Coercions in Hindley-Milner Systems*

Robert Kießling and Zhaohui Luo

Department of Computer Science, University of Durham

Abstract. Coercive subtyping is a theory of abbreviation for dependent type theories. In this paper, we incorporate the idea of coercive subtyping into the traditional Hindley-Milner type systems in functional programming languages. This results in a typing system with coercions, an extension of the Hindley-Milner type system. A type inference algorithm is developed and shown to be sound and complete with respect to the typing system. A notion of derivational coherence is developed to deal with the problem of ambiguity and the corresponding type inference algorithm is shown to be sound and complete.

1 Introduction

The Hindley-Milner type system (HM system for short) [8] is the standard core of the modern typed functional programming languages. Various extensions to the HM system have been proposed in order to enrich a programming language with new and more powerful features. These include, for example, Haskell's class mechanism [10], which provides convenient overloading facilities among other things.

Coercive subtyping [14] is a theory of abbreviation developed in the setting of dependent type theories, where coercions are regarded as abbreviation mechanisms and directly characterised in the proof system (type theory) extended with coercions. It has been implemented in several proof development systems [1, 19, 4] and effectively used in proof development (e.g., [1]).

In this paper, we incorporate the idea of coercive subtyping into the traditional HM type system. There are several motivations in studying the possible combination of coercive subtyping and traditional polymorphic typing systems. First, it leads to a novel approach that increases the power of the HM system with new abbreviation mechanisms, which we believe would be useful in various programming activities. Secondly, coercive subtyping provides a clean and simple theory for abbreviation in dependent type theories. Incorporating its ideas into traditional type systems may lead to simple theoretical development and better understanding of the more powerful facilities (e.g., overloading) found useful in programming. Thirdly, not the least important, studying coercions in polymorphic type systems meets with new challenges, partly because type uniqueness simply does not hold in a polymorphic system.

* This work is partly supported by the UK EPSRC grants GR/M75518, GR/R84092 and GR/R72259, the EU TYPES grant 21900 on the TYPES project and by an EPSRC studentship.

One of the results of our work is a typing system with coercions, an extension of the HM type system, together with a sound and complete type inference algorithm. Since the HM system is polymorphic, where a term may have more than one type, the introduction of coercions has to be very careful; a naive way to introduce coercions causes problems. For example, one of the decisions we have made is that if a term is already typable in the original HM system, then no coercions will be inserted. This also conforms with the intuition and, in practice, an implementation of the extended system will not alter the meanings of the existing programs.

We shall also study a notion of derivational coherence that is developed to deal with the problem of ambiguity of computational meanings of a term. A term may have different completions – there may be different ways to insert coercions to make a term typable. The notion of derivational coherence captures this and we have developed a sound and complete type inference algorithm for derivationally coherent terms.

We regard the Hindley-Milner system as well-known and refer its introduction for example to [21]. In the remainder of this section, we give a summary of work on coercive subtyping and other related work. In Section 2, we give a brief introduction to our approach by considering several simple examples. The extended typing system with coercions is presented and explained formally in Section 3. In Section 4, the type inference algorithm is presented and proved to be sound and complete. Derivational coherence is introduced in Section 5, where we also give the corresponding algorithm and discuss the proofs of its soundness and completeness. We conclude with some discussions about future work.

1.1 Coercive subtyping

Coercive subtyping is a framework of abbreviation for dependent type theories [14]. The basic idea is: if there is a coercion c from A to B, then an object of type A may be regarded as an object of type B via c in appropriate contexts. More precisely, a functional operation f with domain B can be applied to any object a of A and the application fa is definitionally equal to $f(ca)$. Intuitively, we can view f as a context which requires an object of B; then the argument a in the context f stands for its image of the coercion, ca. Therefore, the term fa, originally not well-typed, becomes well-typed and "abbreviates" $f(ca)$.

The second author and his colleagues have studied the above simple idea in the Logical Framework (and type theory), resulting in a very powerful theory of abbreviation and inheritance, including parameterised coercions and coercions between parameterised inductive types. In coercive subtyping, the coercion mechanism is directly characterised in the type theory proof-theoretically. Some important meta-theoretic aspects of coercive subtyping such as the results on conservativity, coherence, and transitivity elimination have been studied. They not only justify the adequacy of the theory from the proof-theoretic consideration, but provide the basis for implementation of coercive subtyping. See [1, 4, 13, 14, 20] for details of some of these development and applications of coercive subtyping.

Coercion mechanisms with certain restrictions have been implemented for dependent type theory both in the proof development system Lego [15] and Coq [2], by Bailey [1] and Saïbi [19], respectively. Callaghan of the Computer Assisted Reasoning Group at Durham has implemented Plastic [4], a proof assistant that supports the Logical Framework LF and coercive subtyping with a mixture of simple coercions, parameterised coercions, coercion rules for parameterised inductive types, and dependent coercions.

Remark 1. Incorporating the idea of coercive subtyping to a polymorphic calculus is not straightforward. Coercive subtyping has been developed in dependent type theories with inductive data types, which are rather sophisticated systems. However, most of them (or at least the standard ones) have the property of type uniqueness; that is every well-typed object has a unique type up to computational equality. Compared with the polymorphic calculi such as the HM type system where an object may have more than one type, one may say that dependent type theories are 'simpler'. It is important to bear this in mind when we consider combining coercive subtyping with a polymorphic calculus.

1.2 Modelling subtyping by coercions

Various notions of coercion have been studied in the literature, particularly when subtyping systems are considered. In subtyping, we have the subsumption rule, which says that if $a : A$ and $A \leq B$, then $a : B$. This can be modelled by means of coercions (maps from A to B). In [3], this idea was proposed and used to give a coercion-based semantic interpretation of Cardelli and Wegner's system Fun [5]. The idea of coercive subtyping discussed above was influenced by this work.

People have used the term coercion to interpret subtyping simpler settings as well. For example, Mitchell [17, 18] considers a system where conceptually a subtype is a subset and thus coercions essentially represent set inclusions. In [6, 12], the term coercion is used to denote a special restricted form of mapping in modelling and explaining subtyping.

Remark 2. Note that, because of the subsumption rule in subtyping, a term obtains more types, while in our setting, a term does not get more types. Rather, in coercive subtyping or the extended HM-system considered in this paper, where there is no subsumption rule, there are more well-typed terms, which are regarded as abbreviations, and typing conflicts are resolved by the insertion of coercions. Furthermore, this is studied in the typing system at the proof-theoretic level.

2 Some simple examples

We consider the HM type system extended with coercions. Coercions are regarded as abbreviations; more precisely, if a term is not well-typed in the original HM type system, and after inserting coercions it becomes well-typed, then

we regard the term to be well-typed and "abbreviate" the completed term with appropriate coercions inserted.

We shall consider extending the HM type system with two forms of coercions: argument coercions and function coercions. By argument coercions, we mean that the argument of a function is coerced according to the typing requirement; more precisely, the term fa abbreviates $f(ca)$ if $f : \sigma \to \tau$, $a : \sigma_0$, and there is a coercion c from σ_0 to σ. By function coercion, we mean that a term in a function position is coerced into an appropriate function accordingly; more precisely, ka abbreviates $(ck)a$ if $k : \sigma$, $a : \sigma_0$, and there is a coercion from σ to a function type $\sigma_0 \to \tau$.

In the following, we give some simple examples to explain the above basic idea. The first two examples explain argument coercions, while the last example about overloading explains how function coercions work. We assume that the types include integers (Int), floating numbers (Float), booleans (Bool), monads (Tσ, where σ is any type), and a unit type (called Plus).

An example of basic coercions

The simplest example of coercions, as often used in programming languages, is to convert integers to floating point numbers. For example, we can declare

$$\texttt{int2float} : \text{Int} \to \text{Float}$$

as a coercion, either in a context or in a program by using the coercion declaration[1] `cdec int2float` : Int $\to$ Float `in` . Then, assuming 2 : Int and `plusone` : Float $\to$ Float, the term `plusone` 2 is typable and abbreviates its "completion" `plusone` (`int2float` 2), where the coercion `int2float` is inserted. Note that the completion is typable in the original HM system. More formally, we say that the term (or program) `cdec int2float` : Int $\to$ Float `in plusone` 2 has type Float. The function `int2float` here is represented as a constant in the typing system. It could be defined externally (e.g., using system call at runtime).

This coercion is usually handled automatically by programming systems, without a formal explanation. We provide a principled explanation of this in a setting where we can, for example, formally answer coherence questions. Note that we can handle the converse coercion, from floating point numbers to integers using e.g. `floor`, in the same way.

Using coercions in monads

Monads are a commonly used vehicle in functional programming to deal with "imperative" features like state, random numbers, partial functions, error handling or input/output. Every Monad consists at least of a unary type constructor (called T here), an injection function (called "return" here) and a lifting function. We refer the reader for example to [22] for a full introduction.

[1] We leave out some type variable annotation; see Sec. 3.2 and 3.3 for more details

Coercions can ease use of monads, by allowing omission of the injection of a value into its "monadified" type (function `return`). T in the types for the examples below can be seen as the error monad. There are two ways to create values of this monadic type: one is a regular, good value ($\texttt{return} : \forall\alpha.\, \alpha \to T\alpha$) and the other is to signal an error or exception ($\texttt{err} : \forall\alpha.\, T\alpha$). We can then define a reciprocal function, from Float to T Float, which captures the division by zero error:

$$\lambda x.\ \texttt{if}\ (\texttt{iszero}\ x)\ \texttt{err}\ (\texttt{return}\ (\texttt{sysdiv}\ 1.0\ x)),$$

where

$$\texttt{if} : \forall\alpha.\, \texttt{Bool} \to \alpha \to \alpha \to \alpha$$

$$\texttt{iszero} : \text{Float} \to \texttt{Bool}$$

$$\texttt{sysdiv} : \text{Float} \to \text{Float} \to \text{Float}$$

Using the coercion abbreviation mechanism, however, we can leave the `return` implicit by declaring it as a coercion:

$$\begin{array}{l}\texttt{cdec}\ \texttt{return} : \forall\alpha.\, \alpha \to T\alpha\ \texttt{in} \\ \lambda x.\, \texttt{if}\ (\texttt{iszero}\ x)\ \texttt{err}\ (\texttt{sysdiv}\ 1.0\ x)\end{array}$$

Similar situations occur frequently when a monadic programming style is used, making this a fairly useful abbreviation, both for code clarity and brevity.

Note that, as shown by this example, coercions are not necessarily representing simple inclusion between types (as considered in the setting of subtyping [17]). They are arbitrary functional maps which one wishes to omit, in preference to the abbreviated form. In particular, the intuition that a type that can be coerced into another type can be viewed as set-theoretic inclusion does not apply.

Using coercions for overloading

Coercions can be used to represent ad hoc polymorphism, or overloading. For example, assume that we have two functions for addition, one for the integers and the other for the floating point numbers:

$$\texttt{plusi} : \text{Int} \to \text{Int} \to \text{Int}$$

$$\texttt{plusf} : \text{Float} \to \text{Float} \to \text{Float}$$

and we wish to use a single notation `plus` in both cases. This can be done by means of coercions. What we need to do is to consider a (unit) type `Plus` which has element $\texttt{plus} : \texttt{Plus}$ and then declare the following two (function) coercions:

$$\texttt{cdec}\ (\lambda x.\, \texttt{plusi}) : \texttt{Plus} \to (\text{Int} \to \text{Int} \to \text{Int})$$

$$\texttt{cdec}\ (\lambda x.\, \texttt{plusf}) : \texttt{Plus} \to (\text{Float} \to \text{Float} \to \text{Float})$$

Then, we can use

$$\texttt{plus}\ 1\ 2 \quad \textit{or} \quad \texttt{plus}\ 1.0\ 2.5$$

as intended, as these two terms abbreviate `plusi` 1 2 and `plusf` 1.0 2.5, respectively.

Note that, in this example, the coercions are defined λ-terms rather than just constants. It also shows that coercions are not just the same as a previously defined function. The idea of using unit types for overloading was studied by the second author [14]. See [1] for more applications of this idea.

Remark 3. We considered `Plus` to be a unit type. In fact, there could be multiple elements in `Plus` (i.e. constants of that type), but they are all treated the same.

3 Typing System

3.1 Base Language

Our starting point in this development is an existing programming language, namely a minimal polymorphic programming language with Hindley-Milner type system [8] which we call the **base language**. We assume readers are familiar with the basic ideas. We omit additional elements necessary to make this into a programming language, namely declaration of new types and recursion. This is because we focus on typing, and those features do not affect type checking. They can be added.

The typing judgment in the base language is denoted by

$$\Gamma \vdash_{HM} e : \tau,$$

which can be read as "term e has type τ in context Γ". We are extending the base language with a coercion mechanism, which leads to our system $\vdash$. We shall explain in Section 3.4 how we can recover the HM system from our rules.

3.2 Syntax and notations

Apart from coercion-specific extensions, we use standard notions of terms, types, type schemes and contexts [8]. The syntactic symbols to be used are as follows.

Type variables
$\alpha, \beta, \gamma, \varrho$

Sets of type variables
$\bar{\alpha}, \bar{\beta}, \bar{\gamma}, \bar{\varrho}$

Types
$\sigma, \tau, \varrho \quad ::= \alpha \mid \sigma \to \sigma$

Type schemes
$\mu \quad ::= \forall \bar{\alpha}.\, \sigma$

(Object language) Variables
x, y, z

Terms
$e, f, g \quad ::= x \mid ee \mid \lambda x.\, e \mid$
$\texttt{let}\ x = e\ \texttt{in}\ e \mid$
$\texttt{cdec}\ c : \forall \bar{\alpha}.\, \sigma \to \sigma\ \texttt{in}\ e$

Contexts
$\Gamma, \Delta \quad ::= \emptyset \mid \Gamma, x : \mu \mid \Gamma, \texttt{cdec}\ c : \forall \bar{\alpha}.\, \sigma \to \tau$

Notations The following notations will be used in our description of the system.

- FV stands for the set of (object) variables declared in a context: $FV(\emptyset) = \emptyset$, $FV(\Gamma, x : \mu) = FV(\Gamma) \cup \{ x \}$ and $FV(\Gamma, \texttt{cdec}\ c : \mu) = FV(\Gamma)$.
- FTV denotes the set of **free type variables** of a context, type, type scheme or term. It is defined as:

$$
\begin{array}{llll}
FTV(\Gamma, x : \mu) & = FTV(\Gamma) \cup FTV(\mu) & FTV(\emptyset) & = \emptyset \\
FTV(\Gamma, \texttt{cdec}\ c : \mu) & = FTV(\Gamma) \cup FTV(c) \cup FTV(\mu) & & \\
FTV(\sigma \to \tau) & = FTV(\sigma) \cup FTV(\tau) & FTV(\alpha) & = \{ \alpha \} \\
FTV(\forall \bar{\alpha}.\, \sigma) & = FTV(\sigma) \setminus \bar{\alpha} & FTV(x) & = \emptyset \\
FTV(e\, f) & = FTV(e) \cup FTV(f) & FTV(\lambda x.\, e) & = FTV(e) \\
FTV(\texttt{let}\ x = e\ \texttt{in}\ f) & = FTV(e) \cup FTV(f) & & \\
FTV(\texttt{cdec}\ c : \mu\ \texttt{in}\ e) & = FTV(\mu) \cup FTV(e) \cup FTV(c) & &
\end{array}
$$

- Let Γ be a context. The coercion-free part of Γ is denoted by $\widehat{\Gamma}$, and defined as $\widehat{\emptyset} = \emptyset$, $\widehat{\Gamma, x : \mu} = \widehat{\Gamma}, x : \mu$ and $\widehat{\Gamma'} = \widehat{\Gamma}$, where Γ' is $\Gamma, \texttt{cdec}\ c : \forall \bar{\alpha}.\, \sigma \to \tau$. Furthermore, we write $\Gamma(x) = \mu$ if $x : \mu$ is an entry of Γ.
- $\forall \emptyset.\, \sigma$ is a special case of $\forall \bar{\alpha}.\, \sigma$, denoting a type scheme with no bound variables. We may omit $\forall \emptyset$ when the context makes it clear we denote a type scheme instead of a type.
- $\sigma \prec_{\bar{\alpha}} \mu$ means that σ is a **generic instance** of μ where all (free) type variables of σ are in $\bar{\alpha}$.

3.3 Judgment Forms and Rules

The rules in fig. 1 define our typing system. The forms of judgments are:

- $\Gamma \vdash^{\bar{\alpha}} e : \tau \ \Rightarrow\ e'$. This should be read as "term e has type τ and completion e' in context Γ with free type variables $\bar{\alpha}$. We extend the usual typing judgment for ML-like languages $\Gamma \vdash e : \tau$ by allowing coercion declarations in the context, adding the completion e' and an explicit annotation for the free type variables which may occur in Γ, τ and e.
- Γ **$\bar{\alpha}$-valid**. To capture the notion that a "context Γ is valid with free type variables in $\bar{\alpha}$", we write Γ **$\bar{\alpha}$-valid**. Note that this judgment is useful as we consider coercions in contexts subject to certain restrictions.
- $\Gamma \vdash^{\bar{\alpha}} \sigma \to_c \tau$. This third form of judgment expresses that "coercion c from σ to τ can be derived from context Γ".

We also use the notation $\Gamma \not\vdash_{HM} e : ?$ to express the side condition that e is not typable in the HM system.

Product Types We can extend the language without affecting the basic results and mechanisms presented. For example some of the examples below will require the use of pairs. We can extend the language to add them to our language in the standard way, using the rules like the following:

$$
\textit{PairIn} \qquad \frac{\Gamma \vdash^{\bar{\alpha}} e_1 : \tau_1 \ \Rightarrow\ e_1' \quad \Gamma \vdash^{\bar{\alpha}} e_2 : \tau_2 \ \Rightarrow\ e_2'}{\Gamma \vdash^{\bar{\alpha}} \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \ \Rightarrow\ \langle e_1', e_2' \rangle}
$$

The results and the type checking algorithm can be extended in straight-forward ways.

CId

$$\frac{}{\emptyset\ \bar{\alpha}\textbf{-valid}}$$

CVar

$$\frac{\Gamma\ \bar{\alpha}\textbf{-valid}}{\Gamma, x{:}\mu\ \bar{\alpha}\textbf{-valid}} \quad \begin{array}{l} x \notin FV(\Gamma), \\ FTV(\mu) \subseteq \bar{\alpha} \end{array}$$

CCoer

$$\frac{\Gamma\ \bar{\alpha}\textbf{-valid} \quad \Gamma \vdash^{\bar{\alpha}\cup\bar{\beta}} c_0 : \sigma \to \tau\ \Rightarrow\ c}{\Gamma, \mathtt{cdec}\ c : \forall\bar{\beta}.\ \sigma \to \tau\ \bar{\alpha}\textbf{-valid}} \quad \bar{\alpha} \cap \bar{\beta} = \emptyset$$

Id

$$\frac{\Gamma\ \bar{\alpha}\textbf{-valid}}{\Gamma \vdash^{\bar{\alpha}} x : \tau\ \Rightarrow\ x} \quad \tau \prec_{\bar{\alpha}} \mu,\ \Gamma(x) = \mu$$

Abs

$$\frac{\Gamma, x : \forall\emptyset.\,\sigma \vdash^{\bar{\alpha}} e : \tau\ \Rightarrow\ e'}{\Gamma \vdash^{\alpha} \lambda x.\, e : \sigma \to \tau\ \Rightarrow\ \lambda x.\, e'}$$

App

$$\frac{\Gamma \vdash^{\bar{\alpha}} e_1 : \sigma \to \tau\ \Rightarrow\ e_1' \quad \Gamma \vdash^{\bar{\alpha}} e_2 : \sigma\ \Rightarrow\ e_2'}{\Gamma \vdash^{\bar{\alpha}} e_1 e_2 : \tau\ \Rightarrow\ e_1' e_2'}$$

App$_{ac}$

$$\frac{\begin{array}{l}\Gamma \vdash^{\bar{\alpha}} e_1 : \sigma \to \tau\ \Rightarrow\ e_1' \quad \Gamma \vdash^{\bar{\alpha}} e_2 : \sigma_0\ \Rightarrow\ e_2' \\ \Gamma \vdash^{\bar{\alpha}} \sigma_0 \to_c \sigma\end{array}}{\Gamma \vdash^{\alpha} e_1 e_2 : \tau\ \Rightarrow\ e_1'(c e_2')} \quad \widehat{\Gamma} \not\vdash_{HM} e_1' e_2' : ?$$

App$_{fc}$

$$\frac{\begin{array}{l}\Gamma \vdash^{\bar{\alpha}} e_1 : \varrho_0\ \Rightarrow\ e_1' \quad \Gamma \vdash^{\bar{\alpha}} e_2 : \sigma\ \Rightarrow\ e_2' \\ \Gamma \vdash^{\bar{\alpha}} \varrho_0 \to_c (\sigma \to \tau)\end{array}}{\Gamma \vdash^{\bar{\alpha}} e_1 e_2 : \tau\ \Rightarrow\ (c e_1') e_2'} \quad \widehat{\Gamma} \not\vdash_{HM} e_1' e_2' : ?$$

Let

$$\frac{\Gamma \vdash^{\bar{\alpha}\cup\bar{\beta}} e_1 : \sigma\ \Rightarrow\ e_1' \quad \Gamma, x : \forall\bar{\beta}.\,\sigma \vdash^{\bar{\alpha}} e_2 : \tau\ \Rightarrow\ e_2'}{\Gamma \vdash^{\bar{\alpha}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau\ \Rightarrow\ \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2'} \quad \bar{\alpha} \cap \bar{\beta} = \emptyset$$

Decl

$$\frac{\begin{array}{l}\Gamma \vdash^{\bar{\alpha}\cup\bar{\beta}} c : \sigma \to \tau\ \Rightarrow\ c' \\ \Gamma, \mathtt{cdec}\ c' : \forall\bar{\beta}.\,\sigma \to \tau \vdash^{\bar{\alpha}} e : \varrho\ \Rightarrow\ e'\end{array}}{\Gamma \vdash^{\bar{\alpha}} \mathtt{cdec}\ c : \forall\bar{\beta}.\,\sigma \to \tau\ \mathtt{in}\ e : \varrho\ \Rightarrow\ e'} \quad \bar{\alpha} \cap \bar{\beta} = \emptyset$$

Lup

$$\frac{}{\Gamma, \mathtt{cdec}\ c : \forall\bar{\beta}.\,\sigma_0 \to \tau_0, \Gamma' \vdash^{\bar{\alpha}} \sigma \to_c \tau} \quad \begin{array}{l}\sigma \to \tau \prec_{\bar{\alpha}} \\ \forall\bar{\beta}.\,\sigma_0 \to \tau_0\end{array}$$

Fig. 1. Typing Rules

3.4 Explanations

We give some informal explanations and prove some basic properties of the system presented above.

Completion and Relation to HM The above system is an extension of the system $\vdash_{HM}$ in the sense that, if we remove rules App_{ac}, App_{fc}, $Decl$, Lup and $CCoer$ and the notation of completion, the resulting system is equivalent to Hindley-Milner typing. We say that a program e is **well-typed** if $\emptyset \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$ for some type τ, completion e', and set of type variables $\bar{\alpha}$.

An addition to the language is **completion**. Informally, we insert all the needed coercion functions in a term e to form its completion e', such that the completed term is typable in the system without the coercion rules App_{ac}, App_{fc} and $CCoer$, i.e. in the base language $\vdash_{HM}$. This is formally captured by lemma 1, which will establish the relationship between our typing judgment $\vdash$ and that of the base language $\vdash_{HM}$. It makes precise why we call e' "completion": because the completion is an expansion of the term e in question, and this completion type checks in the base language.

Definition 1 (Term Expansion). *The notion that a term e_2 expands a term e_1, in symbols $e_1 \leq e_2$, is inductively defined as follows.*

$x \leq x$
$\lambda x.\, e_1 \leq \lambda x.\, e_2$ *if* $e_1 \leq e_2$
$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \leq \texttt{let}\ x = e_3\ \texttt{in}\ e_4$ *if* $e_1 \leq e_3$ *and* $e_2 \leq e_4$
$e_1 e_2 \leq e_3 e_4$ *if* $e_1 \leq e_3$ *and* $e_2 \leq e_4$
$e_2 e_3 \leq (e_1 e_2) e_3$
$e_1 e_3 \leq e_1 (e_2 e_3)$
$\texttt{cdec}\ c : \forall \bar{\alpha}.\, \sigma \to \tau\ \texttt{in}\ e \leq e$
$e_1 \leq e_3$ *if* $e_1 \leq e_2$ *and* $e_2 \leq e_3$

Lemma 1 (Completion). *If $\Gamma \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$, then $\Gamma \vdash_{HM} e' : \tau$, and $e \leq e'$.*

Proof Sketch. We prove the following two statements by simultaneous induction on the derivations of $\Gamma \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$ and Γ $\bar{\alpha}$-**valid**.

- If $\Gamma \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$, then $\Gamma \vdash_{HM} e' : \tau$ and $e \leq e'$.
- If Γ $\bar{\alpha}$-**valid** and $\Gamma \vdash^{\bar{\alpha}} \sigma \to_c \tau$, then $\Gamma \vdash_{HM} c : \sigma \to \tau$.

Free Type Variables $\bar{\alpha}$ The handling of type variables needs some explanation. The standard notation of typing judgment assumes that the free type variables in Γ can be chosen arbitrarily. On the other hand, we require that all variables must either be bound or chosen from the $\bar{\alpha}$ denoted in the judgment. Formally, the role of the free type variable annotations is captured by the following lemma which has three parts, for each of the judgements.

Lemma 2 (Free type variables).

1. *If* $\Gamma \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$, *then* $FTV(\Gamma) \subseteq \bar{\alpha}$, $FTV(e) \subseteq \bar{\alpha}$ *and* $FTV(\tau) \subseteq \bar{\alpha}$.
2. *If* Γ $\bar{\alpha}$-***valid***, *then* $FTV(\Gamma) \subseteq \bar{\alpha}$.
3. *If* $\Gamma \vdash^{\bar{\alpha}} \sigma \to_c \tau$, *then* $FTV(\Gamma) \subseteq \bar{\alpha}$, $FTV(\sigma \to \tau) \subseteq \bar{\alpha}$ *and* $FTV(c) \subseteq \bar{\alpha}$.

By explicitly denoting all possible free type variables, we no longer require the notion of "generalisation" in the formulation of the *Let* rule, which, in our opinion, clarifies its intention.

Remark 4. Another way of looking at this is that there are no free type variables, but all type variables are bound – some explicitly in type schemes, while all others are bound by the global quantification $\forall\bar{\alpha}$. To our knowledge, this is the first time this reformulation of the *Let* rule is published. It is due to McKinna [16].

In the rule *Abs*, we add x to the context, quantifying over no variables. This means that all type variables in σ are non-generic and cannot be instantiated in the derivation of $e : \tau$. This is in contrast to the *Let* rule which allows generic type variables.

Global and local coercions Besides assignments of types (more precisely type schemes) to variables, our contexts also contain declarations of (global) coercions, of the form $\texttt{cdec}\ c : \forall\bar{\alpha}.\, \sigma \to \tau\ \texttt{in}\ e$. The form of coercions is unlimited and can be any expression in the base language, like a constant function between base types or a function between arbitrary types computing the result in a complex way. The coercions declared in a context are well-typed and can be looked up by means of the rule (*Lup*). We have

Lemma 3. *If* $\Gamma \vdash^{\bar{\alpha}} \sigma \to_c \tau$, *then* $\Gamma \vdash^{\bar{\alpha}} c : \sigma \to \tau \Rightarrow c$.

In fact, we know that any declared coercion is well-typed in the HM system (c.f., Lemma 1).

Besides global coercions, we also allow local coercion declarations in programs, similar to the way `let` works.

Example 1 (Localised Coercions). This example shows the scope of coercion declaration. In $\Delta = \texttt{plusone} : \text{Int} \to \text{Int}, 1 : \text{Int}, 1.0 : \text{Float}, plus : \text{Float} \to \text{Float} \to \text{Float}$, the following program is well-typed.

$$\begin{array}{l}\texttt{plus}\ (\texttt{plusone}\ 1)\\ \quad (\texttt{cdec}\ \texttt{floor} : \forall\emptyset.\, \text{Float} \to \text{Int}\ \texttt{in}\ \texttt{plusone}\ 1.0)\end{array}$$

However, since the coercion is not available when `plusone` is first used, the following is not typable in Δ:

$$\begin{array}{l}\texttt{plus}\ (\texttt{plusone}\ 1.0)\\ \quad (\texttt{cdec}\ \texttt{floor} : \forall\emptyset.\, \text{Float} \to \text{Int}\ \texttt{in}\ \texttt{plusone}\ 1.0)\end{array}$$

Rules for argument and function coercions Let us have a closer look at the special rules App_{ac} and App_{fc} for argument and function coercions, in particular on their side condition. By $\widehat{\Gamma} \not\vdash_{HM} e_1' e_2' : ?$ we mean that $e_1' e_2'$ is not typable in the base language, i.e. there is no type τ such that $\widehat{\Gamma} \vdash_{HM} e_1' e_2' : \tau$. We illustrate the necessity of this side condition with an example which shows that otherwise ambiguity arises which would lead to non-unique meaning of certain terms.

Example 2. We assume that A and B are any base types inhabited by the constants $a : A$ and $b_1, b_2 : B$, and we have product types. Using the abbreviations

$$\begin{array}{l} \Gamma = \mathtt{cdec}\ \lambda\langle x, y\rangle.\ \langle b_1, b_2\rangle : \forall\emptyset.\ A \times A \to B \times B \\ f = \lambda\langle x, y\rangle.\ x \\ g = \lambda\langle x, y\rangle.\ \langle y, x\rangle \end{array}$$

we can obviously derive

$$\begin{array}{l} \Gamma \vdash^{\bar{\alpha}} f : B \times B \to B \ \Rightarrow\ f \\ \Gamma \vdash^{\bar{\alpha}} g : A \times A \to A \times A \ \Rightarrow\ g \\ \Gamma \vdash^{\bar{\alpha}} g : B \times B \to B \times B \ \Rightarrow\ g \end{array}$$

Thus using App_{ac} *without* the side condition $\Gamma \not\vdash_{HM} e_1' e_2' : ?$ we could derive the following, where $c = \lambda\langle x, y\rangle.\ \langle b_1, b_2\rangle$:

$$\begin{array}{l} \Gamma \vdash^{\bar{\alpha}} f(g\langle a, a\rangle) : B \ \Rightarrow\ f(c(g\langle a, a\rangle)) \\ \Gamma \vdash^{\bar{\alpha}} f(g\langle a, a\rangle) : B \ \Rightarrow\ f(g(c\langle a, a\rangle)) \end{array}$$

However, $f(c(g\langle a, a\rangle))$ computes to b_1 while $f(g(c\langle a, a\rangle))$ to b_2. This is a very bad situation, since it means that evaluation can no longer be uniquely defined, and thus the term $f(g\langle a, a\rangle)$ no longer has a definite, unique meaning.

The side condition prevents this particular ambiguity, by forbidding the use of App_{ac} and App_{fc} when App can be used. In other words, it gives preference to derivations which does not involve coercions, and a coercion may only be applied if needed since otherwise typing would fail. The side condition is decidable, for example by traditional algorithm $\mathcal{W}$. This side condition does not prevent all forms of ambiguities, however. Section 5 discusses how to deal with them.

The example shows an essential difference to coercive subtyping in Type Theory with its unique and explicit typing, where the type of g would fully determine the type of the coercion function to apply and whether a coercion is needed at all.

The side conditions on rules App_{ac} and App_{fc} have another effect too. In coercive subtyping for Type Theory, the question arises whether identity coercions (i.e. the identity function declared as coercion) are allowed. We do not forbid them, but these side conditions ensure that they will never be used, since an application with an identity coercion can always be typed without it.

Let Expression One noticeable feature of our typing rules is that there are no coercion-specific rules involving `let`. Corresponding to the rules for application, one might expect to find something like:

$$Let_c \qquad \frac{\Gamma \vdash^{\bar{\alpha}\cup\bar{\beta}} e_1 : \sigma_0 \;\Rightarrow\; e_1' \qquad \Gamma, x : \forall\bar{\beta}.\,\sigma \vdash^{\bar{\alpha}} e_2 : \tau \;\Rightarrow\; e_2' \quad \Gamma \vdash^{\bar{\alpha}\cup\bar{\beta}} \sigma_0 \to_c \sigma}{\Gamma \vdash^{\bar{\alpha}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau \;\Rightarrow\; \texttt{let}\ x = ce_1'\ \texttt{in}\ e_2'} \qquad \bar{\alpha}\cap\bar{\beta} = \emptyset$$

With this rule basic soundness conditions still hold, like lemma 1 saying that the completion is well-typed in the base language. Thus it is not obviously wrong to add thus rule. Simple examples show that Let_c is not admissible. Consider (assuming A, B, C and D are any types) $\Gamma = x : A, c : A \to B, \texttt{cdec}\ c : A \to B$. With the new rule we can then derive $\texttt{let}\ y = x\ \texttt{in}\ y : B$, without it we cannot.

Another example shows the complication of the rule Let_c. With $\Gamma = a : A, b : B, c_1 : A \to (C \to D), c_2 : B \to C, \texttt{cdec}\ c_1 : A \to (C \to D), \texttt{cdec}\ c_2 : B \to C$ and assuming the Let_c rule is present, we are able to derive $\Gamma \vdash^{\emptyset} \texttt{let}\ x = a\ \texttt{in}\ xb : D \;\Rightarrow\; \texttt{let}\ x = c_1 a\ \texttt{in}\ c_2 b$. Essentially, this amounts to a simultaneous use of functional and argument coercions which is not admissible in our rules.

These examples illustrate that Let_c would allow a more liberal use of coercions. Our intention however is to restrict the situations in which they can occur to allow a formulation of derivational coherence (see section 5). A consequence of a rule like Let_c is that a type checking algorithm (Section 4) would need to search for σ which is not present in the conclusion of the rule; this may cause difficulties.

4 Type Checking Algorithm

The previous section describes our type system which adds coercions to Hindley-Milner type systems. The rules in fig. 1 describe well-typing, but they do not provide a decision procedure to verify well-typedness. This is mainly due to the application rules (App, App_{ac} and App_{fc}), in which the argument type σ cannot be inferred from the typing judgment whose validity is to be verified, and thus there are infinitely many derivation trees to check.

This section provides a different set of rules to resolve this problem (fig. 2).

4.1 Algorithm

In the tradition of algorithm $\mathcal{W}$ [8], the rules in fig. 2 describe typing for most general types. These rules can be read as an algorithm, which we call "algorithm $\mathcal{W}_C$", to give non-deterministic answers to the question: "Given Γ and e, what are the type and completion of e?". The inputs are context Γ and term e and the outputs substitution S, type τ and completion e'. It is non-deterministic because of the rules $LCdec_1^{\mathcal{W}}$ and $LCdec_2^{\mathcal{W}}$, where multiple coercions c can be found for a given pair of types σ and τ. In Section 5 we will provide a deterministic algorithm together with a characterisation of its modified behaviour.

$CId^{\mathcal{W}}$

$$\frac{}{\emptyset \rightsquigarrow \mathbf{valid}}$$

$CVar^{\mathcal{W}}$

$$\frac{\Gamma \rightsquigarrow \mathbf{valid}}{\Gamma, x{:}\mu \rightsquigarrow \mathbf{valid}} \quad x \notin FV(\Gamma)$$

$CCoer^{\mathcal{W}}$

$$\frac{\Gamma \rightsquigarrow \mathbf{valid} \quad \Gamma \vdash^{\mathcal{W}} c \rightsquigarrow \langle \tau, S, c' \rangle}{\Gamma, \mathtt{cdec}\ c : \forall \bar{\beta}.\ \sigma \to \tau \rightsquigarrow \mathbf{valid}}$$

$Id^{\mathcal{W}}$

$$\frac{\Gamma, x : \forall \alpha_1, \ldots, \alpha_n.\ \tau, \Gamma' \rightsquigarrow \mathbf{valid}}{\Gamma, x : \forall \alpha_1, \ldots, \alpha_n.\ \tau, \Gamma' \vdash^{\mathcal{W}} x \rightsquigarrow \langle [\beta_i \,/\, \alpha_i]\tau, \emptyset, x \rangle} \quad \beta_i \text{ new}$$

$Abs^{\mathcal{W}}$

$$\frac{\Gamma, x : \forall \emptyset.\ \alpha \vdash^{\mathcal{W}} e \rightsquigarrow \langle \tau, S \circ \{\, \alpha \mapsto \sigma \,\}, e' \rangle}{\Gamma \vdash^{\mathcal{W}} \lambda x.\, e \rightsquigarrow \langle \sigma \to \tau, S, \lambda x.\, e' \rangle} \quad \alpha \text{ new}$$

$App^{\mathcal{W}}$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathcal{W}} e_1 \rightsquigarrow \langle S_1, \tau_1, e_1' \rangle \\ S_1\Gamma \vdash^{\mathcal{W}} e_2 \rightsquigarrow \langle S_2, \sigma_2, e_2' \rangle \\ \mathit{unify}_{\mathcal{C}}(\Gamma, S_2\tau_1, \tau_2, e_1', e_2') \rightsquigarrow \langle T, e_3' \rangle\end{array}}{\Gamma \vdash^{\mathcal{W}} e_1 e_2 \rightsquigarrow \langle T S_2 \tau_1, T \circ S_2 \circ S_1, e_3' \rangle} \quad \alpha, \beta \text{ new}$$

$Let^{\mathcal{W}}$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathcal{W}} e_1 \rightsquigarrow \langle \tau_1, S_1, e_1' \rangle \\ S_1\Gamma, x : \mathit{Gen}(\tau_1, S_1\Gamma) \vdash^{\mathcal{W}} e_2 \rightsquigarrow \langle \tau_2, S_2, e_2' \rangle\end{array}}{\begin{array}{l}\Gamma \vdash^{\mathcal{W}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\ \qquad \rightsquigarrow \langle \tau_2, S_2 \circ S_1, \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2' \rangle\end{array}}$$

$Decl^{\mathcal{W}}$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathcal{W}} c \rightsquigarrow \langle \varrho_0, \emptyset, c' \rangle \\ \Gamma, \mathtt{cdec}\ c' : \forall \bar{\beta}.\ \sigma \to \tau \vdash^{\mathcal{W}} e \rightsquigarrow \langle \varrho, S, e' \rangle\end{array}}{\Gamma \vdash^{\mathcal{W}} \mathtt{cdec}\ c : \forall \bar{\beta}.\ \sigma \to \tau\ \mathtt{in}\ e \rightsquigarrow \langle \varrho, S, e' \rangle} \quad \sigma \to \tau \prec \varrho_0$$

$Unc^{\mathcal{W}}$

$$\frac{\mathit{unify}(\beta, \tau) = T}{\mathit{unify}_{\mathcal{C}}(\Gamma, \beta \to \alpha, \tau, e_1, e_2) \rightsquigarrow \langle T, e_1 e_2 \rangle}$$

$Unc_{ac}^{\mathcal{W}}$

$$\frac{\begin{array}{c}\mathit{unify}(\beta \to \tau, \sigma_0 \to \sigma) = T \\ \Gamma \vdash_L \sigma_0 \to \sigma \rightsquigarrow c\end{array}}{\mathit{unify}_{\mathcal{C}}(\Gamma, \beta \to \alpha, \tau, e_1, e_2) \rightsquigarrow \langle T, e_1(c e_2) \rangle} \quad \begin{array}{l}\mathit{unify}(\beta, \tau) \\ \mathit{fails}\end{array}$$

$Unc_{fc}^{\mathcal{W}}$

$$\frac{\begin{array}{c}\mathit{unify}(\beta \to (\tau \to \tau_1), \sigma_0 \to \sigma) = T \\ \Gamma \vdash_L \sigma_0 \to \sigma \rightsquigarrow c\end{array}}{\mathit{unify}_{\mathcal{C}}(\Gamma, \beta, \tau, e_1, e_2) \rightsquigarrow \langle T, (c e_1) e_2 \rangle} \quad \begin{array}{l}\mathit{unify}(\beta, \tau) \\ \mathit{fails}\end{array}$$

$LVar^{\mathcal{W}}$

$$\frac{\Gamma \vdash_L \sigma \to \tau \rightsquigarrow c}{\Gamma, x : \mu \vdash_L \sigma \to \tau \rightsquigarrow c}$$

$LCdec_1^{\mathcal{W}}$

$$\frac{\Gamma \vdash_L \sigma \to \tau \rightsquigarrow c}{\Gamma, \mathtt{cdec}\ c_0 : \forall \bar{\beta}.\ \sigma_0 \to \tau_0 \vdash_L \sigma \to \tau \rightsquigarrow c}$$

$LCdec_2^{\mathcal{W}}$

$$\frac{}{\Gamma, \mathtt{cdec}\ c : \forall \bar{\beta}.\ \sigma \to \tau \vdash_L \sigma \to \tau \rightsquigarrow c}$$

Fig. 2. Algorithm $\mathcal{W}_{\mathcal{C}}$

$\mathcal{W}_C$ is presented with judgments of the following forms:

- $\Gamma \vdash^{\mathcal{W}} e \leadsto \langle S, \tau, e' \rangle$, which can be read as "In context Γ, term e type checks to substitution S, type τ and completion e'.
- $\Gamma \leadsto$ **valid** expresses that "Γ is valid"; in particular, it means that the coercions declared in it are well-typed.
- $\Gamma \vdash_L \sigma \to \tau \leadsto c$ stands for "in context Γ, the lookup for a coercion from type σ to type τ yields the coercion term c".

We use the standard notion of first-order unification *unify*. It is easy to see that the traditional algorithm $\mathcal{W}$ can be recovered from the rules in fig. 2 by removing rules $CCoer^{\mathcal{W}}$, $Decl^{\mathcal{W}}$, $Unc^{\mathcal{W}}_{ac}$ and $Unc^{\mathcal{W}}_{fc}$. (In that case $\vdash_L$ will not be used either.) Using this observation and soundness and completeness of $\mathcal{W}$, we can see that the condition on c in rule $Decl^{\mathcal{W}}$ is actually the same as in *Decl* in fig. 1.

Note that the side condition of rules App_{ac} and App_{fc} in fig. 1 refers to the separate system of HM typing, while the implementation uses a simple unification test in $Unc^{\mathcal{W}}$ and does not need to refer to a separate type checking algorithm.

4.2 Soundness and Completeness

Algorithm $\mathcal{W}_C$ (fig. 2) is a sound and complete implementation of the typing rules (fig. 1), in the following sense.

Soundness expresses that the computed result type and completion can be derived using the typing rules.

Theorem 1 (Soundness). *Assume that we can derive* $\Gamma \vdash^{\mathcal{W}} e \leadsto \langle \tau, S, e' \rangle$. *Let* $\bar{\alpha} = FTV(S\Gamma, \tau, e)$. *Then* $S\Gamma \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$.

Proof Sketch. We can prove this by strengthening it with the additional condition if $\Gamma \leadsto$ **valid** and $\bar{\alpha} = FTV(\Gamma)$, then Γ $\bar{\alpha}$-**valid**. We then do simultaneous induction on the derivations of $\Gamma \vdash^{\mathcal{W}} e \leadsto \langle \tau, S, e' \rangle$ and $\Gamma \leadsto$ **valid**, using much of the structure and lemmas from [7]. Use of $Unc^{\mathcal{W}}$ in $App^{\mathcal{W}}$ by the algorithm corresponds to rule *App*, whereas $Unc^{\mathcal{W}}_{ac}$ and $Unc^{\mathcal{W}}_{fc}$ correspond to App_{ac} and App_{fc}, resp.

Completeness means that for any given completion, every derivable type for a term is an instance of the type computed by the algorithm for the result with this completion.

Theorem 2 (Completeness). *If* $S\Gamma \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$, *then there are exactly one type* σ *and substitution* T *such that* $\Gamma \vdash^{\mathcal{W}} e \leadsto \langle \sigma, T, e' \rangle$, *and there is a substitution* U *with* $\tau = U\sigma$ *and* $S\Gamma = UT\Gamma$.

Proof Sketch. The proof uses induction on the derivation of $S\Gamma \vdash^{\bar{\alpha}} e : \tau \Rightarrow e'$. Thus when looking for the right derivations for $\mathcal{W}_C(\Gamma; e)$ to prove the theorem, we already know the completion in the result. This completion resolves possible ambiguities in the choice of rules $Unc^{\mathcal{W}}_{ac}$ or $Unc^{\mathcal{W}}_{fc}$.

5 Resolving Ambiguities

The rules in fig. 2 allow certain ambiguities, that can occur if there is more than one matching coercion during coercion search in $Unc^{\mathcal{W}}$. Assume, for example, that A and B are base types and Γ is $f : \alpha \times \alpha \to \alpha, a : A, \texttt{cdec}\ c_1 : \forall\emptyset.\ A \to A \times A, \texttt{cdec}\ c_2 : \forall\emptyset.\ A \to B \times B$. Then we have both $\Gamma \vdash^{\mathcal{W}} fa \rightsquigarrow \langle A, \emptyset, f(c_1 a)\rangle$ and $\Gamma \vdash^{\mathcal{W}} fa \rightsquigarrow \langle B, \emptyset, f(c_2 a)\rangle$.

Such a situation is not desirable, since it means that the evaluation behaviour is not uniquely defined. This is the **coherence** problem which needs to be addressed for any system of (coercive) subtyping.

We can solve this problem by replacing $unify_{\mathcal{C}}$ in $App^{\mathcal{W}}$ by $unify^1_{\mathcal{C}}$, which succeeds if and only if $unify_{\mathcal{C}}$ returns a unique result:

Definition 2 ($unify^1_{\mathcal{C}}$). *$unify^1_{\mathcal{C}}(\Gamma, \beta, \tau, e) \rightsquigarrow \langle T, f\rangle$ if $unify_{\mathcal{C}}(\Gamma, \beta, \tau, e) \rightsquigarrow \langle T, f\rangle$ and for all U, g such that $unify^1_{\mathcal{C}}(\Gamma, \beta, \tau, e) \rightsquigarrow \langle U, g\rangle$, $U = T$ and $f = g$.*

$unify^1_{\mathcal{C}}$ is effectively decidable since $unify_{\mathcal{C}}$ is decidable and can only return a finite number of results.

We call algorithm $\mathcal{W}^1_{\mathcal{C}}$ the algorithm obtained from $\mathcal{W}_{\mathcal{C}}$ where the $App^{\mathcal{W}}$ case uses $unify^1_{\mathcal{C}}$ instead of $unify_{\mathcal{C}}$, and $\vdash^{\mathcal{W}}_1$ for the corresponding judgment. Algorithm $\mathcal{W}^1_{\mathcal{C}}$ can return at most one result, and is therefore a deterministic algorithm, in contrast to non-deterministic $\mathcal{W}_{\mathcal{C}}$.

These additional side conditions clearly limit the cases in which the algorithm succeeds. This still allows all the examples presented earlier. However the question is how this restricted behaviour can be described in the typing rules. For this, we introduce the notion of "derivational coherence".

Definition 3 (Derivational Coherence). *A term e is **derivationally coherent** over a context Γ if for each subterm f of e and $\Gamma_1 \vdash^{\bar\alpha} f : \tau_1 \Rightarrow e'_1$ and $\Gamma_2 \vdash^{\bar\alpha} f : \tau_2 \Rightarrow e'_2$ occurring anywhere in any derivation of $\Gamma \vdash^{\bar\alpha} e : \tau \Rightarrow e'$ for any τ_1, τ_2, e'_1 and e'_2, the two completions are the same, i.e. $e'_1 = e'_2$.*

Using this notion, we can formulate a soundness and completeness result for $\mathcal{W}^1_{\mathcal{C}}$.

Theorem 3. *For all Γ, e, the following holds. There are τ, S and e' such that $\Gamma \vdash^{\mathcal{W}}_1 e \rightsquigarrow \langle \tau, S, e'\rangle$ if and only if e is derivationally coherent over Γ and there are σ, f' and $\bar\alpha$ such that $\Gamma \vdash^{\bar\alpha} e : \sigma \Rightarrow f'$. In both directions, $e' = f'$ and $\sigma \prec_{\bar\alpha} \tau$.*

For the proof we note that the derivation trees for typing derivation and for type checking are isomorphic, and thus we can establish the conditions in which ambiguities occur by an inductive analysis of them, using the previous soundness and completeness results (Theorems 1 and 2).

6 Conclusion

We have presented an extension of the Hindley-Milner polymorphic system with coercions by incorporating the idea from coercive subtyping. The extended typing system can be further enriched with other features such as records whose associated inheritance relation can be represented as coercions. More details of the work, including a prototype implementation of the extended system and the details of the proofs, can be found in the forthcoming thesis of the first author [11].

There are several issues to be further studied. For example, in our rules we have not included "transitivity" as found in general subtyping or coercive subtyping systems. For basic types, adding transitivity of coercions is not a problem; it simply becomes a decidable search problem of the transitive closure of the coercions between basic types, representable as a finite graph [19]. However, when coercions parameterised over type variables are considered, as they are allowed here in general, it is not clear to us that the coercion search with transitivity is decidable.

Coercion rules are another field of further study (e.g., see [14]). The current system would allow to add rules to derive new coercions from the rules already declared, like lifting of coercions over lists. The requirement is that coercion search must be decidable.

As mentioned in the introduction, coercion search for type theory is facilitated considerably by the unique typing property. That is no longer given, however, if metavariables are added. Thus we can look to apply the techniques of this paper to type theory with metavariables.

Coercion mechanisms as discussed in this paper facilitate overloading among other things. Another mechanism for overloading is the class mechanism in Haskell [23, 10]. An interesting research topic is to compare these mechanisms formally and consider a possible general framework for abbreviations.

Acknowledgements We thank Paul Callaghan and James McKinna for discussions and comments on a draft.

References

1. A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1999.
2. B. Barras et al. *The Coq Proof Assistant Reference Manual (Version 6.3.1).* INRIA-Rocquencourt, 2000.
3. Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and André Ščedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991. Also in the collection [9].
4. P. Callaghan and Z. Luo. An implementation of LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.
5. Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

6. G. Chen. *Subtyping, Type Conversion and Transitivity Elimination.* PhD thesis, University of Paris VII, 1998.
7. Luis Damas. *Type Assignment in Programming Languages.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1985. CST-33-85.
8. Luis Damas and Robin Milner. Principal type-schemes for functional programming languages. In *Ninth Annual Symposium on Principles of Programming Languages (POPL) (Albuquerque, NM)*, pages 207–212. ACM, January 1982.
9. Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics, and Language Design.* Foundations of Computing Series. MIT Press, 1994.
10. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space, 1997.
11. Robert Kießling. Coercions in Hindley-Milner systems. forthcoming thesis, 2004.
12. Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. Coherence and transitivity of subtyping as entailment. *Journal of Logic and Computation*, 10(4):493–526, August 2000.
13. Y. Luo and Z. Luo. Coherence and transitivity in coercive subtyping. *Proc. of the 8th Inter. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'01), Havana, Cuba. LNAI 2250*, 2001.
14. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.
15. Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.
16. James McKinna. personal communication, 2001.
17. John C. Mitchell. Coercion and type inference. In *Tenth Annual Symposium on Principles of Programming Languages (POPL) (Austin, TX)*, pages 175–185. ACM, January 1983.
18. John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(2):245–286, July 1991.
19. A. Saïbi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.
20. S. Soloviev and Z. Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 113(1-3):297–322, 2002.
21. Simon Thompson. *Type Theory and Functional Programming.* International Computer Science Series. Addison-Wesley, 1991.
22. Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, 1995.
23. Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorhism less *ad-hoc*. In *Proceedings of POPL '89*, pages 60–76. ACM, January 1989.