto \bullet^{r-s} \mu.(\kappa_1 \to \kappa_2)]$$

if $\varphi \notin \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2))$ *and* $s \leq r$
with $d > 0$ *and* $\mu.(\kappa_1 \to \kappa_2)$ *a type*

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \mu.(\kappa_1 \to \kappa_2), \bullet^{s} \varphi) = [\varphi \mapsto \mu.(\kappa_1 \to \kappa_2)]$$

if $\varphi \notin \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2))$ *and* $r < s$
with $d > 0$ *and* $\mu.(\kappa_1 \to \kappa_2)$ *a type*

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \mu.(\kappa_1 \to \kappa_2), \bullet^{s} \varphi) = \mathsf{Unify}^{\mu}_{\leq}(d-1, \mathsf{bPush}[r]([\mathbf{0} \mapsto \mu.(\kappa_1 \to \kappa_2)](\kappa_1 \to \kappa_2)), \bullet^{s} \varphi)$$

if $\varphi \in \mathrm{TV}(\mu.(\kappa_1 \to \kappa_2))$ *and* $d > 0$

Unifying Recursive Type Variables/Head-Recursive Types (Structural Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \mathbf{n}, \bullet^{s} \mathbf{n}) = \mathsf{Id}$$

if $r \leq s$ *and* $d > 0$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^{r} \mu.(\kappa_1 \to \kappa_2), \bullet^{s} \mu.(\kappa_1' \to \kappa_2')) = \mathsf{Unify}^{\mu}_{\leq}(d-1, \kappa_1 \to \kappa_2, \kappa_1' \to \kappa_2')$$

$$\textit{if } r \leq s \textit{ and } d > 0$$

Unifying Function Types (Structural Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(d, \kappa_1 \to \kappa_2, \kappa_1' \to \kappa_2') = \mathsf{O}_2 \circ \mathsf{O}_1$$

$$\textit{if } d > 0$$
$$\textit{where } \mathsf{O}_1 = \mathsf{Unify}^{\mu}_{\leq}(d-1, \kappa_1', \kappa_1)$$
$$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d-1, \mathsf{O}_1(\kappa_2), \mathsf{O}_1(\kappa_2'))$$

Unifying Function Types and Head-Recursive Types (Structural Cases)

$$\mathsf{Unify}^{\mu}_{\leq}(\kappa_1 \to \kappa_2, \check{\iota} \bullet^s \mu.(\kappa_1' \to \kappa_2'))$$
$$= \mathsf{Unify}^{\mu}_{\leq}(\kappa_1 \to \kappa_2, \mathsf{iPush}[\check{\iota}](\mathsf{bPush}[s]([\mathbf{0} \mapsto \mu.\kappa_1' \to \kappa_2'](\kappa_1' \to \kappa_2'))))$$

$$\textit{if } d > 0$$

$$\mathsf{Unify}^{\mu}_{\leq}(\check{\iota} \bullet^r \mu.(\kappa_1 \to \kappa_2), \kappa_1' \to \kappa_2')$$
$$= \mathsf{Unify}^{\mu}_{\leq}(\mathsf{iPush}[\check{\iota}](\mathsf{bPush}[r]([\mathbf{0} \mapsto \mu.\kappa_1 \to \kappa_2](\kappa_1 \to \kappa_2))), \kappa_1' \to \kappa_2')$$

$$\textit{if } d > 0$$

Generic Logical Cases

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \cdot \check{\iota}_n\, \alpha_1, \iota' \cdot \vec{\iota'}_m\, \alpha_2) = \mathsf{Unify}^{\mu}_{\leq}(d, \check{\iota}_n\, \alpha_1, \vec{\iota'}_m\, \alpha_2)$$

$$\textit{if } \iota = \iota' \textit{ and } d, n, m > 0$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \cdot \check{\iota}_n \bullet^r \xi_1, \iota' \cdot \vec{\iota'}_m \bullet^s \xi_2) = \mathsf{O}_2 \circ \mathsf{O}_1$$

$$\textit{if } \iota \neq \iota' \textit{ and } d > 0$$
$$\textit{with either } (r \leq s \;\&\; n > 0) \textit{ or } (s < r \;\&\; m > 0)$$
$$\textit{and either } \xi_1 \textit{ or } \xi_2 \textit{ not a type variable}$$
$$\textit{where } \mathsf{O}_1 = [\iota \mapsto \iota']$$
$$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\check{\iota}_n \bullet^r \xi_1), \mathsf{O}_1(\vec{\iota'}_m \bullet^s \xi_2))$$

$$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \bullet^r \xi_1, \check{\iota} \bullet^s \xi_2) = \mathsf{O}_2 \circ \mathsf{O}_1$$

if $\iota \notin \vec{\iota}$ *and* $r \leq s$ *with* $d > 0$
and either ξ_1 *or* ξ_2 *not a type variable*
where $\mathsf{O}_1 = [\iota \mapsto \vec{\iota} \bullet^{s-r}]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\xi_1), \mathsf{O}_1(\xi_2))$

$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \bullet^r \xi_1, \vec{\iota} \bullet^s \xi_2) = \mathsf{O}_2 \circ \mathsf{O}_1$

if $\iota \in \vec{\iota}$ *and* $r \leq s$ *with* $d > 0$
and either ξ_1 *or* ξ_2 *not a type variable*
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\bullet^r \xi_1), \mathsf{O}_1(\vec{\iota} \bullet^s \xi_2))$

$\mathsf{Unify}^{\mu}_{\leq}(d, \vec{\iota} \bullet^r \xi_1, \iota \bullet^s \xi_2) = \mathsf{O}_2 \circ \mathsf{O}_1$

if $\iota \notin \vec{\iota}$ *and* $s < r$ *with* $d > 0$
and either ξ_1 *or* ξ_2 *not a type variable*
where $\mathsf{O}_1 = [\iota \mapsto \vec{\iota} \bullet^{r-s}]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\xi_1), \mathsf{O}_1(\xi_2))$

$\mathsf{Unify}^{\mu}_{\leq}(d, \vec{\iota} \bullet^r \xi_1, \iota \bullet^s \xi_2) = \mathsf{O}_2 \circ \mathsf{O}_1$

if $\iota \in \vec{\iota}$ *and* $s < r$ *with* $d > 0$
and either ξ_1 *or* ξ_2 *not a type variable*
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\vec{\iota} \bullet^r \xi_1, \mathsf{O}_1(\bullet^s \xi_2))$

$\mathsf{Unify}^{\mu}_{\leq}(d, \iota \cdot \vec{\iota}_n \bullet^r \xi_1, \bullet^s \xi_2) = \mathsf{O}_2 \circ \mathsf{O}_1$

if $n > 0$ *or* $s < r$ *with* $d > 0$
and either ξ_1 *or* ξ_2 *not a type variable*
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\vec{\iota}_n \bullet^r \xi_1), \mathsf{O}_1(\bullet^s \xi_2))$

$\mathsf{Unify}^{\mu}_{\leq}(d, \bullet^r \xi_1, \iota \cdot \vec{\iota}_m \bullet^s \xi_2) = \mathsf{O}_2 \circ \mathsf{O}_1$

if $m > 0$ *or* $r \leq s$ *with* $d > 0$
and either ξ_1 *or* ξ_2 *not a type variable*
where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$
$\mathsf{O}_2 = \mathsf{Unify}^{\mu}_{\leq}(d, \mathsf{O}_1(\bullet^r \xi_1), \mathsf{O}_1(\vec{\iota}_m \bullet^s \xi_2))$

It should be straightforward to show that this algorithm decides unification inference.

Proposition 9.62 (Soundness and Completeness of $\mathsf{Unify}^{\mu}_{\leq}$)**.** *1. If* $\mathsf{Unify}^{\mu}_{\leq}(d, \kappa_1, \kappa_2) = \mathsf{O}$*, then* $\mathsf{O} \vdash \kappa_1 \leq \kappa_2$.

2. Let $\mathcal{D}$ *be the derivation for the judgement* $\mathsf{O} \vdash \kappa_1 \leq \kappa_2$ *and suppose it has height* h*; then for all* $d \geq h$, $\mathsf{Unify}^{\mu}_{\leq}(d, \kappa_1, \kappa_2) = \mathsf{O}$.

Proof technique. 1. By induction on the definition of $\mathsf{Unify}^{\mu}_{\leq}$.

2. By induction on the structure of unification inference derivations.

As for subtype inference, this immediately implies a partial correctness result for the unification procedure.

Conjecture 9.63 (Partial Correctness of $\mathsf{Unify}^{\mu}_{\leq}$)**.** *Let* κ_1, κ_2 *be canonical pretypes and* $d = |\mathcal{UC}(\{\kappa_1, \kappa_2\})|^2$*; then* $\mathsf{Unify}^{\mu}_{\leq}(d, \kappa_1, \kappa_2) = \mathsf{O}$ *if and only if* $\mathsf{O} \vdash \kappa_1 \leq \kappa_2$.

Proof technique. Directly by Proposition 9.62

We must also show that unification algorithm terminates. To do so, we need to define a measure on pretypes, called the *insertion rank*, which is a measure of the maximum depth of nesting of insertion variables in a pretype.

Definition 9.64. *The* insertion rank $\mathsf{iRank}(\pi)$ *of the pretype* π *is defined inductively on the structure of pretypes as follows:*

$$\mathsf{iRank}(\varphi) = 0 \qquad \mathsf{iRank}(\iota\,\pi) = 1 + \mathsf{iRank}(\pi)$$
$$\mathsf{iRank}(\mathbf{n}) = 0 \qquad \mathsf{iRank}(\pi_1 \to \pi_2) = \mathsf{max}(\mathsf{iRank}(\pi_1), \mathsf{iRank}(\pi_2))$$
$$\mathsf{iRank}(\bullet\,\pi) = \mathsf{iRank}(\pi) \qquad \mathsf{iRank}(\mu.\phi) = \mathsf{iRank}(\phi)$$

Certain types of insertions decrease the insertion rank of types.

Lemma 9.65. *Let* $\mathsf{I} = [\iota \mapsto \check{\iota}_n]$ *be an insertion with* $n \leq 1$*, then* $\mathsf{iRank}(\pi) \geq \mathsf{iRank}(\mathsf{I}(\pi))$ *for all pretypes* π.

Proof. By straightforward induction on the structure of pretypes. □

This allows us to prove the termination of $\mathsf{Unify}^{\mu}_{\leq}$.

Theorem 9.66. *The procedure* $\mathsf{Unify}^{\mu}_{\leq}$ *terminates on all inputs.*

Proof. We interpret the input (d, κ_1, κ_2) as the tuple $(d, \mathsf{iRank}(\kappa_1) + \mathsf{iRank}(\kappa_2))$, and prove by well-founded induction using the lexicographic ordering on pairs of natural numbers. □

The final step before defining the type inference procedure itself is to extend the notion of unification to type environments.

Definition 9.67 (Unification of Type Environments)**.** *The unification procedure is extended to type environments as follows:*

$$\mathsf{Unify}^{\mu}_{\leq}(\emptyset,\ \Pi) = \Pi$$
$$\mathsf{Unify}^{\mu}_{\leq}((\Pi, x{:}\sigma),\ (\Pi', x{:}\tau)) = \mathsf{O}_2 \circ \mathsf{O}_1 \qquad \textit{if } \mathsf{Unify}^{\mu}_{\leq}(d, \sigma, \tau) = \mathsf{O}_1$$

$$\mathsf{Unify}^{\mu}_{\leq}((\Pi, x{:}\sigma),\ (\Pi', x{:}\tau)) = \mathsf{O}_2 \circ \mathsf{O}_1 \qquad \textit{if } \mathsf{Unify}^{\mu}_{\leq}(d, \sigma, \tau) \textit{ fails}$$

$$\textit{and } \mathsf{Unify}^{\mu}_{\leq}(d, \tau,\ \sigma) = \mathsf{O}_1$$

$$\mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_1(\Pi_1),\ \mathsf{O}_1(\Pi_2)) = \mathsf{O}_2$$

$$\textit{where } |\mathcal{UC}(\{\sigma, \tau\})|^2 = d$$

$$\mathsf{Unify}^{\mu}_{\leq}((\Pi, x{:}\sigma),\ \Pi') = \mathsf{Unify}^{\mu}_{\leq}(\Pi,\ \Pi') \qquad \textit{if } x \notin \Pi'$$

Notice that since type environments are sets, we cannot assume that $\mathsf{Unify}^{\mu}_{\leq}$ defines a *function* from type environment pairs to operations - it could be that unifying the statements in the two type environment in different orders produces different unifying operations, and so we may only state that $\mathsf{Unify}^{\mu}_{\leq}$ induces a *relation* between pairs of type environments and operations. However, since our unification procedure is *sound*, we do know that any unifying operation it returns does indeed unify type environments modulo subtyping. Note that in practice, when implementing this system, we are at liberty to impose an *ordering* on term variables, meaning that unifying type environments happens in a deterministic fashion.

We point out, though, that we have not yet been able to come up with an example demonstrating that this is the case, and so we consider it at least *possible* that $\mathsf{Unify}^{\mu}_{\leq}$ does indeed compute a function. Notice that this is the question of whether the unification procedure computes *most general* unifiers, which is orthogonal to the question of its completeness. Even though there exist pairs of unifiable pretypes for which our unification procedure fails to produce a unifier, it may still be the case that when our unification procedure does infer a unifier for a pair or pretypes, that unifier is most general. Even if this is not the case, note that it may still hold true for a subset of pretypes. Here we are thinking in particular about inferring types for λ-terms and so the subset of types that we have in mind is that of *principal* types for λ-terms in our type assignment system (if they exist). Answering these questions is an objective for future research.

Proposition 9.68 (Soundness of Unification for Type Environments)**.** *If* $\mathsf{Unify}^{\mu}_{\leq}(\Pi_1, \Pi_2) = \mathsf{O}$ *then for each pair of statements* $(x{:}\sigma, x{:}\tau)$ *such that* $x{:}\sigma \in \Pi_1$ *and* $x{:}\tau \in \Pi_2$ *it is the case that either* $\mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$ *or* $\mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$.

Proof technique. By induction on the definition of $\mathsf{Unify}^{\mu}_{\leq}$ for type environments, using the soundness of unification (Proposition 9.51), and the soundness of operations with respect to subtyping (Proposition 9.26).

9.6. Type Inference

In this section, we will present our type inference algorithm for the type assignment system that was defined in Section 9.2, and discuss its operation using some examples. Since the unification algorithm that we defined in the previous section is not complete, neither is our type inference algorithm and so to give the reader a better idea of where its limitations lie we will also present an example of a term for

which a type cannot be inferred.

Before being able to define our type inference algorithm, we will first have to define an operation that combines two type environments. This operation will be used when inferring a type for an application of two terms. To support the operation of combining type environments, we will also define a measure of height for types so that if the type environments to be combined contain equivalent types for a given term variable, then we can choose the 'smaller' type.

Definition 9.69 (Height of Pretypes)**.** *The* height *of a pretype π is defined inductively as follows:*

$$\left.\begin{array}{l} h(\varphi) \\ h(\mathbf{n}) \end{array}\right\} = 0 \qquad \left.\begin{array}{r} h(\bullet\pi) \\ h(\iota\pi) \end{array}\right\} = h(\pi) \qquad \begin{array}{c} h(\pi_1 \to \pi_2) = 1 + \max(h(\pi_1), h(\pi_2)) \\ h(\mu.\phi) = h(\phi) \end{array}$$

Definition 9.70 (Combining Environments)**.** *We define a* combination *operation $\uplus$ on environments which takes subtyping into account. The set $\Pi_1 \uplus \Pi_2$ is defined as the smallest set satisfying the following conditions:*

$$x{:}\sigma \in \Pi_1 \;\&\; x \notin \Pi_2 \Rightarrow x{:}\sigma \in \Pi_1 \uplus \Pi_2 \tag{9.1}$$

$$x \notin \Pi_1 \;\&\; x{:}\sigma \in \Pi_2 \Rightarrow x{:}\sigma \in \Pi_1 \uplus \Pi_2 \tag{9.2}$$

$$x{:}\sigma \in \Pi_1 \;\&\; x{:}\tau \in \Pi_2 \quad \&\; \vdash \sigma \leq \tau \;\&\; \nvdash \tau \leq \sigma \Rightarrow x{:}\sigma \in \Pi_1 \uplus \Pi_2 \tag{9.3}$$

$$x{:}\sigma \in \Pi_1 \;\&\; x{:}\tau \in \Pi_2 \;\&\; \vdash \tau \leq \sigma \;\&\; \nvdash \sigma \leq \tau \Rightarrow x{:}\tau \in \Pi_1 \uplus \Pi_2 \tag{9.4}$$

$$x{:}\sigma \in \Pi_1 \;\&\; x{:}\tau \in \Pi_2 \quad \&\; \vdash \sigma \simeq \tau \;\&\; h(\sigma) \leq h(\tau) \Rightarrow x{:}\sigma \in \Pi_1 \uplus \Pi_2 \tag{9.5}$$

$$x{:}\sigma \in \Pi_1 \;\&\; x{:}\tau \in \Pi_2 \quad \&\; \vdash \sigma \simeq \tau \;\&\; h(\tau) < h(\sigma) \Rightarrow x{:}\tau \in \Pi_1 \uplus \Pi_2 \tag{9.6}$$

The environment-combining operation is sound.

Lemma 9.71 (Soundness of Environment Combination)**.** *If Π_1 and Π_2 are both type environments, then so is $\Pi_1 \uplus \Pi_2$.*

Proof. Straightforward by Definition 9.70. □

The environment-combining operation also has the property that it creates a *subtype environment* of each of the two combined environments. This property will be crucial when showing the soundness of the type inference procedure itself.

Lemma 9.72. *Let Π_1 and Π_2 be type environments and O be an operation such that, for each pair of types (σ, τ) with $x{:}\sigma \in \Pi_1$ and $x{:}\tau \in \Pi_2$, either $\vdash \mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$ or $\vdash \mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$; then both $(\mathsf{O}(\Pi_1) \uplus \mathsf{O}(\Pi_2)) \leq \mathsf{O}(\Pi_1)$ and $(\mathsf{O}(\Pi_1) \uplus \mathsf{O}(\Pi_2)) \leq \mathsf{O}(\Pi_2)$.*

Proof. Let $\Pi' = \mathsf{O}(\Pi_1) \uplus \mathsf{O}(\Pi_2)$. Take an arbitrary statement $x{:}\mathsf{O}(\sigma) \in \mathsf{O}(\Pi_1)$; there are two possibilities.

($x \notin \mathsf{O}(\Pi_2)$) Then by condition (9.1), $x{:}\mathsf{O}(\sigma) \in \Pi'$. By reflexivity of subtyping, $\mathsf{O}(\sigma) \leq \mathsf{O}(\sigma)$ and so there is a statement $x{:}\delta \in \Pi'$ such that $\delta \leq \mathsf{O}(\sigma)$.

($x \in \mathsf{O}(\Pi_2)$) Then there is a statement of the form $x{:}\mathsf{O}(\tau)$ in $\mathsf{O}(\Pi_2)$. Consequently, $x{:}\sigma \in \Pi_1$ and $x{:}\tau \in \Pi_2$. By assumption, either $\vdash \mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$ or $\vdash \mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$. We consider the cases separately.

- If $\vdash \mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$ then there are two subcases to consider:
  1. $\nvdash \mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$ - then by condition (9.3), $x{:}\mathsf{O}(\sigma) \in \Pi'$; by reflexivity of subtyping, $\mathsf{O}(\sigma) \leq \mathsf{O}(\sigma)$ and so there is a statement $x{:}\delta \in \Pi'$ such that $\delta \leq \mathsf{O}(\sigma)$.
  2. $\vdash \mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$ and so $\vdash \mathsf{O}(\sigma) \simeq \mathsf{O}(\tau)$. If $h(\mathsf{O}(\sigma)) \leq h(\mathsf{O}(\tau))$ then by condition (9.5), $x{:}\mathsf{O}(\sigma) \in \Pi'$; by reflexivity of subtyping, $\mathsf{O}(\sigma) \leq \mathsf{O}(\sigma)$ and so there is a statement $x{:}\delta \in \Pi'$ such that $\delta \leq \mathsf{O}(\sigma)$. If $h(\mathsf{O}(\tau)) < h(\mathsf{O}(\sigma))$ then by condition (9.6), $x{:}\mathsf{O}(\tau) \in \Pi'$; by Lemma 9.31, $\mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$ and so there is a statement $x{:}\delta \in \Pi'$ such that $\delta \leq \mathsf{O}(\sigma)$.
- The reasoning for $\vdash \mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$ is symmetrical.

Thus, for all $x{:}\sigma \in \mathsf{O}(\Pi_1)$ there is $x{:}\delta \in \Pi'$ such that $\delta \leq \sigma$, and so $\Pi' \leq \mathsf{O}(\Pi_1)$. The proof that $\Pi' \leq \mathsf{O}(\Pi_2)$ is symmetric. □

Notice that it may not be the case that the operation returned by unifying two type environments satisfies the properties required of it in the previous lemma in order to ensure that the environment combination is a subtype environment. This is because, while the unification algorithm will return an operation that is sound with respect to the subtype relation, it is not guaranteed to be sound with respect to subtype *inference*.

Example 9.73. *Take the two type environments* $\Pi_1 = \{x{:}\varphi, y{:}\varphi' \to \varphi'\}$ *and* $\Pi_2 = \{x{:}\bullet\varphi, y{:}\varphi\}$. *Unifying the types for x results in the Identity; then, unifying the types for y gives the substitution* $[\varphi \mapsto \varphi' \to \varphi']$. *Applying these operations* $\mathsf{O} = [\varphi \mapsto \varphi' \to \varphi'] \circ \mathsf{Id}$ *to the environments, we obtain* $\mathsf{O}(\Pi_1) = \{x{:}\varphi' \to \varphi', y{:}\varphi' \to \varphi'\}$ *and* $\mathsf{O}(\Pi_2) = \{x{:}\bullet\varphi' \to \bullet\varphi', y{:}\varphi' \to \varphi'\}$. *Then, according to Definition 9.70, we get* $\mathsf{O}(\Pi_1) \cup \mathsf{O}(\Pi_2) = \{y{:}\varphi' \to \varphi'\}$ *since we cannot* infer *that* $\vdash \varphi' \to \varphi' \leq \bullet\varphi' \to \bullet\varphi'$ *(remember this was shown in Example 9.32).*

Since subtype inference is our method of *deciding* when one type is a subtype of another, in order to formulate a type inference algorithm which is *decidable* as well as sound, we must therefore define a further property of operations with respect to type environments which guarantees that when the environments are combined, the result is a subtype environment.

Definition 9.74 (Inferrably Unifying Operations)**.** *We say that an operation* O inferrably *unifies two type environments* Π_1 *and* Π_2 *if it satisfies the property that for each pair of types* (σ, τ) *with* $x{:}\sigma \in \Pi_1$ *and* $x{:}\tau \in \Pi_2$, *either* $\vdash \mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$ *or* $\vdash \mathsf{O}(\tau) \leq \mathsf{O}(\sigma)$.

Notice that it is decidable whether a given operation inferrably unifies two type environments, since we can simply apply the operation to each type in the two environments and use the subtype inference algorithm to check, for each pair of types associated with a given variable in both environments, that one is (inferrably) a subtype of the other.

We can now define our type inference procedure.

Definition 9.75 (Type Inference)**.** *The type inference procedure,* Type, *is a partial function that takes a lambda expression and returns a pair* $\langle \Pi, \sigma \rangle$ *of a type environment* Π *and a canonical type* σ. *It is defined inductively on the structure of terms as follows:*

$$\mathsf{Type}(x) = \langle \{x{:}\varphi\}, \varphi \rangle \qquad \textit{where } \varphi \textit{ fresh}$$

$$\mathsf{Type}(\lambda x.M) = \begin{cases} \langle \Pi, \sigma \to \tau \rangle & \textit{if } \mathsf{Type}(M) = \langle (\Pi,\, x{:}\sigma), \tau \rangle \\ \langle \Pi, \varphi \to \tau \rangle & \textit{if } \mathsf{Type}(M) = \langle \Pi, \tau \rangle \textit{ and } x \notin \Pi \\ & \textit{where } \varphi \textit{ fresh} \end{cases}$$

$$\mathsf{Type}(M\,N) = \langle \mathsf{iPush}[\iota_2](\Pi_1') \cup \Pi_2', \mathsf{iPush}[\iota_2](\mathsf{O}(\varphi)) \rangle$$

$$\begin{array}{l} \textit{if } \mathsf{Type}(M) = \langle \Pi_1, \sigma \rangle \\ \quad \mathsf{Type}(N) = \langle \Pi_2, \tau \rangle \\ \quad \mathsf{Unify}^{\mu}_{\leq}(\sigma,\, \mathsf{iPush}[\iota_1](\tau) \to \varphi) = \mathsf{O}_1 \\ \quad \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_1(\Pi_1),\, \mathsf{O}_1(\Pi_2)) = \mathsf{O}_2 \textit{ and} \\ \quad \mathsf{O} = \mathsf{O}_2 \circ \mathsf{O}_1 \textit{ inferrably unifies } \Pi_1 \textit{ and } \Pi_2 \end{array}$$

$$\begin{array}{l} \textit{where } \varphi, \iota_1, \iota_2 \textit{ fresh} \\ \quad \Pi_1' = \{\, x{:}\sigma \mid x \in \Pi_1 \;\&\; x{:}\sigma \in \mathsf{O}(\Pi_1) \uplus \mathsf{O}(\Pi_2) \,\} \\ \quad \Pi_2' = \{\, y{:}\tau \mid y \in \Pi_2 \;\&\; y \notin \Pi_1 \;\&\; y{:}\tau \in \mathsf{O}(\Pi_1) \uplus \mathsf{O}(\Pi_2) \,\} \end{array}$$

Notice that the case for term variables and for abstractions is identical to the ordinary inference procedure for Curry typing. The difference lies only in the case for application. We conjecture that the type inference procedure we have just defined is *sound* with respect to the type assignment system, i.e. the typing inferred for a term can indeed be assigned to that term.

Conjecture 9.76 (Soundness of Type Inference)**.** *If* $\mathsf{Type}(M) = \langle \Pi, \sigma \rangle$ *then* $\Pi \vdash M{:}\sigma$.

Proof technique. By induction on the definition of Type.

Definition 9.75 straightforwardly defines an algorithm. Since each recursive call is made on a subterm, to show termination of the algorithm, we simply need to argue that all the other procedures that it calls (apart from itself) terminate. In the base case, that for a term variable, no other procedure are called. The case for an abstraction simply makes a recursive call. In the case for an application, besides recursively calling itself, the algorithm makes two calls to the unification procedure which we have shown is terminating (Theorem 9.66). It must also decide if the operation returned by the unification procedure inferrably unifies the type environments resulting from the recursive calls, the decidability of which we have remarked on above. Finally, if they are, it must combine the unified environments, which is a terminating operation since type environments are finite sets, and the subtype inference procedure is decidable (see Section 9.4).

In the next section, we will look at some examples of terms for which types can be inferred and also an example of a typeable term for which type inference fails. These examples also provide some evidence for the principality of our algorithm in the sense that the types inferred are most general, although we are unable to formally show a principal types property for our system at this time. Through these examples, we hope to give the reader a more intuitive understanding of how type assignment in the Nakano system works, as well as the role of insertion variables, both in type inference and in inferring principal typings.

9.6.1. Typing Curry's Fixed Point Operator Y

The original motivation for us in introducing insertion variables into the type system was to be able to infer a type for the fixed point operator $\mathbf{Y} = \lambda f.\delta_f\,\delta_f$ where $\delta_f = \lambda x.f\,(x\,x)$. In this section we will demonstrate how our type inference algorithm can successfully infer a type for this term, where one that does not use insertion variables would falter.

The obstacle to inferring a type for this term without insertion variables is trying to infer a type for the application $\delta_f\,\delta_f$. For the subterm $\delta_f = \lambda x.f\,(x\,x)$, an inference procedure without insertion variables would have to infer the typing $\langle\{f{:}\varphi_1 \to \varphi_2\},\ \mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_2\rangle$. Then, when inferring a type for the application of this subterm to itself, we first 'peel off' a fresh copy of this typing and attempt to unify the first copy of the type with a function type constructed from the second copy and a fresh type variable, as follows:

$$\mathsf{Unify}^{\mu}_{\leq} \qquad \mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_2, \qquad (\mu.(\bullet\mathbf{0} \to \varphi_3) \to \varphi_4) \to \varphi_5$$

The unification procedure would first attempt to unify the left-hand sides of each arrow type contravariantly, as follows:

$$\mathsf{Unify}^{\mu}_{\leq} \qquad \mu.(\bullet\mathbf{0} \to \varphi_3) \to \varphi_4 \qquad \mu.(\bullet\mathbf{0} \to \varphi_1)$$

This would then require the unification procedure to unfold the head-recursive type on the right:

$$\mathsf{Unify}^{\mu}_{\leq} \qquad \mu.(\bullet\mathbf{0} \to \varphi_3) \to \varphi_4 \qquad \bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_1$$

And again, we first try to contravariantly unify the left hand sides of these arrow types:

$$\mathsf{Unify}^{\mu}_{\leq} \qquad \bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \qquad \mu.(\bullet\mathbf{0} \to \varphi_3)$$

Here, of course, is where the unification fails due to a bullet prefixing the left-hand recursive type and not the right-hand one.

The reason underlying the failure of type inference in this instance is that the typing $\langle\{f{:}\varphi_1 \to \varphi_2\},\mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_2\rangle$ is not the *only* typing for the term $\lambda x.f\,(x\,x)$. The typing $\langle f{:}\bullet\varphi_1 \to \varphi_2, \bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_2\rangle$, for example, is also valid as shown in the following derivation $\mathcal{D}_1$:

$$\dfrac{\dfrac{}{\ldots \vdash f{:}\bullet\varphi_1 \to \varphi_2}\,(\textsc{var}) \qquad \dfrac{\dfrac{\dfrac{}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi_1)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \to \bullet\varphi_1}\,(\leq) \qquad \dfrac{\dfrac{}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi_1)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\bullet\mu.(\bullet\mathbf{0} \to \varphi_1)}\,(\leq)}{\ldots \vdash x\,x{:}\bullet\varphi_1}\,(\to E)}{f{:}\bullet\varphi_1 \to \varphi_2, x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \vdash f\,(x\,x){:}\varphi_2}\,(\to E)}{f{:}\bullet\varphi_1 \to \varphi_2 \vdash \lambda x.f\,(x\,x){:}\bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_2}\,(\to I)$$

The reason that this is a problem from the point of view of inferring a typing for the term $\delta_f\,\delta_f$, is that we need to use *both* typings: we must use the first typing for the left-hand occurrence of δ_f, and the latter typing for the right-hand occurrence.

Unfolding the type from the first typing we obtain the following:

$$\mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_2 \leq (\bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_1) \to \varphi_2$$

Notice that the left-hand side of the unfolded type is of the same shape as type $\bullet\mu.(\bullet\mathbf{0} \to \varphi_1) \to \varphi_2$ from the second typing. To make it exactly the same as the operand type of the unfolding above (and therefore allow us to type the application $\delta_f\,\delta_f$) we must unify the two type variables φ_1 and φ_2. Having done so we can then construct the following derivation $\mathcal{D}_2$ for the left-hand δ_f term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash f{:}\varphi \to \varphi}\,(\textsc{var}) \quad \dfrac{\dfrac{\dfrac{}{\ldots \vdash x{:}\mu.(\bullet\mathbf{0} \to \varphi)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi}\,(\leq) \quad \dfrac{\dfrac{}{\ldots \vdash x{:}\mu.(\bullet\mathbf{0} \to \varphi)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi)}\,(\leq)}{\ldots \vdash x\,x{:}\varphi}\,(\to E)}{f{:}\varphi \to \varphi, x{:}\mu.(\bullet\mathbf{0} \to \varphi) \vdash f\,(x\,x){:}\varphi}\,(\to I)}{f{:}\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi}\,(\leq)}{f{:}\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}(\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi) \to \varphi}\,(\leq)$$

To type the application of δ_f to itself, we must *weaken* the derivation so that the term variable f has the (more specific) type $\bullet\varphi \to \varphi$, as this is the type for f in the other typing that we will use to type the right-hand version of δ_f. Interestingly, we can do this in two ways. The first is by weakening the subderivation typing the subterm f to give $\mathcal{D}_2'$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash f{:}\bullet\varphi \to \varphi}\,(\textsc{var})}{\ldots \vdash f{:}\varphi \to \varphi}\,(\leq) \quad \dfrac{\dfrac{\dfrac{}{\ldots \vdash x{:}\mu.(\bullet\mathbf{0} \to \varphi)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi}\,(\leq) \quad \dfrac{\dfrac{}{\ldots \vdash x{:}\mu.(\bullet\mathbf{0} \to \varphi)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi)}\,(\leq)}{\ldots \vdash x\,x{:}\varphi}\,(\to E)}{f{:}\bullet\varphi \to \varphi, x{:}\mu.(\bullet\mathbf{0} \to \varphi) \vdash f\,(x\,x){:}\varphi}\,(\to I)}{f{:}\bullet\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi}\,(\leq)}{f{:}\bullet\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}(\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi) \to \varphi}\,(\leq)$$

The other is by weakening the subderivation typing the subterm $x\,x$, to give $\mathcal{D}_2''$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash f{:}\bullet\varphi \to \varphi}\,(\textsc{var}) \quad \dfrac{\dfrac{\dfrac{}{\ldots \vdash x{:}\mu.(\bullet\mathbf{0} \to \varphi)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi}\,(\leq) \quad \dfrac{\dfrac{}{\ldots \vdash x{:}\mu.(\bullet\mathbf{0} \to \varphi)}\,(\textsc{var})}{\ldots \vdash x{:}\bullet\mu.(\bullet\mathbf{0} \to \varphi)}\,(\leq)}{\ldots \vdash x\,x{:}\varphi}\,(\to E)}{\ldots \vdash x\,x{:}\bullet\varphi}\,(\leq)}{f{:}\bullet\varphi \to \varphi, x{:}\mu.(\bullet\mathbf{0} \to \varphi) \vdash f\,(x\,x){:}\varphi}\,(\to I)}{f{:}\bullet\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi}\,(\leq)}{f{:}\bullet\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}(\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi) \to \varphi}\,(\leq)$$

We can now type the application $\delta_f\,\delta_f$ itself:

$$\dfrac{\overbrace{f{:}\bullet\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}(\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi) \to \varphi}^{\mathcal{D}_2' \text{ or } \mathcal{D}_2''} \quad \overbrace{f{:}\bullet\varphi \to \varphi \vdash \lambda x.\,f\,(x\,x){:}\bullet\mu.(\bullet\mathbf{0} \to \varphi) \to \varphi}^{\mathcal{D}_1}}{f{:}\bullet\varphi \to \varphi \vdash \delta_f\,\delta_f{:}\varphi}\,(\to E)$$

A key observation motivating our approach to type inference is that there is no *substitution* of variables that can transform the first typing into the second one, not even modulo the subtyping relation. This is part of the reason for the necessity of insertion variables and the insertion operation itself, which allow us to place bullets at locations which would not be possible using the substitution operation alone. However,

this is not the whole story: to type the term $\delta_f\,\delta_f$, we had to use two *different* typings - one containing bullets, and the other not. In the system with insertion variables, the typing that we infer for the term δ_f is

$$\langle\{\,f\!:\iota_4\iota_3\iota_2\,\varphi_1 \to \iota_4\,\varphi_2\,\},\iota_2\,\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_1) \to \iota_4\,\varphi_2\rangle \tag{9.7}$$

The important point to note here is that the insertion variables in this typing, as well as allowing us to prefix the recursive type that forms the left-hand side of the arrow type with a bullet by using the insertion $[\iota_2 \mapsto \bullet]$, also allow for the possibility of obtaining the other typing via the empty insertion which removes all insertion variables. Thus, using insertion variables enables us to defer, until the last possible moment, the decision of whether to infer a type with or without bullets. More fundamentally, though, what insertion variables serve to do is mark the places in a typing derivation where the subtyping rule can be used (to introduce bullets into a type). Notice that the *expansion* variables of Kfoury and Wells [75] perform exactly the same function, marking the points in a derivation where a subderivation may be duplicated and used as the premises for an intersection *introduction* rule.

The typing above is also more general than just the two particular typings we have considered that allow us to type the self application $\delta_f\,\delta_f$. In fact, for all $n,m \in \mathbb{N}$ the term $\lambda x.\,f\,(x\,x)$ has the typing:

$$\langle\{\,f\!:\bullet^n\,\varphi_1 \to \varphi_2\,\},\bullet^n\,\mu.(\bullet^m\,\mathbf{0} \to \varphi_1) \to \varphi_2\rangle$$

None of these typings are related to any of the others via the subtyping relation $\leq$, meaning that none can be considered principal, but each of these typings can be generated from 9.7 above. We conjecture that it is *this* typing which is principal for the term $\lambda x.\,f\,(x\,x)$, i.e. any valid typing for the term can be generated from it by applying some operation.

Now to finish off, let us examine in detail exactly how the type inference procedure generates the typing for the term $\delta_f\,\delta_f$ that we gave above, with all its insertion variables.

For completeness, we will proceed from the very beginning and so start with the inference of a typing for $x\,x$. For each occurrence of x, a fresh type variable is used to generate the typings $\langle\{\,x\!:\!\varphi_1\,\},\varphi_1\rangle$ and $\langle\{\,x\!:\!\varphi_2\,\},\varphi_2\rangle$. To infer a typing for the term $x\,x$ we first unify φ_1 (the type of the first occurrence of x) with the arrow type $\iota_1\,\varphi_2 \to \varphi_3$, constructed from the type of the second occurrence of x and fresh insertion and type variables. This yields the type substitution $\mathsf{O}_1 = [\varphi_1 \mapsto \iota_1\,\varphi_2 \to \varphi_3]$, which is then applied to the type environments in the typings for each occurrence of x giving $\{\,x\!:\iota_1\,\varphi_2 \to \varphi_3\,\}$ and $\{\,x\!:\!\varphi_2\,\}$. Unifying these requires unifying the two types $\iota_1\,\varphi_2 \to \varphi_3$ and φ_2. Since φ_2 occurs in the type $\iota_1\,\varphi_2 \to \varphi_3$, unifying them involves generating a recursive type, which is done by first inserting a bullet at ι_1, and then promoting the re-occuring type variable φ_2 and recursively closing the type. This yields the operation $\mathsf{O}_2 = [\varphi_2 \mapsto \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)] \circ [\iota_1 \mapsto \iota_1 \bullet]$, and so φ_2 gets replaced by a recursive type. The penultimate step is to combine the environments $\mathsf{O}_2 \circ \mathsf{O}_1(\{\,x\!:\!\varphi_1\,\}) = \{\,x\!:\iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \varphi_3\,\}$ and $\mathsf{O}_2 \circ \mathsf{O}_1(\{\,x\!:\!\varphi_2\,\}) = \{\,x\!:\!\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\,\}$. It is easy to verify that the types in these two environments are *inferrably* equivalent, that is $\vdash \iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \varphi_3 \leq \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)$ and $\vdash \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \leq \iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \varphi_3$. Therefore, we take the *smaller* of the two types for the combined environment

$$\mathsf{O}_2 \circ \mathsf{O}_1(\{\,x\!:\!\varphi_1\,\}) \cup \mathsf{O}_2 \circ \mathsf{O}_1(\{\,x\!:\!\varphi_2\,\}) = \{\,x\!:\!\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\,\}$$

The final step is to push a fresh insertion variable onto the typing:

$$\begin{aligned}\mathsf{Type}(x\,x) &= \langle \mathsf{iPush}[\iota_2](\{x{:}\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\}) \cup \emptyset, \mathsf{iPush}[\iota_2](\mathsf{O}_2 \circ \mathsf{O}_1(\varphi_3))\rangle \\ &= \langle \{x{:}\iota_2\,\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\}, \iota_2\,\varphi_3\rangle\end{aligned}$$

Using the typing that we have just inferred, the inference procedure for the term $f\,(x\,x)$ follows similar lines. A fresh type variable forms the typing $\langle\{f{:}\varphi_4\},\varphi_4\rangle$ for the subterm f. This is then unified with an arrow type that allows the application with $x\,x$ to be typed, yielding the type subtitution $[\varphi_4 \mapsto \iota_3\iota_2\,\varphi_3 \to \varphi_5]$. Unifying the type environments of the two subterms is unnecessary since they are disjoint in their variable range. Thus, the typing for the whole term is again obtained by pushing a fresh insertion variable onto the type in the environment associated with f and the result type:

$$\mathsf{Type}(f\,(x\,x)) = \langle\{f{:}\iota_4\iota_3\iota_2\,\varphi_3 \to \iota_4\,\varphi_5,\ x{:}\iota_2\,\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\}, \iota_4\,\varphi_5\rangle$$

The typing for the term $\lambda x.\,f\,(x\,x)$ is obtained straightforwardly by abstracting over the type for x:

$$\mathsf{Type}(\lambda x.\,f\,(x\,x)) = \langle\{f{:}\iota_4\iota_3\iota_2\,\varphi_3 \to \iota_4\,\varphi_5\}, \iota_2\,\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_4\,\varphi_5\rangle$$

At this point we arrive to where we ran into trouble previously - remember that type inference without insertion variables failed because we could not unify the types inferred for the two occurrences of the subterm $\lambda x.\,f\,(x\,x)$. Having inferred a typing *with* insertion variables, however, the unification succeeds. Taking a fresh instance of the above typing for the right-hand occurrence of δ_f gives us the two typings:

$$\begin{aligned}\langle\Pi_1,\sigma\rangle &= \langle\{f{:}\iota_4\iota_3\iota_2\,\varphi_3 \to \iota_4\,\varphi_5\}, \iota_2\,\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_4\,\varphi_5\rangle \\ \langle\Pi_2,\tau\rangle &= \langle\{f{:}\iota_8\iota_7\iota_6\,\varphi_8 \to \iota_8\,\varphi_{10}\}, \iota_6\,\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_8\,\varphi_{10}\rangle\end{aligned}$$

The unification then proceeds as follows. We have underlined the unification call which led to failure in the approach without using insertion variables. This call is now easily handled because the insertion variable ι_6 prefixing the right-hand recursive type is able to 'consume' the bullet prefixing the left-hand recursive type:

$$\begin{aligned}\mathsf{O} &= \mathsf{Unify}^{\mu}_{\leq}(\sigma, \mathsf{iPush}[\iota_9](\tau) \to \varphi_{11}) \\ &= \mathsf{Unify}^{\mu}_{\leq}(\iota_2\,\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_4\,\varphi_5, (\iota_9\iota_6\,\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_9\iota_8\,\varphi_{10}) \to \varphi_{11}) \\ &= \mathsf{O}_2 \circ \mathsf{O}_1 \text{ where} \\ &\quad \mathsf{O}_1 = \mathsf{Unify}^{\mu}_{\leq}(\iota_9\iota_6\,\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_9\iota_8\,\varphi_{10}, \iota_2\,\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)) \\ &\qquad = \mathsf{Unify}^{\mu}_{\leq}(\iota_9\iota_6\,\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_9\iota_8\,\varphi_{10}, \iota_2\iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_2\,\varphi_3) \\ &\qquad = \mathsf{O}_4 \circ \mathsf{O}_3 \text{ where} \\ &\qquad\quad \mathsf{O}_3 = \mathsf{Unify}^{\mu}_{\leq}(\iota_2\iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3), \iota_9\iota_6\,\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8)) \\ &\qquad\qquad = \mathsf{O}_6 \circ \mathsf{O}_5 \text{ where} \\ &\qquad\qquad\quad \mathsf{O}_5 = [\iota_2 \mapsto \iota_9] \\ &\qquad\qquad\quad \mathsf{O}_6 = \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_5(\iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)), \mathsf{O}_5(\iota_6\,\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8))) \\ &\qquad\qquad\qquad = \underline{\mathsf{Unify}^{\mu}_{\leq}(\iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3), \iota_6\,\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8))} \\ &\qquad\qquad\qquad = \mathsf{O}_8 \circ \mathsf{O}_7 \text{ where}\end{aligned}$$

$$
\begin{aligned}
\mathsf{O}_7 &= [\iota_6 \mapsto \iota_1 \bullet] \\
\mathsf{O}_8 &= \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_7(\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)), \mathsf{O}_7(\mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8))) \\
&= \mathsf{Unify}^{\mu}_{\leq}(\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3), \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8)) \\
&= \mathsf{Unify}^{\mu}_{\leq}(\iota_1 \bullet \mathbf{0} \to \varphi_3, \iota_5 \bullet \mathbf{0} \to \varphi_8) \\
&= \mathsf{O}_{10} \circ \mathsf{O}_9 \text{ where} \\
&\quad \mathsf{O}_9 = \mathsf{Unify}^{\mu}_{\leq}(\iota_5 \bullet \mathbf{0}, \iota_1 \bullet \mathbf{0}) \\
&\quad\quad = \mathsf{O}_{12} \circ \mathsf{O}_{11} \text{ where} \\
&\quad\quad\quad \mathsf{O}_{11} = [\iota_5 \mapsto \iota_1] \\
&\quad\quad\quad \mathsf{O}_{12} = \mathsf{Unify}^{\mu}_{\leq}(\mathbf{0}, \mathbf{0}) \\
&\quad\quad\quad\quad = \mathsf{Id} \\
&\quad \mathsf{O}_{10} = \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_9(\varphi_3), \mathsf{O}_9((\varphi_8)) \\
&\quad\quad = \mathsf{Unify}^{\mu}_{\leq}(\varphi_3, \varphi_8) \\
&\quad\quad = [\varphi_3 \mapsto \varphi_8]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{O}_4 &= \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_3(\iota_9\iota_8\varphi_{10}), \mathsf{O}_3(\iota_2\varphi_3)) \\
&= \mathsf{Unify}^{\mu}_{\leq}(\iota_9\iota_8\varphi_{10}, \iota_9\varphi_8) \\
&= \mathsf{O}_{14} \circ \mathsf{O}_{13} \text{ where} \\
&\quad \mathsf{O}_{13} = [\iota_9 \mapsto \epsilon] \\
&\quad \mathsf{O}_{14} = \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_{13}(\iota_8\varphi_{10}), \varphi_8) \\
&\quad\quad = \mathsf{Unify}^{\mu}_{\leq}(\iota_8\varphi_{10}, \varphi_8) \\
&\quad\quad = \mathsf{O}_{16} \circ \mathsf{O}_{15} \text{ where} \\
&\quad\quad\quad \mathsf{O}_{15} = [\iota_8 \mapsto \epsilon] \\
&\quad\quad\quad \mathsf{O}_{16} = \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_{15}(\varphi_{10}), \varphi_8) \\
&\quad\quad\quad\quad = \mathsf{Unify}^{\mu}_{\leq}(\varphi_{10}, \varphi_8) \\
&\quad\quad\quad\quad = [\varphi_{10} \mapsto \varphi_8]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{O}_2 &= \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_1(\iota_4\varphi_5), \mathsf{O}_1(\varphi_{11})) \\
&= \mathsf{Unify}^{\mu}_{\leq}(\iota_4\varphi_5, \varphi_{11}) \\
&= \mathsf{O}_{18} \circ \mathsf{O}_{17} \text{ where} \\
&\quad \mathsf{O}_{17} = [\iota_4 \mapsto \epsilon] \\
&\quad \mathsf{O}_{18} = \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_{17}(\varphi_5), \varphi_{11}) \\
&\quad\quad = \mathsf{Unify}^{\mu}_{\leq}(\varphi_5, \varphi_{11}) \\
&\quad\quad = [\varphi_5 \mapsto \varphi_{11}]
\end{aligned}
$$

$$
\begin{aligned}
&= \mathsf{O}_{18} \circ \mathsf{O}_{17} \circ \mathsf{O}_{16} \circ \mathsf{O}_{15} \circ \mathsf{O}_{13} \circ \mathsf{O}_{10} \circ \mathsf{O}_{12} \circ \mathsf{O}_{11} \circ \mathsf{O}_7 \circ \mathsf{O}_5 \\
&= [\varphi_5 \mapsto \varphi_{11}] \circ [\iota_4 \mapsto \epsilon] \circ [\iota_{10} \mapsto \varphi_8] \circ [\iota_8 \mapsto \epsilon] \circ [\iota_9 \mapsto \epsilon] \\
&\qquad \circ [\varphi_3 \mapsto \varphi_8] \circ \mathsf{Id} \circ [\iota_5 \mapsto \iota_1] \circ [\iota_6 \mapsto \iota_1 \bullet] \circ [\iota_2 \mapsto \iota_9]
\end{aligned}
$$

Having successfully carried out the unification above, we must apply the resulting operation to the two type environments Π_1 and Π_2 and then unify them. This involves applying the operation to the type associated with f in each environment and unifying the resulting types, which gives the following:

$$
\begin{aligned}
\mathsf{O}' &= \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}(\iota_4\iota_3\iota_2\varphi_3 \to \iota_4\varphi_5), \mathsf{O}(\iota_8\iota_7\iota_6\varphi_8 \to \iota_8\varphi_{10})) \\
&= \mathsf{Unify}^{\mu}_{\leq}(\iota_3\varphi_3 \to \varphi_{11}, \iota_7\iota_1 \bullet \varphi_3 \to \varphi_3) \\
&= \mathsf{O}'_2 \circ \mathsf{O}'_1 \text{ where} \\
&\quad \mathsf{O}'_1 = \mathsf{Unify}^{\mu}_{\leq}(\iota_7\iota_1 \bullet \varphi_3, \iota_3\varphi_3)
\end{aligned}
$$

$$
\begin{aligned}
&= [\iota_3 \mapsto \iota_7\iota_1 \bullet] \\
\mathsf{O}'_2 &= \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}'_1(\varphi_{11}), \mathsf{O}'_1(\varphi_3)) \\
&= \mathsf{Unify}^{\mu}_{\leq}(\varphi_{11}, \varphi_3) \\
&= [\varphi_{11} \mapsto \varphi_3] \\
= \quad & [\varphi_{11} \mapsto \varphi_3] \circ [\iota_3 \mapsto \iota_7\iota_1 \bullet]
\end{aligned}
$$

Applying the combined operation $\mathsf{O}' \circ \mathsf{O}$ to the two type environments, in this case, results in identical environments and thus the combined type environment is the same:

$$
\begin{aligned}
\mathsf{O}' \circ \mathsf{O}(\Pi_1) \uplus \mathsf{O}' \circ \mathsf{O}(\Pi_2) &= \{ f{:}\iota_7\iota_1 \bullet \varphi_3 \to \varphi_3 \} \uplus \{ f{:}\iota_7\iota_1 \bullet \varphi_3 \to \varphi_3 \} \\
&= \{ f{:}\iota_7\iota_1 \bullet \varphi_3 \to \varphi_3 \}
\end{aligned}
$$

The typing that we will infer for $\delta_f\,\delta_f$ is formed by pushing a fresh insertion variable onto the type associated with every variable that is present in Π_1, and also onto the result type of the application:

$$
\begin{aligned}
\mathsf{Type}((\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x))) &= \langle \mathsf{iPush}[\iota_{10}](\Pi'_1) \cup \Pi'_2, \mathsf{iPush}[\iota_{10}](\mathsf{O}' \circ \mathsf{O}(\varphi_{11})) \rangle \\
&= \langle \mathsf{iPush}[\iota_{10}](\{ f{:}\iota_7\iota_1 \bullet \varphi_3 \to \varphi_3 \}) \cup \emptyset, \mathsf{iPush}[\iota_{10}](\varphi_3) \rangle \\
&= \langle \{ f{:}\iota_{10}\iota_7\iota_1 \bullet \varphi_3 \to \iota_{10}\,\varphi_3 \}, \iota_{10}\,\varphi_3 \rangle
\end{aligned}
$$

where $\Pi'_1 = \{ x{:}\sigma \mid x \in \Pi_1 \;\&\; x{:}\sigma \in \mathsf{O}' \circ \mathsf{O}(\Pi_1) \uplus \mathsf{O}' \circ \mathsf{O}(\Pi_2) \}$

$\Pi'_2 = \{ y{:}\tau \mid y \in \Pi_2 \;\&\; y \notin \Pi_1 \;\&\; y{:}\tau \in \mathsf{O}' \circ \mathsf{O}(\Pi_1) \uplus \mathsf{O}' \circ \mathsf{O}(\Pi_2) \}$

Lastly, the typing for $\mathbf{Y}$ is obtained by abstracting over the type for f:

$$
\begin{aligned}
\mathsf{Type}(\mathbf{Y}) &= \mathsf{Type}(\lambda f.(\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x))) \\
&= \langle \emptyset, (\iota_{10}\iota_7\iota_1 \bullet \varphi_3 \to \iota_{10}\,\varphi_3) \to \iota_{10}\,\varphi_3 \rangle
\end{aligned}
$$

The typing for $\mathbf{Y}$ that we demonstrated back in Section 8.2.3 can easily be obtained from this via the operation $[\iota_{10} \mapsto \epsilon] \circ [\iota_7 \mapsto \epsilon] \circ [\iota_1 \mapsto \epsilon]$ which removes all the insertion variables.

9.6.2. Incompleteness of the Algorithm

Due to the incompleteness of the unification procedure we defined in the previous section, our type inference procedure also fails to be complete. In other words, there are typeable terms for which our procedure *cannot* infer a type. An example of such a term is the λ-term $\mathbf{Y}' = \mathbf{Y}\,(\lambda xy.y\,(xy))$. Notice that this term is also a fixed point combinator:

$$
\begin{aligned}
\mathbf{Y}'\,M &= \mathbf{Y}\,(\lambda xy.y\,(xy))\,M \\
&=_{\beta} (\lambda xy.y\,(xy))\,(\mathbf{Y}\,(\lambda xy.y\,(xy)))\,M \\
&\to_{\beta} (\lambda y.y\,(\mathbf{Y}\,(\lambda xy.y\,(xy))\,y))\,M \\
&\to_{\beta} M\,(\mathbf{Y}\,(\lambda xy.y\,(xy))\,M) \\
&= M\,(\mathbf{Y}'\,M)
\end{aligned}
$$

As such, it can be assigned the characteristic fixed-point operator type $\sigma_{\mathbf{Y}} = (\bullet\varphi \to \varphi) \to \varphi$. Take the following derivation $\mathcal{D}$, in which $\Pi = \{x{:}\bullet\sigma_{\mathbf{Y}}, y{:}\bullet\varphi \to \varphi\}$ (notice that $\bullet\sigma_{\mathbf{Y}} = \bullet((\bullet\varphi \to \varphi) \to \varphi) \simeq (\bullet\bullet\varphi \to \bullet\varphi) \to \bullet\varphi$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash y{:}\bullet\varphi \to \varphi}\,(\textsc{var}) \quad \dfrac{\dfrac{\dfrac{}{\Pi \vdash x{:}\bullet\sigma_{\mathbf{Y}}}\,(\textsc{var})}{\Pi \vdash x{:}(\bullet\bullet\varphi \to \bullet\varphi) \to \bullet\varphi}\,(\leq) \quad \dfrac{\dfrac{}{\Pi \vdash y{:}\bullet\varphi \to \varphi}\,(\textsc{var})}{\Pi \vdash y{:}\bullet\bullet\varphi \to \bullet\varphi}\,(\leq)}{\Pi \vdash xy{:}\bullet\varphi}\,(\to E)}{\Pi \vdash y(xy){:}\varphi}\,(\to E)}{\{x{:}\bullet\sigma_{\mathbf{Y}}\} \vdash \lambda y.y(xy){:}\sigma_{\mathbf{Y}}}\,(\to I)}{\vdash \lambda xy.y(xy){:}\bullet\sigma_{\mathbf{Y}} \to \sigma_{\mathbf{Y}}}\,(\to I)$$

Notice that the type $(\bullet\sigma_{\mathbf{Y}} \to \sigma_{\mathbf{Y}}) \to \sigma_{\mathbf{Y}}$ is a substitution instance of the type $\sigma_{\mathbf{Y}}$ itself, and so we can easily assign it to the term $\mathbf{Y}$ (one way to do this is to take the derivations given in the previous section and replace all occurrences of the type variable φ with the type $\sigma_{\mathbf{Y}}$). Thus we can assign the type $\sigma_{\mathbf{Y}}$ to the term $\mathbf{Y}'$:

$$\dfrac{\begin{array}{c}\\ \vdash \mathbf{Y}{:}(\bullet\sigma_{\mathbf{Y}} \to \sigma_{\mathbf{Y}}) \to \sigma_{\mathbf{Y}}\end{array} \quad \begin{array}{c}\mathcal{D}\\ \vdash \lambda xy.y(xy){:}\bullet\sigma_{\mathbf{Y}} \to \sigma_{\mathbf{Y}}\end{array}}{\vdash \mathbf{Y}'{:}\sigma_{\mathbf{Y}}}\,(\to E)$$

We will now show where type inference for the term $\mathbf{Y}'$ breaks down. As we have seen, we are able to infer a type for the term $\mathbf{Y}$. We leave it as an exercise to the reader to verify that the our type inference procedure returns the following typing for the term $\lambda xy.y(xy)$ (or an α-equivalent one, that is equivalent up to the renaming of type and insertion variables):

$$\mathsf{Type}(\lambda xy.y(xy)) = \langle \emptyset, ((\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1) \to (\iota_5\iota_4\iota_3\varphi_1 \to \iota_5\varphi_2) \to \iota_5\varphi_2 \rangle$$

This means that type inference fails because the unification procedure fails when it attempts to make their respective types compatible for application. The execution of the relevant call to the unification procedure is given below:

$$\begin{aligned}
\mathsf{O} &= \mathsf{Unify}^{\mu}_{\leq}((\iota_{10}\iota_7\iota_1\bullet\varphi_3 \to \iota_{10}\varphi_3) \to \iota_{10}\varphi_3,\\
&\qquad (((\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1) \to (\iota_5\iota_4\iota_3\varphi_1 \to \iota_5\varphi_2) \to \iota_5\varphi_2) \to \varphi_4)\\
&= \mathsf{O}_2 \circ \mathsf{O}_1 \text{ where}\\
&\quad \mathsf{O}_1 = \mathsf{Unify}^{\mu}_{\leq}(((\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1) \to (\iota_5\iota_4\iota_3\varphi_1 \to \iota_5\varphi_2) \to \iota_5\varphi_2,\\
&\qquad\qquad \iota_{10}\iota_7\iota_1\bullet\varphi_3 \to \iota_{10}\varphi_3)\\
&\quad\ = \mathsf{O}_4 \circ \mathsf{O}_3 \text{ where}\\
&\quad\quad \mathsf{O}_3 = \mathsf{Unify}^{\mu}_{\leq}(\iota_{10}\iota_7\iota_1\bullet\varphi_3, (\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1)\\
&\quad\quad\ = \mathsf{O}_6 \circ \mathsf{O}_5 \text{ where}\\
&\quad\quad\quad \mathsf{O}_5 = [\iota_{10} \mapsto \epsilon]\\
&\quad\quad\quad \mathsf{O}_6 = \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_5(\iota_7\iota_1\bullet\varphi_3), \mathsf{O}_5((\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1))\\
&\quad\quad\quad\ = \mathsf{Unify}^{\mu}_{\leq}(\iota_7\iota_1\bullet\varphi_3, (\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1)\\
&\quad\quad\quad\ = \mathsf{O}_8 \circ \mathsf{O}_7 \text{ where}\\
&\quad\quad\quad\quad \mathsf{O}_7 = [\iota_7 \mapsto \epsilon]\\
&\quad\quad\quad\quad \mathsf{O}_8 = \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_7(\iota_1\bullet\varphi_3), \mathsf{O}_7((\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1))\\
&\quad\quad\quad\quad\ = \mathsf{Unify}^{\mu}_{\leq}(\iota_1\bullet\varphi_3, (\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1)
\end{aligned}$$

$$
\begin{aligned}
&= \mathsf{O}_{10} \circ \mathsf{O}_9 \text{ where} \\
&\quad \mathsf{O}_9 \;=\; [\iota_1 \mapsto \epsilon] \\
&\quad \mathsf{O}_{10} \;=\; \mathsf{Unify}^{\mu}_{\leq}(\bullet\varphi_3, \mathsf{O}_9((\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1)) \\
&\qquad\quad = \mathsf{Unify}^{\mu}_{\leq}(\bullet\varphi_3, (\iota_3\iota_2\iota_4\iota_3\varphi_1 \to \iota_3\iota_2\varphi_2) \to \iota_3\varphi_1) \\
&\qquad\quad = \textsc{fail}
\end{aligned}
$$

As can easily be seen, the unification procedure fails at the point that it has to unify a bulleted type variable with a function type, since there is no unification inference rule that matches this pattern of input types. This is also exactly the case that we highlighted in the previous section when mentioning the incompleteness of unification.

A possible solution to the problem of unifying a bulleted type variable $\bullet\varphi$ with a function type $\kappa_1 \to \kappa_2$ (in which the type variable φ itself does *not* occur, of course) is to unify the latter type with a *freshly generated* function type $\bullet\varphi_1 \to \bullet\varphi_2$. If this succeeds, it returns an operation O such that $\mathsf{O}(\bullet\varphi_1 \to \bullet\varphi_2) \leq \mathsf{O}(\kappa_1 \to \kappa_2)$ (or $\mathsf{O}(\kappa_1 \to \kappa_2) \leq \mathsf{O}(\bullet\varphi_1 \to \bullet\varphi_2)$ depending on which way round we attempt the unification). We can then build a unifying operation for the original type $\bullet\varphi$, namely $\mathsf{O} \circ [\varphi \mapsto \varphi_1 \to \varphi_2]$. Notice that since φ does not occur in $\kappa_1 \to \kappa_2$, it is the case that $[\varphi \mapsto \varphi_1 \to \varphi_2](\kappa_1 \to \kappa_2) = \kappa_1 \to \kappa_2$ and so $\mathsf{O} \circ [\varphi \mapsto \varphi_1 \to \varphi_2](\kappa_1 \to \kappa_2) = \mathsf{O}(\kappa_1 \to \kappa_2)$. Thus:

$$
\begin{aligned}
\mathsf{O} \circ [\varphi \mapsto \varphi_1 \to \varphi_2](\bullet\varphi) &= \mathsf{O}(\bullet\varphi_1 \to \bullet\varphi_2) \\
&\leq \mathsf{O}(\kappa_1 \to \kappa_2) = \mathsf{O} \circ [\varphi \mapsto \varphi_1 \to \varphi_2](\kappa_1 \to \kappa_2)
\end{aligned}
$$

The reason that we have not incorporated this approach into our unification algorithm is that we would then have been unable to prove its decidability, or more accurately its *termination*. This is due to the proof technique that we have used, which is based on the structural closure of a type. The approach we have just outlined introduces *fresh* type variables which are not present in either of the original types that we were trying to unify. This means that, in a unification inference system incorporating this approach, the subtype statements in subderivations are not guaranteed to be in the structural closure of the types in the final conclusion. Thus, our technique for proving termination would no longer be valid.

9.6.3. On Principal Typings

Although we have not yet been able to show a formal principal typings property for our variant of Nakano's type assignment, we do believe that such a property holds, since there is evidence to believe that our type inference procedure infers such principal typings (when it succeeds in inferring a typing at all, of course).

We hinted at this above when we talked about the different typings that can be assigned to the λ-term $\lambda x.f\,(x\,x)$, and their role in typing the fixed point combinator $\mathbf{Y}$. There, we noted that many of the (subtype-incompatible) typings assignable to this term can be generated by applying some composition of insertion operations to the typing returned by our inference algorithm, and that furthermore, these typing could not be obtained via substitution alone (from a typing without insertion variables).

Another example supporting our conjecture of principal typings is the typing that is inferred by our algorithm for the term $\lambda xy.y\,(x\,y)$. The 'vanilla', principal Curry type for this term is

$$((\varphi_1 \to \varphi_2) \to \varphi_1) \to (\varphi_1 \to \varphi_2) \to \varphi_2$$

Again, this is assignable to the term in our system too. Notice, though, that in the Nakano system it can also be assigned the type

$$((\bullet\bullet\varphi_1 \to \bullet\varphi_2) \to \bullet\varphi_1) \to (\bullet\varphi_1 \to \varphi_2) \to \varphi_2$$

In fact, it was a variant of *this* type (in which the type variables φ_1 and φ_2 were unified) that we used in order to type the fixed point combinator $\mathbf{Y}' = \mathbf{Y}(\lambda xy.y(xy))$. The important point to note is that neither one of these types is a subtype of the other, and furthermore we cannot generate the second one from the first via substitution alone, since the latter type contains both the type variables φ_1 and φ_2 in both their bulleted and unbulleted (or double-bulleted, in the case of φ_1) form. Both of these types, however, *can* be generated from the typing inferred by our algorithm. The former using the operation $[\iota_5 \mapsto \epsilon] \circ [\iota_4 \mapsto \epsilon] \circ [\iota_3 \mapsto \epsilon] \circ [\iota_2 \mapsto \epsilon]$, and the latter using the operation $[\iota_5 \mapsto \epsilon] \circ [\iota_4 \mapsto \epsilon] \circ [\iota_3 \mapsto \bullet] \circ [\iota_2 \mapsto \epsilon]$.

We will give one more example to bolster the case for the principal typings conjecture, that of the familiar λ-term $\mathbf{S} = \lambda xyz.xz(yz)$. In the simply typed λ-calculus, of course, **S** has the (principal) type

$$(\varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3$$

In Nakano's sytem (and thus also our variant), it can also be assigned that type. However, as with the previous example, it is possible to assign it another type which is not obtainable from the Curry type either via the subtype relation or via substitution:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash x{:}\varphi_1 \to \varphi_2 \to \varphi_3}\,(\textsc{var}) \quad \dfrac{}{\Pi \vdash z{:}\varphi_1}\,(\textsc{var})}{\Pi \vdash xz{:}\varphi_2 \to \varphi_3}\,(\to E) \quad \dfrac{\dfrac{}{\Pi \vdash y{:}\bullet\varphi_1 \to \varphi_2}\,(\textsc{var}) \quad \dfrac{\dfrac{}{\Pi \vdash z{:}\varphi_1}\,(\textsc{var})}{\Pi \vdash z{:}\bullet\varphi_1}\,(\leq)}{\Pi \vdash yz{:}\varphi_2}\,(\to E)}{\{x{:}\varphi_1 \to \varphi_2 \to \varphi_3, y{:}\bullet\varphi_1 \to \varphi_2, z{:}\varphi_1\} \vdash xz(yz){:}\varphi_3}\,(\to E)}{\{x{:}\varphi_1 \to \varphi_2 \to \varphi_3, y{:}\bullet\varphi_1 \to \varphi_2\} \vdash \lambda z.xz(yz){:} \to \varphi_1 \to \varphi_3}\,(\to I)}{\{x{:}\varphi_1 \to \varphi_2 \to \varphi_3\} \vdash \lambda yz.xz(yz){:}(\bullet\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3}\,(\to I)}{\vdash \lambda xyz.xz(yz){:}(\varphi_1 \to \varphi_2 \to \varphi_3) \to (\bullet\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3}\,(\to I)$$

where $\Pi = \{x{:}\varphi_1 \to \varphi_2 \to \varphi_3, y{:}\bullet\varphi_1 \to \varphi_2, z{:}\varphi_1\}$.

The typing inferred by our algorithm for the term **S** is:

$$\mathsf{Type}(\lambda xyz.xz(yz)) = \langle \emptyset, (\iota_5\iota_1\varphi_1 \to \iota_5\iota_4\iota_3\varphi_2 \to \iota_5\varphi_3) \to (\iota_3\iota_2\varphi_1 \to \iota_3\varphi_2) \to \iota_5\varphi_1 \to \iota_5\varphi_3 \rangle$$

Notice that the type we assigned to **S** above can be generated from this type via the operation

$$[\iota_5 \mapsto \epsilon] \circ [\iota_4 \mapsto \epsilon] \circ [\iota_3 \mapsto \epsilon] \circ [\iota_2 \mapsto \bullet] \circ [\iota_1 \mapsto \epsilon]$$

that removes all the insertion variables with the exception of ι_2, which it replaces with a bullet. Another type that can be assigned to **S** is

$$(\varphi_1 \to \bullet\varphi_2 \to \varphi_3) \to (\bullet\varphi_1 \to \bullet\varphi_2) \to \varphi_1 \to \varphi_3$$

The interesting thing about this type is that, while it is not related to the previous two types we have considered via subtyping, nor obtainable from the Curry type via substitution, it can be generated from

the inferred typing in two different ways: the first, via the operation

$$[\iota_5 \mapsto \epsilon] \circ [\iota_4 \mapsto \epsilon] \circ [\iota_3 \mapsto \bullet] \circ [\iota_2 \mapsto \epsilon] \circ [\iota_1 \mapsto \epsilon]$$

which removes all insertion variables except ι_3, which it replaces with a bullet; the second way is via the operation

$$[\varphi_2 \mapsto \bullet\varphi_2] \circ [\iota_5 \mapsto \epsilon] \circ [\iota_4 \mapsto \epsilon] \circ [\iota_3 \mapsto \epsilon] \circ [\iota_2 \mapsto \bullet] \circ [\iota_1 \mapsto \epsilon]$$

that removes all the insertion variables except ι_2 which it replaces with a bullet, and substitutes the type variable φ_2 with a bulleted version of itself.

Lastly, notice that the following three types assignable to **S** can be generated either by replacing insertion variables ι_1, ι_4 and ι_5 respectively by bullets, or via empty insertions and the subtype relation since they are all supertypes of the Curry type that we gave above:

$$(\bullet\varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3$$
$$(\varphi_1 \to \bullet\varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3$$
$$(\bullet\varphi_1 \to \bullet\varphi_2 \to \bullet\varphi_3) \to (\varphi_1 \to \varphi_2) \to \bullet\varphi_1 \to \bullet\varphi_3$$

Thus, there is a certain degree of redundancy in the operations that we can use to obtain typings. The main point however, is that all these typings are obtainable in *some* way, only obtainable because we have incorporated insertion variables into the system.

We end this section by remarking that, while there is evidence to believe that our system has a principal typings property, this property as it applies to our system can only partially hold, or hold modulo a stronger equivalence relation on types than the one we have defined through the $\leq$ subtyping relation. The reason for this is that the subtyping relation we have defined only relates recursive types via a *finitary* unfolding - that is, two recursive types are equivalent ($\simeq$) to one another if and only if they can both be unfolded some finite number of times such that they become the same type. This is *weak* μ-equality. A stronger notion of equality on recursive types can be obtained if we consider the *infinite* unfolding - thus, two recursive types are *strongly* μ-equivalent if and only if they have the same infinite unfolding. For example, the two types $\mu.(\varphi \to \bullet\mathbf{0})$ and $\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0})$ are strongly μ-equivalent, but they are *not* weakly μ-equivalent - that is, there is no finitary unfolding of these types such that they are (syntactically) equal.

This has a bearing on the (conjectured) principle typings property of our system, since it may be that a particular term can be assigned two (or more) typings which are strongly μ-equivalent but not weakly μ-equivalent, and so there is no way to generalise one to the other via the $\leq$ subtyping relation. Thus, neither (or none) of the typings can be considered 'principal', in the strictest sense of the word.

Take, for example, the term $\mathbf{K}' = \lambda xyz.x$. Using the types that we gave above, notice that the following (weak) μ-equalities hold:

$$\begin{aligned}\bullet\mu.(\varphi \to \bullet\mathbf{0}) \to \mu.(\varphi \to \bullet\mathbf{0}) &\simeq \bullet\mu.(\varphi \to \bullet\mathbf{0}) \to \varphi \to \bullet\mu.(\varphi \to \bullet\mathbf{0}) \\ &\simeq \bullet\mu.(\varphi \to \bullet\mathbf{0}) \to \varphi \to \bullet\varphi \to \bullet\bullet\mu.(\varphi \to \bullet\mathbf{0})\end{aligned}$$

$$\bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \to \mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0})$$
$$\simeq \bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \to \varphi \to \bullet\varphi \to \bullet\bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0})$$

Let $\sigma_1 = \mu.(\varphi \to \bullet\mathbf{0})$ and $\sigma_2 = \mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0})$. Then for each $i \in \{1,2\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x{:}\bullet\sigma_i, y{:}\varphi, z{:}\bullet\varphi\} \vdash x{:}\bullet\sigma_i}(\textsc{var})}{\{x{:}\bullet\sigma_i, y{:}\varphi, z{:}\bullet\varphi\} \vdash x{:}\bullet\bullet\sigma_i}(\leq)}{\{x{:}\bullet\sigma_i, y{:}\varphi\} \vdash \lambda z.x{:}\bullet\varphi \to \bullet\bullet\sigma_i}(\to I)}{\{x{:}\bullet\sigma_i\} \vdash \lambda yz.x{:}\varphi \to \bullet\varphi \to \bullet\bullet\sigma_i}(\to I)}{\vdash \lambda xyz.x{:}\bullet\sigma_i \to \varphi \to \bullet\varphi \to \bullet\bullet\sigma_i}(\to I)}{\vdash \lambda xyz.x{:}\bullet\sigma_i \to \sigma_i}(\leq)$$

Thus, the term $\mathbf{K}'$ can be assigned both of the types $\bullet\sigma_1 \to \sigma_1$ and $\bullet\sigma_2 \to \sigma_2$. This, in itself, does not constitute a problem for our conjectured principal typings property, since the 'principal' typing for $\mathbf{K}'$ (i.e. the typing inferred by our algorithm) is

$$\mathsf{Type}(\mathbf{K}') = \langle \emptyset, \varphi_1 \to \varphi_2 \to \varphi_3 \to \varphi_1 \rangle$$

and both of the types we have assigned are supertypes of types that can be generated from this principal type via substitution:

$$\begin{aligned}\mathsf{O}_1(\varphi_1 \to \varphi_2 \to \varphi_3 \to \varphi_1) &= \mu.(\varphi \to \bullet\mathbf{0}) \to \varphi \to \varphi \to \mu.(\varphi \to \bullet\mathbf{0}) \\ &\leq \bullet\mu.(\varphi \to \bullet\mathbf{0}) \to \bullet\varphi \to \bullet\varphi \to \bullet\mu.(\varphi \to \bullet\mathbf{0}) \\ &\leq \bullet\mu.(\varphi \to \bullet\mathbf{0}) \to \bullet\varphi \to \varphi \to \bullet\mu.(\varphi \to \bullet\mathbf{0}) \\ &\leq \bullet\mu.(\varphi \to \bullet\mathbf{0}) \to \bullet\varphi \to \varphi \to \bullet\bullet\mu.(\varphi \to \bullet\mathbf{0})\end{aligned}$$

where $\mathsf{O}_1 = [\varphi_3 \mapsto \varphi] \circ [\varphi_2 \mapsto \varphi] \circ [\varphi_1 \mapsto \mu.(\varphi \to \bullet\mathbf{0})]$.

$$\begin{aligned}\mathsf{O}_2(\varphi_1 \to \varphi_2 \to \varphi_3 \to \varphi_1) &= \mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \to \varphi \to \varphi \to \mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \\ &\leq \bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \to \bullet\varphi \to \bullet\varphi \to \bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \\ &\leq \bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \to \bullet\varphi \to \varphi \to \bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \\ &\leq \bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \to \bullet\varphi \to \varphi \to \bullet\bullet\mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0})\end{aligned}$$

where $\mathsf{O}_2 = [\varphi_3 \mapsto \varphi] \circ [\varphi_2 \mapsto \varphi] \circ [\varphi_1 \mapsto \mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0})]$.

On the other hand, let us consider the term $\mathbf{YK}'$. We have seen that $\mathbf{Y}$ can be assigned types of the form $(\bullet\sigma \to \sigma) \to \sigma$ for any type σ, and so this allows us to type the term $\mathbf{YK}'$ in two different ways, since for each $i \in \{1,2\}$:

$$\dfrac{\vdash \mathbf{Y}{:}(\bullet\sigma_i \to \sigma_i) \to \sigma_i \qquad \vdash \mathbf{K}'{:}\bullet\sigma_i \to \sigma_i}{\vdash \mathbf{YK}'{:}\sigma_i}(\to E)$$

Thus, we can assign both the typings $\langle \emptyset, \sigma_1 \rangle$ and $\langle \emptyset, \sigma_2 \rangle$ to $\mathbf{YK}'$. The type inference algorithm returns

the typing $\langle \emptyset, \mu.(\varphi_1 \to \varphi_2 \to \bullet\mathbf{0}) \rangle$. Notice that

$$[\varphi_2 \mapsto \varphi] \circ [\varphi_1 \mapsto \varphi](\langle \emptyset, \mu.(\varphi_1 \to \varphi_2 \to \bullet\mathbf{0}) \rangle) = \langle \emptyset, \mu.(\varphi \to \varphi \to \bullet\mathbf{0}) \rangle$$
$$\leq \langle \emptyset, \mu.(\varphi \to \bullet\varphi \to \bullet\bullet\mathbf{0}) \rangle$$

However, the type σ_1, while *strongly* μ-equivalent to σ_2, is not itself directly obtainable from the 'principal' typing returned by the type inference algorithm. This leads us to formulate the following principal typings conjecture, stated modulo strong μ-equivalence.

Conjecture 9.77. *If* $\Pi \vdash M{:}\sigma$ *and* $\mathsf{Type}(M) = \langle \Pi', \tau \rangle$*, then there is an operation* O*, an environment* Π'' *and type* τ' *such that* $\mathsf{O}(\langle \Pi', \tau \rangle) = \langle \Pi'', \tau' \rangle$*, with* Π'' *strongly* μ*-equivalent to* Π *and* τ' *strongly* μ*-equivalent to* σ*.*

10. Extending Nakano Types to Featherweight Java

Having demonstrated in the previous chapter that some kind of type inference is possible for Nakano-style type assignment in the context of λ-calculus, we will continue by describing how we might now apply the concepts and approach of this system in an object-oriented setting, and present a type assignment system for FJ$^{\text{¢}}$ that assigns Nakano-style simple recursive structural types to programs. The aim of doing this is to obtain a type assignment system for FJ$^{\text{¢}}$ which is both semantic and fully decidable for all programs. By basing the system on the Nakano approach, we claim that we obtain a system with a semantic foundation; furthermore, since it is also a system of recursive types, this allows us to 'tame' the recursion inherent in the class-based object-oriented paradigm, and facilitates an effective algorithm approach to type inference.

Although we do not give a formal result showing the semantic nature of our system, we will work through several illustrative examples showing how they can be typed by our system. These will provide what we believe to be strong evidence in support of the semantic nature of the system. We will also discuss how type inference technique for Nakano's system that we outlined in the previous chapter might be merged with the type inference procedure for FJ$^{\text{¢}}$ that we gave in Section 7.3, and thus applied to the new system. Finally, we consider how to introduce intersections into this system, with the aim of gaining semantic completeness.

We see the work in this chapter as providing a starting point, upon which we can build both formal and practical results. The definitions, descriptions and examples in this chapter should be detailed enough to convince the reader that such results are feasible. As such, it can be considered to constitute a roadmap for future research.

10.1. Classes As Functions

Our aim in studying the Nakano system of recursive types, and techniques for the inference of its type assignment, is to be able to apply it in an object-oriented setting in order to obtain a semantic type assignment system for which typings can be inferred in a flexible and comprehensive manner. To see just how a system of recursive types can lead to a generally terminating type inference procedure, we will first examine the precise reason for the *non*-termination of type inference in the intersection type system. This, as we have remarked in Section 7.3, lies in the ability to define classes *recursively*.

The type assignment system of 'simple' types (whether intersections are allowed or not) studied in the first part of this thesis treats class definitions simply as lookup tables for fields and methods, and operates by 'unfolding' these definitions as many times as required for a given analysis, or to be able to type an object in a given context. Type analyses for objects are obtained via analyses of their method bodies. So, if a method body (or more precisely, the execution of the method body - i.e. the invocation

of the method) results in the creation of a new instance of its containing class, then the analysis of that method body will necessarily entail an analysis of this new instance obtained by again 'unfolding' the same class definition, resulting in a non-terminating loop.

Notice that this is not just a property or artefact of the type inference algorithm, but a property of the type assignment system itself, which allows instances of recursively defined classes to be assigned types of arbitrary size and, in general, arbitrary irregularity. For a semantic analysis, this is exactly what we want since it allows the system to capture the infinity of potential behaviours that such an object will have. However, for practical type inference of any kind of real-world object-oriented program, this expressive power is entirely prohibitive.

One possible approach to ensuring the termination of a type inference algorithm for this system would be to simply place a limit on the number of times any particular class definition can be 'unfolded' during type inference. In other words, we would be placing a limit on the contexts in which the program can (typeably) be placed in, this limit being defined as some maximum number of successive method calls. However, as well as resulting in a woefully incomplete inference algorithm this is, of course, overly restrictive from a practical programming point of view. Think of the list example of Section 6.3 - such an approach would impose a maximum length for any typeable list. While programmers are used to being limited on this front by hardware constraints, being limited in this way by the type system would be more or less unacceptable.

So, the question we must answer becomes: is there a way to limit the unfolding of class definitions during type analysis, while still allowing programs to be used in contexts consisting of arbitrary-length sequences of method calls? The key to answering this question affirmatively lies in viewing classes not just as lookup tables, but as *functions* from objects to objects - in other words, object *constructor* functions. We will make this interpretation more concrete, and explain how it leads to a new typing approach, by first considering a possible interpretation of classes in the ς-calculus. Abadi and Cardelli also describe such a translation in [2], similar to approach adopted by Reddy in his semantic model for Smalltalk [92]. We will discuss a slightly different translation, however, which more suits the way we will eventually type new objects. We will not use, for instance, Abadi and Cardelli's notion of *premethods*.

The principal feature of Abadi and Cardelli's encoding is that classes are themselves represented as objects, containing a special method named `new`. The class-based approach to creating objects by instantiation is translated to an invocation of the `new` method on the object that is the translation of the instantiated class, i.e. the class-based expression `new` $C(\vec{e})$ becomes the object calculus expression $\mathsf{classObj}_C$`.new(`$\vec{e}$`)`, where $\mathsf{classObj}_C$ is the translation of the class `C`. The `new` method of the class object $\mathsf{classObj}_C$ is defined such that it returns a new object equivalent to an instance of the class `C`, so it contains all the methods and fields declared in `C`.

Consider an FJ$^{¢}$ class:

```
class C {
    C1 f1; ... Cn fn;
    D1 m1(E x⃗) { return e1; }   ...  Dm mm(E x⃗) { return em; }
}
```

The corresponding C class object, $\mathsf{classObj}_C$, in the ς-calculus would be:

$$\mathsf{classObj}_C \triangleq [\texttt{new} = \varsigma(C).\lambda(\vec{g}_n).[\overrightarrow{f = g}_n, \overrightarrow{m = \varsigma(\texttt{this}).\lambda(\vec{x}).[\![e]\!]_m}]]$$

We will discuss the translation of the method bodies $[\![e_i]\!]$ shortly, but first we point out an important aspect of this translation. This is that the variable `this` in the translation of each method has been bound by the ς self binder, and so references to `this` within method bodies will correctly refer to the receiving object. Also the class name C has been self bound within the body of the special `new` method. This means that any occurrences of C in the bodies of the object's methods will also refer back to the translated class object, and so new instances of the class C itself can be created by its own methods. In fact, this latter observation is the crucial element in the translation. Apart from this we can see that the body of the `new` method is a (lambda) function that takes some arguments $(\vec{g}_n)$ which are then used as the values of the object's fields.

The translation $[\![e_i]\!]$ of the method bodies is slightly subtle. What the translation must do, quite straightforwardly, is convert `new` C`(...)` expressions into the form $\mathsf{classObj}_C$`.new`, translating the class C into a class object, and invoking its `new` method. There is an exception to this, however - when converting `new` C`(...)` expressions *within* the translation of the class C itself, we must refer to the self-bound variable C, and not translate the class into an object a second time.

How does this translation of classes to a calculus of pure objects and functions help us to see how we might better type class-based programs? Well, notice that in our translation, class objects contain the *sole* method `new` - its object nature plays no other role (in our translation) other than providing the `new` method for creating new instances. Since the body of the `new` method is a function, we might as well dispose of the notion of a class as an object, and treat it as a raw function. We must be careful, however, because the body of this function may well contain invocations of the `new` method on occurrences of the previously recursively bound variable C. We must also now treat these method invocations as recursive calls to the function itself. So, our view of classes has now moved from a ς-calculus object with a `new` method, to a recursively defined function named C`.new`:

$$\mathsf{class}_C \triangleq \mathbf{FIX}\, C\texttt{.new}.\lambda(\vec{g}_n).[\overrightarrow{f = g}_n, \overrightarrow{m = \varsigma(\texttt{this}).\lambda(\vec{x}).[\![e]\!]_m}]$$

To type such recursively defined functions, we use a (FIX) rule as follows:

$$\frac{\Gamma, g{:}\sigma \vdash M : \sigma}{\Gamma \vdash \mathbf{FIX}\, g.M : \sigma}\ (\textsc{fix})$$

So, we assume a type for the function identifier in the typing environment and try to assign the same type to the function body. If this is possible, then we can type the entire recursive definition itself with this type. This typing rule is based upon the recursive definition $\mathbf{FIX}\ g.M$ being a shorthand for $\mathbf{Y}(\lambda g.M)$, where $\mathbf{Y}$ is a fixed point operator with the type scheme $(A \to A) \to A$. As we have observed, in Nakano's system fixed point operators have the type scheme $(\bullet A \to A) \to A$, and so we must reformulate our typing rule for recursive definitions as follows:

$$\frac{\Gamma, g{:}\bullet\sigma \vdash M : \sigma}{\Gamma \vdash \mathbf{FIX}\, g.M : \sigma}\ (\textsc{fix})$$

That is, we must assume a *bulleted* version of the type we are trying to derive.

In the following section, we will define a type system for Featherweight that assigns Nakano types to programs. It will very much follow in the footsteps of the intersection type assignment system from the first part of this thesis. The key difference, however, will lie in our approach to assigning method types, which we will now do based on the above idea of treating the `new` keyword as denoting a recursively defined function that returns an object.

10.2. Nakano Types for Featherweight Java

In this section, we will define a type system for FJ$^{¢}$ that incorporates Nakano recursive types and insertion variables, which we will call FJ$\bullet\mu$. The types of this new system will essentially be the (Curry) types that we defined in Chapter 7 (i.e. field and method types), but augmented with recursive types, the bullet type constructor and insertion variables.

We will not give this notion of type assignment as full a treatment as we gave to the type assignment systems of Part I. Rather, we focus on formulating the details of Nakano-style type assignment for FJ$^{¢}$, motivating this formulation through typed examples and demonstrating that our approach points in the 'right direction'. We conjecture that the semantic properties (i.e. subject reduction and head-normalisation) enjoyed by Nakano's systems for the λ-calculus will hold for this system as well. We also believe that when intersection types are added as well, the full complement of semantic results (i.e. subject expansion and approximation) will follow.

We make a final remark before proceeding to the definition of Nakano type assignment for FJ$^{¢}$: for convenience of notation, we will reuse the meta-variables that range over pretypes from the previous chapter, and redefine the various lookup functions and operations that we defined there so that they apply to the new set of FJ$\bullet\mu$ pretypes. This is intentional, and the reader is asked to treat this chapter in (formal) isolation from the previous one, so that no confusion can arise.

Definition 10.1 (FJ$\bullet\mu$ Object Pretypes)**.** *1. The set of FJ$\bullet\mu$* object pretypes *(ranged over by π), and its (strict) subset of* functional *object pretypes (ranged over by ϕ) are defined by the following grammar, where de Bruijn indices* **n** *range over the set of natural numbers, φ ranges over a denumerable set of type variables, and ι ranges over a denumerable set of insertion variables:*

$$\begin{array}{lcccccccc} \pi & ::= & \varphi & | & \mathbf{n} & | & C & | & \bullet\pi & | \\ & & \iota\pi & | & \langle f{:}\pi \rangle & | & \phi & & & \\ \\ \phi & ::= & \langle m{:}(\vec{\pi}) \to \pi \rangle & & & | & \bullet\phi & | & \iota\phi & | & \mu.\phi \end{array}$$

2. *We use the shorthand notation $\bullet^n\pi$ (where $n \geq 0$) to denote the pretype π prefixed by n occurrences of the $\bullet$ operator, i.e. $\underbrace{\bullet \ldots \bullet}_{n \text{ times}}\pi$.*
3. *We use the shorthand notation $\vec{\iota}_n\pi$ (where $n \geq 0$) to denote the pretype π prefixed by each ι_k in turn, i.e. $\iota_1 \ldots \iota_n\pi$.*

The following lookup functions take an object pretype as input, and return a set of de Bruijn indices. In the first case, this is the set of free indices (those that do not correspond to a μ type constructor), and in the second case it is the set of such free indices that do not occur within the scope of a bullet type constructor. In the following definitions, the decrement (postfix) operator is taken from Definition 9.3.

Definition 10.2 (Free Variable Set)**.** *The function* FV *takes an object pretype π and returns the set of de Bruijn indices representing the free recursive 'variables' of π. It is defined inductively on the structure of pretypes as follows:*

$$\left.\begin{array}{l}\mathrm{FV}(\varphi)\\ \mathrm{FV}(C)\end{array}\right\} = \emptyset \qquad \left.\begin{array}{l}\mathrm{FV}(\bullet\pi)\\ \mathrm{FV}(\iota\pi)\\ \mathrm{FV}(\langle f:\pi\rangle)\end{array}\right\} = \mathrm{FV}(\pi)$$

$$\mathrm{FV}(\mathbf{n}) = \{\mathbf{n}\}$$

$$\mathrm{FV}(\mu.\phi) = \mathrm{FV}(\phi)\downarrow \qquad \mathrm{FV}(\langle m:(\vec{\pi}_n)\to\pi\rangle) = \left(\bigcup_{i\in\overline{n}}\mathrm{FV}(\pi_i)\right)\cup\mathrm{FV}(\pi)$$

Definition 10.3 (Raw Variable Set)**.** *The function* RAW_μ *takes an object pretype π and returns the set of its* raw *recursive variables - the free recursive variables (i.e. de Bruijn indices) occurring in π which do not occur within the scope of a $\bullet$. It is defined inductively on the structure of pretypes as follows:*

$$\mathrm{RAW}_\mu(\mathbf{n}) = \{\mathbf{n}\} \qquad \left.\begin{array}{l}\mathrm{RAW}_\mu(\iota\pi)\\ \mathrm{RAW}_\mu(\langle f:\pi\rangle)\end{array}\right\} = \mathrm{RAW}_\mu(\pi)$$

$$\left.\begin{array}{l}\mathrm{RAW}_\mu(\varphi)\\ \mathrm{RAW}_\mu(C)\\ \mathrm{RAW}_\mu(\bullet\pi)\end{array}\right\} = \emptyset \qquad \mathrm{RAW}_\mu(\mu.\phi) = \mathrm{RAW}_\mu(\phi)\downarrow$$

$$\mathrm{RAW}_\mu(\langle m:(\vec{\pi}_n)\to\pi\rangle) = \left(\bigcup_{i\in\overline{n}}\mathrm{RAW}_\mu(\pi_i)\right)\cup\mathrm{RAW}_\mu(\pi)$$

As before, these allows us to define a notion of *adequacy* for object pretypes.

Definition 10.4 (Adequacy)**.** *The set of* adequate *objects pretypes are those pretypes for which every μ binder binds at least one occurrence of its associated recursive variable, and every* bound *recursive variable occurs within the scope of a bullet. It is defined as the smallest set of pretypes satisfying the following conditions:*

1. *φ, $\mathbf{n}$ and C are adequate, for all φ, $\mathbf{n}$ and C;*
2. *if π is adequate, then so are $\bullet\pi$, $\iota\pi$ and $\langle f:\pi\rangle$;*
3. *if both π and each π_i (where $i\in\overline{n}$) are adequate, then so is $\langle m:(\vec{\pi}_n)\to\pi\rangle$;*
4. *if ϕ is adequate and $\mathbf{0}\in\mathrm{FV}(\phi)\setminus\mathrm{RAW}_\mu(\phi)$, then $\mu.\phi$ is adequate.*

This notion, in turn, allows us to define the set of proper FJ•μ *types*.

Definition 10.5 (FJ•μ Types)**.**

1. *We say that an object pretype π is* closed *when it contains no free recursive variables, i.e.* $\mathrm{FV}(\pi)=\emptyset$.
2. *We call an object pretype π a (proper)* type *whenever it is* both *adequate* and *closed. The meta-variables σ, τ, γ, α and β will be used to range over proper object types.*

Using these object types, we can now define a set of types which we will use when deriving types for instances of recursively defined classes. To match our intuition that classes define *functions* from objects to objects, which we call using the `new` syntax, the types that we will assign to classes will be (first order) function types constructed using object types.

Definition 10.6 (FJ•μ Class Types)**.** *FJ•μ* class *types are defined by the following grammar (where σ and τ range over object* types*):*

$$\delta \quad ::= \quad (\vec{\sigma}) \to \tau \quad | \quad \bullet\delta \quad | \quad \iota\delta$$

We will now define a subtyping relation on object and class types. This is defined in a similar way to the subtyping relation in Definition 9.11 - we merely add the obvious cases for field and methods types. First, though, we must define how to fold and unfold recursive types, as we did in the previous chapter.

Definition 10.7 (FJ•μ μ-substitution)**.** *A μ-substitution is a function from object pretypes to object pretypes. Let ϕ be a functional object pretype, then the μ-substitution $[\mathbf{n} \mapsto \mu.\phi]$ is defined by induction on the structure of pretypes simultaneously for every $\mathbf{n} \in \mathbb{N}$ as follows:*

$$\begin{aligned}
[\mathbf{n} \mapsto \mu.\phi](\varphi) &= \varphi \\
[\mathbf{n} \mapsto \mu.\phi](C) &= C \\
[\mathbf{n} \mapsto \mu.\phi](\mathbf{n}') &= \begin{cases} \mu.\phi & \text{if } \mathbf{n} = \mathbf{n}' \\ \mathbf{n}' & \text{otherwise} \end{cases} \\
[\mathbf{n} \mapsto \mu.\phi](\bullet\pi) &= \bullet([\mathbf{n} \mapsto \mu.\phi](\pi)) \\
[\mathbf{n} \mapsto \mu.\phi](\iota\pi) &= \iota([\mathbf{n} \mapsto \mu.\phi](\pi)) \\
[\mathbf{n} \mapsto \mu.\phi](\langle f{:}\pi\rangle) &= \langle f{:}([\mathbf{n} \mapsto \mu.\phi](\pi))\rangle \\
[\mathbf{n} \mapsto \mu.\phi](\langle m{:}(\vec{\pi}_n) \to \pi\rangle) &= \langle m{:}([\mathbf{n} \mapsto \mu.\phi](\pi_1), \ldots, [\mathbf{n} \mapsto \mu.\phi](\pi_n)) \\
&\qquad \to [\mathbf{n} \mapsto \mu.\phi](\pi)\rangle \\
[\mathbf{n} \mapsto \mu.\phi](\mu.\phi') &= \mu.([\mathbf{n}+1 \mapsto \mu.\phi](\phi'))
\end{aligned}$$

This allows us to define a subtyping relation on both object pretypes and then class types.

Definition 10.8 (FJ•μ Subtyping)**.** *1. The relation $\leq$ on object pretypes is defined as the smallest preorder on pretypes satisfying the following conditions:*

$$\pi \leq \bullet\pi \qquad \pi \leq \iota\pi \qquad \pi_1 \leq \pi_2 \Rightarrow \begin{cases} \bullet\pi_1 \leq \bullet\pi_2 \\ \iota\pi_1 \leq \iota\pi_2 \end{cases}$$

$$\bullet\iota\pi \leq \iota\bullet\pi \qquad \iota\bullet\pi \leq \bullet\iota\pi \qquad \iota_1\iota_2\pi \leq \iota_2\iota_1\pi$$

$$\bullet\langle f{:}\pi\rangle \leq \langle f{:}\bullet\pi\rangle \qquad \langle f{:}\bullet\pi\rangle \leq \bullet\langle f{:}\pi\rangle$$

$$\bullet\langle m{:}(\vec{\pi}) \to \pi\rangle \leq \langle m{:}(\overline{\bullet\pi}) \to \bullet\pi\rangle \qquad \langle m{:}(\overline{\bullet\pi}) \to \bullet\pi\rangle \leq \bullet\langle m{:}(\vec{\pi}) \to \pi\rangle$$
$$\iota\langle m{:}(\vec{\pi}) \to \pi\rangle \leq \langle m{:}(\overline{\iota\pi}) \to \iota\pi\rangle \qquad \langle m{:}(\overline{\iota\pi}) \to \iota\pi\rangle \leq \iota\langle m{:}(\vec{\pi}) \to \pi\rangle$$

$$\mu.\phi \leq [\mathbf{0} \mapsto \mu.\phi](\phi) \qquad [\mathbf{0} \mapsto \mu.\phi](\phi) \leq \mu.\phi$$

$$\phi_1 \leq \phi_2 \Rightarrow \mu.\phi_1 \leq \mu.\phi_2$$

$$\pi \leq \pi' \;\&\; \pi'_i \leq \pi_i \,(\forall\, i \in \overline{n}) \quad \Rightarrow \quad \langle m : (\vec{\pi}_n) \to \pi \rangle \leq \langle m : (\overrightarrow{\pi'}_n) \to \pi' \rangle$$

2. *The relation $\leq$ on class types is defined as the smallest preorder on class types satisfying the following conditions:*

$$\delta \leq \bullet\delta \qquad \delta \leq \iota\delta \qquad\qquad \delta_1 \leq \delta_2 \Rightarrow \begin{cases} \bullet\delta_1 \leq \bullet\delta_2 \\ \iota\delta_1 \leq \iota\delta_2 \end{cases}$$

$$\bullet\iota\delta \leq \iota\bullet\delta \qquad \iota\bullet\delta \leq \bullet\iota\delta \qquad\qquad \iota_1\iota_2\delta \leq \iota_2\iota_1\delta$$

$$\bullet((\vec{\sigma}) \to \tau) \leq (\overrightarrow{\bullet\sigma}) \to \bullet\tau \qquad (\overrightarrow{\bullet\sigma}) \to \bullet\tau \leq \bullet((\vec{\sigma}) \to \tau)$$
$$\iota((\vec{\sigma}) \to \tau) \leq (\overrightarrow{\iota\sigma}) \to \iota\tau \qquad (\overrightarrow{\iota\sigma}) \to \iota\tau \leq \iota((\vec{\sigma}) \to \tau)$$

$$\tau \leq \tau' \;\&\; \sigma'_i \leq \sigma_i \,(\forall\, i \in \overline{n}) \quad \Rightarrow \quad (\vec{\sigma}_n) \to \tau \leq (\overrightarrow{\sigma'}_n) \to \tau'$$

We now come to define the rules of our type assignment system. At this juncture, we are presented with a problem, or rather, we have a design decision to make. We explained in the previous section that we will view the expression `new C(`$\vec{e}$`)` as an *application* of the function `new C` to the field values $\vec{e}$. To type such an application, the obvious thing to do is to split the expression apart into function and operands and type them separately. So far, so good; but this brings us to the crucial question - how do we now type the *function* `new C`?

We can take one of two approaches here. The first follows in the tradition of the ς-calculus: we express our understanding of the object-oriented concept of classes in terms of the behaviour of another, perhaps better understood, computational medium via some translation. Our approach to typing the higher-level concepts, then, is to type the lower-level translation. We could decide to abandon the use of the Featherweight Java model, and develop a Nakano-style type system for the ς-calculus. In a sense, we have already taken this approach by discussing the further translation of FJ$^{\text{¢}}$ into a kind of λ-calculus, albeit one with records. We could 'go all the way', translating records into pure λ-calculus and then, in a sense, our job is done since Nakano's system is already defined for the λ-calculus. We can type FJ$^{\text{¢}}$ programs by translating them and typing the translation using Nakano's original system. The focus of our study would perhaps shift, then, to the question of the comparative reduction behaviour of program in the original model and their 'compiled' versions, and we would need to show that any properties we consider 'transfer back' to FJ$^{\text{¢}}$.

We will *not* take this approach. Indeed, theoretical study of the object-oriented programming model began in this way, and it was a frustration with having to deal with the technicalities of translation itself rather than being able to focus on the key issues that led to the development of the ς-calculus in the first place [28, Introduction]. While we believe it is right that an understanding and a consideration of the concepts of class-based programming in terms of more fine-grained components should *inform* our type analysis, it is our view that the actual analysis itself should be done at the *same* level of abstraction as that found in the object of study. For us, that level of abstraction is the Featherweight Java model.

$$(\text{VAR}): \frac{}{\Sigma;\Gamma, x{:}\sigma \vdash x{:}\sigma} \quad (\text{SUB}): \frac{\Sigma;\Gamma \vdash e{:}\sigma}{\Sigma;\Gamma \vdash e{:}\tau}\,(\sigma \leq \tau)$$

$$(\text{FLD}): \frac{\Sigma;\Gamma \vdash e{:}\langle f{:}\sigma\rangle}{\Sigma;\Gamma \vdash e.f{:}\sigma} \quad (\text{INVK}): \frac{\Sigma;\Gamma \vdash e{:}\langle m{:}(\vec{\sigma}_n)\to\tau\rangle \quad \Sigma;\Gamma \vdash e_i{:}\sigma_i \quad (\forall\, i \in \overline{n})}{\Sigma;\Gamma \vdash e.m(\vec{e}_n){:}\tau}$$

$$(\bullet): \frac{\bullet\Sigma;\bullet\Gamma \vdash e{:}\bullet\sigma}{\Sigma;\Gamma \vdash e{:}\sigma} \quad (\iota): \frac{\iota\Sigma;\iota\Gamma \vdash e{:}\iota\sigma}{\Sigma;\Gamma \vdash e{:}\sigma} \quad (\text{INST-OBJ}): \frac{\Sigma;\Gamma \vdash e_i{:}\sigma_i \quad (\forall\, i \in \overline{n})}{\Sigma;\Gamma \vdash \texttt{new}\ C(\vec{e}_n){:}C}\,(\mathcal{F}(C) = \vec{f}_n)$$

$$(\text{SELF-FLD}): \frac{}{\Sigma;\Gamma, \texttt{this}{:}C, f{:}\sigma \vdash \texttt{this}{:}\langle f{:}\sigma\rangle}\,(f \in \mathcal{F}(C)) \qquad (\text{SELF-METH}): \frac{}{\Sigma, C{:}(\vec{\tau})\to\sigma;\Gamma, \texttt{this}{:}C \vdash \texttt{this}{:}\sigma}$$

$$(\text{INST-FLD}): \frac{\Sigma;\Gamma \vdash e_i{:}\sigma_i \quad (\forall\, i \in \overline{n})}{\Sigma;\Gamma \vdash \texttt{new}\ C(\vec{e}_n){:}\langle f_j{:}\sigma_j\rangle}\,(\mathcal{F}(C) = \vec{f}_n,\, j \in \overline{n}) \qquad (\text{REC-METH}): \frac{\Sigma, C{:}\delta;\Gamma \vdash e_i{:}\tau_i \quad (\forall\, i \in \overline{n})}{\Sigma, C{:}\delta;\Gamma \vdash \texttt{new}\ C(\vec{e}_n){:}\sigma}\,(\delta \leq (\vec{\tau}_n)\to\sigma)$$

$$(\text{INST-METH}): \frac{\Sigma, C{:}\delta;\{\texttt{this}{:}C, x_1{:}\sigma_1, \ldots, x_{n'}{:}\sigma_{n'}, f_1{:}\tau_1, \ldots, f_n{:}\tau_n\} \vdash e_b{:}\gamma \qquad \Sigma;\Gamma \vdash e_i{:}\tau_i \quad (\forall\, i \in \overline{n})}{\Sigma;\Gamma \vdash \texttt{new}\ C(\vec{e}_n){:}\langle m{:}(\vec{\sigma}_{n'})\to\gamma\rangle}$$
$$(\mathcal{M}b(C,m) = (\vec{x}_{n'}, e_b), \mathcal{F}(C) = \vec{f}_n \text{ and } \bullet((\vec{\tau}_n)\to\langle m{:}(\vec{\sigma}_{n'})\to\gamma\rangle) \leq \delta)$$

Figure 10.1.: Predicate Assignment for FJ•μ

Therefore, we will develop our type system purely in terms of the syntactic components of FJ itself. We feel that such an approach leads to a more intuitive understanding of the relationship between programs and their types.

We will now proceed to define our Nakano type system for FJ$^{¢}$.

Definition 10.9 (Self Environments)**.** *1. A* self type statement *is of the form $C{:}\delta$, where C is a class name and δ is an FJ•μ class type; the class name C is called the* subject *of the statement.*

2. A self *environment Σ is a set of self type statements where the subject of each statement is* unique.

3. The notation $\Sigma, C{:}\delta$ stands for the self environment $\Sigma \cup \{C{:}\delta\}$ where Σ does not contain any statement with the subject C.

Definition 10.10 (Type Environments)**.** *1. A* variable type statement *is of the form $x{:}\sigma$, where x is an expression variable and σ is an FJ•μ type; the variable x is called the* subject *of the statement.*

2. A field type statement *is of the form $f{:}\sigma$, where f is a field identifier and σ is an FJ•μ type; the field identifier f is called the* subject *of the statement.*

3. An FJ•μ type *environment Γ is a set of variable type and field type statements where the subject of each statement is* unique.

4. The notation $\Gamma, x{:}\sigma$ stands for the type environment $\Gamma \cup \{x{:}\sigma\}$ where Γ does not contain any statement with the subject x; similarly, $\Gamma, f{:}\sigma$ stands for the type environment $\Gamma \cup \{f{:}\sigma\}$ where Γ does not contain any statement with the subject f.

Definition 10.11 (Type Assignment for FJ•μ)**.** *FJ•μ type assignment $\Sigma;\Gamma \vdash e{:}\sigma$ is a relation between self environments, type environments, expressions and FJ•μ types. It is defined by the natural deduction*

system whose rules are given in Figure 10.1. When either the self or type environment is empty, we may write the judgement simply as $\Sigma \vdash e{:}\sigma$ *or* $\Gamma \vdash e{:}\sigma$.

The type rules of Figure 10.1 are not as big a leap from our intersection type assignment of Part I (or rather, its Curry counterpart, discussed in Chapter 7) as they might seem. We inherit the (FLD) and (INVK) rules as is, and the (VAR) rule with the slight modification that it cannot apply to the self variable. This minor addition is to ensure that typing the self is now carried out consistent with the new recursive typing approach. We include a separate subtyping rule (instead of allowing subtyping only for variables) because we want to allow (recursive) method types to be unfolded or folded as necessary. The ($\bullet$) and (ι) rules are straightforward extensions of their cousins in the type system of the previous section for the λ-calculus.

The (INST-METH) rule replaces the (NEWM) of the original intersection type system, and assigns a method type to an object value. As we mentioned in the previous section, our approach to assigning a method type $\langle m{:}(\vec{\sigma}) \to \tau\rangle$ to an object `new C(...)` involves viewing the syntactic subcomponent `new C` as a function defined by the body of the method m in the class `C`, which is then applied to its field value 'arguments'. In general, this 'function' can be defined recursively, and so we type it with a form of (FIX) rule. Bringing all of this together, the (INST-METH) rule acts as an arrow introduction, (FIX) and arrow elimination rule *combined*. A type for the `new C` 'function' is derived by typing the method body using an environment with type assumptions for its 'arguments', the fields. Since the range of this type should be a function type itself (i.e. a method type), a second function type is implicitly inferred by also including in the environment type assumptions for the parameters of the method body being typed. This implicit function type is then matched against an explicit type included in the self environment, and which is used to type recursive occurrences of the `new C` 'function', constituting the (FIX) part of the rule. As mentioned in the previous section, since we are building a Nakano-style system, the (FIX) rule must ensure that the explicit class type used in the self environment is a *bulleted* version of the implicitly derived one. Finally, after having inferred a function type for the recursively defined class, it is then applied to concrete field values, constituting the final arrow elimination step. Thus, one way of understanding, or visualising, the (INST-METH) rule could be in terms of the following derivation scheme:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, C{:}\bullet\delta; \{\vec{f}{:}\vec{\tau}_n, \texttt{this}{:}C, x{:}\vec{\sigma}_{n'}\} \vdash e_b{:}\gamma}{\Sigma, C{:}\bullet\delta; \{\vec{f}{:}\vec{\tau}_n, \texttt{this}{:}C\} \vdash [m = \lambda\vec{x}_{n'}.e_b]{:}\langle m{:}(\vec{\sigma}_{n'}) \to \gamma\rangle}(\to I)}{\Sigma, C{:}\bullet\delta; \{\texttt{this}{:}C\} \vdash \lambda\vec{f}_n.[m = \lambda\vec{x}_{n'}.e_b]{:}(\vec{\tau}_n) \to \langle m{:}(\vec{\sigma}_{n'}) \to \gamma\rangle}(\to I)}{\Sigma, C{:}\bullet\delta; \{\texttt{this}{:}C\} \vdash \lambda\vec{f}_n.[m = \lambda\vec{x}_{n'}.e_b]{:}\delta}(\leq)}{\Sigma \vdash \texttt{new}\ \ C{:}(\vec{\tau}_n) \to \langle m{:}(\vec{\tau}_{n'}) \to \gamma\rangle}(\text{FIX}) \qquad \Sigma;\Gamma \vdash e_i{:}\tau_i \quad (\forall\, i \in \overline{n})}{\Sigma;\Gamma \vdash \texttt{new}\ \ C(\vec{e}_n){:}\langle m{:}(\vec{\tau}_{n'}) \to \gamma\rangle}(\to E)$$

where $\mathcal{F}(C) = \vec{f}_n$ and $\mathcal{M}b(C, m) = (\vec{x}_{n'}, e_b)$

We hasten to point out that the above figure is not intended to be taken as a true derivation; it is merely intended to illustrate to the reader the parallel that we have drawn between the operational behaviour of FJ$^{\text{¢}}$ objects on the one hand, and that of recursively defined (labelled) functions on the other, and how

this has directed the formulation of our (INST-METH) typing rule. Notice that judgements at the top and bottom of the above figure are exactly those judgements that appear in the (INST-METH) rule itself.

Since we now intend to type method bodies as if they are the bodies of recursively defined functions (i.e. in an environment containing a type assumption for the function name itself) we have to make sure that, when we encounter a recursive occurrence of the 'function' we are typing, then we type it appropriately. This is the reason for including the (REC-METH) rule in the system. It assigns a type to a `new C(...)` object value, but applies *only* when there is a type statement for the class `C` in the self environment, indicating that we are typing a method body obtained from an 'unfolding' of the definition for the class `C`.

Having just established, then, that the presence of a class `C` in the self environment indicates that we are typing the recursive definition of the function `new C`, our (INST-OBJ) and (INST-FLD) rules would thus seem to violate our stated approach to typing recursive definitions, since they do not care whether there is a type assumption in the self environment for instances of the class that they type. In the case that there is such a statement in the self environment, they may assign types *other* than the one assumed. This apparent error is resolved by observing that these aforementioned rules type recursive occurrences of `new C(...)` expressions not in contexts where the function definition needs to be recursively 'unfolded' (i.e. a method call), but in contexts where a field is accessed or when we simply want to express the identity of the object (i.e. know that we have an instance of `C`). In these contexts, the syntax `new C` merely acts as a *datatype constructor*, rather than (a recursive call to) an object constructor *function*. As such, it is not necessary (and not unsafe) to assign field types or class name constants in these cases. Thus, the view here is that the `new` syntax is *overloaded* - it denotes both a datatype constructor *and* an object constructor function.

Finally, we comment on the rules (SELF-FLD) and (SELF-METH). They are the counterparts to the (INST-FLD) and (REC-METH) rules respectively. The intuition behind them is that, operationally (in the *functional* context of FJ$^{\mathfrak{C}}$), the self-reference `this` is equivalent (in the body of a method in the class `C`) to using the expression `new C(this.`f_1`, ..., this.`f_n`)` (where $f_1, \ldots, f_n$ are the fields of class `C`). They should also, therefore, be indistinguishable from the point of view of type assignment.

The main result that we conjecture for this system is that, just for Nakano's original type systems for LC, our type system gives a guarantee of head normalisation.

Conjecture 10.12. *If* $\Gamma \vdash e{:}\sigma$ *in* FJ$\bullet\mu$*, then* e *has a head normal form.*

Proving this conjecture, and thus that our object-oriented variant of Nakano type assignment is *logical*, is a main objective for future research. We imagine that a similar approach to the one we used for the intersection type system can be employed (i.e. via an approximation result based on a head-normalisation result for derivation reduction). Nakano's technique using realizability interpretations should also apply. In addition, the question of *completeness* (i.e. whether types are preserved under conversion) for Nakano-style type assignment is still open.

10.3. Typed Examples

We will now give some examples of how programs can be typed in this new system. We consider some of the same examples that we used to demonstrate typeability for the simple intersection type assignment system. In this way, we can directly compare the two. We have mentioned that the aim of formulating

a Nakano-style type system for FJ$^{¢}$ has been motivated by a desire to obtain a semantic type system for FJ$^{¢}$ for which *useful* type assignment is decidable. In this section, we will give (as we did for the simple intersection type system, and for Nakano's original system for LC) a non-terminating, or more precisely, unsolvable program and show that it cannot be typed in our system. This provides evidence for the semantic nature of the type system. We will also reconsider the examples that were problematic for our Curry type inference algorithm of Chapter 7, and show how they are naturally handled by the Nakano system. This provides evidence for the flexibility, or usefulness of the system. In the following section, we will discuss type inference for the system.

10.3.1. A Self-Returning Object

Consider the class `SR` (short for `SelfReturning`) which we gave back in Section 6.1, whose instances have methods that return the receiver and new instances of `SR` respectively. For reference, we list the class declaration again below:

```
class SR extends Object {
    SR self() { return this; }
    SR newInst() { return new SR(); }
}
```

Instances of this class can now be typed in a finitary way using Nakano-style type assignment. Recall that we said the type of such objects should be either $\mu X.\langle \texttt{self}:() \to X\rangle$ or $\mu X.\langle \texttt{newInst}:() \to X\rangle$. Of course, we must modify these types slightly to be proper types in our variant of the Nakano system, but these are essentially the types that are now assignable to `new SR()` objects using our new type system, as shown in the derivations below.

$$\dfrac{\dfrac{\dfrac{}{\texttt{SR}:() \to \bullet\mu.\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle;\texttt{this}:\texttt{SR} \vdash \texttt{this}:\bullet\mu.\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle}\,(\textsc{self-meth})}{\vdash \texttt{new SR()}:\langle \texttt{self}:() \to \bullet\mu.\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle\rangle}\,(\textsc{inst-meth})}{\vdash \texttt{new SR()}:\mu.\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle}\,(\leq)$$

$$\dfrac{\dfrac{\dfrac{}{\texttt{SR}:() \to \bullet\mu.\langle \texttt{newInst}:() \to \bullet\mathbf{0}\rangle;\texttt{this}:\texttt{SR} \vdash \texttt{new SR()}:\bullet\mu.\langle \texttt{newInst}:() \to \bullet\mathbf{0}\rangle}\,(\textsc{rec-meth})}{\vdash \texttt{new SR()}:\langle \texttt{newInst}:() \to \bullet\mu.\langle \texttt{newInst}:() \to \bullet\mathbf{0}\rangle\rangle}\,(\textsc{inst-meth})}{\vdash \texttt{new SR()}:\mu.\langle \texttt{newInst}:() \to \bullet\mathbf{0}\rangle}\,(\leq)$$

The application of the INST-METH rule in each case is valid because the class type we use for `SR` (in the self environment) when typing the method body is a supertype of the *implicit* (bulleted) class type that we construct from the remaining type information in typing. To be more explicit, let us take the first derivation above, which is an analysis of the `m1` method. The class type we have used in the self environment for `SR` is $\delta = () \to \bullet\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle$. The type we derive for the method body is $\bullet\mu.\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle$; the method that this body belongs to is `self`, and has no formal parameters (and thus no corresponding variable statements in the type environment); lastly, the class `SR` has no fields (and thus there are no field statements in the type environment). Thus we construct the implicit class type $\delta_{Imp} = () \to \langle \texttt{self}:() \to \bullet\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle\rangle$. Now, the class type that is associated with `C` in the self environment is a supertype of the *bulleted* version of this implicit class type:

$$\bullet\,\delta_{Imp} \quad = \quad \bullet(() \to \langle \texttt{self}:() \to \bullet\langle \texttt{self}:() \to \bullet\mathbf{0}\rangle\rangle)$$

$$
\begin{array}{lll}
 & \leq & () \to \bullet\langle \texttt{self}{:}() \to \bullet\langle \texttt{self}{:}() \to \bullet\,\mathbf{0}\rangle\rangle \\
 & \leq & () \to \bullet\langle \texttt{self}{:}() \to \bullet\,\mathbf{0}\rangle \\
 & = & \delta
\end{array}
$$

An interesting and important point to note is that the infinite family of method types which were assignable to `new SR()` objects in the simple intersection type system are now *no longer* assignable in the Nakano-style system, with the exception of the most basic one $\langle \texttt{self}{:}() \to \texttt{SR}\rangle$ whose derivation is given below (a similar derivation exists for `newInst` as well, of course):

$$
\dfrac{\dfrac{}{\{\texttt{SR}{:}() \to \bullet\langle \texttt{self}{:}() \to \texttt{SR}\rangle\}; \texttt{this}{:}\texttt{SR} \vdash \texttt{this}{:}\texttt{SR}}\,(\textsc{var})}{\vdash \texttt{new SR()}{:}\langle \texttt{self}{:}() \to \texttt{SR}\rangle}\,(\textsc{inst-meth})
$$

In this case, the class type we assume for `SR` is exactly the bulleted implicit class type. However, if we try to derive any of the 'nested' (and, crucially, non-recursive) method types, $\langle \texttt{self}{:}() \to \sigma\rangle$ where the type σ is itself non-recursive and of the form $\langle \texttt{self}{:}() \to \tau\rangle$, then we cannot make the self type equivalent to the implicit class type. This is because whatever class type we assume for `SR`, the implicit class type that it must match will always be a strictly larger type.

As a concrete example, suppose we want to assign $\langle \texttt{self}{:}() \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle\rangle$ to the object expression `new SR()`. The (INST-METH) rule tells us that in order to do this, two conditions must hold. Firstly, we must be able to assign the type $\langle \texttt{self}{:}() \to \texttt{SR}\rangle$ to the body of the `self` method (i.e. the expression `this`) in a self environment containing `SR`. Since this is itself a method type, the only typing rule that we could use to do this is (SELF-METH) (we could also subsequently apply the ($\leq$) rule), which tells us that the type associated with `SR` in the self environment must be a subtype of $() \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle$. A quick examination of the subtyping relation reveals that the only such type is $() \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle$ itself. Thus, the `self` method body is typed using the self environment $\{\texttt{SR}{:}() \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle\}$ and the empty type environment, since the `self` method has no formal parameters. The `SR` class also has no fields, and thus the implicit self type for the `self` method of the `SR` class that we construct from this typing is $() \to \langle \texttt{self}{:}() \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle\rangle$.

Secondly, the (INST-METH) rule says that this implicit self type must be a proper subtype of the self type we used when typing the method body. That is, the following must hold:

$$
\bullet(() \to \langle \texttt{self}{:}() \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle\rangle) \quad \leq \quad () \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle
$$

Or, equivalently:

$$
() \to \langle \texttt{self}{:}() \to \bullet\langle \texttt{self}{:}() \to \texttt{SR}\rangle\rangle \quad \leq \quad () \to \langle \texttt{self}{:}() \to \texttt{SR}\rangle
$$

However we can see that this is impossible: the result type of the (bulleted) implicit class type (on the left of the above inequality) cannot be a subtype of the result type of the self type used in the derivation (on the right-hand side of the inequality above), for the simple reason that it contains as a substructure the very type which it must be a subtype of! Such a thing is only possible for *recursive* types, and thus the *only* types of this form which can be assigned to the expression `new SR()` are *recursive* ones.

10.3.2. A Nonterminating Program

Just as we demonstrated the non-typeability of a non-terminating program in the intersection type assignment system (Section 6.2), we can show that our Nakano-style type assignment system also rejects this as ill-typed. Recall that the program in question uses the following class:

```
class NT extends Object {
    NT loop() { return this.loop(); }
}
```

What happens if we try to find a type σ which we can assign to the non-terminating expression `new NT().loop()`? Let's begin by assuming that such a type exists, and we will quickly run into a contradiction. Any derivation assigning σ to this expression must have (INVK) as its final rule, meaning that the type $\langle \texttt{loop:}() \to \sigma \rangle$ must be assignable to `new NT()`. Assigning such a type must have been done using the (INST-METH) rule, which means that we can derive the judgement $\{\texttt{NT:}\delta\} \vdash \texttt{this.loop():}\sigma$ for some class type δ. Let us now examine the structure of the derivation of this judgement, in order to discover what δ must be. The expression `this.loop()` is a method invocation and therefore the only rule that could have been applied to derive this is the (INVK) rule. This tells us that $\{\texttt{NT:}\delta\} \vdash \texttt{this:}\langle \texttt{loop:}() \to \sigma \rangle$ is derivable. This judgement can only be derived by an application of the (SELF-METH) rule followed by some number of applications of ($\leq$). Thus, the class type δ must be a subtype of $() \to \langle \texttt{loop:}() \to \sigma \rangle$.

We also know that another subtyping inequality holds for this class type δ, since we have assumed that there was a valid application of the (INST-METH) rule that derives $\vdash \texttt{new NT()}:\langle \texttt{loop:}() \to \sigma \rangle$. Specifically, we know that δ is a *supertype* of the implicit class type $\bullet(() \to \langle \texttt{loop:}() \to \sigma \rangle)$. Thus, combining the subtyping inequalities we have inferred for δ, we obtain the following:

$$\bullet(() \to \langle \texttt{loop:}() \to \sigma \rangle) \quad \leq \quad \delta \quad \leq \quad () \to \langle \texttt{loop:}() \to \sigma \rangle$$

Hence, we know that no such class type δ can exist since if it did, then via transitivity of the subtyping relation we would have that:

$$\bullet(() \to \langle \texttt{loop:}() \to \sigma \rangle) \quad \leq \quad () \to \langle \texttt{loop:}() \to \sigma \rangle$$

which is impossible. Therefore, we can conclude that no derivation exists for $\vdash \texttt{new NT()}:\langle \texttt{loop:}() \to \sigma \rangle$, and thus that we cannot assign a type to the (non-terminating) expression `new NT().loop()`.

A similar analysis holds for the following variant that invokes the `loop` method on a *new instance* of the class, rather than the receiver `this`:

```
class NTVariant extends Object {
    NTVariant loop() { return new NTVariant().loop(); }
}
```

10.3.3. Mutually Recursive Class Definitions

We will now look at an example containing two classes which are mutually recursive in that they each contain a method which returns an instance of the other. Thus each class depends upon the other (and

via transitivity itself, closing the recursive loop):

```
class C extends Object {
    D newInst() { return new D(); }
}

class D extends Object {
    C newInst() { return new C(); }
}
```

This is actually a variation on the first example we considered, the self returning object. In each case, by invoking the method `newInst` we obtain an object on which we can again invoke the method `newInst`, and so on. Just as in the self returning object example, each of these two classes is defined recursively, but this time via a loop of length two. As a consequence, recursive types can be used to type instances of these classes as well.

The interesting thing about this example is that it demonstrates how, as we recurse through class definitions to analyse nested method calls, self types for different classes can accrue in the self environment until they are needed. For example, in typing instances of each of the classes above, the self type for `C` (respectively `D`) is needed not for typing the body of the `newInst` method contained in the class `C` itself (respectively `D`), but in typing the body of the `newInst` method contained in its *sister* class. This means that for a sufficiently complex analysis of any recursively defined class (in the sense of Definition 7.26), no matter how indirect the recursion we will eventually reach the recursive reference which we *must* type using a type assumption from the environment. Thus, at some point, we stop 'looking inside' the class definition (i.e. recursively analysing method bodies) which, in turn, implies that a type inference algorithm for this system will only have to recurse to a finite level, and thus terminate.

Sufficiently simple non-recursive types *can* of course be assigned to `new C()` and `new D()` objects. Apart from the trivial types `C` and `D`, the following non-recursive method types can also be assigned.

$$\dfrac{\dfrac{}{\{\mathsf{C}{:}() \to \langle \mathtt{newInst}{:}() \to \bullet \mathsf{D}\rangle\};\{\mathtt{this}{:}\mathsf{C}\} \vdash \mathtt{new\ D()}{:}\mathsf{D}}\,(\textsc{inst-obj})}{\vdash \mathtt{new\ C()}{:}\langle \mathtt{newInst}{:}() \to \mathsf{D}\rangle}\,(\textsc{inst-meth})$$

$$\dfrac{\dfrac{}{\{\mathsf{D}{:}() \to \langle \mathtt{newInst}{:}() \to \bullet \mathsf{C}\rangle\};\{\mathtt{this}{:}\mathsf{D}\} \vdash \mathtt{new\ C()}{:}\mathsf{C}}\,(\textsc{inst-obj})}{\vdash \mathtt{new\ D()}{:}\langle \mathtt{newInst}{:}() \to \mathsf{C}\rangle}\,(\textsc{inst-meth})$$

We can even go slightly further:

$$\dfrac{\dfrac{\dfrac{}{\{\mathsf{C}{:}() \to \bullet\sigma_1, \mathsf{D}{:}() \to \bullet\langle \mathtt{newInst}{:}() \to \mathsf{C}\rangle\};\{\mathtt{this}{:}\mathsf{D}\} \vdash \mathtt{new\ C()}{:}\mathsf{C}}\,(\textsc{inst-obj})}{\{\mathsf{C}{:}() \to \bullet\sigma_1\};\{\mathtt{this}{:}\mathsf{C}\} \vdash \mathtt{new\ D()}{:}\langle \mathtt{newInst}{:}() \to \mathsf{C}\rangle}\,(\textsc{inst-meth})}{\vdash \mathtt{new\ C()}{:}\langle \mathtt{newInst}{:}() \to \langle \mathtt{newInst}{:}() \to \mathsf{C}\rangle\rangle}\,(\textsc{inst-meth})$$

$$\dfrac{\dfrac{\dfrac{}{\{\mathsf{D}{:}() \to \bullet\sigma_2, \mathsf{C}{:}() \to \bullet\langle \mathtt{newInst}{:}() \to \mathsf{D}\rangle\};\{\mathtt{this}{:}\mathsf{C}\} \vdash \mathtt{new\ D()}{:}\mathsf{D}}\,(\textsc{inst-obj})}{\{\mathsf{D}{:}() \to \bullet\sigma_2\};\{\mathtt{this}{:}\mathsf{D}\} \vdash \mathtt{new\ C()}{:}\langle \mathtt{newInst}{:}() \to \mathsf{D}\rangle}\,(\textsc{inst-meth})}{\vdash \mathtt{new\ D()}{:}\langle \mathtt{newInst}{:}() \to \langle \mathtt{newInst}{:}() \to \mathsf{D}\rangle\rangle}\,(\textsc{inst-meth})$$

where $\sigma_1 = \langle \texttt{newInst:}() \to \langle \texttt{newInst:}() \to \texttt{C} \rangle \rangle$
$\sigma_2 = \langle \texttt{newInst:}() \to \langle \texttt{newInst:}() \to \texttt{D} \rangle \rangle$

Such non-recursive types can be assigned because they do not reach down 'far enough' - the analysis they provide of the functional behaviour of the method that they type does not reach the point where an instance of the original class reappears in a context where it must again be typed with a method type, such as the following:

$$\texttt{new C().newInst().newInst().newInst()} \to \texttt{new D().newInst().newInst()} \to \texttt{new C().newInst()}$$

Since we only allow a *single* (i.e. non-intersection) type for each class in the self environment, in the case that a nested occurrence of `new C()` or `new D()` must be assigned a method type, it must be assigned *same* method type we are trying to derive for the outer occurrence. Then, because this type must also form a sub-part of the type we are trying to derive, it must form a sub-part of *itself*. Only recursive types are capable of doing this and thus, to assign method types of further complexity than those already considered, recursive types are required. In the specific case of the example we are looking at, the recursive type that expresses this behaviour is $\sigma = \mu.\langle \texttt{newInst:}() \to \bullet\langle \texttt{newInst:}() \to \bullet \mathbf{0} \rangle \rangle$, which can be assigned to `new C()` as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\texttt{C:}() \to \bullet\sigma, \texttt{D:}() \to \bullet\langle \texttt{newInst:}() \to \bullet\sigma \rangle\}; \{\texttt{this:D}\} \vdash \texttt{new C()}:\bullet\sigma}\,(\text{REC-METH})}{\{\texttt{C:}() \to \bullet\sigma\}; \{\texttt{this:C}\} \vdash \texttt{new D()}:\langle \texttt{newInst:}() \to \bullet\sigma \rangle}\,(\text{INST-METH})}{\{\texttt{C:}() \to \bullet\sigma\}; \{\texttt{this:C}\} \vdash \texttt{new D()}:\bullet\langle \texttt{newInst:}() \to \bullet\sigma \rangle}\,(\leq)}{\vdash \texttt{new C()}:\langle \texttt{newInst:}() \to \bullet\langle \texttt{newInst:}() \to \bullet\sigma \rangle \rangle}\,(\text{INST-METH})}{\vdash \texttt{new C()}:\mu.\langle \texttt{newInst:}() \to \bullet\langle \texttt{newInst:}() \to \bullet\mathbf{0} \rangle \rangle}\,(\leq)$$

By simply switching the class names (from `C` to `D`, and vice-versa), we can obtain a derivation assigning this type to `new D()` as well.

Moving on to a separate theme illustrated by this example, the typing derivation above assigns a (recursive) type to the object `new C()` using an *empty* typing environment, and more importantly an empty self environment. We can view this derivation as performing an analysis on the object in which no assumptions have been made - that is, it *fully* examines the behaviour of both `new C()` and `new D()` objects and returns the result. However, the type system also allows for a *partial* analysis. The following derivation assigns a recursive type to a `new C()` object but using a self environment containing a typing assumption about the class `D`, or more precisely an assumption about the type of the object constructor function that the class `D` encodes. Let $\Sigma = \{\texttt{D:}() \to \bullet\langle \texttt{newInst:}() \to \bullet\sigma \rangle\}$, then:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma, \texttt{C:}() \to \bullet\sigma; \{\texttt{this:C}\} \vdash \texttt{new D()}:\bullet\langle \texttt{newInst:}() \to \bullet\sigma \rangle}\,(\text{REC-METH})}{\Sigma \vdash \texttt{new C()}:\langle \texttt{newInst:}() \to \bullet\langle \texttt{newInst:}() \to \bullet\sigma \rangle \rangle}\,(\text{INST-METH})}{\Sigma \vdash \texttt{new C()}:\mu.\langle \texttt{newInst:}() \to \bullet\langle \texttt{newInst:}() \to \bullet\mathbf{0} \rangle \rangle}\,(\leq)$$

This suggests an alternative technique for type inference, where class types could be inferred in *isolation* for each class in a program by typing its method bodies in a self environment containing type assumptions for *every other* class in the program. Then, when all possible class types have been inferred

in this way, their consistency with one another could be checked. This would be more akin to the standard approach of nominal type checking (as outlined for FJ$^{¢}$ in Section 6.6), and we will expand upon this in the next section.

10.3.4. A Fixed-Point Operator Construction

Recall the object-oriented fixed-point combinator that was discussed at the end of Section 6.5:

```
class T extends Combinator {
    Combinator app(Combinator x) {
        return x.app(this.app(x));
    }
}
```

Since 'applying' the object `new T()` to an expression results in the same reduction behaviour as applying the OOCL encoding of any CL fixed point operator Y[1] (i.e. the expressions `new T().app(`e`)` and $\llbracket Y \rrbracket$`.app(`e`)` have the *same* set of approximants), it is not surprising that we can assign to it the FJ$^{¢}$ translation of the characteristic Nakano type for fixed point operators:

$$\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}:\bullet\langle \texttt{app}:(\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle)\to\varphi\rangle}\,(\text{SELF-METH})}{\Sigma;\Gamma \vdash \texttt{this}:\langle \texttt{app}:(\bullet\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle)\to\bullet\varphi\rangle}\,(\leq) \qquad \dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{x}:\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle}\,(\text{VAR})}{\Sigma;\Gamma \vdash \texttt{x}:\bullet\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle}\,(\leq)}{\Sigma;\Gamma \vdash \texttt{this.app(x)}:\bullet\varphi}\,(\text{INVK})$$

$$\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{x}:\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle}\,(\text{VAR}) \qquad \vdots}{\Sigma;\Gamma \vdash \texttt{x.app(this.app(x))}:\varphi}\,(\text{INVK})}{\vdash \texttt{new T()}:\langle \texttt{app}:(\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle)\to\varphi\rangle}\,(\text{INST-METH})$$

where $\Sigma = \{\texttt{T}:() \to \bullet\langle \texttt{app}:(\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle)\to\varphi\rangle\}$ and
$\Gamma = \{\texttt{this}:\texttt{T}, \texttt{x}:\langle \texttt{app}:(\bullet\varphi)\to\varphi\rangle\}$

It is interesting to note that in Nakano's systems for LC, derivations of this type for fixed-point combinators *require* the use of recursive types, while the derivation above for the OO fixed-point combinator object `new T()` does not use recursive types at all. This notable difference between the two systems lies in the fact that recursion in the Lambda Calculus is *explicit*, while the recursive nature of classes is highly *implicit*. This can be illustrated with surprising clarity by considering an interpretation of the object `new T()` as a direct translation of the body of its `app` method into LC. We can do this almost trivially by taking the method body, translating invocations of the `app` method into standard LC application, and abstracting over its formal parameter as follows:

$$\texttt{new T()} \quad \triangleq \quad \lambda\texttt{x}.\,\texttt{x}\,(\,\texttt{new T()}\,\texttt{x}\,)$$

We are not quite done, as this interpretation of the object `new T()` is still defined in terms of itself. We can quite easily 'solve' this equation by using the standard technique of abstracting over the recursive

[1] We could, for example, consider the encoding of Curry's fixed point operator in Combinatory Logic – the term **S**(**K**(**S**(**SKK**)(**SKK**)))(**S**(**S**(**KS**)**K**)(**K**(**S**(**SKK**)(**SKK**)))).

occurrences in the definition of the term itself, and then applying a fixed point operator (for the sake of argument let's choose Curry's fixed point operator **Y**):

$$\texttt{new T()} \quad \triangleq \quad \mathbf{Y}(\lambda \texttt{t}\,\texttt{x}.\texttt{x}(\texttt{t}\,\texttt{x}))$$

The λ-term above should be familiar, as we discussed it at length in Section 9.6.2 (or rather, its meta-level representative, the term $\mathbf{Y}(\lambda xy.y(xy))$), in the context of type inference for our variant of Nakano's system for the λ-calculus. There, we noted that the term was a fixed point combinator itself (just like our FJ$^{¢}$ object `new T()`) and that it could therefore be assigned the type $(\bullet\varphi \to \varphi) \to \varphi$ (just as the expression `new T()` can be assigned the FJ$\bullet\mu$ version of this type). Furthermore and most importantly, while we had to use recursive types in the subderivation typing **Y**, in the subderivation typing $\lambda xy.y(xy)$ (our *interpretation* of the expression `new T()`) no recursive types were needed at all.

10.3.5. Lists

Let us now turn our attention to the FJ$^{¢}$ programs that we introduced in Section 6.3 and 6.4, which encode lists, and certain arithmetic operations on natural numbers. Like the previous examples in Section 10.3.1 and 10.3.3 of the individually and mutually recursive classes that give rise to self returning objects, the instances of the classes in these programs have methods which return objects of the same kind as themselves. However, in addition, these are programs which very naturally illustrate the concept of binary methods – the argument(s) to these methods must also be objects of the same kind as the receiver. In this section, we will demonstrate how these objects can be typed in our system with the obvious recursive types that express this requirement.

Let's start with the program that encodes lists, reproduced below:

```
class List extends Object {
    List append(List l) { return this; }
    List cons(Object o) { return new NEL(o, this); }
}

class EL extends List {
    List append(List l) { return l; }
}

class NEL extends List {
    Object head;
    List tail;
    List append(List l) {
        return new NEL(this.head, this.tail.append(l));
    }
}
```

Here, `Lists` (i.e. either `EL` objects or `NEL` objects) have a binary method `append`, which takes another `List` object as an argument and returns another `List`. This behaviour is expressed by the recursive type $\sigma = \mu.\langle \texttt{append}:(\bullet\mathbf{0}) \to \bullet\mathbf{0}\rangle$. Showing that all `List` objects have this type involves two parts. Every

(well-formed) `List` object is an expression generated by the following grammar:

$$l \;::=\; \texttt{new EL()} \;\mid\; \texttt{new NEL(}e\texttt{, }l\texttt{)}$$

We must therefore show that `new EL()` objects have this type, and that whenever some list object l has this type, so does `new NEL(`e`, `l`)`. The first of these two conditions is witnessed by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{EL}{:}() \to \bullet\sigma\};\{\texttt{this}{:}\texttt{EL}, \texttt{l}{:}\bullet\sigma\} \vdash \texttt{l}{:}\bullet\sigma}\,(\textsc{var})}{\vdash \texttt{new EL()}{:}\langle \texttt{append}{:}(\bullet\sigma) \to \bullet\sigma\rangle}\,(\textsc{inst-meth})}{\vdash \texttt{new EL()}{:}\mu.\langle \texttt{append}{:}(\bullet\mathbf{0}) \to \bullet\mathbf{0}\rangle}\,(\leq)$$

The following derivation schema demonstrates how to assign this type to a non-empty `new NEL(`e`, `l`)` list object when the tail l already has this type and the head e is typeable with some type τ:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}{:}\langle \texttt{tail}{:}\sigma\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma \vdash \texttt{this.tail}{:}\sigma}\,(\textsc{fld})}{\Sigma;\Gamma \vdash \texttt{this.tail}{:}\langle \texttt{append}{:}(\bullet\sigma) \to \bullet\sigma\rangle}\,(\leq) \quad \dfrac{}{\Sigma;\Gamma \vdash \texttt{l}{:}\bullet\sigma}\,(\textsc{var})}{\Sigma;\Gamma \vdash \texttt{this.tail.append(l)}{:}\bullet\sigma}\,(\textsc{invk})}{\Sigma;\Gamma \vdash \texttt{new NEL(this.head, this.tail.append(l))}{:}\bullet\sigma}\,(\textsc{rec-meth})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}{:}\langle \texttt{head}{:}\tau\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma \vdash \texttt{this.head}{:}\tau}\,(\textsc{fld})}{\vdots}}{\vdash e{:}\tau} \qquad \dfrac{\dfrac{}{\vdots}}{\vdash l{:}\mu.\langle \texttt{append}{:}(\bullet\mathbf{0}) \to \bullet\mathbf{0}\rangle}$$

$$\dfrac{\vdash \texttt{new NEL(}e\texttt{, }l\texttt{)}{:}\langle \texttt{append}{:}(\bullet\sigma) \to \bullet\sigma\rangle}{\vdash \texttt{new NEL(}e\texttt{, }l\texttt{)}{:}\mu.\langle \texttt{append}{:}(\bullet\mathbf{0}) \to \bullet\mathbf{0}\rangle}\,(\leq)\;(\textsc{inst-meth})$$

where $\Sigma = \{\texttt{NEL}{:}(\bullet\tau, \bullet\sigma) \to \bullet\sigma\}$ and
$\Gamma = \{\texttt{this}{:}\texttt{NEL}, \texttt{head}{:}\tau, \texttt{tail}{:}\sigma, \texttt{l}{:}\bullet\sigma\}$

Notice, however, that the type system does not constrain lists to be ‘well-formed’. Obviously, the intention in writing the above program is that any list objects that we might want to create and use should adhere to the aforementioned structure. Notwithstanding, there are ways of using the `NEL` class to create objects which, although incorrect with respect to the intended and standard semantics of lists, are nonetheless ‘safe’ to use.

We have mentioned previously that the intention in declaring the FJ¢ superclass `List` was to create a program that was also type correct in Featherweight Java, and that one should view this construction as representing an interface, or abstract class, as in full Java. However, since FJ¢ allows us to create instances of this class we can use it to illustrate the point (in any case, one could imagine defining a concrete subclass – or implementation – of this ‘interface’ in full Java which overrides the `append` method with some incorrect behaviour). This `List` class acts as an empty list, but one which simply

throws the argument to its `append` method away. Thus, the actual behaviour of invoking the append method on this incorrect empty list is that the empty list itself is again returned. This is expressed by the fact that we can assign a `new List()` object the type $\sigma = \mu.\langle \texttt{append}:(\alpha) \to \bullet\mathbf{0}\rangle$ for any type α, as in the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{EL}:() \to \bullet\sigma\};\{\texttt{this}:\texttt{EL}, \texttt{l}:\alpha\} \vdash \texttt{this}:\bullet\sigma}\,(\textsc{self-meth})}{\vdash \texttt{new EL()}:\langle \texttt{append}:(\alpha) \to \bullet\sigma\rangle}\,(\textsc{inst-meth})}{\vdash \texttt{new EL()}:\mu.\langle \texttt{append}:(\alpha) \to \bullet\mathbf{0}\rangle}\,(\leq)$$

By extension then, it also follows that any *non*-empty list we create starting from an instance of this incorrect class will also simply return itself, throwing away the argument to its `append` method since non-empty lists simply delegate to their tails. This behaviour is perfectly consistent with the actual object that we have created (in the sense that a programmer can be assumed to have intended the behaviour of the program that they have actually written), and more importantly it is *safe*: invoking the `append` method does not result in any runtime error - indeed, an object (i.e. a value) is returned and moreover it is safe to invoke the `append` method on this too. The type system confirms this alternative for our program:

$$\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}:\langle \texttt{head}:\tau\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma \vdash \texttt{this.head}:\tau}\,(\textsc{fld}) \quad \vdots}{\Sigma;\Gamma \vdash \texttt{new NEL(this.head, this.tail.append(l))}:\bullet\sigma}\,(\textsc{rec-meth})$$

$$\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}:\langle \texttt{tail}:\sigma\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma \vdash \texttt{this.tail}:\sigma}\,(\textsc{fld})}{\Sigma;\Gamma \vdash \texttt{this.tail}:\langle \texttt{append}:(\alpha) \to \bullet\sigma\rangle}\,(\leq) \quad \dfrac{}{\Sigma;\Gamma \vdash \texttt{l}:\alpha}\,(\textsc{var})}{\Sigma;\Gamma \vdash \texttt{this.tail.append(l)}:\bullet\sigma}\,(\textsc{invk})$$

$$\dfrac{\dfrac{\vdots \quad \vdash e:\tau \quad \vdash l:\mu.\langle \texttt{append}:(\alpha) \to \bullet\mathbf{0}\rangle}{\vdash \texttt{new NEL(}e\texttt{, }l\texttt{)}:\langle \texttt{append}:(\alpha) \to \bullet\sigma\rangle}\,(\textsc{inst-meth})}{\vdash \texttt{new NEL(}e\texttt{, }l\texttt{)}:\mu.\langle \texttt{append}:(\alpha) \to \bullet\mathbf{0}\rangle}\,(\leq)$$

where $\Sigma = \{\texttt{NEL}:(\bullet\tau, \bullet\sigma) \to \bullet\sigma\}$ and
$\Gamma = \{\texttt{this}:\texttt{NEL}, \texttt{head}:\tau, \texttt{tail}:\sigma, \texttt{l}:\alpha\}$

The type analysis performed by the system for the `cons` method is actually very similar to this 'incorrect' `append` method, since the `cons` method accepts any arbitrary object (of some type α) and returns a new list object (with the object at the head and the original list, which was the receiver of the method invocation, at the tail) onto which, of course, we can `cons` another object. This is exactly the behaviour expressed in the type $\sigma = \mu.\langle \texttt{cons}:(\tau) \to \bullet\mathbf{0}\rangle$ when assigned to a list object.

The `EL` and `NEL` classes share the same body for the `cons` method, since they both inherit their definition of this method from the `List` class. Since this method body creates a new `NEL` object, it

is the analysis of the method when invoked on an instance of the `NEL` class itself which is the more fundamental:

$$\dfrac{\dfrac{\dfrac{}{\Sigma_2;\Gamma_2 \vdash \mathtt{o}:\tau}\,(\textsc{var}) \quad \dfrac{}{\Sigma_2;\Gamma_2 \vdash \mathtt{this}:\bullet\sigma}\,(\textsc{self-meth})}{\Sigma_2;\Gamma_2 \vdash \mathtt{new\ NEL(o,\ this)}:\bullet\sigma}\,(\textsc{rec-meth}) \quad \vdots \quad \boxed{\Sigma_1;\Gamma_1 \vdash e:\tau} \quad \boxed{\Sigma_1;\Gamma_1 \vdash l:\mu.\langle \mathtt{cons}:(\tau)\to\bullet\mathbf{0}\rangle}}{\Sigma_1;\Gamma_1 \vdash \mathtt{new\ NEL}(e,\ l):\langle \mathtt{cons}:(\tau)\to\bullet\sigma\rangle}\,(\textsc{inst-meth})}{\Sigma_1;\Gamma_1 \vdash \mathtt{new\ NEL}(e,\ l):\mu.\langle \mathtt{cons}:(\tau)\to\bullet\mathbf{0}\rangle}\,(\leq)$$

where $\Sigma_2 = \{\mathtt{NEL}:(\bullet\tau, \bullet\sigma)\to\bullet\sigma\}$ and
$\Gamma_2 = \{\mathtt{this}:\mathtt{NEL}, \mathtt{head}:\tau, \mathtt{tail}:\sigma, \mathtt{o}:\alpha\}$

The `cons` method when invoked on instances of the `EL` (or indeed `List`) class does not refer to the class of its receiver, and uses an instantiation of the derivation scheme given above:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_2';\Gamma_2' \vdash \mathtt{o}:\bullet\tau}\,(\textsc{var}) \quad \dfrac{}{\Sigma_2';\Gamma_2' \vdash \mathtt{this}:\bullet\bullet\sigma}\,(\textsc{self-meth})}{\Sigma_2';\Gamma_2' \vdash \mathtt{new\ NEL(o,\ this)}:\bullet\bullet\sigma}\,(\textsc{rec-meth}) \quad \vdots \quad \dfrac{}{\Sigma_1;\Gamma_1 \vdash \mathtt{o}:\tau}\,(\textsc{var}) \quad \dfrac{}{\Sigma_1;\Gamma_1 \vdash \mathtt{this}:\bullet\sigma}\,(\textsc{self-meth})}{\Sigma_1;\Gamma_1 \vdash \mathtt{new\ NEL(o,\ this)}:\langle \mathtt{cons}:(\bullet\tau)\to\bullet\bullet\sigma\rangle}\,(\textsc{inst-meth})}{\Sigma_1;\Gamma_1 \vdash \mathtt{new\ NEL(o,\ this)}:\bullet\sigma}\,(\leq)}{\vdash \mathtt{new\ EL()}:\langle \mathtt{cons}:(\tau)\to\bullet\sigma\rangle}\,(\textsc{inst-meth})}{\vdash \mathtt{new\ EL()}:\mu.\langle \mathtt{cons}:(\tau)\to\bullet\mathbf{0}\rangle}\,(\leq)$$

where $\Sigma_1 = \{\mathtt{EL}:()\to\bullet\sigma\}, \Gamma_1 = \{\mathtt{this}:\mathtt{EL}, \mathtt{o}:\tau\}$
$\Sigma_2' = \bullet\Sigma_2 = \{\mathtt{NEL}:(\bullet\bullet\tau, \bullet\bullet\sigma)\to\bullet\bullet\sigma\}$ and
$\Gamma_2' \leq \bullet\Gamma_2 = \{\mathtt{this}:\mathtt{NEL}, \mathtt{head}:\bullet\tau, \mathtt{tail}:\bullet\sigma, \mathtt{o}:\bullet\alpha\}$

Notice that if we want to assign a recursive method type for `cons`, as above, the list is forced to be *homogeneous*. That is, the elements that we `cons` onto the original list must have the same type as the elements already in the list.

So far, we have only considered typing expressions that *add* elements to a list; we have not considered how we might (typeably) retrieve them. If we have a (typeable) list *value*

$$\mathtt{new\ NEL(}e_1\mathtt{,\ ...\ new\ NEL(}e_n\mathtt{,\ new\ EL())\ ...\ )}$$

(so all the e_i are typeable), then this is trivial. For any element e_i in the list we can assign a type to the list value which refers to that element specifically:

$$\dfrac{\Sigma;\Gamma\vdash e_i:\sigma_i \qquad \Sigma;\Gamma\vdash l:\tau}{\Sigma;\Gamma\vdash \texttt{new NEL}(e_i,\ l):\langle\texttt{head}:\sigma_i\rangle}\ (\textsc{fld})$$

$$\vdots$$

$$\dfrac{\Sigma;\Gamma\vdash e_1:\sigma_1 \qquad \dfrac{}{\Sigma;\Gamma\vdash \ldots\ \texttt{new NEL}(e_i,\ l)\ \ldots : \ldots\ \langle\texttt{head}:\sigma_i\rangle\ \ldots}}{\Sigma;\Gamma\vdash \texttt{new NEL}(e_1,\ \ldots\ \texttt{new NEL}(e_i,\ \ldots)\ \ldots\ ):\langle\texttt{tail}:\ldots\ \langle\texttt{head}:\sigma_i\rangle\ \ldots\rangle}\ (\textsc{fld})$$

On the other hand, if we would like to first build a list by invoking a sequence of `cons` and `append` methods on some (possibly empty) list value, and then subsequently access its elements, our options are considerably more limited. It is possible to assign the type $\sigma = \langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle$ to a list object, which then allows the newly consed object to be retrieved:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\dfrac{}{\Sigma';\Gamma'\vdash \texttt{o}:\alpha}\,(\textsc{var}) \qquad \dfrac{}{\Sigma';\Gamma'\vdash\texttt{this}:\texttt{EL}}\,(\textsc{var})}{\Sigma';\Gamma'\vdash\texttt{new NEL(o, this)}:\langle\texttt{head}:\alpha\rangle}\,(\textsc{inst-fld})}{\Sigma;\Gamma\vdash\texttt{new EL()}:\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle}\,(\textsc{inst-meth})\\ \vdots\end{array} \qquad \Sigma;\Gamma\vdash e:\alpha}{\Sigma;\Gamma\vdash\texttt{new EL().cons}(e):\langle\texttt{head}:\alpha\rangle}\,(\textsc{invk})}{\Sigma;\Gamma\vdash\texttt{new EL().cons}(e)\texttt{.head}:\alpha}\,(\textsc{fld})$$

where $\Sigma' = \{\texttt{EL}:()\to\bullet\sigma\}$ and
$\Gamma' = \{\texttt{this}:\texttt{EL}, \texttt{o}:\alpha\}$

And for non-empty lists:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\dfrac{}{\Sigma';\Gamma'\vdash \texttt{o}:\alpha}\,(\textsc{var}) \qquad \dfrac{}{\Sigma';\Gamma'\vdash\texttt{this}:\texttt{NEL}}\,(\textsc{var})}{\Sigma';\Gamma'\vdash\texttt{new NEL(o, this)}:\langle\texttt{head}:\alpha\rangle}\,(\textsc{inst-fld}) \qquad \Sigma;\Gamma\vdash e:\tau \qquad \Sigma;\Gamma\vdash l:\beta}{\Sigma;\Gamma\vdash\texttt{new EL}(e,\ l):\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle}\,(\textsc{inst-meth})\\ \vdots\end{array} \qquad \Sigma;\Gamma\vdash e':\alpha}{\Sigma;\Gamma\vdash\texttt{new EL}(e,\ l)\texttt{.cons}(e'):\langle\texttt{head}:\alpha\rangle}\,(\textsc{invk})}{\Sigma;\Gamma\vdash\texttt{new NEL}(e,\ l)\texttt{.cons}(e')\texttt{.head}:\alpha}\,(\textsc{fld})$$

where $\Sigma' = \{\texttt{NEL}:(\bullet\tau, \bullet\beta)\to\bullet\sigma\}$ and
$\Gamma' = \{\texttt{this}:\texttt{NEL}, \texttt{head}:\tau, \texttt{tail}:\beta, \texttt{o}:\alpha\}$

Notice that in the both the case of typing a list value and the case of typing a single invocation of the `cons` method, there is no requirement for the list to be homogeneous. Furthermore, when assigning the type for the single-invocable `cons` method, we are able to type the tail of the list however we like - since

it plays no active part in the reduction of the final typed expression, it is essentially allowed to be any arbitrary (typeable) expression.

Unfortunately, as with the mutually recursively defined self-returning objects, more informative types allowing multiple *heterogeneous* calls to the `cons` method (e.g. types such as $\langle \texttt{cons}:(\alpha) \to \langle \texttt{cons}:(\beta) \to \langle \texttt{head}:\beta \rangle \rangle \rangle$ or $\langle \texttt{cons}:(\alpha) \to \langle \texttt{cons}:(\beta) \to \langle \texttt{tail}:\langle \texttt{head}:\alpha \rangle \rangle \rangle \rangle \rangle$) cannot be assigned, since we only allow a *single* type for each class in the self environment. This certainly is a major disadvantage to the type system that we have presented, however we can overcome this rather pronounced lack of expressivity by allowing intersections back into the type language. We will discuss this possibility later (in Section 10.5), and point out that the type system we have presented is only intended as a proof-of-concept first attempt at a Nakano-style type assignment for oo.

10.3.6. Object-Oriented Arithmetic

The object-oriented arithmetic example bears a strong similarity to the list example we have just considered. The `add` method that performs the addition operation on the receiver and method argument behaves in almost exactly the same way as the `append` method on lists. Both methods (when invoked on 'well-formed' objects) require as input an object of the same 'kind' as the receiver, and return the same. The only difference lies in the name of the method being invoked, and the classes of the objects involved (which are actually abstracted away into recursively bound type variables).

```
class Nat extends Object {
    Nat add(Nat x) { return this; }
    Nat mult(Nat x) { return this; }
}

class Zero extends Nat {
    Nat add(Nat x) { return x; }
    Nat mult(Nat x) { return this; }
}

class Suc extends Nat {
    Nat pred;
    Nat add(Nat x) { return new Suc(this.pred.add(x)); }
    Nat mult(Nat x) { return x.add(this.pred.mult(x)); }
}
```

Notice that the following derivations assigning the type $\sigma = \mu.\langle \texttt{add}:(\bullet \mathbf{0}) \to \bullet \mathbf{0} \rangle$ to object-oriented natural numbers are only slight variations on the derivations assigning types for the `append` method to list objects.

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{Zero}:() \to \bullet\sigma\};\{\texttt{this}:\texttt{Zero}, \texttt{x}:\bullet\sigma\} \vdash \texttt{x}:\bullet\sigma}\,(\textsc{var})}{\vdash \texttt{new Zero()}:\langle \texttt{add}:(\bullet\sigma) \to \bullet\sigma \rangle}\,(\textsc{inst-meth})}{\vdash \texttt{new Zero()}:\mu.\langle \texttt{add}:(\bullet\mathbf{0}) \to \bullet\mathbf{0} \rangle}\,(\leq)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}:\langle \texttt{pred}:\sigma\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma \vdash \texttt{this.pred}:\sigma}\,(\textsc{fld})}{\Sigma;\Gamma \vdash \texttt{this.pred}:\langle \texttt{add}:(\bullet\sigma)\to\bullet\sigma\rangle}\,(\leq) \quad \dfrac{}{\Sigma;\Gamma \vdash \texttt{x}:\bullet\sigma}\,(\textsc{var})}{\Sigma;\Gamma \vdash \texttt{this.pred.add(x)}:\bullet\sigma}\,(\textsc{invk})}{\Sigma;\Gamma \vdash \texttt{new Suc(this.pred.add(x))}:\bullet\sigma}\,(\textsc{rec-meth})$$

$$\dfrac{\dfrac{\vdots}{\vdash n:\mu.\langle \texttt{add}:(\bullet\mathbf{0})\to\bullet\mathbf{0}\rangle}}{\vdash \texttt{new Suc}(n):\langle \texttt{add}:(\bullet\sigma)\to\bullet\sigma\rangle}\,(\textsc{inst-meth})$$

$$\dfrac{\vdash \texttt{new Suc}(n):\langle \texttt{add}:(\bullet\sigma)\to\bullet\sigma\rangle}{\vdash \texttt{new Suc}(n):\mu.\langle \texttt{add}:(\bullet\mathbf{0})\to\bullet\mathbf{0}\rangle}\,(\leq)$$

where $\Sigma = \{\texttt{Suc}:(\bullet\sigma)\to\bullet\sigma\}$ and
$\Gamma = \{\texttt{this}:\texttt{Suc}, \texttt{pred}:\sigma, \texttt{x}:\bullet\sigma\}$

We can also assign a similarly recursive type for the `mult` method, $\tau = \mu.\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\mathbf{0})\to\bullet\mathbf{0}\rangle)\to\bullet\mathbf{0}\rangle.$:

$$\dfrac{\dfrac{}{\{\texttt{Zero}:()\to\bullet\tau\};\{\texttt{this}:\texttt{Zero}, \texttt{x}:\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle\}\vdash \texttt{this}:\bullet\tau}\,(\textsc{self-meth})}{\vdash \texttt{new Zero}():\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle)\to\bullet\tau\rangle}\,(\textsc{inst-meth})$$

$$\dfrac{\vdash \texttt{new Zero}():\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle)\to\bullet\tau\rangle}{\vdash \texttt{new Zero}():\mu.\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\mathbf{0})\to\bullet\mathbf{0}\rangle)\to\bullet\mathbf{0}\rangle}\,(\leq)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}:\langle \texttt{pred}:\tau\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma \vdash \texttt{this.pred}:\tau}\,(\textsc{fld})}{\Sigma;\Gamma \vdash \texttt{this.pred}:\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle)\to\bullet\tau\rangle}\,(\leq) \quad \dfrac{}{\Sigma;\Gamma \vdash \texttt{x}:\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle}\,(\textsc{var})}{\Sigma;\Gamma \vdash \texttt{this.pred.mult(x)}:\bullet\tau}\,(\textsc{invk})}{\Sigma;\Gamma \vdash \texttt{x.add(this.pred.mult(x))}:\bullet\tau}\,(\textsc{invk}) \quad \dfrac{\vdots}{\vdash n:\tau}}{\vdash \texttt{new Suc}(n):\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle)\to\bullet\tau\rangle}\,(\textsc{inst-meth})$$

$$\dfrac{\vdash \texttt{new Suc}(n):\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle)\to\bullet\tau\rangle}{\vdash \texttt{new Suc}(n):\mu.\langle \texttt{mult}:(\langle \texttt{add}:(\bullet\mathbf{0})\to\bullet\mathbf{0}\rangle)\to\bullet\mathbf{0}\rangle}\,(\leq)$$

where $\Sigma = \{\texttt{Suc}:(\bullet\tau)\to\bullet\tau\}$ and
$\Gamma = \{\texttt{this}:\texttt{Suc}, \texttt{pred}:\tau, \texttt{x}:\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle\}$

It is interesting to note that the type we have derived for the `mult` method requires its argument to have a *different* type for the `add` method than the one we derived above. We can assign the type $\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle$ for `new Zero()`:

$$\dfrac{\dfrac{}{\{\texttt{Zero}:()\to\bullet\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle\};\{\texttt{this}:\texttt{Zero}, \texttt{x}:\bullet\tau\}\vdash \texttt{x}:\bullet\tau}\,(\textsc{var})}{\vdash \texttt{new Zero}():\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle}\,(\textsc{inst-meth})$$

However, in the type system as we have presented it, we are unable to assign this type to any positive number. This is because we are only allowed to associate a single class type to each class in the self

environment. In this particular case, since we would like to derive the type $\langle \texttt{add:}(\bullet\tau) \to \bullet\tau \rangle$, this is also the type that we must assume for `Suc` in the self environment (or rather a bulleted version) when typing the body of the `add` method. The problem is that then, by the (REC-METH) rule, we are only allowed to derive the statement $\Sigma;\Gamma \vdash \texttt{new Suc(this.pred.add(x))}:\bullet\langle \texttt{add:}(\bullet\tau) \to \bullet\tau \rangle$, when what we need to derive in order to apply the (INST-METH) is $\Sigma;\Gamma \vdash \texttt{new Suc(this.pred.add(x))}:\bullet\tau$.

10.4. Extending The Type Inference Algorithm

We will now discuss how we might adapt the type inference and unification procedures from the previous chapter to infer typings for FJ$^{¢}$ programs in the new Nakano-style type assignment system.

The procedure that we will outline is one that we hinted at in Section 7.3, when we were discussing the type inference algorithm for the system of simple types. There, we commented on how we could make the type inference procedure terminating by keep track of all the classes that had already been 'looked inside of'. This is exactly the approach that we will outline here for the Nakano style system. The difference is that instead of simply keeping a list of all these classes, we will now also use this list to determine whether we should infer a class type for a `new C(...)` expression from the self environment, or look inside its definitions and derive method types based on further analysis of its method bodies. Essentially, we will use the list of already examined classes to determine if we should apply the (REC-METH) or the (INST-METH) rule; in other words, to determine if we should treat `new C(...)` as a function definition, or an occurrence of a recursive function identifier.

Unification

In the same way that we have adapted Robinson's standard unification algorithm for Curry types into a unification procedure for recursive (Nakano) types, it should be relatively straightforward to adapt the unification procedure of Section 7.2 to unify the recursive types of Definition 10.1.

The procedure given in Definition 7.14 already deals with the unification of field and method types - the field identifiers or method names in the two types are checked, and if they match, unification proceeds on the corresponding argument and result types. The extension consists in dealing with bullets and insertion variables, and inferring recursive types. These questions can all be answered in the same way as described in the previous chapter. Namely, we can a) define a notion of canonical pretypes and canonicalising substitution for FJ$\bullet\mu$, which will allow us to deal with bullets and insertion variables separately from the structural elements of types; and b) we can infer recursive types in the same way by, instead of failing on an occurs check, promoting the variable and forming a recursive type. Since recursion is guarded by method types, such creation of recursive types should only happen when unifying a type variable with a *method* type in which it occurs.

Decidability results for this procedure should follow in the same way as outlined in the previous chapter. We can define notions of structural and unification closure for the object types of FJ$\bullet\mu$ by a simple extension of these notions for Curry types. The proof should follow the same structure, and demonstrate that only a finite number of statements (or recursive calls) need to be made during unification.

Unification for class types follows straightforwardly from the unification of object types, since class types cannot be nested, and consist only of object types. The argument and result types of the two class types to be unified are themselves pairwise unified contravariantly, as standard for function types.

Unification can also then be easily extended to type and self environments. As before, unification would return an operation, consisting of a sequence of substitutions and insertions.

Type Inference

A type inference algorithm for FJ•μ could be obtained by a fairly straightforward extension of the algorithm given in Definition 7.23. As for that algorithm, it would be defined by cases over the structure of expressions. Also, in this system as in the intersection type assignment system, the principal typing of a term is in fact a set of typings, and a typing is now a triple of a self-environment, a type environment and a type. For each kind of expression, we generate principal typings according the different type assignment rules that are applicable to that kind of expression.

As we mentioned above, the algorithm should keep track of which classes have already had their method bodies analysed. To do this, we define the type inference algorithm in two stages. First of all, we define an auxiliary algorithm which takes as an argument both an expression and a list of classes. The main type inference algorithm is then simply a wrapper for this auxiliary procedure, which takes an expression as an argument and calls the auxiliary procedure on this expression and an empty set of classes.

We will now give an informal description of the steps that the algorithm should perform for each kind of expression.

(x): Firstly we generate a typing according the to (VAR) rule: we take a fresh type variable φ, and add the typing $[\emptyset, \{x{:}\varphi\}, \varphi]$ to the principal typing set. If $x \neq \texttt{this}$, then we are done. Otherwise we generate additional typings according to the (SELF-FLD) and (SELF-METH) rules. For each class $\texttt{C}$ in the program, with fields $\mathcal{F}(\texttt{C}) = \vec{\texttt{f}}_n$, we:

1. take fresh type variables $\varphi_0, \ldots, \varphi_n$ and add the typing $[\{\texttt{C}{:}(\vec{\varphi}_n) \to \varphi_0\}, \{\texttt{this}{:}\texttt{C}\}, \varphi_0]$ to the principal typing set;
2. and for each field $\texttt{f} \in \vec{\texttt{f}}_n$, we take a fresh type variable and add the typing $[\emptyset, \{\texttt{this}{:}\texttt{C}, \texttt{f}{:}\varphi\}, \langle \texttt{f}{:}\varphi \rangle]$ to the principal typing set.

($\texttt{e.f}$): This case actually remains unchanged from the simple type inference procedure: we recursively generate the principal typing set for the expression $\texttt{e}$ and then for each typing $[\Sigma, \Gamma, \sigma]$ in the set, we generate a fresh type variable φ and try to unify the type σ with $\langle \texttt{f}{:}\varphi \rangle$. If unification succeeds, returning the Operation O, then we add the typing $[\mathsf{O}(\Sigma), \mathsf{O}(\Gamma), \mathsf{O}(\varphi)]$ to the principal typing set of $\texttt{e.f}$.

($\texttt{e}_0\texttt{.m}(\vec{\texttt{e}}_n)$): This case is a straightforward extension of its counterpart in Definition 7.23 to introduce insertion variables, in the same way as for the case for application in the λ-calculus setting (Definition 9.75). First, we recursively call the procedure to generate the principal typing set of each expression $\texttt{e}_i$. Then, for each possible combination for selecting a typing $[\Sigma_i, \Gamma_i, \sigma_i]$ from each of the sets we do the following. We take a fresh type variable φ and a fresh insertion variable ι. Then we try to unify σ_0 with the type $\langle \texttt{m}{:}(\overline{\iota\sigma}_n) \to \varphi \rangle$, to make sure that the method invocation can be well typed. If this succeeds, then we apply the resulting operation O_1 to each type environment, and then try to unify them all. If this succeeds, returning operation O_2, then we apply $\mathsf{O}_2 \circ \mathsf{O}_1$ to each self environment, and try to unify all of these. If this succeeds with operation O_3, then apply

the operation $\mathsf{O}_3 \circ \mathsf{O}_2 \circ \mathsf{O}_1$ to each self and type environment, and combine them to generate a single self environment Σ, and a single type environment Γ. We then take a fresh insertion variable ι', and prepend it on to the class type in Σ of each class C that was present in Σ_0, and the type in Γ of each variable x and field f that was present in Γ_0. At this point, we form a typing from the resulting environments, and the result type $\mathsf{O}_3 \circ \mathsf{O}_2 \circ \mathsf{O}_1(\varphi)$, and add it to the set of principal typings that we return for $e_0.m(\vec{e}_n)$.

(new $C(\vec{e}_n)$): For each expression of the form new $C(\vec{e}_n)$ we infer three different kinds of typing, corresponding to the (INST-OBJ) and (INST-FLD) rules, and either the (REC-METH) or (INST-METH) rule depending on whether then class C is in the list of already encountered classes. Firstly, we lookup the fields $\mathcal{F}(C) = \vec{f}$. If the number of fields does not match the number of expressions $\vec{e}_n$, then we return the empty set. Otherwise, we recursively call the procedure to generate the principal typing sets of each expression e_i. Then for each combination of selecting a typing $[\Sigma_i, \Gamma_i, \sigma_i]$ from each set, we unify the type environments and the self environments. If this succeeds, returning operation O_1, we form the combined self environment Σ and type environment Γ by applying O to each self (respectively type) environment and then taking the union. Then, we do the following three things:

1. We add the typing $[\Sigma, \Gamma, C]$ to the set of principal typings.
2. For each field $f_i \in \vec{f}$, we add the typing $[\Sigma, \Gamma, \mathsf{O}_1(\sigma_i)]$ to the set of principal typings.
3. We check to see whether C is in the list of classes (passed as a parameter to the procedure) that have already had their methods analysed.
   a) If it is, then we do one of two things. If C occurs in the combined self environment Σ, then we take the class type $\mathsf{O}_1((\vec{\sigma}_n) \to \varphi)$, where φ is a fresh type variable, and unify it with the class type for C in Σ. If this succeeds, resulting in operation O_2, then we add the typing $[\mathsf{O}_2(\Sigma), \mathsf{O}_2(\Gamma), \mathsf{O}_2(\varphi)]$ to the set of principal typings. If C does not occur in the combine self environment Σ, then we take a fresh type variable φ, and add the typing $[\Sigma \cup \{C{:}(\vec{\sigma}_n) \to \varphi\}, \Gamma, \varphi]$ to the set of principal typings.
   b) If C has not already been 'unfolded', then we perform this unfolding now, and analyse the method bodies of C. For each method m in C, we lookup its method body e_b and formal parameters $\vec{x}_{n'}$. Then we recursively call the algorithm on this expression e_b but, crucially, when we do so we add the class C to the list of classes already encountered. This will ensure the termination of the algorithm, even for recursively defined programs.

      Now, for each typing $[\Sigma_b, \Gamma_b, \gamma]$ in the principal typing set of e_b, we unify the types we have for each expression e_i with the type for the corresponding field in Γ_b, and unify the type for the class C in Σ_b with the bulleted implicit self type constructed from the types in Γ_b and the type γ we have derived for the method body itself. If Γ_b does not contain a statement for any field $f \in \vec{f}_n$ or variable $x \in \vec{x}_{n'}$, then we can extend it using freshly generated type variables. Similarly, if Σ_b does not contain a class type for C, then we can extend it with one generated entirely out of fresh type variables. If this unification process succeeds returning operation O_2, then we add the typing $[\mathsf{O}_2(\Sigma), \mathsf{O}_2(\Gamma), \mathsf{O}_2(\langle m{:}(\vec{\tau}_{n'}) \to \gamma\rangle)]$ to the set of principal typing, where $\vec{\tau}_{n'}$ are the types for the formal parameters $\vec{x}_{n'}$ of m in the type environment Γ_b.

Class Compositionality

By modifying the above type inference procedure slightly, we can obtain an approach to type inference which is more akin to that of nominal type checking, described in Section 6.6. The philosophy behind that approach is that each class should be typed in isolation, separately from the others. This is achieved using nominal type *annotations* - each field is annotated with a nominal class type, as are the parameters and bodies of each method. These annotations serve, essentially, as type *assumptions*, and each class implicitly imports these type assumption for every other class in the program. The method bodies of each class are then checked, in the presence of these assumptions, to verify that the class satisfies its own type annotations. If all the classes pass this type checking phase, then it is guaranteed that each class satisfies the assumptions that were made of it by the others.

This procedure can be emulated using the type inference algorithm we have given above by applying it, for each class C, to an expression of the form $\texttt{new } C(\vec{x})$ and passing it a list of 'already analysed' classes consisting of the *full* set of classes in the program except C itself. The algorithm will then look inside all of the methods of C, analysing the method bodies, however it will not look inside any other classes during this process, since it will believe them to already have been analysed. If any of the method bodies use instances of other classes, the algorithm will actually *infer* the types required for these other classes, returning them in the self environments of the typings that it generates. The typings returned by such a use of the above algorithm would correspond to the ones we gave at the end of Section 10.3.3. The typings would also consist of type environments containing type assumptions for the variables $\vec{x}_n$, which would actually correspond to the types required of the object's field values in order to assign those types to the object itself. Thus, these typings also constitute a form if *implicit* class type: if we have that $[\Sigma, \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}, \tau]$ is a typing for the expression $\texttt{new } C(\vec{x})$, then this expresses that $(\vec{\sigma}_n) \to \tau$ is a valid class type for C.

Once such an analysis has been done for each class, we can check that the classes satisfy their mutual type requirements through unification. The basic principle is this: if we have typings for instances of C and D as follows

$$\Sigma, D{:}(\vec{\tau}_{n'}) \to \gamma; \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\} \vdash \texttt{new } C(\vec{x}_n) : \delta$$
$$\Sigma, C{:}(\vec{\sigma}_n) \to \delta; \{y_1{:}\tau_1, \ldots, y_{n'}{:}\tau_{n'}\} \vdash \texttt{new } D(\vec{y}_{n'}) : \gamma$$

then we know that C and D satisfy their mutual type requirements. Using the notion of valid class types we described above, an alternative way of seeing this mutual satisfaction of type constraints is as the following 'cut' rules:

$$\frac{\Sigma, D : B \vdash C : A \quad \Sigma, C : A \vdash D : B}{\Sigma \vdash C : A} \qquad \frac{\Sigma, D : B \vdash C : A \quad \Sigma \vdash D : B}{\Sigma \vdash C : A}$$

Thus, if we can unify the type assumed for D in (the self environment of) a typing inferred for $\texttt{new } C(\vec{x})$ with the class type implicit in a typing inferred for $\texttt{new } D(\vec{y})$, then we have satisfied one of the typing requirements in these typings. If we can repeat this process until we have eliminated all the class type assumptions in the self environment, then the resulting implicit class type is valid.

Having collected all the valid class types for the classes of a program, we can now use them 'as-is'. That is, they can be used to infer typings for executable expressions without having to look inside method

bodies any further. Thus, we have a *compositional* method for typing classes 'once and for all'.

10.5. Nakano Intersection Types

During the worked examples of Section 10.3, we commented several times about the inability of the system to type methods which invoke *other* methods of the receiver, such as the `mult` method in the object-oriented program, or to give an analysis for methods which incorporates slightly different views on their containing classes, such a heterogeneous types for lists. We will now describe an approach to extending the system to include intersections, which will allows us to type these problematic examples.

It is fairly straightforward to incorporate intersections into the type language:

Definition 10.13 (FJ•μ Intersection types)**.** *1. The set of FJ•μ object intersection pretypes (ranged over by ψ), and its subset of strict object and functional pretypes (ranged over by π and ϕ respectively) are defined by the following grammar:*

$$\begin{array}{lllllllllll} \pi & ::= & \varphi & | & \mathbf{n} & | & C & | & \bullet\pi & | & \\ & & \iota\pi & | & \langle f{:}\pi\rangle & | & \phi & & & & \\ \phi & ::= & \langle m{:}(\vec{\psi}) \to \pi\rangle & & & | & \bullet\phi & | & \iota\phi & | & \mu.\phi \\ \psi & ::= & \pi_1 \cap \ldots \cap \pi_n \quad (n \geq 1) & & & | & \iota\psi & | & \bullet\psi & & \end{array}$$

2. The definitions of free recursive variables and adequacy of pretypes are extended in the obvious way; FJ•μ types are defined as pretypes which are both adequate and closed; we will use meta-variables σ, τ, α, β, γ, and ρ to range over strict FJ•μ types. We will use the meta-variable θ to range over intersection types (and sometimes also the meta-variable ψ when it is clear from the context that it should be a proper type).

3. The set of FJ•μ class intersection types (ranged over by ζ) is defined by the following grammar:

$$\begin{array}{lllllll} \delta & ::= & (\vec{\sigma}) \to \tau & | & \bullet\delta & | & \iota\delta \\ \zeta & ::= & \delta_1 \cap \ldots \cap \delta_n \quad (n \geq 1) & | & \iota\zeta & | & \bullet\zeta \end{array}$$

Notice that in the definition above we have not included the universal type ω. This type should be added when considering a full formal treatment of this system, however for the purposes of the present discussion it is not required.

The subtyping relation can also easily be extended with the obvious cases for intersections:

Definition 10.14 (Subtyping for FJ•μ Intersection Types)**.** *1. The operation of μ-substitution from Definition 10.7 is extended to operate over intersections as follows:*

$$[\mathbf{n} \mapsto \mu.\phi](\bullet\psi) = \bullet([\mathbf{n} \mapsto \mu.\phi](\psi))$$
$$[\mathbf{n} \mapsto \mu.\phi](\iota\psi) = \iota([\mathbf{n} \mapsto \mu.\phi](\psi))$$
$$[\mathbf{n} \mapsto \mu.\phi](\pi_1 \cap \ldots \cap \pi_n) = ([\mathbf{n} \mapsto \mu.\phi](\pi_1)) \cap \ldots \cap ([\mathbf{n} \mapsto \mu.\phi](\pi_n))$$

2. *The subtyping relation on intersection pretypes is defined as in Definition 10.8, extended by the following:*

$$\pi_1 \cap \ldots \cap \pi_n \leq \pi_i \quad (\textit{for all } i \in \overline{n})$$

$$\psi \leq \pi_i \textit{ for each } i \in \overline{n},\ n \geq 1 \Rightarrow \psi \leq \pi_1 \cap \ldots \cap \pi_n$$

$$\bullet(\pi_1 \cap \ldots \cap \pi_n) \leq \bullet\pi_1 \cap \ldots \cap \bullet\pi_n \qquad \iota(\pi_1 \cap \ldots \cap \pi_n) \leq \iota\pi_1 \cap \ldots \cap \iota\pi_n$$

$$\bullet\pi_1 \cap \ldots \cap \bullet\pi_n \leq \bullet(\pi_1 \cap \ldots \cap \pi_n) \qquad \iota\pi_1 \cap \ldots \cap \iota\pi_n \leq \iota(\pi_1 \cap \ldots \cap \pi_n)$$

3. *The subtyping relation on class intersection types is defined as in Definition 10.8, extended by the following:*

$$\delta_1 \cap \ldots \cap \delta_n \leq \delta_i \quad (\textit{for all } i \in \overline{n})$$

$$\zeta \leq \delta_i \textit{ for each } i \in \overline{n},\ n \geq 1 \Rightarrow \zeta \leq \delta_1 \cap \ldots \cap \delta_n$$

$$\bullet(\delta_1 \cap \ldots \cap \delta_n) \leq \bullet\delta_1 \cap \ldots \cap \bullet\delta_n \qquad \iota(\delta_1 \cap \ldots \cap \delta_n) \leq \iota\delta_1 \cap \ldots \cap \iota\delta_n$$

$$\bullet\delta_1 \cap \ldots \cap \bullet\delta_n \leq \bullet(\delta_1 \cap \ldots \cap \delta_n) \qquad \iota\delta_1 \cap \ldots \cap \iota\delta_n \leq \iota(\delta_1 \cap \ldots \cap \delta_n)$$

To extend the type system, we obviously need to allow variables and fields to have intersection types in the typing environment, and we modify the (VAR) and (SELF-FLD) rules to assign a variable or field any of its strict types:

$$(\textsc{var}): \quad \frac{}{\Sigma;\Gamma,x{:}\psi \vdash x{:}\sigma}\,(\psi \leq \sigma) \qquad (\textsc{self-fld}): \quad \frac{}{\Sigma;\Gamma,f{:}\phi \vdash \texttt{this}{:}\langle f{:}\sigma\rangle}\,(\psi \leq \sigma)$$

Furthermore, we will need to allow *classes* to have intersections of class types in the self environment. This leads to the following obvious modification to the (SELF-METH) rule:

$$(\textsc{self-meth}): \quad \frac{}{\Sigma,C{:}\zeta;\Gamma,\texttt{this}{:}C \vdash \texttt{this}{:}\sigma}\,(\zeta \leq (\vec{\psi}_n) \to \sigma)$$

We also need to add the (JOIN) rule to allow intersections to be derived for arbitrary expressions:

$$(\textsc{join}): \quad \frac{\Sigma;\Gamma \vdash e{:}\sigma_i \quad (\forall\, i \in \overline{n})}{\Sigma;\Gamma \vdash e{:}\sigma_1 \cap \ldots \cap \sigma_n}\,(n \geq 2)$$

Lastly, we will need to modify the (INST-METH) rule. This modification, however, is slightly more subtle. Remember that the (INST-METH) rule is our version of the (Nakano) (FIX) rule:

$$(\textsc{fix}): \quad \frac{\Gamma, g : \bullet\sigma \vdash M : \sigma}{\Gamma \vdash \mathbf{FIX}\, g.M : \sigma}$$

In the context of intersection types, the above rule is extended as follows:

$$(\textsc{fix}_{\cap}): \quad \frac{\Gamma, g : \bullet\sigma_1 \cap \ldots \cap \bullet\sigma_n \vdash M : \sigma_i \quad (\forall\, i \in \overline{n})}{\Gamma \vdash \mathbf{FIX}\, g.M : \sigma_j}\,(n \geq 2,\ j \in \overline{n})$$

Thus, assuming an intersection type ψ for the recursive identifier, we must type the body of the definition with *each* type σ_i in the intersection. Then we are permitted to assign any of these types for the whole recursive definition (and thus also, via the (JOIN) rule, the full intersection type itself). Recall that the analogue to the recursive identifier g in the above inference rule is a class name C, and that the definition body M corresponds to the body of the method m for which we would like to assign a type $\langle m{:}(\vec{\sigma} \to \tau)\rangle$. Thus, the generalisation of our (INST-METH) rule should type a number of method bodies (possibly

analysing the same method body in multiple different ways):

$$(\textsc{inst-meth2}):\quad \frac{\Sigma, C{:}\delta_1 \cap \ldots \cap \delta_n; \Gamma_i \vdash e_i{:}\sigma_i \quad (\forall\, i \in \overline{n}) \qquad \Sigma; \Gamma \vdash e'_j{:}\theta_j \quad (\forall\, j \in \overline{r})}{\Sigma; \Gamma \vdash \texttt{new C}(\overrightarrow{e'}_r){:}\tau_j}$$

$$(j \in \overline{n}\ \&\ \mathcal{F}(C) = \overrightarrow{f}_r\ \&\ \forall\, i \in \overline{n}\ .\ \exists\, \overrightarrow{\psi}_r, \overrightarrow{\psi'}_s, \overrightarrow{x}_s, m\ .$$
$$\mathcal{M}b(C,m) = (\overrightarrow{x}_s, e_i), \Gamma_i = \{\texttt{this}{:}C, \overrightarrow{f{:}\psi}_r, \overrightarrow{x{:}\psi'}_s\},$$
$$\tau_i = \langle m{:}(\overrightarrow{\psi'}_s) \rightarrow \sigma_i\rangle, \bullet((\overrightarrow{\psi}_r) \rightarrow \tau_i) \leq \delta_i$$
$$\&\ \text{if } i = j \text{ then } \psi_k = \theta_k\ (\forall\, k \in \overline{r}))$$

This rule looks rather forbidding, but it is more straightforward than it appears at first glance. It simply ensures that for each class type δ_i in the intersection $\delta_1 \cap \ldots \cap \delta_n$ that is assumed for C, the appropriate method body e_i can be typed according to an implicit class type consistent with δ_i. Then, for one of the class types δ_j in particular, the rule checks that the field values $e'_1, \ldots, e'_r$ can be assigned the types assumed for the fields when typing its associated method body. The expressivity of this extended rule derives from the fact that recursively created `new C(...)` objects in any of the methods analysed can be typed according to *any* of the class types δ_i assumed for C, and not just the one corresponding to the current method.

Using this extended system with intersections, we are now able to assign more expressive types which allow sequences of calls to the same method to be made, with each call having a heterogeneous type. We will illustrate this using the oo list program.

We can assign the type $\langle \texttt{cons}{:}(\beta) \rightarrow \langle \texttt{cons}{:}(\alpha) \rightarrow \langle \texttt{head}{:}\alpha\rangle\rangle\rangle$ to lists which allows us to add two elements to the list and then retrieve the latter element:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_2;\Gamma_2 \vdash \texttt{o}{:}\alpha}\,(\textsc{var}) \quad \dfrac{}{\Sigma_2;\Gamma_2 \vdash \texttt{this}{:}\texttt{NEL}}\,(\textsc{var})}{\Sigma_2;\Gamma_2 \vdash \texttt{new NEL(o, this)}{:}\langle \texttt{head}{:}\alpha\rangle}\,(\textsc{inst-fld}) \quad \vdots \quad \dfrac{}{\Sigma_1;\Gamma_1 \vdash \texttt{o}{:}\beta}\,(\textsc{var}) \quad \dfrac{}{\Sigma_1;\Gamma_1 \vdash \texttt{this}{:}\texttt{EL}}\,(\textsc{var})}{\Sigma_1;\Gamma_1 \vdash \texttt{new NEL(o, this)}{:}\langle \texttt{cons}{:}(\alpha) \rightarrow \langle \texttt{head}{:}\alpha\rangle\rangle}\,(\textsc{inst-meth2})}{\vdash \texttt{new EL()}{:}\langle \texttt{cons}{:}(\beta) \rightarrow \langle \texttt{cons}{:}(\alpha) \rightarrow \langle \texttt{head}{:}\alpha\rangle\rangle\rangle}\,(\textsc{inst-meth2})$$

where $\Sigma_1 = \{\texttt{EL}{:}() \rightarrow \bullet\langle \texttt{cons}{:}(\beta) \rightarrow \langle \texttt{cons}{:}(\alpha) \rightarrow \langle \texttt{head}{:}\alpha\rangle\rangle\rangle\}$

$\Gamma_1 = \{\texttt{this}{:}\texttt{EL}, \texttt{o}{:}\beta\}$

$\Sigma_2 = \Sigma_1, \{\texttt{NEL}{:}(\bullet\beta, \bullet\texttt{EL}) \rightarrow \bullet\langle \texttt{cons}{:}(\alpha) \rightarrow \langle \texttt{head}{:}\alpha\rangle\rangle\}$

$\Gamma_2 = \{\texttt{this}{:}\texttt{NEL}, \texttt{head}{:}\beta, \texttt{tail}{:}\texttt{EL}, \texttt{o}{:}\alpha\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_3;\Gamma_3 \vdash \texttt{o}:\bullet\beta}\,(\textsc{var}) \quad \dfrac{\dfrac{}{\Sigma_3;\Gamma_3 \vdash \texttt{this}:\texttt{NEL}}\,(\textsc{var})}{\Sigma_3;\Gamma_3 \vdash \texttt{this}:\bullet\texttt{NEL}}\,(\leq)}{\Sigma_3;\Gamma_3 \vdash \texttt{new NEL(o, this)}:\bullet\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle}\,(\textsc{rec-meth}) \quad \dfrac{\dfrac{}{\Sigma_3;\Gamma_4 \vdash \texttt{o}:\alpha}\,(\textsc{var}) \quad \dfrac{}{\Sigma_3;\Gamma_4 \vdash \texttt{this}:\texttt{NEL}}\,(\textsc{var})}{\Sigma_3;\Gamma_4 \vdash \texttt{new NEL(o, this)}:\langle\texttt{head}:\alpha\rangle}\,(\textsc{inst-fld}) \quad \vdots\ e:\rho \quad \vdots\ l:\gamma}{\vdash \texttt{new NEL}(e,\ l):\langle\texttt{cons}:(\bullet\beta)\to\bullet\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle\rangle}\,(\textsc{inst-meth2})$$
$$\dfrac{\vdash \texttt{new NEL}(e,\ l):\langle\texttt{cons}:(\bullet\beta)\to\bullet\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle\rangle}{\vdash \texttt{new NEL}(e,\ l):\bullet\langle\texttt{cons}:(\beta)\to\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle\rangle}\,(\leq)$$
$$\dfrac{\vdash \texttt{new NEL}(e,\ l):\bullet\langle\texttt{cons}:(\beta)\to\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle\rangle}{\vdash \texttt{new NEL}(e,\ l):\langle\texttt{cons}:(\beta)\to\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle\rangle}\,(\bullet)$$

where $\Sigma_3 = \{\texttt{NEL}:(\bullet\rho,\bullet\gamma)\to\bullet\langle\texttt{cons}:(\bullet\beta)\to\bullet\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle\rangle$
$\cap\ (\bullet\beta,\bullet\texttt{NEL})\to\bullet\langle\texttt{cons}:(\alpha)\to\langle\texttt{head}:\alpha\rangle\rangle\}$
$\Gamma_3 = \{\texttt{this}:\texttt{NEL},\texttt{head}:\rho,\texttt{tail}:\gamma,\texttt{o}:\bullet\beta\}$
$\Gamma_4 = \{\texttt{this}:\texttt{NEL},\texttt{head}:\beta.\texttt{tail}:\texttt{NEL},\texttt{o}:\alpha\}$

We can also assign the type $\langle\texttt{cons}:(\beta)\to\langle\texttt{cons}:(\alpha)\to\langle\texttt{tail}:\langle\texttt{head}:\beta\rangle\rangle\rangle\rangle$ which allows access to the first of the two elements added to the list:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_2;\Gamma_2 \vdash \texttt{o}:\alpha}\,(\textsc{var}) \quad \dfrac{}{\Sigma_2;\Gamma_2 \vdash \texttt{this}:\langle\texttt{head}:\beta\rangle}\,(\textsc{self-fld})}{\Sigma_2;\Gamma_2 \vdash \texttt{new NEL(o, this)}:\langle\texttt{tail}:\langle\texttt{head}:\beta\rangle\rangle}\,(\textsc{inst-fld}) \quad \vdots \quad \dfrac{}{\Sigma_1;\Gamma_1 \vdash \texttt{o}:\beta}\,(\textsc{var}) \quad \dfrac{}{\Sigma_1;\Gamma_1 \vdash \texttt{this}:\texttt{EL}}\,(\textsc{var})}{\Sigma_1;\Gamma_1 \vdash \texttt{new NEL(o, this)}:\langle\texttt{cons}:(\alpha)\to\langle\texttt{tail}:\langle\texttt{head}:\beta\rangle\rangle\rangle}\,(\textsc{inst-meth2})}{\vdash \texttt{new EL()}:\langle\texttt{cons}:(\beta)\to\langle\texttt{cons}:(\alpha)\to\langle\texttt{tail}:\langle\texttt{head}:\beta\rangle\rangle\rangle\rangle}\,(\textsc{inst-meth2})$$

where $\Sigma_1 = \{\texttt{EL}:()\to\bullet\langle\texttt{cons}:(\beta)\to\langle\texttt{cons}:(\alpha)\to\langle\texttt{tail}:\langle\texttt{head}:\beta\rangle\rangle\rangle\rangle\}$
$\Gamma_1 = \{\texttt{this}:\texttt{EL},\texttt{o}:\beta\}$
$\Sigma_2 = \Sigma_1, \{\texttt{NEL}:(\bullet\beta,\bullet\texttt{EL})\to\bullet\langle\texttt{cons}:(\alpha)\to\langle\texttt{tail}:\langle\texttt{head}:\beta\rangle\rangle\rangle\}$
$\Gamma_2 = \{\texttt{this}:\texttt{NEL},\texttt{head}:\beta,\texttt{tail}:\texttt{EL},\texttt{o}:\alpha\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_3;\Gamma_3 \vdash \mathtt{o}:\bullet\beta}\,(\textsc{var}) \quad \dfrac{\dfrac{}{\Sigma_3;\Gamma_3 \vdash \mathtt{this}:\mathtt{NEL}}\,(\textsc{var})}{\Sigma_3;\Gamma_3 \vdash \mathtt{this}:\bullet\mathtt{NEL}}\,(\leq)}{\Sigma_3;\Gamma_3 \vdash \mathtt{new\ NEL(o,\ this)}:\bullet\langle\mathtt{cons}:(\alpha)\to\langle\mathtt{tail}:\langle\mathtt{head}:\beta\rangle\rangle\rangle}\,(\textsc{rec-meth}) \quad \dfrac{\dfrac{}{\Sigma_3;\Gamma_4 \vdash \mathtt{o}:\alpha}\,(\textsc{var}) \quad \dfrac{}{\Sigma_3;\Gamma_4 \vdash \mathtt{this}:\langle\mathtt{head}:\beta\rangle}\,(\textsc{self-fld})}{\Sigma_3;\Gamma_4 \vdash \mathtt{new\ NEL(o,\ this)}:\langle\mathtt{tail}:\langle\mathtt{head}:\beta\rangle\rangle}\,(\textsc{inst-fld}) \quad \vdots\ \vdash e:\rho \quad \vdots\ \vdash l:\gamma}{\vdash \mathtt{new\ NEL}(e,\ l):\langle\mathtt{cons}:(\bullet\beta)\to\bullet\langle\mathtt{cons}:(\alpha)\to\langle\mathtt{tail}:\langle\mathtt{head}:\beta\rangle\rangle\rangle\rangle}\,(\textsc{inst-meth}2)}{\vdash \mathtt{new\ NEL}(e,\ l):\bullet\langle\mathtt{cons}:(\beta)\to\langle\mathtt{cons}:(\alpha)\to\langle\mathtt{tail}:\langle\mathtt{head}:\beta\rangle\rangle\rangle\rangle}\,(\leq)}{\vdash \mathtt{new\ NEL}(e,\ l):\langle\mathtt{cons}:(\beta)\to\langle\mathtt{cons}:(\alpha)\to\langle\mathtt{tail}:\langle\mathtt{head}:\beta\rangle\rangle\rangle\rangle}\,(\bullet)$$

where $\Sigma_3 = \{\mathtt{NEL}:(\bullet\rho,\bullet\gamma)\to\bullet\langle\mathtt{cons}:(\bullet\beta)\to\bullet\langle\mathtt{cons}:(\alpha)\to\langle\mathtt{tail}:\langle\mathtt{head}:\beta\rangle\rangle\rangle\rangle$
$\cap\ (\bullet\beta,\bullet\mathtt{NEL})\to\bullet\langle\mathtt{cons}:(\alpha)\to\langle\mathtt{tail}:\langle\mathtt{head}:\beta\rangle\rangle\rangle\}$
$\Gamma_3 = \{\mathtt{this}:\mathtt{NEL},\mathtt{head}:\rho,\mathtt{tail}:\gamma,\mathtt{o}:\bullet\beta\}$
$\Gamma_4 = \{\mathtt{this}:\mathtt{NEL},\mathtt{head}:\beta.\mathtt{tail}:\mathtt{NEL},\mathtt{o}:\alpha\}$

Regarding the object-oriented arithmetic program of Section 10.3.6, recall that we could not assign the argument type $\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle$ of the type for the `mult` method $\tau = \mu.\langle\mathtt{mult}:(\langle\mathtt{add}:(\bullet\mathbf{0})\to\bullet\mathbf{0}\rangle)\to\bullet\mathbf{0}\rangle$ to positive object-oriented natural numbers, i.e. expressions of the form `new Suc(`n`)`. Using the system extended with intersections however this is now possible, allowing us to type invocations of the form n`.mult(`m`)` (where n and m are object-oriented natural numbers). Let $\mathcal{D}$ be the following derivation, which we reproduce from Section 10.3.6:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma_1 \vdash \mathtt{this}:\langle\mathtt{pred}:\tau\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma_1 \vdash \mathtt{this.pred}:\tau}\,(\textsc{fld})}{\Sigma;\Gamma_1 \vdash \mathtt{this.pred}:\langle\mathtt{mult}:(\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle)\to\bullet\tau\rangle}\,(\leq) \quad \dfrac{}{\Sigma;\Gamma_1 \vdash \mathtt{x}:\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle}\,(\textsc{var})}{\Sigma;\Gamma_1 \vdash \mathtt{this.pred.mult(x)}:\bullet\tau}\,(\textsc{invk}) \quad \dfrac{\dfrac{}{\Sigma;\Gamma_1 \vdash \mathtt{x}:\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle}\,(\textsc{var}) \quad \vdots}{\Sigma;\Gamma_1 \vdash \mathtt{x.add(this.pred.mult(x))}:\bullet\tau}\,(\textsc{invk})$$

where $\Sigma = \{\mathtt{Suc}:(\bullet\tau)\to\bullet\tau\cap(\bullet\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle)\to\bullet\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle\}$ and
$\Gamma_1 = \{\mathtt{this}:\mathtt{Suc},\mathtt{pred}:\tau,\mathtt{x}:\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle\}$

Then, given that we can assign the type $\langle\mathtt{add}:(\bullet\tau)\to\bullet\tau\rangle$ to a number n, we can construct the following

derivation which assigns that type to its successor:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma_2 \vdash \texttt{this}:\langle \texttt{pred}:\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma_2 \vdash \texttt{this.pred}:\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle}\,(\textsc{fld}) \quad \dfrac{}{\Sigma;\Gamma_2 \vdash \texttt{x}:\bullet\tau}\,(\textsc{var})}{\Sigma;\Gamma_2 \vdash \texttt{this.pred.add(x)}:\bullet\tau}\,(\textsc{invk})}{\Sigma;\Gamma_2 \vdash \texttt{new Suc(this.pred.add(x))}:\bullet\tau}\,(\textsc{rec-meth})}{\dfrac{\begin{array}{c}\mathcal{D}\\ \Sigma;\Gamma_1 \vdash \texttt{x.add(this.pred.mult(x))}:\bullet\tau\end{array} \quad \vdots \quad \vdash n:\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle}{\vdash \texttt{new Suc}(n):\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle}\,(\textsc{inst-meth})}$$

where $\Gamma_2 = \{\texttt{this}:\texttt{Suc}, \texttt{pred}:\langle \texttt{add}:(\bullet\tau)\to\bullet\tau\rangle, \texttt{x}:\bullet\tau\}$

Using intersection class types, we can also give numbers an alternative type for the `mult` method, namely $\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle$, where $\sigma = \mu.\langle \texttt{add}:(\bullet\mathbf{0})\to\bullet\mathbf{0}\rangle$.

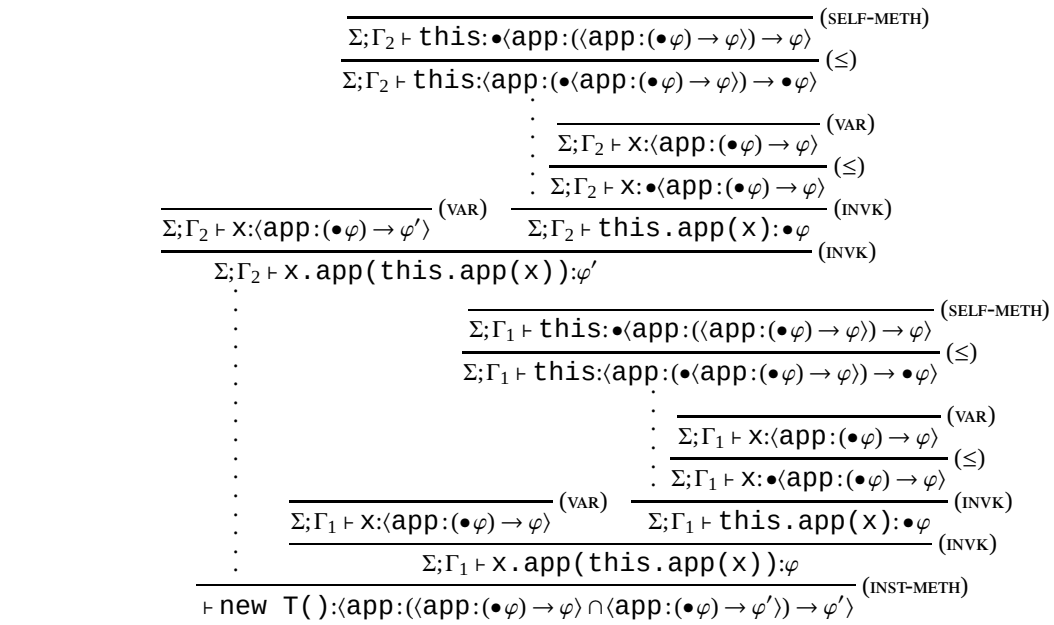
$$\dfrac{\dfrac{}{\Sigma;\Gamma_1 \vdash \texttt{this}:\bullet\sigma}\,(\textsc{self-meth}) \quad \dfrac{}{\Sigma;\Gamma_2 \vdash \texttt{x}:\bullet\sigma}\,(\textsc{var})}{\vdash \texttt{new Zero}():\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle}\,(\textsc{inst-meth})$$

where $\Sigma = \{\texttt{Zero}:() \to \bullet\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle \cap () \to \bullet\sigma\}$,
$\Gamma_1 = \{\texttt{this}:\texttt{Zero}, \texttt{x}:\sigma\}$ and $\Gamma_2 = \{\texttt{this}:\texttt{Zero}, \texttt{x}:\bullet\sigma\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma;\Gamma \vdash \texttt{this}:\langle \texttt{pred}:\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle\rangle}\,(\textsc{self-fld})}{\Sigma;\Gamma \vdash \texttt{this.pred}:\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle}\,(\textsc{fld}) \quad \dfrac{}{\Sigma;\Gamma \vdash \texttt{x}:\sigma}\,(\textsc{var})}{\Sigma;\Gamma \vdash \texttt{this.pred.mult(x)}:\bullet\sigma}\,(\textsc{invk}) \quad \dfrac{}{\Sigma;\Gamma \vdash \texttt{x}:\sigma}\,(\textsc{var})}{\Sigma;\Gamma \vdash \texttt{x}:\langle \texttt{add}:(\bullet\sigma)\to\bullet\sigma\rangle}\,(\leq)}{\Sigma;\Gamma \vdash \texttt{x.add(this.pred.mult(x))}:\bullet\sigma}\,(\textsc{invk})}{\dfrac{\vdots \quad \vdash n:\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle}{\vdash \texttt{new Suc}(n):\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle}\,(\textsc{inst-meth})}$$

where $\Sigma = \{\texttt{Suc}:(\bullet\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle)\to\bullet\langle \texttt{mult}:(\sigma)\to\bullet\sigma\rangle\}$
$\Gamma = \{\texttt{this}:\texttt{Zero}, \texttt{pred}:\sigma, \texttt{x}:\sigma\}$

The intersection class types that we have used in the above derivations could be said to be simply *records* rather than true intersections, since the result type of each class type refers to a different method. However we make the final observation that, as for the intersection type system of Part I, the intersections that we have introduced to FJ$\bullet\mu$ allow more than just records. This can be illustrated using the fixed point combinator that we considered in Section 10.3.4. Using the extended system, we can assign to it a whole family of intersection types which are exactly analogous to the family of intersection types that



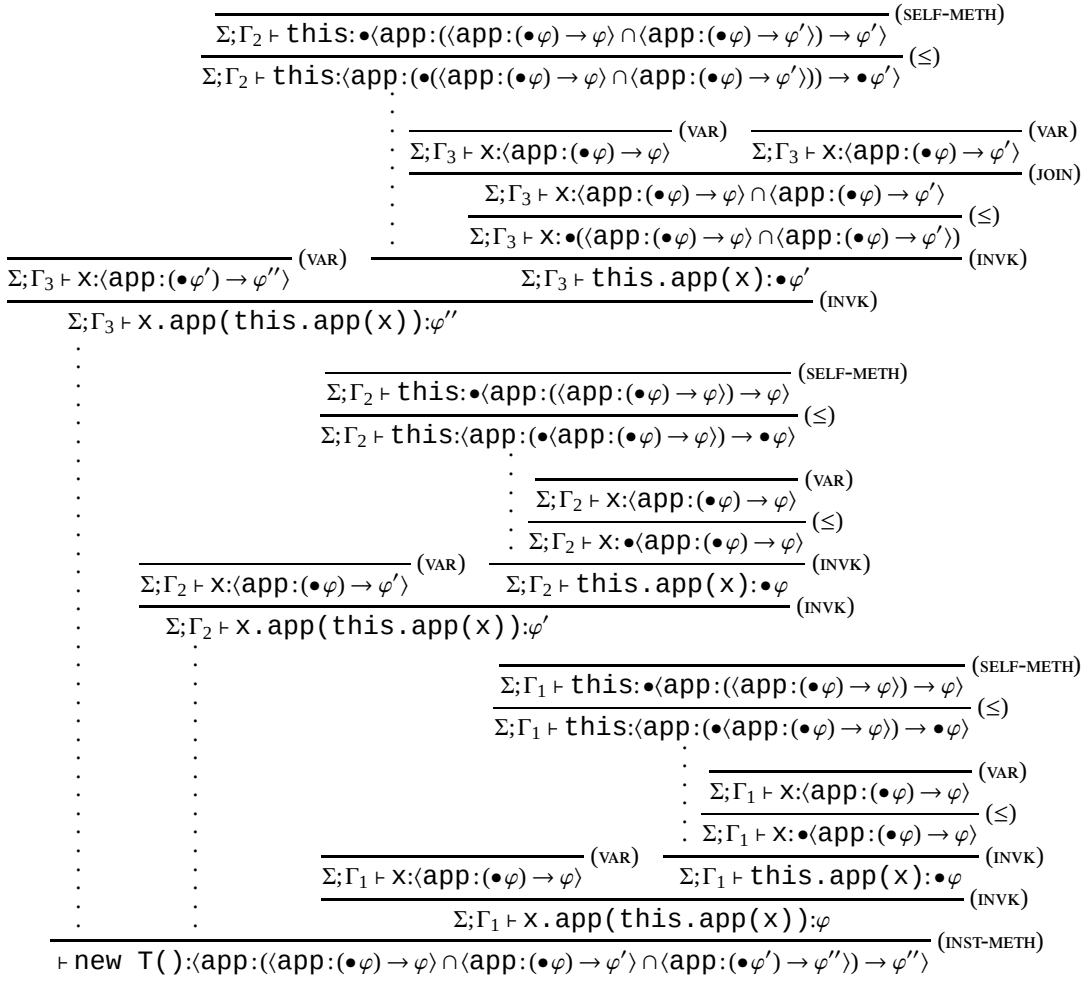
where $\Sigma = \{\mathtt{T}{:}() \to \bullet\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle) \to \varphi'\rangle$
$\cap\, () \to \bullet\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle) \to \varphi\rangle\}$,
$\Gamma_1 = \{\mathtt{this}{:}\mathtt{T}, \mathtt{x}{:}\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle\}$ and
$\Gamma_2 = \{\mathtt{this}{:}\mathtt{T}, \mathtt{x}{:}\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle\}$

Figure 10.2.: Type Derivation for FJ$\bullet\mu$ Intersection Type Assignment for a Fixed Point Combinator (1)

are assignable to fixed point combinators in λ-calculus. That is, we can assign to `new T()` the following family of intersection Nakano types:

$$\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle) \to \varphi\rangle$$
$$\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle) \to \varphi'\rangle$$
$$\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi') \to \varphi''\rangle) \to \varphi''\rangle$$
$$\vdots$$

The first of these is the familiar Nakano type for fixed point operators, and we gave a derivation assigning this type back in Section 10.3.4. In the Figures below, we give derivations for the next two types in this family. The interesting thing to note here is that the derivation for each of these types requires that `T` have an *intersection* class type in the self environment, which is formed from all the types in the sequence up to and including the one that is being derived. None of these types is related to any other via subtyping and thus this is a true class intersection type.



where

$\Sigma = \{\mathtt{T}{:}() \to \bullet\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi') \to \varphi''\rangle) \to \varphi''\rangle$
$\quad \cap () \to \bullet\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle) \to \varphi'\rangle$
$\quad \cap () \to \bullet\langle \mathtt{app}{:}(\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle) \to \varphi\rangle\}$,
$\Gamma_1 = \{\mathtt{this}{:}\mathtt{T}, \mathtt{x}{:}\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle\}$
$\Gamma_2 = \{\mathtt{this}{:}\mathtt{T}, \mathtt{x}{:}\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle\}$ and
$\Gamma_3 = \{\mathtt{this}{:}\mathtt{T}, \mathtt{x}{:}\langle \mathtt{app}{:}(\bullet\varphi) \to \varphi\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi) \to \varphi'\rangle \cap \langle \mathtt{app}{:}(\bullet\varphi') \to \varphi''\rangle\}$

Figure 10.3.: Type Derivation for FJ$\bullet\mu$ Intersection Type Assignment for a Fixed Point Combinator (2)

Conclusion

11. Summary of Contributions & Future Work

In this thesis, we have aimed to develop a notion of type assignment for the class-based approach to object-oriented programming that expresses detailed properties of the behaviour of programs, including their termination. It has also been our aim to firmly base our type assignment systems in the denotational semantics of programs. Our initial approach to this task was to use the intersection type discipline, the type systems belonging to which have been shown to demonstrate this ability in many different computational settings from λ-calculus to process and sequent calculi.

Our work is not the first to apply intersection types to the study of object-orientation. De'Liguoro and van Bakel have undertaken a comprehensive programme of research in which they develop an intersection type system for Abadi and Cardelli's ς-calculus, and use it to analyse many aspects of that model of computation. While the ς-calculus provides a fundamental basis for the object-oriented paradigm in its extensive variety, it is still fairly far removed from the styles of oo programming used in practice, at least as far as class-based programming is concerned. Thus, even though one can consider 'compiling' class-based programs down to the ς object-based level, our research has been an attempt at bringing the success of de'Liguoro and van Bakel's analysis a little closer to the 'source language' level that a large number of oo developers are used to in their every-day lives. Thus we have built our type analysis around the Featherweight Java formal model.

Our work has been influenced and inspired by the intersection type system for the ς-calculus, but there are important differences and we have extended that work in a number of ways. Since we are still interested in analysing objects, it was natural to inherit the structural types of the ς-calculus and its subsequent intersection type system. An important decision that was made early on in our research, however, was to divorce our intersection type analysis from the existing nominal type system of Featherweight Java. The motivation for this lay in our desire to obtain a system that fully characterised normalisation. As we have discussed in Section 6.6, the rigid constraints imposed by the nominal typing approach exclude a number of terminating programs, and thus restricting our system to only typing programs that 'pass' nominal type assignment defeats this aim.

The other major difference between our system and that of van Bakel and de'Liguoro is in our semantic treatment. A principal result of our research is that of the approximation semantics for Featherweight Java and the approximation theorem linking these semantics with our notion of type assignment. Our approximation semantics is the first such model of object-oriented programming, as far as we are aware. Furthermore, our approximation result, as discussed in Section 2.1, demonstrates that our intersection types precisely predict the output of computation. We have also made the comment that our types give a characterisation of the observable properties and observational equivalence of programs, although we do not have a formal result to this effect.

A more subtle difference between our treatment and that of van Bakel and de'Liguoro is that, because we do not have to deal with method override (in the sense of modifying an individual object's method *bodies*), we can move from a *late* self-typing approach to an early one. This manifests itself in our system

in the fact that our (NEWM) rule requires, as a premise, that the type assumed for the variable `this` when typing the method body can be assigned to the object itself for which we are deriving the method type, i.e the eventual receiver of the method invocation. This is in contrast to de'Liguoro and van Bakel's system, where the requirements on the self are coded into the method type and these requirements are then checked at *invocation* time by the (Val Select) rule. An early version of this work [16] did have a (functional, i.e. stateless) *field* update construction, and that system employed late self-typing. As we have observed however (cf. Section 3.2), this feature may be encoded and so we felt that, at this stage, our research would benefit from a simpler presentation. The early self-typing approach seems to lead to a leaner system, since it roots out as early as possible those 'counter-factual' types which are predicated on the receiver satisfying impossible typing constraints. For example, in a system with late self-typing we could derive types for the `loop` method of the non-terminating program of Section 6.2, by *assuming* that the receiver `new NT()` can be assigned the very type that we are deriving. The relationship between late and early self-typing is an interesting one though, and one which we feel deserves further investigation. At the very least, the fact that field update can be encoded suggests an alternative mechanism to de'Liguoro and van Bakel's for typing such a construction.

The technique that we have used in this thesis to show the approximation result, derivation reduction, also constitutes an extension over its previous application. We were motivated to use this technique through observing the similarity between term rewriting systems and the (class-based) oo programming model. Specifically, our encoding of Combinatory Logic bears a strong resemblance to a 'curryfied' term rewriting version of that system. Fernández and van Bakel successfully used the technique of derivation reduction to show an approximation result for combinator systems [15]. The key difference between that work and ours, is that the type assignment system considered for combinator system was *partial* - the types for combinators were derived from an environment, essentially a look-up table, and so are in a sense *external* to the system. In this respect, our type system can be considered a *full* type assignment system - the types assigned to our 'combinators' (i.e. objects) are derived *within* the type assignment system itself, via an analysis of the method body (i.e. the right-hand side of the combinator's reduction rule). Because of this subtle, but important difference, our proof for the strong normalisation of derivation reduction can be done by a straightforward induction on the structure of derivations, rather than appealing to more abstract relationships between terms as done in [15]. The aforementioned similarity between the oo reduction model and TRS also suggests that our research can be used to define 'full' systems of type assignment for a generalised notion of term rewriting.

Having established the type-theoretical expressivity and semantic soundness of our system, we turned our attention to the problem of type inference. A chief motivating factor throughout this research has been to determine to what extent our type analysis can be used for automatic program verification. We noted that there is a well-defined hierarchy of restrictions that makes intersection type assignment decidable in the setting of the λ-calculus. We considered the most restrictive variant in this hierarchy, essentially equivalent to Curry's type system. While we were able to show a principal typings property for this restriction, we also observed that the inherent ability present in the class-based paradigm to define classes *recursively* raises serious barriers to the inference of useful types in our system.

This failure in the satisfactory application of our type system motivated us to look for extensions which would allow us to infer more meaningful types, while still allowing us to capture the functional behaviour and convergence of object-oriented programs. We identified Nakano's systems of logical recursive types

as a suitable candidate, since it gives a guarantee of head normalisation. Before extending our system to include Nakano-style recursive types, we first showed that type inference for Nakano's notion of type assignment is feasible by giving an algorithm for inferring typings which we conjectured to be sound. We showed that the problem of when to infer bullets in the typings for terms is not a trivial one. In order to give a systematic solution to this problem, we proposed extending the type language with 'placeholder' elements (insertion variables) to mark the points in typing derivations where bullets may be used. In this, we were inspired by the work of Kfoury and Wells on expansion variables for intersection type inference.

We then considered how Nakano-style recursive types could be applied to the object-oriented model by defining a system (FJ$\bullet\mu$) for assigning them to Featherweight Java programs. We did present formal results for this type system regarding its semantic properties (i.e. normalisation, approximation, soundness and completeness) - this is an important task for future research. We did, however, demonstrate through the use of case studies and worked examples, the ability of this extended type system to assign useful and informative types to programs which were problematic for our intersection type inference algorithm. We then discussed how our type inference procedures for our intersection type system and Nakano's original λ-calculus-based system might be merged and extended to provide type inference for FJ$\bullet\mu$. We also discussed how intersections might be added to this system.

Future Work

The wide scope of our work provides many directions for the future continuation of this research. The immediate priority is to derive formal results for our Nakano-style type assignment systems. Specifically, for our unification and type inference algorithms we must formally prove its soundness. For the Nakano-style type system for FJ$^{\text{¢}}$, we must prove at least soundness (i.e. that typeability is preserved by reduction). An interesting question is the *completeness* of type assignment, or in other words whether typeability is preserved under expansion. This is an open question for Nakano's original system too. This property does not hold for non-logical systems of recursive types [34, Remark 2.6(i)], and so it may be that such a result requires the use of intersections. In addition, we would like to show an approximation result for this system. We also believe that extension of the modal approach of Nakano might lead to a system which gives stronger normalisation guarantees, and this is something we would like to investigate further. On the type inference side, we must extend the algorithm of Chapter 9 in order to make it complete. Formally, this could involve showing a principal typings property for our extension of Nakano's system. Furthermore, as we have remarked, our approach type inference is fundamentally unification-based, but it would be interesting to consider what advantages or difficulties would arise out of a constraint-based approach. On a practical note, we would like to produce a working prototype of our system for Featherweight Java and analyse its effectiveness in an operational setting.

Beyond that discussed above, there is a need to extend our treatment beyond the limited set of features modelled by Featherweight Java. An important extension to the calculus involves adding imperative, or state-based features. Such an extension is already considered by Abadi and Cardelli for the ς-calculus, and there are also state-based extensions of Featherweight Java (e.g. Middleweight Java[22]). We would like to apply the intersection type discipline to this important feature, which would move our systems one step closer to practical application, as well providing an alternative theoretical treatment to state-based

issues. There are other computational features. One example is the use of exceptions, and other similar mechanisms for *control flow*. There is a wide body of research connecting such control mechanisms, via a Curry-Howard correspondence, with the notion of *classical* logic. Van Bakel's work on the application of IRTD to such issues suggests that a similar approach could work in the context of Featherweight Java and OO.

There are interesting directions for future research outside of the sphere of the object-oriented world. As we have suggested above, our approach to type assignment could afford improvements or alternatives in the field of more general term rewriting systems. There is also the question of applying Nakano's systems not only to the object-oriented paradigm, but to the functional one too. Since Nakano's formulation applies to the λ-calculus, it should be even more straightforward to apply it to functional languages such as ML or Haskell, than to OO. Type-based termination guarantees, even partial ones, would provide immediate productivity benefits in the practical programming community.

Bibliography

[1] *Ruby online documentation*. `http://www.ruby-lang.org/en/`.

[2] M. Abadi and L. Cardelli. *A Theory Of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *POPL*, pages 396–409, 1996.

[4] Jim Alves-Foss and Fong Shing Lam. Dynamic Denotational Semantics of Java. In *Formal Syntax and Semantics of Java*, pages 201–240, 1999.

[5] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[6] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Type inference for polymorphic methods in java-like languages. In *ICTCS*, pages 118–129, 2007.

[7] S. van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102:135–163, 1992.

[8] S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, University of Nijmegen, 1993.

[9] S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 151(2):385–435, 1995.

[10] S. van Bakel. The Heart of Intersection Type Assignment: Normalisation Proofs Revisited. *Theor. Comput. Sci.*, 398(1-3):82–94, 2008.

[11] S. van Bakel and U. de'Liguoro. Logical semantics for the first order varsigma-calculus. In *ICTCS*, pages 202–215, 2003.

[12] S. van Bakel and U. de'Liguoro. Subtyping object and recursive types logically. In *ICTCS*, pages 66–80, 2005.

[13] S. van Bakel and U. de'Liguoro. Logical Equivalence for Subtyping Object and Recursive Types. *Theory of Computing Systems*, 42(3):306–348, 2008.

[14] S. van Bakel and M. Fernández. Normalization Results for Typeable Rewrite Systems. *Information and Computation*, 133(2):73–116, 1997.

[15] S. van Bakel and M. Fernández. Normalization, Approximation, and Semantics for Combinator Systems. *Theor. Comput. Sci.*, 290(1):975–1019, 2003.

[16] S. van Bakel and R. N. S. Rowe. Semantic Predicate Types and Approximation for Class-based Object-Oriented Programming. In *proceedings of 11th Workshop on Formal Techniques for Java-like Programs (FTfJP'09)*, Genova, Italy, 2009.

[17] A. Banerjee and T. P. Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathematical Structures in Computer Science*, 13(1):87–124, 2003.

[18] F. Barbanera and U. de'Liguoro. Type Assignment for Mobile Objects. *Electr. Notes Theor. Comput. Sci.*, 104:25–38, 2004.

[19] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.

[20] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[21] L. Bettini, S. Capecchi, and B. Venneri. Featherweight Java with Multi-Methods. In *PPPJ, volume 272 of ACM International Conference Proceeding Series*, pages 83–92. ACM, 2007.

[22] G. Bierman, M. J. Parkinson, and A. Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, University of Cambridge Computer Laboratory, 15 JJ Thompson Ave., Cambridge, CB3 0FD, UK, April 2003.

[23] George Boolos. *The Logic of Provability*. Cambridge University Press, 1993.

[24] Gérard Boudol. On strong normalization and type inference in the intersection type discipline. *Theor. Comput. Sci.*, 398(1-3):63–81, 2008.

[25] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA*, pages 183–200, 1998.

[26] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inform.*, 33(4):309–338, 1998.

[27] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Funct. Program.*, 4(2):127–206, 1994.

[28] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. In *TACS*, pages 415–438, 1997.

[29] Michele Bugliesi and Santiago M. Pericás-Geertsen. Type inference for variant object types. *Inf. Comput.*, 177(1):2–27, 2002.

[30] M.T. Burt. *Games, Call-by-Value and Featherweight Java*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, England, 2004.

[31] L. Cardelli. A Semantics of Multiple Inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[32] L. Cardelli and J. C. Mitchell. Operations on Records. In *Proceedings of the fifth international conference on Mathematical foundations of programming semantics*, pages 22–52, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[33] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[34] Felice Cardone and Mario Coppo. Type Inference with Recursive Types: Syntax and Semantics. *Inf. Comput.*, 92(1):48–80, 1991.

[35] Felice Cardone and Mario Coppo. Decidability properties of recursive types. In *ICTCS*, pages 242–255, 2003.

[36] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

[37] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Daniele Varacca. Encoding cduce in the cpi-calculus. In *CONCUR*, pages 310–326, 2006.

[38] A. Church. A Note on the Entscheidungsproblem. 1(1):40–41, 1936.

[39] W.R. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In G. Bosworth, editor, *Conference on Object-Oriented Programming: Systems, Languages, And Applications (OOPSLA'89), New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings*, pages 433–443. ACM, 1989.

[40] M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame, Journal of Formal Logic*, 21(4):685–693, 1980.

[41] M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame, Journal of Formal Logic*, 21(4):685–693, 1980.

[42] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[43] Mario Coppo and Paola Giannini. Principal Types and Unification for a Simple Intersection Type System. *Inf. Comput.*, 122(1):70–96, 1995.

[44] H. B. Curry. Grundlagen der kombinatorischen logik. *Amer. J. Math.*, 52:509–536, 1930.

[45] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[46] O-J. Dahl and K. Nygaard. SIMULA: an ALGOL-based Simulation Language. *Commun. ACM*, 9(9):671–678, 1966.

[47] F. Damiani and F. Prost. Detecting and Removing Dead-Code using Rank 2 Intersection. In *Types for Proofs and Programs (International Workshop TYPES'96, Selected Papers)*, LNCS 1512, pages 66–87. Springer, 1998.

[48] U. de'Liguoro. Characterizing Convergent Terms in Object Calculi via Intersection Types. In *Proc. of TCLA'01*, volume 2004 of *Lecture Notes in Computer Science*, pages 315–328, 2001.

[49] Ugo de'Liguoro. Subtyping in Logical Form. *Electr. Notes Theor. Comput. Sci.*, 70(1), 2002.

[50] Mariangiola Dezani-Ciancaglini and J. Roger Hindley. Intersection Types for Combinatory Logic. *Theor. Comput. Sci.*, 100(2):303–324, 1992.

[51] Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, pages 169–184, 1995.

[52] Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to oop. *Electr. Notes Theor. Comput. Sci.*, 1:132–153, 1995.

[53] Jörg Endrullis, Clemens Grabmayer, Jan Willem Klop, and Vincent van Oostrom. On equal μ-terms. *Theor. Comput. Sci.*, 412(28):3175–3202, 2011.

[54] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *In POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282. ACM Press, 2006.

[55] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[56] Martin Odersky et al. An overview of the scala programming language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, 2004.

[57] S. Feferman. A language and axioms for explicit mathematics. In J. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*. Springer, Berlin, 1975.

[58] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic J. of Computing*, 1(1):3–37, 1994.

[59] K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtying. In *FCT*, pages 42–61, 1995.

[60] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[61] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd Edition)*. Prentice Hall, 2005.

[62] J. Roger Hindley. The Completeness Theorem for Typing λ-Terms. *Theor. Comput. Sci.*, 22:1–17, 1983.

[63] Martin Hofmann and Benjamin C. Pierce. A unifying type-theoretic framework for objects. *J. Funct. Program.*, 5(4):593–635, 1995.

[64] W.A. Howard. The Formulae-as-Types Notion of Construction. In J. Hindley and J. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1969.

[65] A. Igarashi and B. C. Pierce. On Inner Classes. In *Information and Computation*, 2000.

[66] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[67] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In *APLAS*, pages 161–177, 2005.

[68] ECMA International. *ECMA Language Specification (ECMA-262), 3rd Edition*. December 2006. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[69] ECMA International. *The C# Language Specification (ECMA-334), 4th Edition*. June 2006. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[70] T. P. Jensen. Types in program analysis. pages 204–222, 2002.

[71] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping, 1997. Manuscript.

[72] S'N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In *POPL*, pages 80–87, 1988.

[73] S.N. Kamin and U.S. Reddy. Two semantic models of object-oriented languages. In C.A. Gunter and J.C. Mitchell, editors, *Theoretical aspects of object-oriented programming*, pages 463–495. MIT Press, Cambridge, MA, USA, 1994.

[74] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL*, pages 161–174, 1999.

[75] A. J. Kfoury and J. B. Wells. Principality and Type Inference for Intersection Types Using Expansion Variables. *Theor. Comput. Sci.*, 311(1-3):1–70, 2004.

[76] Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30(2), 2008.

[77] David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. *Information and Control*, 71(1/2):95–130, 1986.

[78] N.P. Mendler. Recursive Types and Type Constraints in Second Order Lambda Calculus. In *Proceedings Second Symposium on Logic in Computer Science*, pages 30–36. IEEE, 1987.

[79] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[80] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[81] J. C. Mitchell. Toward A Typed Foundation for Method Specialization and Inheritance. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–124, New York, NY, USA, 1990. ACM.

[82] H. Nakano. A modality for recursion. Technical report.

[83] H. Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

[84] Hiroshi Nakano. Fixed-Point Logic with the Approximation Modality and its Kripke Completeness. In *TACS*, pages 165–182, 2001.

[85] Jens Palsberg. Efficient inference of object types. *Inf. Comput.*, 123(2):198–209, 1995.

[86] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Asp. Comput.*, 9(1):49–67, 1997.

[87] Mauro Piccolo. Strong normalization in the π-calculus with intersection and union types. *Fundamenta Informaticae*, 2010. Accepted.

[88] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[89] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *J. Funct. Program.*, 4(2):207–247, 1994.

[90] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, pages 324–340, 1994.

[91] D. Prawitz. *Natural Deduction: A Proof Theoretical Study*. Almquist & Wiksell, 1965.

[92] U.S. Reddy. Objects as Closures: Abstract Semantics of Object-Oriented Languages. In *LISP and Functional Programming*, pages 289–297, 1988.

[93] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[94] Simona Ronchi Della Rocca. Principal Type Scheme and Unification for Intersection Type Discipline. *Theor. Comput. Sci.*, 59:181–209, 1988.

[95] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

[96] Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG–2, Oxford University Computing Laboratory, Oxford, England, November 1970.

[97] Dana S. Scott. Domains for denotational semantics. In *ICALP*, pages 577–613, 1982.

[98] Bjarne Stroustrup. *The C++ programming language (3. ed.)*. Addison-Wesley-Longman, 1997.

[99] Thomas Studer. Constructive foundations for featherweight java. In *Proof Theory in Computer Science*, pages 202–238, 2001.

[100] W. Tait. Intensional interpretation of functionals of finite type i. *Journal of Symbolic Logic*, 32, 2:198–223, 1967.

[101] Steffen van Bakel. Completeness and partial soundness results for intersection and union typing for $\overline{\lambda}\mu\tilde{\mu}$. *Ann. Pure Appl. Logic*, 161(11):1400–1430, 2010.

[102] Steffen van Bakel. Completeness and soundness results for § with intersection and union types. *Fundamenta Informaticae*, 2011. to appear.

[103] G. van Rossum and F.L. Drake, editors. *Python Language Reference*. PythonLabs, 2003.

[104] R. Viswanathan. Full Abstraction for First-Order Objects with Recursive Types and Subtyping. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 380–391. IEEE Computer Society, 1998.

[105] C. Wadsworth. The Relation Between Computational and Denotational Properties for Scott's d_∞-Models of the Lambda-Calculus. *SIAM J. Comput.*, 5:488–521, 1976.

[106] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *ECOOP*, pages 99–117, 2001.

Appendix

A. Type-Based Analysis of Ackermann's Function

In this appendix, we will consider an implementation of the Ackermann function in FJ$^{¢}$, and its typeability using the intersection type system considered in the first part of this thesis. We will show that the implementation is strongly normalising, and conjecture that each level of the parameterized hierarchy is typeable using some finitely bounded level of nested intersection types. The aim of this presentation is to demonstrate the analytical expressiveness of the type systems that we have been considering.

Definition A.1 (Ackermann Function)**.** *The Ackermann function* $\mathsf{Ack} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *is defined as follows:*

$$\mathsf{Ack}(m,n) = \begin{cases} n+1 & \textit{(if } m = 0\textit{)} \\ \mathsf{Ack}(m-1,1) & \textit{(if } m > 0, n = 0\textit{)} \\ \mathsf{Ack}(m-1,\mathit{Ack}(m,n-1)) & \textit{(if } m,n > 0\textit{)} \end{cases}$$

We can also define a *parameterized* version of the Ackermann function, by fixing the first argument:

Definition A.2 (Parameterized Ackermann Function)**.** *For every m, the function* $\mathsf{Ack[m]}$ *is defined by*

$$\mathsf{Ack[m]}(n) = \mathsf{Ack}(m,n)$$

A.1. The Ackermann Function in Featherweight Java

The Ackermann function can be implemented quite straightforwardly in an object-oriented style. We use the same approach as in Section 6.4 of defining a class for zero and a class for successor, with each class containing methods that implement the Ackermann function:

Definition A.3 (Ackermann Program)**.** *The* FJ *program* $\mathsf{Ack}_{\mathrm{FJ}}$ *is defined by the following class table:*

```
class Nat extends Object {
    Nat ackM(Nat n) { return this; }
    Nat ackN(Nat m) { return this; }
}

class Zero extends Nat {
    Nat ackM(Nat n) { return new Suc(n); }
    Nat ackN(Nat m) { return m.ackM(new Suc(new Zero())); }
}

class Suc extends Nat {
    Nat pred;
    Nat ackM(Nat n) { return n.ackN(this.pred) }
    Nat ackN(Nat m) { return m.ackM(new Suc(m).ackM(this.pred)); }
}
```

Natural numbers, as discussed in Section 6.4, have a straightforward encoding using the above FJ¢ program.

Definition A.4 (Translation of Naturals)**.** *The translation function* $⟦\cdot⟧_{\mathbb{N}}$ *maps natural numbers to expressions of* Ack_{FJ}*, and is defined inductively as follows:*

$$⟦0⟧_{\mathbb{N}} = \texttt{new Zero()}$$
$$⟦i+1⟧_{\mathbb{N}} = \texttt{new Suc(}⟦i⟧_{\mathbb{N}}\texttt{)}$$

Notice that for every n, $⟦n⟧_{\mathbb{N}}$ is a *normal form* (this is easily proved by induction on n). The following result shows that the Ackermann program computes the Ackermann function.

Theorem A.5. $\forall m,n\,.\,\exists k\,.\,⟦m⟧_{\mathbb{N}}\texttt{.ackM(}⟦n⟧_{\mathbb{N}}\texttt{)} \to^* ⟦k⟧_{\mathbb{N}}$ *and* $k = \mathsf{Ack}(m,n)$.

Proof. By well-founded induction on the pair (m,n) using the lexicographic ordering $<_{\text{LEX}}$ on natural numbers. Take arbitrary (m,n); then we have the following cases.

($m = 0$): Then $\mathsf{Ack}(0,n) = n+1$, and we have the following reduction sequence:

$$\begin{array}{ll} & ⟦m⟧_{\mathbb{N}}\texttt{.ackM(}⟦n⟧_{\mathbb{N}}\texttt{)} \\ = & \texttt{new Zero().ackM(}⟦n⟧_{\mathbb{N}}\texttt{)} \\ \to & \texttt{new Suc(}⟦n⟧_{\mathbb{N}}\texttt{)} \\ = & ⟦n+1⟧_{\mathbb{N}} \end{array}$$

($m > 0, n = 0$): Then $m = i+1$ for some i and $⟦m⟧_{\mathbb{N}} = \texttt{new Suc(}⟦i⟧_{\mathbb{N}}\texttt{)}$. Notice that $i = m-1$, so $i < m$ and therefore $(i,1) <_{\text{LEX}} (m,n)$. Thus it follows by the inductive hypothesis that there is some k such that $⟦i⟧_{\mathbb{N}}\texttt{.ackM(}⟦1⟧_{\mathbb{N}}\texttt{)} \to^* ⟦k⟧_{\mathbb{N}}$ and $k = \mathsf{Ack}(i,1)$. Notice also that $⟦n⟧_{\mathbb{N}} = ⟦0⟧_{\mathbb{N}} = \texttt{new Zero()}$ and $⟦1⟧_{\mathbb{N}} = \texttt{new Suc(new Zero())}$. Then we have the following reduction sequence:

$$\begin{array}{ll} & ⟦m⟧_{\mathbb{N}}\texttt{.ackM(}⟦n⟧_{\mathbb{N}}\texttt{)} \\ = & \texttt{new Suc(}⟦i⟧_{\mathbb{N}}\texttt{).ackM(new Zero())} \\ \to & \texttt{new Zero().ackN(new Suc(}⟦i⟧_{\mathbb{N}}\texttt{).pred)} \\ \to & \texttt{new Zero().ackN(}⟦i⟧_{\mathbb{N}}\texttt{)} \\ \to & ⟦i⟧_{\mathbb{N}}\texttt{.ackM(new Suc(new Zero()))} \\ \to & ⟦i⟧_{\mathbb{N}}\texttt{.ackM(}⟦1⟧_{\mathbb{N}}\texttt{)} \\ \to^* & ⟦k⟧_{\mathbb{N}} \\ = & ⟦\mathsf{Ack}(i,1)⟧_{\mathbb{N}} \\ = & ⟦\mathsf{Ack}(m-1,1)⟧_{\mathbb{N}} \end{array}$$

($m > 0, n > 0$): Then $m = i+1$ and $n = j+1$ for some i and j. So $j = n-1 < n$, therefore $(m,j) <_{\text{LEX}} (m,n)$ and thus by the inductive hypothesis there is some k_1 such that $⟦m⟧_{\mathbb{N}}\texttt{.ackM(}⟦j⟧_{\mathbb{N}}\texttt{)} \to^* ⟦k_1⟧_{\mathbb{N}}$ and $\mathsf{Ack}(m,j) = k_1$. Also $i = m-1 < m$, therefore $(i,k_1) <_{\text{LEX}} (m,n)$ and so by the inductive hypothesis there is some k_2 such that $k_2 = \mathsf{Ack}(i,k_1)$ and $⟦i⟧_{\mathbb{N}}\texttt{.ackM(}⟦k_1⟧_{\mathbb{N}}\texttt{)} \to^* ⟦k_2⟧_{\mathbb{N}}$. Notice that $⟦m⟧_{\mathbb{N}} =$

$\llbracket i+1 \rrbracket_{\mathbb{N}} = \texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}})$ and $\llbracket n \rrbracket_{\mathbb{N}} = \llbracket j+1 \rrbracket_{\mathbb{N}} = \texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}})$. Then we have the following reduction sequence:

$$
\begin{array}{lll}
 & \llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket n \rrbracket_{\mathbb{N}}) \\
= & \texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}})\texttt{.ackM}(\texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}})) \\
\to & \texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}})\texttt{.ackN}(\texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}})\texttt{.pred}) \\
\to & \texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}})\texttt{.ackN}(\llbracket i \rrbracket_{\mathbb{N}}) \\
\to & \llbracket i \rrbracket_{\mathbb{N}}\texttt{.ackM}(\texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}})\texttt{.ackM}(\texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}})\texttt{.pred})) \\
\to & \llbracket i \rrbracket_{\mathbb{N}}\texttt{.ackM}(\texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}})\texttt{.ackM}(\llbracket j \rrbracket_{\mathbb{N}})) \\
= & \llbracket i \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket j \rrbracket_{\mathbb{N}})) \\
\to^* & \llbracket i \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket k_1 \rrbracket_{\mathbb{N}}) \\
\to^* & \llbracket k_2 \rrbracket_{\mathbb{N}} \\
= & \llbracket \mathsf{Ack}(i, k_1) \rrbracket_{\mathbb{N}} \\
= & \llbracket \mathsf{Ack}(i, \mathsf{Ack}(m, j)) \rrbracket_{\mathbb{N}} \\
= & \llbracket \mathsf{Ack}(m-1, \mathsf{Ack}(m, n-1)) \rrbracket_{\mathbb{N}}
\end{array}
$$

□

Notice that this implies that every instance of the Ackermann program is *normalisable*. In the following section we will show the stronger result that every instance of the Ackermann program is *strongly* normalisable.

A.2. Strong Normalisation of $\mathsf{Ack}_{\mathbf{FJ}}$

Recall that one of the main properties of our intersection type system is that programs typeable with *strong* derivations (Definition 4.8) – i.e. without using the top type ω – are *strongly* normalising (Theorem 5.20). In this section we will show that every instance of the Ack_{FJ} program is strongly normalising by using this result and showing that every instance is typeable with a strong derivation. This follows from two main lemmas: firstly, that strong derivations are preserved by expansion for Ack_{FJ} - that is if $\llbracket k \rrbracket_{\mathbb{N}}$ is typeable using a strong derivation and $\llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket n \rrbracket_{\mathbb{N}}) \to^* \llbracket k \rrbracket_{\mathbb{N}}$ for some m and n, then $\llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket n \rrbracket_{\mathbb{N}})$ is also typeable with a strong derivation; secondly we show that every number $\llbracket k \rrbracket_{\mathbb{N}}$ has at least one strong derivation. Then, using the result from the previous section that every instance of the Ackermann function in Ack_{FJ} reduces to a number $\llbracket k \rrbracket_{\mathbb{N}}$, it immediately follows that every instance is typeable with a strong derivation and is thus strongly normalisable.

We will first need the following lemma, which is an extension to derivations of *type extraction* - Lemma 3.10(2).

Lemma A.6. *Let* $\mathsf{S} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ *be a term substitution and* e *be an expression such that* $\text{VARS}(e) = \{x_1, \ldots, x_n\}$*; if there is a (strong) derivation* $\mathcal{D}$ *such that* $\mathcal{D} :: \Pi \vdash e^{\mathsf{S}} : \phi$*, then there exists another (strong) derivation* $\mathcal{D}'$ *such that* $\mathcal{D}' :: \Pi' \vdash e : \phi$ *with* $\Pi' = \{x_1{:}\phi_1, \ldots, x_n{:}\phi_n\}$*, and a (strong) derivation substitution* $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1 :: \Pi \vdash e_1 : \phi_1, \ldots, x_n \mapsto \mathcal{D}_n :: \Pi \vdash e_n : \phi_n\}$ *such that* $(\mathcal{D}')^{\mathcal{S}} = \mathcal{D}$*.*

Proof. By induction on derivations, similar to Lemma 3.10(2). □

We can now show that strong derivations are preserved by expansion for the Ack_{FJ} program.

Lemma A.7 (Expansion for Strong Derivations)**.** *For all m and n:*

$$\forall \mathcal{D}, \sigma \,.\, \llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} \rightarrow^* \llbracket k \rrbracket_{\mathbb{N}} \textit{ and } \mathcal{D} :: \vdash \llbracket k \rrbracket_{\mathbb{N}} : \sigma \textit{ with } \mathcal{D} \textit{ strong} \Rightarrow \exists \mathcal{D}' \,.\, \vdash \llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} : \sigma \textit{ with } \mathcal{D}' \textit{ strong}$$

Proof. By well-founded induction on (m,n) using the lexicographic ordering $<_{\text{LEX}}$ on natural numbers. It is sufficient to consider the following cases:

($m = 0$): Then $\mathsf{Ack}(m,n) = \mathsf{Ack}(0,n) = n+1$ and so by Theorem A.5 it follows that

$$\texttt{new Zero().ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} \rightarrow^* \llbracket n+1 \rrbracket_{\mathbb{N}}$$

Notice $\llbracket n+1 \rrbracket_{\mathbb{N}} = \texttt{new Suc(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)}$. Assume $\mathcal{D} :: \vdash \texttt{new Suc(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} : \sigma$ with $\mathcal{D}$ strong. Notice that $\texttt{new Suc(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} = \texttt{new Suc(n)}^{\mathsf{S}}$ where $\mathsf{S} = \{\texttt{n} \mapsto \llbracket n \rrbracket_{\mathbb{N}}\}$. Thus, by Lemma A.6, we have that there is a strong derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi' \vdash \texttt{new Suc(n)} : \sigma$ with $\Pi' = \{\texttt{n}{:}\phi\}$ and there is a strong derivation substitution $\mathcal{S} = \{\texttt{n} \mapsto \mathcal{D}''\}$ with $\mathcal{D}'' :: \vdash \llbracket n \rrbracket_{\mathbb{N}} : \phi$. Since $\mathcal{S}$ is strong, so too is $\mathcal{D}''$. Now we can build the following strong derivation:

$$\dfrac{\dfrac{\boxed{\mathcal{D}''[\Pi \trianglelefteq \Pi']}}{\Pi \vdash \texttt{new Suc(n)} : \sigma} \quad \dfrac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\,(\textsc{newO})}{\vdash \texttt{new Zero()} : \langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle}\,(\textsc{newM}) \quad \boxed{\mathcal{D}''}\ \vdash \llbracket n \rrbracket_{\mathbb{N}} : \phi \Big/ \vdash \texttt{new Zero().ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} : \sigma\,(\textsc{invk})$$

where $\Pi = \{\texttt{this}{:}\texttt{Zero}, \texttt{n}{:}\phi\}$.

($m = i+1, n = 0$): Then $\mathsf{Ack}(m,n) = \mathsf{Ack}(i+1,0) = \mathsf{Ack}(i,1)$ and by Theorem A.5 $\llbracket i \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket 1 \rrbracket_{\mathbb{N}}\texttt{)} \rightarrow^* \llbracket k \rrbracket_{\mathbb{N}}$ with $\mathsf{Ack}(i,1) = k$. Thus, also by Theorem A.5 it follows that $\llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} \rightarrow^* \llbracket k \rrbracket_{\mathbb{N}}$. Assume there is a strong derivation $\mathcal{D}$ such that $\mathcal{D} :: \vdash \llbracket k \rrbracket_{\mathbb{N}} : \sigma$. Since $i = m-1 < m$, $(i,1) <_{\text{LEX}} (m,n)$ and so by the inductive hypothesis there is a strong derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \vdash \llbracket i \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket 1 \rrbracket_{\mathbb{N}}\texttt{)} : \sigma$. Then by rule (INVK) there are strong derivations $\mathcal{D}''$ and $\mathcal{D}'''$ such that $\mathcal{D}'' :: \vdash \llbracket i \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle$ and $\mathcal{D}''' :: \vdash \llbracket 1 \rrbracket_{\mathbb{N}} : \phi$. Notice $\llbracket 1 \rrbracket_{\mathbb{N}} = \texttt{new Suc(new Zero())}$, $\llbracket m \rrbracket_{\mathbb{N}} = \llbracket i+1 \rrbracket_{\mathbb{N}} = \texttt{new Suc(}\llbracket i \rrbracket_{\mathbb{N}}\texttt{)}$ and $\llbracket n \rrbracket_{\mathbb{N}} = \llbracket 0 \rrbracket_{\mathbb{N}} = \texttt{new Zero()}$, so we can build the following strong derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi_2 \vdash \texttt{m} : \langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle}\,(\textsc{var}) \quad \dfrac{\boxed{\mathcal{D}'''[\Pi_2 \trianglelefteq \emptyset]}}{\Pi_2 \vdash \texttt{new Suc(new Zero())} : \phi}\,(\textsc{invk})}{\Pi_2 \vdash \texttt{m.ackM(new Suc(new Zero()))} : \sigma} \quad \dfrac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\,(\textsc{newO})}{\vdash \texttt{new Zero()} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle \rightarrow \sigma \rangle}\,(\textsc{newM}) \quad \vdots}{\dfrac{\dfrac{}{\Pi_1 \vdash \texttt{n} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle \rightarrow \sigma \rangle}\,(\textsc{var}) \quad \dfrac{\dfrac{}{\Pi_1 \vdash \texttt{this} : \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle \rangle}\,(\textsc{var})}{\Pi_1 \vdash \texttt{this.pred} : \langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle}\,(\textsc{fld})}{\Pi_1 \vdash \texttt{n.ackN(this.pred)} : \sigma}\,(\textsc{invk}) \quad \vdots}{\vdash \texttt{new Suc(}\llbracket i \rrbracket_{\mathbb{N}}\texttt{)} : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle \rightarrow \sigma \rangle \rightarrow \sigma \rangle}\,(\textsc{newM}) \quad \vdots}{\vdash \texttt{new Suc(}\llbracket i \rrbracket_{\mathbb{N}}\texttt{).ackM(new Zero())} : \sigma}\,(\textsc{invk})$$

$$\dfrac{\boxed{\mathcal{D}''}\ \vdash \llbracket i \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle}{\vdash \texttt{new Suc(}\llbracket i \rrbracket_{\mathbb{N}}\texttt{)} : \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\phi \rightarrow \sigma \rangle \rangle}\,(\textsc{newF})$$

$(m = i+1, n = j+1)$: Then $\mathsf{Ack}(m,n) = \mathsf{Ack}(i,\mathsf{Ack}(m,j))$. By Theorem A.5, $⟦m⟧_{\mathbb{N}}\,.\,\texttt{ackM}(⟦j⟧_{\mathbb{N}}) \to^* ⟦k⟧_{\mathbb{N}}$ with $\mathsf{Ack}(m,j) = k$. Also by Theorem A.5 we have $⟦i⟧_{\mathbb{N}}\,.\,\texttt{ackM}(⟦k⟧_{\mathbb{N}}) \to^* r$ with $\mathsf{Ack}(i,k) = r$. Thus it follows that $⟦m⟧_{\mathbb{N}}\,.\,\texttt{ackM}(⟦n⟧_{\mathbb{N}}) \to^* ⟦r⟧_{\mathbb{N}}$. Assume there is a strong derivation $\mathcal{D}$ such that $\mathcal{D} :: \vdash ⟦r⟧_{\mathbb{N}} : \sigma$. Since $i = m-1 < m$, $(i,k) <_{\text{LEX}} (m,n)$ and so by the inductive hypothesis there is a strong derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \vdash ⟦i⟧_{\mathbb{N}}\,.\,\texttt{ackM}(⟦k⟧_{\mathbb{N}}) : \sigma$. Then, by rule (INVK) it must be that there are strong derivations $\mathcal{D}''$ and $\mathcal{D}'''$ such that $\mathcal{D}'' :: \vdash ⟦i⟧_{\mathbb{N}} : \langle \texttt{ackM}{:}\phi \to \sigma\rangle$ and $\mathcal{D}''' :: \vdash ⟦k⟧_{\mathbb{N}} : \phi$. We can assume without loss of generality that $\phi = \tau_1 \cap \ldots \cap \tau_t$ (for some $t > 0$ since $\mathcal{D}'''$ is strong), then by rule (JOIN) there are strong derivations $\mathcal{D}_1, \ldots, \mathcal{D}_t$ such that $\mathcal{D}_s :: \vdash ⟦k⟧_{\mathbb{N}} : \tau_s$ for each $s \in \overline{t}$.

Now, since $j = n-1 < n$, therefore $(m,j) <_{\text{LEX}} (m,n)$ and by the inductive hypothesis there are strong derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_t$ such that for each $s \in \overline{t}$, $\mathcal{D}'_s :: \vdash ⟦m⟧_{\mathbb{N}}\,.\,\texttt{ackM}(⟦j⟧_{\mathbb{N}}) : \tau_s$. So by rule (INVK) there are strong derivations $\mathcal{D}'_1{}_1, \ldots, \mathcal{D}'_t{}_1$ such that $\mathcal{D}'_s :: \vdash ⟦m⟧_{\mathbb{N}} : \langle \texttt{ackM}{:}\phi'_s \to \tau_s\rangle$ and derivations $\mathcal{D}'''_1, \ldots, \mathcal{D}'''_r$ such that $\mathcal{D}'''_s :: \vdash ⟦j⟧_{\mathbb{N}} : \phi'_s$ for each $s \in \overline{t}$.

We can assume without loss of generality that, for each $s \in \overline{t}$, $\phi'_s = \delta^s_1 \cap \ldots \cap \delta^s_{v_s}$ (with $v_s > 0$ since $\mathcal{D}'''_s$ is strong, and each δ strict). Thus by rule (JOIN) there are strong derivations $\mathcal{D}^{(6,1)}_1$, …, $\mathcal{D}^{(6,1)}_{v_1}$, …, $\mathcal{D}^{(6,t)}_1$, …, $\mathcal{D}^{(6,t)}_{v_t}$ such that $\mathcal{D}^{(6,s)}_u :: \vdash ⟦j⟧_{\mathbb{N}} : \delta^s_u$ for each $s \in \overline{t}, u \in \overline{v_s}$. Let $\mathcal{D}^7$ be the following strong derivation:

$$\dfrac{\dfrac{\overset{\mathcal{D}^{(6,1)}_1}{\vdash ⟦j⟧_{\mathbb{N}} : \delta^1_1}}{\vdash \texttt{new Suc}(⟦j⟧_{\mathbb{N}}) : \langle \texttt{pred}{:}\delta^1_1\rangle}\,(\text{NEWF}) \quad \ldots \quad \dfrac{\overset{\mathcal{D}^{(6,t)}_{v_t}}{\vdash ⟦j⟧_{\mathbb{N}} : \delta^t_{v_t}}}{\vdash \texttt{new Suc}(⟦j⟧_{\mathbb{N}}) : \langle \texttt{pred}{:}\delta^t_{v_t}\rangle}\,(\text{NEWF})}{\vdash \texttt{new Suc}(⟦j⟧_{\mathbb{N}}) : \langle \texttt{pred}{:}\delta^1_1\rangle \cap \ldots \cap \langle \texttt{pred}{:}\delta^t_{v_t}\rangle}\,(\text{JOIN})$$

Let $\Pi' = \{\texttt{this}{:}\langle \texttt{pred}{:}\delta^1_1\rangle \cap \ldots \cap \langle \texttt{pred}{:}\delta^t_{v_t}\rangle\}$ and for each $s \in \overline{t}$ $\mathcal{D}^8_s$ be the following strong derivation:

$$\dfrac{\dfrac{\dfrac{}{\Pi' \vdash \texttt{this} : \langle \texttt{pred}{:}\delta^s_1\rangle}\,(\text{VAR})}{\Pi' \vdash \texttt{this.pred} : \delta^s_1}\,(\text{FLD}) \quad \ldots \quad \dfrac{\dfrac{}{\Pi' \vdash \texttt{this} : \langle \texttt{pred}{:}\delta^s_{v_s}\rangle}\,(\text{VAR})}{\Pi' \vdash \texttt{this.pred} : \delta^s_{v_s}}\,(\text{FLD})}{\Pi' \vdash \texttt{this.pred} : \phi'_s}\,(\text{JOIN})$$

Notice that $⟦m⟧_{\mathbb{N}} = ⟦i+1⟧_{\mathbb{N}} = \texttt{new Suc}(⟦i⟧_{\mathbb{N}}) = \texttt{new Suc(m)}^{\mathsf{S}}$ where $\mathsf{S} = \{\texttt{m} \mapsto ⟦i⟧_{\mathbb{N}}\}$. Thus by Lemma A.6 there are strong derivations $\mathcal{D}^4_1, \ldots, \mathcal{D}^4_t$ and $\mathcal{D}^5_1, \ldots, \mathcal{D}^5_r$ such that $\mathcal{D}^4_s :: \{\texttt{m}{:}\phi''_s\} \vdash \texttt{new Suc(m)} : \langle \texttt{ackM}{:}\phi'_s \to \tau_s\rangle$ and $\mathcal{D}^5_s :: \vdash ⟦i⟧_{\mathbb{N}} : \phi''_s$ for each $s \in \overline{t}$.

We can assume without loss of generality that $\phi''_s = \pi^s_1 \cap \ldots \cap \pi^s_{w_s}$ for each $s \in \overline{t}$ (with $w_s > 0$ since $\mathcal{D}^5_s$ is strong, and each π strict). Thus by rule (JOIN) there are strong derivations $\mathcal{D}^{(9,1)}_1$, …, $\mathcal{D}^{(9,1)}_{w_1}$, …, $\mathcal{D}^{(9,t)}_1$, …, $\mathcal{D}^{(9,t)}_{w_t}$ such that $\mathcal{D}^{(9,s)}_u :: \vdash ⟦i⟧_{\mathbb{N}} : \pi^s_u$ for each $s \in \overline{r}, u \in \overline{w_s}$. Let $\mathcal{D}^{10}$ be the following strong derivation:

$$\dfrac{\dfrac{\overset{\mathcal{D}^{(9,1)}_1}{\vdash ⟦i⟧_{\mathbb{N}} : \pi^1_1}}{\vdash \texttt{new Suc}(⟦i⟧_{\mathbb{N}}) : \langle \texttt{pred}{:}\pi^1_1\rangle}\,(\text{NEWF}) \quad \ldots \quad \dfrac{\overset{\mathcal{D}^{(9,t)}_{w_t}}{\vdash ⟦i⟧_{\mathbb{N}} : \delta^t_{w_t}}}{\vdash \texttt{new Suc}(⟦i⟧_{\mathbb{N}}) : \langle \texttt{pred}{:}\delta^t_{w_t}\rangle}\,(\text{NEWF}) \quad \dfrac{\overset{\mathcal{D}''}{\vdash ⟦i⟧_{\mathbb{N}} : \langle \texttt{ackM}{:}\phi \to \sigma\rangle}}{\vdash \texttt{new Suc}(⟦i⟧_{\mathbb{N}}) : \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\phi \to \sigma\rangle\rangle}\,(\text{NEWF})\ \vdots}{\vdash \texttt{new Suc}(⟦i⟧_{\mathbb{N}}) : \langle \texttt{pred}{:}\delta^1_1\rangle \cap \ldots \cap \langle \texttt{pred}{:}\delta^t_{w_t}\rangle \cap \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\phi \to \sigma\rangle\rangle}\,(\text{JOIN})$$

Let $\Pi'' = \{\texttt{this}{:}\langle \texttt{pred}{:}\pi^1_1\rangle \cap \ldots \cap \langle \texttt{pred}{:}\pi^t_{w_t}\rangle \cap \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\phi \to \sigma\rangle\rangle\}$ and $\mathcal{D}^{11}$ be the follow-

ing strong derivation:

$$\dfrac{\dfrac{}{\Pi'' \vdash \texttt{this} : \langle \texttt{pred} : \pi^1_1 \rangle}\,(\textsc{var})}{\Pi'' \vdash \texttt{this.pred} : \pi^1_1}\,(\textsc{fld}) \quad \ldots \quad \dfrac{\dfrac{}{\Pi'' \vdash \texttt{this} : \langle \texttt{pred} : \pi^t_{w_t} \rangle}\,(\textsc{var})}{\Pi'' \vdash \texttt{this.pred} : \pi^t_{w_t}}\,(\textsc{fld})}{\Pi'' \vdash \texttt{this.pred} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle}\,(\textsc{join})$$

where $\Pi'' = \dfrac{}{\Pi'' \vdash \texttt{this} : \langle \texttt{pred} : \langle \texttt{ackM} : \phi \to \sigma \rangle \rangle}\,(\textsc{var})$ / $\Pi'' \vdash \texttt{this.pred} : \langle \texttt{ackM} : \phi \to \sigma \rangle\,(\textsc{fld})$

We can now build the following strong derivation:

$$\dfrac{\begin{array}{c}\mathcal{D}_{12} \\ \vdash \texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}}) : \langle \texttt{ackM} : \langle \texttt{ackN} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle \to \sigma \rangle \to \sigma \rangle \end{array} \quad \begin{array}{c}\mathcal{D}_{13} \\ \vdash \texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}}) : \langle \texttt{ackN} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle \to \sigma \rangle \end{array}}{\vdash \texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}}).\texttt{ackM}(\texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}})) : \sigma}\,(\textsc{invk})$$

where $\mathcal{D}_{12}$ is the following (strong) derivation:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}^{10} \\ \vdash \texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}}) : \langle \texttt{pred} : \delta^1_1 \rangle \cap \ldots \cap \langle \texttt{pred} : \delta^t_{w_t} \rangle \cap \langle \texttt{pred} : \langle \texttt{ackM} : \phi \to \sigma \rangle \rangle \end{array} \quad \begin{array}{c}\mathcal{D}^{11}[\Pi_1 \trianglelefteq \Pi''] \\ \Pi_1 \vdash \texttt{this.pred} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle \end{array}}{\dfrac{\dfrac{}{\Pi_1 \vdash \texttt{n} : \langle \texttt{ackN} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle \to \sigma \rangle}\,(\textsc{var})}{\Pi_1 \vdash \texttt{n.ackN(this.pred)} : \sigma}\,(\textsc{invk})}\,(\textsc{newM})}{\vdash \texttt{new Suc}(\llbracket i \rrbracket_{\mathbb{N}}) : \langle \texttt{ackM} : \langle \texttt{ackN} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle \to \sigma \rangle \to \sigma \rangle}$$

with $\Pi_1 = \Pi'' \cup \{\texttt{n} : \langle \texttt{ackN} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle \to \sigma \rangle\}$, and $\mathcal{D}_{13}$ is the following (strong) derivation:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\mathcal{D}^{14}_1 \\ \Pi_2 \vdash \texttt{new Suc(m).ackM(this.pred)} : \tau_1 \end{array} \quad \ldots \quad \begin{array}{c}\mathcal{D}^{14}_t \\ \Pi_2 \vdash \texttt{new Suc(m).ackM(this.pred)} : \tau_t \end{array}}{\Pi_2 \vdash \texttt{new Suc(m).ackM(this.pred)} : \phi}\,(\textsc{join}) \quad \dfrac{}{\Pi_2 \vdash \texttt{m} : \langle \texttt{ackM} : \phi \to \sigma \rangle}\,(\textsc{var})}{\Pi_2 \vdash \texttt{m.ackM(new Suc(m).ackM(this.pred))} : \sigma}\,(\textsc{invk})}{\vdash \texttt{new Suc}(\llbracket j \rrbracket_{\mathbb{N}}) : \langle \texttt{ackN} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle \to \sigma \rangle}\,(\textsc{newM})$$

with $\Pi_2 = \Pi' \cup \{\texttt{m} : \phi''_1 \cap \ldots \cap \phi''_t \cap \langle \texttt{ackM} : \phi \to \sigma \rangle\}$, and where each $\mathcal{D}^{14}_i$ ($i \in \bar{t}$) is a derivation of the following form:

$$\dfrac{\begin{array}{c}\mathcal{D}^4_i[\Pi_2 \trianglelefteq \{\texttt{m} : \phi''_i\}] \\ \Pi_2 \vdash \texttt{new Suc(m)} : \langle \texttt{ackM} : \phi'_i \to \tau_i \rangle \end{array} \quad \begin{array}{c}\mathcal{D}^8_i 1[\Pi_2 \trianglelefteq \Pi'] \\ \Pi_2 \vdash \texttt{this.pred} : \phi'_i \end{array}}{\Pi_2 \vdash \texttt{new Suc(m).ackM(this.pred)} : \tau_i}\,(\textsc{invk})$$

□

The final lemma that we need is that all numbers $\llbracket k \rrbracket_{\mathbb{N}}$ are strongly typeable.

Lemma A.8 (Strong Typeability of Numbers)**.** *For all k there exists a strong derivation $\mathcal{D}$ such that $\mathcal{D} :: \vdash \llbracket k \rrbracket_{\mathbb{N}} : \sigma$ for some σ.*

Proof. By induction on k.

($n = 0$): Then $\llbracket n \rrbracket_{\mathbb{N}} = \llbracket 0 \rrbracket_{\mathbb{N}} = \texttt{new Zero()}$ Notice that the following derivation is strong:

$$\frac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\ (\textsc{newO})$$

($n = k+1$): Then $\llbracket n \rrbracket_{\mathbb{N}} = \llbracket k+1 \rrbracket_{\mathbb{N}} = \texttt{new Suc(}\llbracket k \rrbracket_{\mathbb{N}}\texttt{)}$. By the inductive hypothesis there is a strong derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash \llbracket k \rrbracket_{\mathbb{N}} : \sigma$ for some σ. Then we can build the following strong derivation:

$$\frac{\begin{array}{c}\boxed{\mathcal{D}} \\ \vdash \llbracket k \rrbracket_{\mathbb{N}} : \sigma\end{array}}{\vdash \texttt{new Suc(}\llbracket k \rrbracket_{\mathbb{N}}\texttt{)} : \texttt{Suc}}\ (\textsc{newO})$$

□

Theorem A.9 (Strong Normalisation for Ack_{FJ})**.** *For all m and n, $\llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)}$ is strongly normalising.*

Proof. Take arbitrary m and n. By Theorem A.5 there is some k such that $\llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} \to^* \llbracket k \rrbracket_{\mathbb{N}}$. By Lemma A.8 there is a strong derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash \llbracket k \rrbracket_{\mathbb{N}} : \sigma$, and then by lemma A.7 it follows that there is also a strong derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \ \vdash \llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} : \sigma$. Thus, by Theorem 5.20, $\llbracket m \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)}$ is strongly normalising. Since m and n were arbitrary, this holds for all m and n. □

A.3. Typing the Parameterized Ackermann Function

In this section, we consider the typeability of the *parameterized* Ackermann function in various *subsystems* of the intersection type system for FJ. These subsystems are defined by restricting where intersections can occur in the argument position of method predicates (i.e. to the left of the $\to$ type constructor).

Definition A.10 (Rank-based Predicate Hierarchy)**.** *We stratify the set of predicates into an inductively defined hierarchical family based on* rank*. For each n, the set $\mathcal{T}_n$ of rank n predicates is defined as follows:*

$$\mathcal{T}_0 = \varphi \mid \texttt{C} \mid \langle \texttt{f} : \mathcal{T}_0 \rangle \mid \langle \texttt{m} : (\mathcal{T}_0, \ldots, \mathcal{T}_0) \to \mathcal{T}_0 \rangle$$

$$\mathcal{T}_{i+1} = \begin{cases} \mathcal{T}_i \cap \ldots \cap \mathcal{T}_i & (i > 0, i \textit{ even}) \\ \mathcal{T}_{i-1} \mid \langle \texttt{f} : \mathcal{T}_{i+1} \rangle \mid \langle \texttt{m} : (\mathcal{T}_i, \ldots, \mathcal{T}_i) \to \mathcal{T}_{i+1} \rangle & (i > 0, i \textit{ odd}) \end{cases}$$

where φ ranges of predicate variables, $\texttt{C}$ ranges over class names, $\texttt{f}$ ranges over field identifiers, and $\texttt{m}$ ranges over method names.

Definition A.11 (Rank n Typing Derivations)**.** *A derivation $\mathcal{D}$ is called* rank *n if each instance of the typing rules used to in $\mathcal{D}$ contains* only *predicates of rank n.*

The results of this section are that every instance of the $\mathsf{Ack}[0]$ and $\mathsf{Ack}[1]$ parameterized Ackermann functions is typeable in the rank 0 system (essentially corresponding to the simply typed lambda calculus), while every instance of $\mathsf{Ack}[2]$ is typeable in the rank 4 system. This leads us to conjecture that every level of the parameterized Ackermann hierarchy is typeable in some rank-bounded subsystem:

Conjecture A.12 (Rank-Stratified Type Classification of Ack)**.** *For each m, there exists some k such that each instance of* Ack[m] *is typeable using only predicates of rank k, i.e.*

$$\forall m \,.\, \exists k \,.\, \forall n \,.\, \exists \mathcal{D}, \sigma \,.\, \mathcal{D} :: \; \vdash [\![m]\!]_{\mathbb{N}}\,.\,\mathtt{ackM}([\![n]\!]_{\mathbb{N}}) : \sigma \text{ with } \mathcal{D} \text{ rank } k$$

The following family of (rank 0) predicates constitutes the set of predicates that we will be able to assign to instances of the Ackermann function. Since the result of (each instance of) the Ackermann function is a natural number, we call them *ν-predicates.*

Definition A.13 (*ν*-predicates)**.** *The family of ν-predicates is defined inductively as follows:*

$$\begin{aligned} \nu_0 &= \mathsf{Suc} \\ \nu_{i+1} &= \langle \mathtt{ackN}{:}\langle \mathtt{ackM}{:}\nu_i \to \nu_i \rangle \to \nu_i \rangle \end{aligned}$$

The *ν*-predicates will also act as the building blocks for *argument* types: we will later show that to type instances of the Ack function we will have to derive predicates of the form $\langle \mathtt{ackM}{:}\phi \to \nu_j \rangle$ where the predicate ϕ is constructed in terms of *ν*-predicates. The ability of the *ν*-predicates to perform this function hinges on the fact that we can assign *each ν*-predicate to *every* natural number (with the obvious exception that we cannot assign the predicate $\nu_0 = \mathsf{Suc}$ to $[\![0]\!]_{\mathbb{N}}$), a result which we now prove.

We start by showing that if we can assign a *ν*-predicate to a number, then we can assign that same *ν*-predicate to its successor. This result is the crucial element to showing that the whole family of *ν*-predicates are assignable to *each* number.

Lemma A.14. *If* $\mathcal{D} :: \Pi \vdash e : \nu_i$ *with* $\mathcal{D}$ *a rank* 0 *derivation, then there exists a rank* 0 *derivation* $\mathcal{D}'$ *such that* $\mathcal{D}' :: \Pi \vdash \mathtt{new\ Suc}(e) : \nu_i$.

Proof. Assuming $\mathcal{D} :: \Pi \vdash e : \nu_i$ with $\mathcal{D}$ rank 0, then there are two cases to consider:

($i = 0$): Then $\nu_i = \mathsf{Suc}$. The derivation $\mathcal{D}'$ is given below. Notice that since $\mathcal{D}$ is rank 0, so too then is $\mathcal{D}'$.

$$\dfrac{\begin{array}{c}\boxed{\mathcal{D}} \\ \Pi \vdash e : \mathsf{Suc}\end{array}}{\Pi \vdash \mathtt{new\ Suc}(e) : \mathsf{Suc}}\ (\textsc{new}\mathrm{O})$$

($i > 0$): Then $\nu_i = \langle \mathtt{ackN}{:}\langle \mathtt{ackM}{:}\nu_{i-1} \to \nu_{i-1} \rangle \to \nu_{i-1} \rangle$. Since $\mathcal{D}$ is rank 0, it follows that ν_i is also rank 0, and thus so too are $\langle \mathtt{ackM}{:}\nu_{i-1} \to \nu_{i-1} \rangle$ and ν_{i-1}. Therefore, the following derivation $\mathcal{D}'$ is rank 0:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi_2 \vdash \texttt{this} : \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle\rangle}\,(\textsc{var})}{\Pi_2 \vdash \texttt{this.pred} : \langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle}\,(\textsc{fld}) \quad \vdots}{\Pi_2 \vdash \texttt{n} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle \to \nu_{i-1}\rangle}\,(\textsc{var})}{\Pi_2 \vdash \texttt{n.ackN(this.pred)} : \nu_{i-1}}\,(\textsc{invk}) \quad \vdots}{\Pi_1 \vdash \texttt{new Suc(m)} : \langle \texttt{ackM}{:}\nu_i \to \nu_{i-1}\rangle}\,(\textsc{newM}) \quad \vdots}{\Pi_1 \vdash \texttt{m} : \langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle}\,(\textsc{var}) \quad \dfrac{\dfrac{\dfrac{}{\Pi_1 \vdash \texttt{this} : \langle \texttt{pred}{:}\nu_i\rangle}\,(\textsc{var})}{\Pi_1 \vdash \texttt{this.pred} : \nu_i}\,(\textsc{fld}) \quad \vdots}{\Pi_1 \vdash \texttt{new Suc(m).ackM(this.pred)} : \nu_{i-1}}\,(\textsc{invk})}{\Pi_1 \vdash \texttt{m.ackM(new Suc(m).ackM(this.pred))} : \nu_{i-1}}\,(\textsc{invk}) \quad \vdots}{\Pi \vdash \texttt{new Suc(e)} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle \to \nu_{i-1}\rangle}\,(\textsc{newM})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi_2 \vdash \texttt{this} : \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle\rangle}\,(\textsc{var})}{\Pi_2 \vdash \texttt{this.pred} : \langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle}\,(\textsc{fld}) \quad \vdots}{\Pi_2 \vdash \texttt{n} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle \to \nu_{i-1}\rangle}\,(\textsc{var})}{\Pi_2 \vdash \texttt{n.ackN(this.pred)} : \nu_{i-1}}\,(\textsc{invk}) \quad \vdots}{\Pi_1 \vdash \texttt{new Suc(m)} : \langle \texttt{ackM}{:}\nu_i \to \nu_{i-1}\rangle}\,(\textsc{newM}) \quad \vdots}{\Pi_1 \vdash \texttt{m} : \langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle}\,(\textsc{var}) \quad \dfrac{\dfrac{\dfrac{}{\Pi_1 \vdash \texttt{this} : \langle \texttt{pred}{:}\nu_i\rangle}\,(\textsc{var})}{\Pi_1 \vdash \texttt{this.pred} : \nu_i}\,(\textsc{fld}) \quad \vdots}{\Pi_1 \vdash \texttt{new Suc(m).ackM(this.pred)} : \nu_{i-1}}\,(\textsc{invk})}{\Pi_1 \vdash \texttt{m.ackM(new Suc(m).ackM(this.pred))} : \nu_{i-1}}\,(\textsc{invk}) \quad \vdots}{\Pi \vdash \texttt{new Suc(e)} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle \to \nu_{i-1}\rangle}\,(\textsc{newM})$$

where

$$\Pi_1 = \{\texttt{this}{:}\langle \texttt{pred}{:}\nu_i\rangle, \texttt{m}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle\}$$
$$\Pi_2 = \{\texttt{this}{:}\langle \texttt{pred}{:}\langle \texttt{ackM}{:}\nu_{i-1} \to \nu_{i-1}\rangle\rangle, \texttt{n}{:}\nu_i\}$$

□

The predicate ν_0 is the only ν-predicate not assignable to every natural number (it is not assignable to zero). Because of this special case, our result showing the assignability of ν-predicates to natural numbers is formlated as two separate lemmas.

The first states that all ν-predicates *except* ν_0 are assignable to zero. The second states that all ν-predicates are assignable to every positive natural number.

Lemma A.15. $\forall i > 0 \,.\, \exists \mathcal{D} \,.\, \mathcal{D} :: \ \vdash [\![0]\!]_{\mathbb{N}} : \nu_i$ *with* $\mathcal{D}$ *rank 0.*

Proof. By induction on i.

($i = 1$): Then $\nu_i = \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\texttt{Suc} \to \texttt{Suc}\rangle \to \texttt{Suc}\rangle$. Notice that the following derivation is rank 0:

$$\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{m} : \langle \texttt{ackM}{:}\texttt{Suc} \to \texttt{Suc}\rangle}\,(\textsc{var}) \quad \dfrac{\dfrac{}{\Pi \vdash \texttt{new Zero()} : \texttt{Suc}}\,(\textsc{newO})}{\Pi \vdash \texttt{new Suc(new Zero())} : \texttt{Suc}}\,(\textsc{newO})}{\Pi \vdash \texttt{m.ackM(new Suc(new Zero()))} : \texttt{Suc}}\,(\textsc{invk}) \quad \vdots}{\vdash \texttt{new Zero()} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\texttt{Suc} \to \texttt{Suc}\rangle \to \texttt{Suc}\rangle}\,(\textsc{newM})$$

$$\dfrac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\,(\textsc{newO})$$

where $\Pi = \{\texttt{this}{:}\texttt{Zero}, \texttt{m}{:}\langle \texttt{ackM}{:}\texttt{Suc} \to \texttt{Suc}\rangle\}$.

($i = j + 1, j > 0$): Then $\nu_i = \nu_{j+1} = \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_j \to \nu_j\rangle \to \nu_j\rangle$. Notice that $[\![0]\!]_{\mathbb{N}} = \texttt{new Zero()}$ and since $j > 0$, by the inductive hypothesis, there exists a rank 0 derivation $\mathcal{D}$ such that

$$\mathcal{D} :: \ \vdash \texttt{new Zero()} : \nu_j$$

Then by Lemma A.14 there is a rank 0 derivation $\mathcal{D}'$ such that

$$\mathcal{D}' \vdash \texttt{new Suc(new Zero())} : \nu_j$$

Then we can build the following rank 0 derivation:

$$\dfrac{\dfrac{}{\Pi \vdash \texttt{m} : \langle \texttt{ackM}{:}\nu_j \to \nu_j \rangle}\,(\textsc{var}) \quad \boxed{\begin{array}{c}\mathcal{D}'[\Pi \trianglelefteq \emptyset] \\ \Pi \vdash \texttt{new Suc(new Zero())} : \nu_j\end{array}}}{\Pi \vdash \texttt{m.ackM(new Suc(new Zero()))} : \nu_j}\,(\textsc{invk})$$
$$\vdots$$
$$\dfrac{\qquad\qquad\qquad\qquad\qquad\qquad \dfrac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\,(\textsc{newO})}{\vdash \texttt{new Zero()} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_j \to \nu_j \rangle \to \nu_j \rangle}\,(\textsc{newM})$$

where $\Pi = \{\texttt{this}{:}\texttt{Zero}, \texttt{m}{:}\langle \texttt{ackM}{:}\nu_j \to \nu_j \rangle\}$. □

Lemma A.16. *$\forall n > 0 \,.\, \forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} ::\ \vdash \llbracket n \rrbracket_{\mathbb{N}} : \nu_i$ with $\mathcal{D}$ rank 0.*

Proof. By induction on n.

($n = 1$): Then $\llbracket n \rrbracket_{\mathbb{N}} = \llbracket 1 \rrbracket_{\mathbb{N}} = \texttt{new Suc}(\llbracket 0 \rrbracket_{\mathbb{N}}) = \texttt{new Suc(new Zero())}$. Take arbitrary i; there are two cases to consider:

($i = 0$): Then $\nu_i = \nu_0 = Suc$. Notice that the following derivation is rank 0:

$$\dfrac{\dfrac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\,(\textsc{newO})}{\vdash \texttt{new Suc(new Zero())} : \texttt{Suc}}\,(\textsc{newO})$$

($i > 0$): Then since $i > 0$, by Lemma A.15 there is a rank 0 derivation $\mathcal{D}$ such that $\mathcal{D} ::\ \vdash \texttt{new Zero()} : \nu_i$ and then by Lemma A.14 there is another rank 0 derivation $\mathcal{D}'$ such that $\mathcal{D}' ::\ \vdash \texttt{new Suc(new Zero())} : \nu_i$.

($n = k + 1, k > 0$): Take arbitrary i; then since $k > 0$, by the inductive hypothesis there is a rank 0 derivation $\mathcal{D}$ such that $\mathcal{D} ::\ \vdash \llbracket k \rrbracket_{\mathbb{N}} : \nu_i$, and by Lemma A.14 there is another rank 0 derivation $\mathcal{D}'$ such that $\mathcal{D}' ::\ \vdash \texttt{new Suc}(\llbracket k \rrbracket_{\mathbb{N}}) : \nu_i$, that is $\mathcal{D}' ::\ \vdash \llbracket n \rrbracket_{\mathbb{N}} : \nu_i$. □

A.3.1. Rank 0 Typeability of Ack[0]

We can now begin to consider the typeability of some of the different levels of the parameterized Ackermann function. We will start by showing that every instance of the Ack[0] function can be typed using rank 0 derivations.

Lemma A.17. *1. $\exists \mathcal{D} \,.\, \mathcal{D} ::\ \vdash \llbracket 0 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\texttt{Zero} \to \texttt{Suc} \rangle$ with $\mathcal{D}$ rank 0.*

2. $\forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} ::\ \vdash \llbracket 0 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle$ with $\mathcal{D}$ rank 0.

Proof. 1. Notice that the following derivation is rank 0:

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{this}{:}\texttt{Zero}, \texttt{n}{:}\texttt{Zero}\} \vdash \texttt{n} : \texttt{Zero}}\,(\textsc{var})}{\{\texttt{this}{:}\texttt{Zero}, \texttt{n}{:}\texttt{Zero}\} \vdash \texttt{new Suc(n)} : \texttt{Suc}}\,(\textsc{newO}) \quad \dfrac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\,(\textsc{newO})}{\vdash \texttt{new Zero()} : \langle \texttt{ackM}{:}\texttt{Zero} \to \texttt{Suc} \rangle}\,(\textsc{newM})$$

2. Take arbitrary i. Notice that by rule (VAR), we can build the following rank 0 derivation $\mathcal{D}$:

$$\frac{}{\{\texttt{this:Zero}, \texttt{n}:\nu_i\} \vdash \texttt{n} : \nu_i}\ (\textsc{var})$$

Thus, by Lemma A.14 there is a rank 0 derivation $\mathcal{D}'$ such that

$$\mathcal{D}' :: \{\texttt{this:Zero}, \texttt{n}:\nu_i\} \vdash \texttt{new Suc(n)} : \nu_i$$

Then we can build the following rank 0 derivation:

$$\frac{\dfrac{\boxed{\mathcal{D}'}}{\{\texttt{this:Zero}, \texttt{n}:\nu_i\} \vdash \texttt{new Suc(n)} : \nu_i} \quad \frac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}}{\vdash \texttt{new Zero()} : \langle \texttt{ackM}: \nu_i \to \nu_i \rangle}\ (\textsc{newM})$$

□

Theorem A.18 (Rank 0 Typeability of Ack[0])**.** *Every ν-predicate may be assigned to* each *instance of the* Ack[0] *function using a rank 0 derivation, i.e.*

$$\forall n \,.\, \forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} :: \ \vdash [\![0]\!]_{\mathbb{N}} \texttt{.ackM(}[\![n]\!]_{\mathbb{N}}\texttt{)} : \nu_i \text{ with } \mathcal{D} \text{ rank 0}$$

Proof. Take arbitrary n and i. Then it is sufficient to consider the following cases:

($n = 0, i = 0$): Then $[\![n]\!]_{\mathbb{N}} = \texttt{new Zero()}$ and $\nu_i = \texttt{Suc}$. By Lemma A.17(1) there is a rank 0 derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash \texttt{new Zero()} : \langle \texttt{ackM:Zero} \to \texttt{Suc} \rangle$. Then we can build the following rank 0 derivation:

$$\frac{\dfrac{\boxed{\mathcal{D}}}{\vdash \texttt{new Zero()} : \langle \texttt{ackM:Zero} \to \texttt{Suc} \rangle} \quad \frac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\ (\textsc{newO})}{\vdash \texttt{new Zero().ackM(new Zero())} : \texttt{Suc}}\ (\textsc{invk})$$

($n = 0, i > 0$): By Lemma A.17(2) there is a rank 0 derivation $\mathcal{D}_1$ such that $\mathcal{D}_1 :: \ \vdash [\![0]\!]_{\mathbb{N}} : \langle \texttt{ackM}: \nu_i \to \nu_i \rangle$. Since $i > 0$, by Lemma A.15 there is a rank 0 derivation $\mathcal{D}_2$ such that $\mathcal{D}_2 :: \ \vdash [\![0]\!]_{\mathbb{N}} : \nu_i$. Then we can build the following rank 0 derivation:

$$\frac{\dfrac{\boxed{\mathcal{D}_1}}{\vdash [\![0]\!]_{\mathbb{N}} : \langle \texttt{ackM}: \nu_i \to \nu_i \rangle} \quad \frac{\boxed{\mathcal{D}_2}}{\vdash [\![0]\!]_{\mathbb{N}} : \nu_i}}{\vdash [\![0]\!]_{\mathbb{N}} \texttt{.ackM(}[\![0]\!]_{\mathbb{N}}\texttt{)} : \nu_i}\ (\textsc{invk})$$

($n > 0$): By Lemma A.17(2) there is a rank 0 derivation $\mathcal{D}_1$ such that $\mathcal{D}_1 :: \ \vdash [\![0]\!]_{\mathbb{N}} : \langle \texttt{ackM}: \nu_i \to \nu_i \rangle$. Since $n > 0$, by Lemma A.16 there is a rank 0 derivation $\mathcal{D}_2$ such that $\mathcal{D}_2 :: \ \vdash [\![n]\!]_{\mathbb{N}} : \nu_i$. Then we can build the following rank 0 derivation:

$$\frac{\dfrac{\boxed{\mathcal{D}_1}}{\vdash [\![0]\!]_{\mathbb{N}} : \langle \texttt{ackM}: \nu_i \to \nu_i \rangle} \quad \frac{\boxed{\mathcal{D}_2}}{\vdash [\![n]\!]_{\mathbb{N}} : \nu_i}}{\vdash [\![0]\!]_{\mathbb{N}} \texttt{.ackM(}[\![n]\!]_{\mathbb{N}}\texttt{)} : \nu_i}\ (\textsc{invk})$$

□

A.3.2. Rank 0 Typeability of Ack[1]

Showing the rank 0 typeability of the Ack[1] function is similar, with the difference that we must derive a slightly different predicate for invoking the ackM method.

Lemma A.19. $\forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} :: \ \vdash [\![1]\!]_{\mathbb{N}} : \langle \texttt{ackM}: \nu_{i+1} \to \nu_i \rangle$ *with $\mathcal{D}$ rank 0.*

Proof. Take arbitrary i. Notice that by Lemma A.17(2) there is a rank 0 derivation $\mathcal{D}$ such that $\mathcal{D} ::\ \vdash \texttt{new Zero()} : \langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle$. Then we can build the following rank 0 derivation:

$$\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this} : \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle\rangle}\,(\textsc{var})}{\Pi \vdash \texttt{this.pred} : \langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle}\,(\textsc{fld}) \quad \vdots}{\dfrac{\Pi \vdash \texttt{n} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle \to \nu_i \rangle}{\Pi \vdash \texttt{n.ackN(this.pred)} : \nu_i}\,(\textsc{var}) \quad \vdots} \qquad \dfrac{\dfrac{\overbrace{\quad}^{\mathcal{D}}}{\vdash \texttt{new Zero()} : \langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle}}{\vdash \texttt{new Suc(new Zero())} : \langle \texttt{pred}{:}\langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle\rangle}\,(\textsc{newF})}{\vdash \texttt{new Suc(new Zero())} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle}\,(\textsc{newM})$$

where $\Pi = \{\texttt{this}{:}\langle \texttt{pred}{:}\langle \texttt{ackM}{:}\nu_i \to \nu_i \rangle\rangle, \texttt{n}{:}\nu_{i+1}\}$. □

Theorem A.20 (Rank 0 Typeability of Ack[1])**.** *Every ν-predicate may be assigned to* each *instance of the* Ack[1] *function using a rank 0 derivation, i.e.*

$$\forall n\,.\,\forall i\,.\,\exists \mathcal{D}\,.\,\mathcal{D} ::\ \vdash \llbracket 1 \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} : \nu_i \textit{ with } \mathcal{D} \textit{ rank 0}$$

Proof. Take arbitrary n and i. It is sufficient to consider the following two cases:

($n = 0$): By Lemma A.19 there is a rank 0 derivation $\mathcal{D}_1$ such that $\mathcal{D}_1 ::\ \vdash \llbracket 1 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle$. Notice that $i + 1 > 0$ and so by Lemma A.15, there is a rank 0 derivation $\mathcal{D}_2$ such that $\mathcal{D}_2 ::\ \vdash \llbracket 0 \rrbracket_{\mathbb{N}} : \nu_{i+1}$. Then we can build the following rank 0 derivation:

$$\dfrac{\overbrace{\quad}^{\mathcal{D}_1} \quad \overbrace{\quad}^{\mathcal{D}_2}}{\vdash \llbracket 1 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle \quad \vdash \llbracket 0 \rrbracket_{\mathbb{N}} : \nu_{i+1}}\,(\textsc{invk})$$
$$\vdash \llbracket 1 \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket 0 \rrbracket_{\mathbb{N}}\texttt{)} : \nu_i$$

($n > 0$): By Lemma A.19 there is a rank 0 derivation $\mathcal{D}_1$ such that $\mathcal{D}_1 ::\ \vdash \llbracket 1 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle$. By Lemma A.16, there is a rank 0 derivation $\mathcal{D}_2$ such that $\mathcal{D}_2 ::\ \vdash \llbracket n \rrbracket_{\mathbb{N}} : \nu_{i+1}$. Then we can build the following rank 0 derivation:

$$\dfrac{\overbrace{\quad}^{\mathcal{D}_1} \quad \overbrace{\quad}^{\mathcal{D}_2}}{\vdash \llbracket 1 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle \quad \vdash \llbracket n \rrbracket_{\mathbb{N}} : \nu_{i+1}}\,(\textsc{invk})$$
$$\vdash \llbracket 1 \rrbracket_{\mathbb{N}}\texttt{.ackM(}\llbracket n \rrbracket_{\mathbb{N}}\texttt{)} : \nu_i$$

□

A.3.3. Rank 4 Typeability of Ack[2]

In giving a bound on the rank of derivations typing the Ack[0] and Ack[1] functions, the argument predicates were simple the ν-predicates themselves. To give a bound on the rank of derivations assigning ν-predicates to instances of the Ack[2] function, we must design more complex argument predicates. We must also expand the proof technique a little compared to the previous cases of Ack[0] and Ack[1]: for each ν_i we now cannot show that there is a *single* predicate $\langle \texttt{ackM}{:}\sigma \to \nu_i \rangle$ assignable to $\llbracket 2 \rrbracket_{\mathbb{N}}$ such that each possible argument $\llbracket n \rrbracket_{\mathbb{N}}$ has the type σ. Instead, for each i we must now build a *family* of n predicates $\langle \texttt{ackM}{:}\tau_{(n,i)} \to \nu_i \rangle$, each of which can be assigned to $\llbracket 2 \rrbracket_{\mathbb{N}}$, and show additionally that each number $\llbracket n \rrbracket_{\mathbb{N}}$ can be assigned the argument predicate $\tau_{(n,i)}$ for *every* i. Thus, the proof technique is a sort

of '2-D' analogue of the '1-D' technique used previously. Additionally, the predicates that we must now define contain *intersections*.

Definition A.21 (μ-Predicates)**.** *The set of rank 1 μ-*predicates *is defined inductively as follows:*

$$\mu_{(0,j)} = \langle \texttt{ackM}{:}\nu_{j+1} \to \nu_j \rangle \qquad \text{for all } j \geq 0$$
$$\mu_{(i+1,j)} = \langle \texttt{ackM}{:}\nu_{i+j+2} \to \nu_{i+j+1} \rangle \cap \mu_{(i,j)}$$

Lemma A.22. $\mu_{(i,j+1)} \cap \langle \texttt{ackM}{:}\nu_{j+1} \to \nu_j \rangle = \mu_{(i+1,j)}$.

Proof. By induction on i.

($i = 0$): $\mu_{(0,j+1)} \cap \langle \texttt{ackM}{:}\nu_{j+1} \to \nu_j \rangle = \langle \texttt{ackM}{:}\nu_{j+2} \to \nu_{j+1} \rangle \cap \langle \texttt{ackM}{:}\nu_{j+1} \to \nu_j \rangle$ (Def. A.21)
$= \langle \texttt{ackM}{:}\nu_{j+2} \to \nu_{j+1} \rangle \cap \mu_{(0,j)}$ (Def. A.21)
$= \mu_{(i+1,j)}$ (Def. A.21)

($i = k+1$): $\mu_{(i,j+1)} \cap \langle \texttt{ackM}{:}\nu_{j+1} \to \nu_j \rangle$
$= \mu_{(k+1,j+1)} \cap \langle \texttt{ackM}{:}\nu_{j+1} \to \nu_j \rangle$ ($i = k+1$)
$= \langle \texttt{ackM}{:}\nu_{k+(j+1)+2} \to \nu_{k+(j+1)+1} \rangle \cap \mu_{(k,j+1)} \cap \langle \texttt{ackM}{:}\nu_{j+1} \to \nu_j \rangle$ (Def. A.21)
$= \langle \texttt{ackM}{:}\nu_{k+(j+1)+2} \to \nu_{k+(j+1)+1} \rangle \cap \mu_{(k+1,j)}$ (Ind. Hyp.)
$= \langle \texttt{ackM}{:}\nu_{(k+1)+j+2} \to \nu_{(k+1)+j+1} \rangle \cap \mu_{(k+1,j)}$ (arith.)
$= \langle \texttt{ackM}{:}\nu_{i+j+2} \to \nu_{i+j+1} \rangle \cap \mu_{(i,j)}$ ($i = k+1$)
$= \mu_{(i+1,j)}$ (Def. A.21) □

Lemma A.23. *Let $\mu_{(i,j)} = \sigma_1 \cap \ldots \cap \sigma_n$ for some $n > 0$; if there are rank 0 derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ such that $\mathcal{D}_k :: \Pi \vdash e : \sigma_k$ for each $k \in \overline{n}$, then there is a rank 4 derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash \texttt{new Suc}(e) : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\mu_{(i,j)} \to \nu_m \rangle \to \nu_m \rangle$ for any m.*

Proof.

$$\dfrac{\dfrac{\dfrac{}{\Pi \vdash e : \sigma_1}\,(\textsc{var})}{\Pi \vdash \texttt{new Suc}(e) : \langle \texttt{pred}{:}\sigma_1 \rangle}\,(\textsc{newF}) \quad \ldots \quad \dfrac{\dfrac{}{\Pi \vdash e : \sigma_n}\,(\textsc{var})}{\Pi \vdash \texttt{new Suc}(e) : \langle \texttt{pred}{:}\sigma_n \rangle}\,(\textsc{newF})}{\Pi \vdash \texttt{new Suc}(e) : \langle \texttt{pred}{:}\sigma_1 \rangle \cap \ldots \cap \langle \texttt{pred}{:}\sigma_n \rangle}\,(\textsc{join})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi' \vdash \texttt{this} : \langle \texttt{pred}{:}\sigma_1 \rangle}\,(\textsc{var})}{\Pi' \vdash \texttt{this.pred} : \sigma_1}\,(\textsc{fld}) \quad \ldots \quad \dfrac{\dfrac{}{\Pi' \vdash \texttt{this} : \langle \texttt{pred}{:}\sigma_n \rangle}\,(\textsc{var})}{\Pi' \vdash \texttt{this.pred} : \sigma_n}\,(\textsc{fld})}{\Pi' \vdash \texttt{this.pred} : \sigma_1 \cap \ldots \cap \sigma_n}\,(\textsc{join}) \quad \vdots}{\Pi' \vdash \texttt{n.ackN(this.pred)} : \nu_m}\,(\textsc{invk})$$

$$\dfrac{\vdots}{\Pi \vdash \texttt{new Suc}(e) : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\mu_{(i,j)} \to \nu_m \rangle \to \nu_m \rangle}\,(\textsc{newM})$$

where $\Pi' = \{\texttt{this}{:}\langle \texttt{pred}{:}\sigma_1 \rangle \cap \ldots \cap \langle \texttt{pred}{:}\sigma_n \rangle, \texttt{n}{:}\langle \texttt{ackN}{:}\mu_{(i,j)} \to \nu_m \rangle\}$. □

Lemma A.24. $\forall n \,.\, \forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} :: \vdash [\![1]\!]_{\mathbb{N}} : \mu_{(n,i)}$ *with $\mathcal{D}$ rank 1.*

Proof. By induction on n.

($n = 0$): Take arbitrary i; then $\mu_{(n,i)} = \mu_{(0,i)} = \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle$. By Lemma A.19 there is a rank 0 derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash [\![1]\!]_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle$. Since $\mathcal{D}$ is rank 0, it is also rank 1, and since i was arbitrary, this holds for all i.

($n = k+1$): Take arbitrary i; then $\mu_{(n,i)} = \mu_{(k+1,i)} = \langle ackM : \nu_{k+i+2} \to \nu_{k+i+1} \rangle \cap \mu_{(k,1)}$. By Lemma A.19 there is a rank 0 derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash [\![1]\!]_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{k+i+2} \to \nu_{k+i+1} \rangle$. Also, by the inductive hypothesis there is a rank 1 derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \ \vdash [\![1]\!]_{\mathbb{N}} : \mu_{(k,i)}$. Without loss of generality we can assume that $\mu_{(k,i)} = \sigma_1 \cap \ldots \cap \sigma_m$ for some $m > 0$ (since $\mathcal{D}'$ is strong). Then by rule (JOIN) it follows that there are rank 0 derivations $\mathcal{D}_1, \ldots, \mathcal{D}_m$ such that $\mathcal{D}_j :: \ \vdash [\![1]\!]_{\mathbb{N}} : \sigma_j$ for each $j \in \overline{m}$. Then, we can build the following rank 1 derivation:

$$\dfrac{\begin{array}{cccc} \boxed{\mathcal{D}} & \boxed{\mathcal{D}_1} & \cdots & \boxed{\mathcal{D}_m} \\ \vdash [\![1]\!]_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{k+i+2} \to \nu_{k+i+1} \rangle & \vdash [\![1]\!]_{\mathbb{N}} : \sigma_1 & \cdots & \vdash [\![1]\!]_{\mathbb{N}} : \sigma_m \end{array}}{\vdash [\![1]\!]_{\mathbb{N}} : \langle \texttt{ackM}{:}\nu_{k+i+2} \to \nu_{k+i+1} \rangle \cap \sigma_1 \cap \ldots \cap \sigma_m}\ (\textsc{join})$$

Since i was arbitrary, this holds for all i. □

Lemma A.25. *$\forall n \,.\, \forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} :: \ \vdash [\![2]\!]_{\mathbb{N}} : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\mu_{(n,i)} \to \nu_i \rangle \to \nu_i \rangle$ with $\mathcal{D}$ rank 4.*

Proof. Take arbitrary n and i. By Lemma A.24 there is a rank 1 derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash [\![1]\!]_{\mathbb{N}} : \mu_{(n,i)}$. Without loss of generality we can assume that $\mu_{(n,i)} = \sigma_1 \cap \ldots \cap \sigma_m$ for some $m > 0$ (since $\mathcal{D}$ is strong) with each σ_j strict. Thus by rule (JOIN) there are rank 0 derivations $\mathcal{D}_1, \ldots, \mathcal{D}_m$ such that $\mathcal{D}_j :: \ \vdash [\![1]\!]_{\mathbb{N}} : \sigma_j$ for each $j \in \overline{m}$. Then by Lemma A.23 there is a rank 4 derivation $\mathcal{D}'$ such that

$$\mathcal{D}' :: \ \vdash \texttt{new Suc(}[\![1]\!]_{\mathbb{N}}\texttt{)} : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\mu_{(n,i)} \to \nu_i \rangle \to \nu_i \rangle$$

Since n and i were arbitrary, such a derivation exists for all n and i. □

Lemma A.26. *$\forall n \,.\, \forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} :: \ \vdash [\![n]\!]_{\mathbb{N}} : \langle \texttt{ackN}{:}\mu_{(n,i)} \to \nu_i \rangle$ with $\mathcal{D}$ rank 4.*

Proof. By induction on n.

($n = 0$): Take arbitrary i; then $\mu_{(n,i)} = \mu_{(0,i)} = \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle$. By Lemma A.16 there is a rank 0 derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash [\![1]\!]_{\mathbb{N}} : \nu_{i+1}$. Notice that $[\![1]\!]_{\mathbb{N}} = \texttt{new Suc(new Zero())}$. Notice also that $\mu_{(0,i)}$ is a rank 1 predicate, and so the following derivation is rank 2 (and therefore also rank 4):

$$\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{m} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle}\ (\textsc{var}) \quad \boxed{\begin{array}{c}\mathcal{D}[\Pi \trianglelefteq \emptyset] \\ \Pi \vdash \texttt{new Suc(new Zero())} : \nu_{i+1}\end{array}}}{\Pi \vdash \texttt{m.ackM(new Suc(new Zero()))} : \nu_i}\ (\textsc{invk}) \quad \dfrac{}{\vdash \texttt{new Zero()} : \texttt{Zero}}\ (\textsc{newO})}{\vdash \texttt{new Zero()} : \langle \texttt{ackN}{:}\langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle \to \nu_i \rangle}\ (\textsc{newM})$$

where $\Pi = \{\texttt{this:Zero}, \texttt{m:}\langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle\}$. Since i was arbitrary, we can build such a derivation for all i.

($n = k+1$): Take arbitrary i; then by the inductive hypothesis there is a rank 2 derivation $\mathcal{D}$ such that $\mathcal{D} :: \ \vdash [\![k]\!]_{\mathbb{N}} : \langle \texttt{ackN}{:}\mu_{(k,i+1)} \to \nu_{i+1} \rangle$. By Lemma A.22,

$$\mu_{(n,i)} = \mu_{(k+1,i)} = \mu_{(k,i+1)} \cap \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i \rangle$$

Notice that $\llbracket n \rrbracket_{\mathbb{N}} = \llbracket k+1 \rrbracket_{\mathbb{N}} = \texttt{new Suc}(\llbracket k \rrbracket_{\mathbb{N}})$. We can also assume without loss of generality that $\mu_{(k,i+1)} = \sigma_1 \cap \ldots \cap \sigma_m$ for some m, with each σ_j strict. Let

$$\Pi = \{\texttt{this}{:}\langle \texttt{pred}{:}\langle \texttt{ackN}{:}\mu_{(k,i+1)} \to \nu_{i+1}\rangle, \texttt{m}{:}\mu_{(k,i+1)} \cap \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i\rangle\}$$

Then notice that by rule (VAR) we can derive $\Pi \vdash \texttt{m} : \sigma_j$ for each $j \in \overline{m}$. Thus, by Lemma A.23 there is a rank 4 derivation $\mathcal{D}'$ such that

$$\mathcal{D}' :: \Pi \vdash \texttt{new Suc(m)} : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\mu_{(k,i+1)} \to \nu_{i+1}\rangle \to \nu_{i+1}\rangle$$

Then we can then build the following rank 4 derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{m} : \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i\rangle}\,(\textsc{var}) \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this} : \langle \texttt{pred}{:}\langle \texttt{ackN}{:}\mu_{(k,i+1)} \to \nu_{i+1}\rangle\rangle}\,(\textsc{var})}{\Pi \vdash \texttt{this.pred} : \langle \texttt{ackN}{:}\mu_{(k,i+1)} \to \nu_{i+1}\rangle}\,(\textsc{fld})}{\Pi \vdash \texttt{new Suc(m).ackM(this.pred)} : \nu_{i+1}}\,(\textsc{invk})}{\Pi \vdash \texttt{m.ackM(new Suc(m).ackM(this.pred))} : \nu_i}\,(\textsc{invk})}{\vdots}}{\vdash \texttt{new Suc}(\llbracket k \rrbracket_{\mathbb{N}}) : \langle \texttt{ackN}{:}\mu_{(k,i+1)} \cap \langle \texttt{ackM}{:}\nu_{i+1} \to \nu_i\rangle \to \nu_i\rangle}\,(\textsc{newM})}$$

Since i was arbitrary, such a derivation exists for all i. □

Theorem A.27 (Rank 4 Typeability of Ack[2])**.** *Every ν-predicate may be assigned to* each *instance of the* Ack[2] *function using a rank 4 derivation, i.e.*

$$\forall n \,.\, \forall i \,.\, \exists \mathcal{D} \,.\, \mathcal{D} :: \ \vdash \llbracket 2 \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket n \rrbracket_{\mathbb{N}}) : \nu_i \textit{ with } \mathcal{D} \textit{ rank 4.}$$

Proof. Take arbitrary n and i. By Lemma A.25 there is a rank 4 derivation $\mathcal{D}_1$ such that $\mathcal{D}_1 :: \ \vdash \llbracket 2 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\mu_{(n,i)} \to \nu_i\rangle \to \nu_i\rangle$. By Lemma A.26 there exists a rank 4 derivation $\mathcal{D}_2$ such that $\mathcal{D}_2 :: \ \vdash \llbracket n \rrbracket_{\mathbb{N}} : \langle \texttt{ackN}{:}\mu_{(n,i)} \to \nu_i\rangle$. Then we can build the following rank 4 derivation:

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdash \llbracket 2 \rrbracket_{\mathbb{N}} : \langle \texttt{ackM}{:}\langle \texttt{ackN}{:}\mu_{(n,i)} \to \nu_i\rangle \to \nu_i\rangle & \vdash \llbracket n \rrbracket_{\mathbb{N}} : \langle \texttt{ackN}{:}\mu_{(n,i)} \to \nu_i\rangle\end{array}}{\vdash \llbracket 2 \rrbracket_{\mathbb{N}}\texttt{.ackM}(\llbracket n \rrbracket_{\mathbb{N}}) : \nu_i}\,(\textsc{invk})$$

Since n and i were arbitrary, this holds for all n and i. □