Under consideration for publication in Math. Struct. in Comp. Science

Coercions in a polymorphic type system

Zhaohui Luo[†]

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey TW20 0EX, U.K.

Email: zhaohui@cs.rhul.ac.uk

Received 12 February 2007 and Revised 18 December 2007

Abstract

The idea of coercive subtyping, a theory of abbreviation for dependent type theories, is incorporated into the polymorphic type system in functional programming languages. The traditional type system with let-polymorphism is extended with argument coercions and function coercions, and a corresponding type inference algorithm is presented and proved to be sound and complete.

1. Introduction

The Hindley-Milner let-polymorphic type system (the HM system for short) (Damas and Milner 1982; Damas 1985; Milner 1978) is the standard core of the modern typed functional programming languages such as ML (Milner, Tofts and Harper 1990), Haskell (Peyton-Jones 2003), OCaml (Leroy 2000) and Clean (Brus *et al.* 1987). Ever since the notion of principal type was studied ((Hindley 1969) for the simply-typed λ-calculus and (Milner 1978) for the HM system), it has gained tremendous popularity. Various extensions to the HM system have also been proposed in order to enrich a programming language with new and more powerful features. These include, for example, Haskell's class mechanism (Peyton-Jones, Jones and Meijer 1997), which provides convenient overloading facilities among other things.

Coercive subtyping (Luo 1997; Luo 1999) is a theory of abbreviation developed in the setting of dependent type theories with inductive types such as Martin-Löf's type theory (Nordström, Petersson and Smith 1990; Martin-Löf 1984) and UTT (Luo 1994), where coercions are regarded as an abbreviation mechanism which has shown great expressiveness in various applications of formalisation and proof development. Coercion mechanisms have been implemented and effectively used in several type theory based

[†] This work is partially supported by the following research grants: UK EPSRC grant GR/R84092, Leverhulme Trust grant F/07 537/AA and EU TYPES grant 510996.

proof assistants such as Coq (Saïbi 1997), Lego (Bailey 1999) and Plastic (Callaghan and Luo 2001).

In this paper, we incorporate the idea of coercive subtyping into the Hindley-Milner type system. There are several motivations in studying the possible combination of coercive subtyping and polymorphic typing systems. First, it leads to a novel and disciplined approach that increases the power of the HM system with new abbreviation mechanisms, which we believe would be useful in gaining a better understanding of the introduction and use of abbreviation mechanisms in typed programming. Secondly, coercive subtyping provides a clean and simple theory for abbreviation in dependent type theories. Incorporating its ideas into traditional polymorphic type systems may lead to simple theoretical development 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 that type uniqueness simply does not hold in a polymorphic system, while it was one of the key features of the type systems for which coercive subtyping has been considered so far. Such a study may be viewed as a precursor to that of the more general combination of dependent types and polymorphic types.

We present an extension of the HM typing system with coercions. In particular, we consider two kinds of coercions – argument coercions and function coercions. The argument coercions are inserted to make the arguments of a function have the types needed for correct typing, while the function coercions make terms in function positions into those with correct types. Both of them are abbreviation mechanisms that enable omissions of parts of a term to improve readability etc.

The HM typing system is extended with such coercions, resulting in the system HM_c which allow the abbreviated terms to be typable. A corresponding type inference algorithm $\mathcal{W}_c$ is developed by extending the algorithm $\mathcal{W}$ of the HM system (Damas and Milner 1982) and proved to be sound and complete with respect to the typing system HM_c.

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 such as those in libraries.

An earlier attempt on introducing coercions into the HM system was made by Robert Kießling and the present author in (Kießling and Luo 2004). It is worth noting that the typing system HM_c and the algorithm $\mathcal{W}_c$ as presented in the current paper (in particular, their rules) are very much different from those in (Kießling and Luo 2004), which have been improved and simplified. One of the simplifications involves the removal of “coercion declarations” and “local coercions” and hence keeps the distinction between types and terms, which has played an important part in the simplification of the meta-theoretic proofs.

The remainder of this section consists of two subsections, one to describe some of the notational conventions used in this paper, particularly about the let-polymorphic typing, and the other to summarise the existing work on coercive subtyping and related work. In Section 2, we give a brief introduction to our approach by considering several simple examples. HM$_c$, the typing system extended with coercions, is presented and explained in Section 3. In Section 4, the type inference algorithm $\mathcal{W}_c$ is presented and proved to be sound and complete. Section 5 considers a further extension of the system, with simultaneous insertions of argument and function coercions, and discusses the issue of ambiguity after coercions are introduced. The conclusion section mentions several issues of possible future work.

1.1. *Notational conventions for let-polymorphic typing*

We regard the Hindley-Milner let-polymorphic type system and its associated type inference algorithm (Damas and Milner 1982) as well-known and refer it to (Leroy 1992) for a very clear introduction and other books on typed functional programming.

In this paper, we shall mostly adopt the standard notations of terms, types, type schemes and contexts (or typing environments) as used in (Damas and Milner 1982). Specifically, we shall use the following syntactic conventions.

— *Type variables* are denoted by α, β and γ and ordinary variables by x, y and z.
— *Types* are denoted by σ, τ and ρ and are of the form α (type variables) or $\sigma \to \tau$ (arrow types).
— *Type schemes* are denoted by μ and of the form $\forall\alpha_1...\alpha_n.\sigma$. We identify α-convertible type schemes.
— *Contexts* are sets of pairs, each consisting of a variable and a type scheme. A context is either $\emptyset$ (the empty context) or $\Gamma, x : \mu$ (the union of context Γ with $\{x{:}\mu\}$), where x does not occur free in Γ.
— *Terms* are the usual (untyped) λ-terms (variables x, terms of the form $\lambda x.e$ and applications $f\,a$) together with those of the let-form:
— *Type substitutions* (or substitutions for short) are finite mappings from type variables to types. The substitution of type σ for α is written as $[\sigma/\alpha]$. Substitutions are denoted by S, T and U. We usually write $dom(S)$ for the domain of substitution S, ST for the composition $S \circ T$, and $S\sigma$ for the application $S(\sigma)$. The application of substitutions is extended to contexts as well.
— Substitutions are extended to type schemes. First, let's define that a type variable α is *out of reach* for a substitution S if $\alpha \notin dom(S)$ and $\alpha \notin FTV(\sigma)$ for any type σ in the range of S. Then, substitutions are extended to type schemes as follows:

$$S(\forall\bar{\alpha}.\sigma) = \forall\bar{\alpha}.S(\sigma),$$

assuming, after renaming the variables in $\bar{\alpha}$ if necessary, that the variables in $\bar{\alpha}$ are out of reach of S.

Furthermore, we shall adopt the following notational conventions:

— We do not distinguish between the type σ and the trivial type scheme $\forall\emptyset.\sigma$ (with zero quantified type variables) and write σ for both. With this convention, types are special cases of type schemes.

— As usual, we write $FTV(\mu)$ for the set of the type variables occurring free in type scheme μ (or just type τ if $\mu = \forall\emptyset.\tau$). It extends to contexts in the natural way.
— If $x : \mu \in \Gamma$, then $\Gamma(x)$ denotes μ.
— We sometimes use $\bar{k}$ to stand for a set of k's, say $\{k_1, ..., k_n\}$. For instance, $\bar{\alpha}$ would stand for a set of type variables.

The following notions of instance and closure (or generalisation) are defined as usual.

Definition 1.1 (instance).

— Let σ be a type and $\mu = \forall\alpha_1...\alpha_n.\tau$ a type scheme. Then, σ is an *instance* of μ, notation $\sigma \preceq \mu$, if there exists a substitution S, whose domain is a subset of $\{\alpha_1, ..., \alpha_n\}$, such that $\sigma = S\tau$.
— Let u and u' be type schemes. Then u' is *more general than* u, notation $u' \geq u$, if all instances of u are instances of u'. □

Definition 1.2 (closure). Let Γ be a context and σ a type. Then, the closure of σ with respect to Γ is the type scheme defined as follows:

$$\mathcal{C}_\Gamma(\sigma) = \forall\alpha_1...\alpha_n.\sigma,$$

where $\{\alpha_1, ..., \alpha_n\} = FTV(\sigma) \setminus FTV(\Gamma)$. □

1.2. *Coercive subtyping*

Coercive subtyping, as introduced in (Luo 1997; Luo 1999), is a framework of abbreviation for dependent type theories, where coercions play the role of abbreviation. 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 function f with domain B can be applied to any object a of A and the application fa stands for $f(ca)$. 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 above simple idea, which was studied in the literature for simple type systems (see for example, (Mitchell 1991; Mitchell 1983)), has been studied for dependent type theories with inductive types, resulting in a very powerful theory of abbreviation and inheritance, including coercions between parameterised inductive types (Luo, Luo and Soloviev 2002; Luo and Luo 2005; Luo 2005) and dependent coercions (Luo and Soloviev 1999). 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 (see, for example, (Soloviev and Luo 2002)). They not only justify the adequacy of the theory from the proof-theoretic consideration, but provide the basis for implementations of coercive subtyping. See (Bailey 1999; Luo and Callaghan 1998; Luo and Luo 2005; Luo 1999) for details of some of these development and applications of coercive subtyping.

Coercion mechanisms have been implemented for dependent type theories in the proof assistants Lego (Luo and Pollack 1992), Coq (Coq 04) and Plastic (Callaghan and Luo 2001) by Bailey (Bailey 1999), Saïbi (Saïbi 1997) and Callaghan (Callaghan and Luo 2001), respectively.

Remark Incorporating the idea of coercive subtyping into 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 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. □

Concerning related work of coercive subtyping, we would like to point out that some notions of coercion have been studied in the literature, although they are not quite the same or general as studied in coercive subtyping. Particularly, coercions are considered in modelling simple subtyping, where the subsumption rule is modelled by means of coercions. In (Breazu-Tannen *et al.* 1991), this idea was used to give a coercion-based semantic interpretation of Cardelli and Wegner's system Fun (Cardelli and Wegner 1985); the idea of coercive subtyping was influenced by this work. In (Longo, Milsted and Soloviev 1999; Chen 1998), the term coercion is used to denote a special restricted form of mapping in modelling and explaining subtyping.

2. Some simple examples of coercion

We shall consider the Hindley-Milner 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 extend the HM type system with two forms of coercions: argument coercions and function coercions, which have been studied for dependent type theories in, for example, (Bailey 1999; Luo 1999). By argument coercions, we mean that the argument of a function may be coerced due 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 may be coerced into an appropriate function accordingly; more precisely, ka abbreviates $(ck)a$ if $k : \rho$ and $a : \sigma$ and the coercion c is from ρ to a function type with domain σ.

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 those of integers (`Int`), floating numbers (`Float`), booleans (`Bool`), monads ($\mathtt{T}\sigma$, 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

$$\texttt{i2f} : \texttt{Int} \to \texttt{Float}$$

as a coercion, notation

$$\texttt{Int} \xrightarrow{\texttt{i2f}} \texttt{Float}.$$

Then, assuming $2 : \texttt{Int}$ and $\texttt{plusone} : \texttt{Float} \to \texttt{Float}$, the term `plusone` 2, not well-typed in the original system, becomes typable and abbreviates its "completion" `plusone` (`i2f` 2), where the coercion `i2f` is inserted. Note that the completion is typable in the original HM system. The coercion function `i2f` is represented here as a constant in the typing system. It could be defined externally (e.g., using a system call at runtime).

A coercion such as the above one 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 states, 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 (Wadler 1995) for a full introduction.

Coercions can ease the 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 for regular and good values ($\texttt{return} : \forall\alpha.\alpha \to \texttt{T}\alpha$) and the other is to signal an error or exception ($\texttt{err} : \forall\alpha.\texttt{T}\alpha$). 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

$$\begin{array}{rcl} \texttt{if} & : & \forall\alpha.\texttt{Bool} \to \alpha \to \alpha \to \alpha \\ \texttt{iszero} & : & \texttt{Float} \to \texttt{Bool} \\ \texttt{sysdiv} & : & \texttt{Float} \to \texttt{Float} \to \texttt{Float} \end{array}$$

Using the coercion abbreviation mechanism, however, we can leave the `return` implicit by declaring it as a coercion, where γ is a type variable:

$$\gamma \xrightarrow{\texttt{return}} \texttt{T}\gamma;$$

i.e., `return` is a coercion from σ to $\mathtt{T}\sigma$, for any type σ. In other words, with the above coercion, we only need to write

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

with `return` omitted. 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 (Mitchell 1983)). They are arbitrary functional maps which one wishes to omit, in preference to the abbreviated form. In particular, the intuition that the existence of a coercion means a set-theoretic inclusion between the types does *not* apply in general.

Using coercions for overloading

Coercions can be used to represent ad hoc polymorphism or overloading (Strachey 2000).[†] For example, assume that we have two functions for addition, one for the integers and the other for the floating point numbers:

$$\begin{aligned} \mathtt{plusi} &: \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \\ \mathtt{plusf} &: \mathtt{Float} \to \mathtt{Float} \to \mathtt{Float} \end{aligned}$$

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 the (only) object `plus` : `Plus` and to declare the following two functions as coercions:

$$\begin{aligned} \lambda x.\mathtt{plusi} &: \mathtt{Plus} \to (\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}) \\ \lambda x.\mathtt{plusf} &: \mathtt{Plus} \to (\mathtt{Float} \to \mathtt{Float} \to \mathtt{Float}) \end{aligned}$$

one coercion from `Plus` to $\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}$ and the other from `Plus` to $\mathtt{Float} \to \mathtt{Float} \to \mathtt{Float}$. Then, we can use

$$\mathtt{plus}\ 1\ 2 \qquad \textit{or} \qquad \mathtt{plus}\ 1.0\ 2.5$$

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

Remark As in this example, the coercions are defined λ-terms rather than just constants. In general, a coercion can be any well-typed term in the language. □

3. Hindley-Milner typing with coercions

The Hindley-Milner type system extended with coercions, HM_c, is presented in this section, after the introduction of the formal form of coercions in the following subsection.

Our starting point in this development is an existing programming language, namely

[†] The idea of using coercions and unit types to express overloading was studied in (Luo 1999). See (Bailey 1999) for applications of this idea to proof development.

a minimal polymorphic programming language with the Hindley-Milner type system (Damas and Milner 1982; Damas 1985; Milner 1978), which we call the *base language*. The base language we shall consider is a simple one, consisting of only λ-terms and let-expressions. This gives us enough to deal with typing. The additional elements necessary to make it into a programming language, such as declaration of new types and recursion, can be added, but they generally do not cause problems in typing.

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 (in the HM system)". We say that a term e is *well-typed under* Γ in the HM system if $\Gamma \vdash_{HM} e : \sigma$ for some σ.

The base language can be extended with further constructs without affecting the basic results presented in this paper, although we have kept it simple for the sake of simplicity of proofs. For example, one may extend it with pairs, as considered in (Leroy 1992), with the following rule:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (e_1, e_2) : \sigma_1 \times \sigma_2}$$

We shall use the product types in our examples below.

3.1. *Coercions*

We shall extend the polymorphic HM type system with argument coercions and function coercions. In general, the form of coercions is unlimited and can be any expression in the base language, like a constant function between base types or a functional term computing the result in a complex way.

Coercion judgements are of the form

$$\sigma \xrightarrow{c} \tau,$$

where σ and τ are types. The above judgement has the preassumption that $c : \sigma \to \tau$ is derivable in the Hindley-Milner system (Damas and Milner 1982). In other words, a coercion judgement is meaningful only if c is of type $\sigma \to \tau$ in the HM system.

Coercions are closed under type substitutions, as expressed by the following rule:

$$(*) \qquad \frac{\sigma \xrightarrow{c} \tau}{S\sigma \xrightarrow{c} S\tau}$$

where S is any type substitution. This implies that a coercion $\sigma \xrightarrow{c} \tau$ has in fact the implicit universal quantification of the type variables occurring in σ and τ. For instance, if $\alpha \xrightarrow{c} T\alpha$, then $\tau \xrightarrow{c} T\tau$ for any type τ. Note that the above rule $(*)$ preserves the preassumption of coercion judgements since substitution preserves typing in the HM system: $c : S\sigma \to S\tau$ if $c : \sigma \to \tau$.

In the following, coercion judgements are either declared (as axioms) or derived from the above rule $(*)$. The following assumptions are made:

— Only finitely many coercion judgements are declared (as axioms). These axiomatic coercion judgements are called *initial coercions* and form a (finite) set $\mathcal{C}$.[‡]

— In any initial coercion $\sigma_1 \xrightarrow{c} \sigma_2 \in \mathcal{C}$, the free type variables in σ_i $(i = 1, 2)$ never occur free in a context of any judgement. In other words, $FTV(\sigma_i) \cap FTV(\Gamma) = \emptyset$ for any judgement with context Γ.

Note that, the second convention of the above only applies to the *initial* coercions. By the above rule $(*)$, free type variables occurring in a coercion judgement can be changed.

Remark Here, coercions are introduced independently with the typing derivations as considered in the next subsection. This is different from (Kießling and Luo 2004), where coercion declarations in contexts and local coercions are considered. Not considering local coercions simplifies the system. In particular, we keep the distinction between types and terms; this has simplified the meta-theoretic proofs considerably. □

3.2. *The typing system*

Our typing system extends the HM system with coercions. The judgement form of our system is:

$$\Gamma \vdash e : \sigma \Rightarrow e'.$$

It should be read as "term e has type σ and completion e' in context Γ". The typing judgement $\Gamma \vdash e : \sigma$ is extended with the "completion" e', which is a completed form of e after appropriate coercions are inserted. Note that, according to the above convention at the end of the last subsection, for any judgement we consider, $FTV(\Gamma) \cap FTV(\sigma_1, \sigma_2) = \emptyset$, where $\sigma_1 \xrightarrow{c} \sigma_2 \in \mathcal{C}$ is an initial coercion.

The rules of HM$_c$, the typing system extended with coercions, are given in Figure 1, where the instantiation relation in (var) and the closure operator in (let) are defined in Definitions 1.1 and 1.2, respectively, and the side condition $\Gamma \not\vdash_{HM} e'_1 e'_2 :?$ in (app_{ac}) and (app_{fc}) expresses that $e'_1 e'_2$ is not well-typed under Γ in the HM system.

3.3. *Informal explanations and properties*

We give some informal explanations of the typing rules of HM$_c$ (Figure 1) and prove some of its basic properties.

Relationship with the HM system. The above system is an extension of the HM system $\vdash_{HM}$ in the sense that, if we remove the rules (app_{ac}) and (app_{fc}), and the notation of completion, the resulting system is equivalent to Hindley-Milner typing. Actually, we have the following lemma.

Lemma 3.1. If $\Gamma \vdash e : \sigma \Rightarrow e'$ and e is well-typed under Γ in the HM system, then $\Gamma \vdash_{HM} e : \sigma$.

[‡] In practice, for example, the initial coercions can be declared by the user.

Assumption, abstraction and let

(var) $$\frac{\tau \preceq \Gamma(x)}{\Gamma \vdash x : \tau \Rightarrow x}$$

(λ) $$\frac{\Gamma, x : \sigma \vdash e : \tau \Rightarrow e'}{\Gamma \vdash \lambda x.e : \sigma \rightarrow \tau \Rightarrow \lambda x.e'}$$

(let) $$\frac{\Gamma \vdash e_1 : \sigma \Rightarrow e_1' \quad \Gamma, x : \mathcal{C}_\Gamma(\sigma) \vdash e_2 : \tau \Rightarrow e_2'}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau \Rightarrow let\ x = e_1'\ in\ e_2'}$$

Applications

(app) $$\frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \Rightarrow e_1' \quad \Gamma \vdash e_2 : \sigma \Rightarrow e_2'}{\Gamma \vdash e_1 e_2 : \tau \Rightarrow e_1' e_2'}$$

(app_{ac}) $$\frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \Rightarrow e_1' \quad \Gamma \vdash e_2 : \sigma_0 \Rightarrow e_2' \quad \sigma_0 \xrightarrow{c} \sigma}{\Gamma \vdash e_1 e_2 : \tau \Rightarrow e_1'(c e_2')} \quad (\Gamma \not\vdash_{HM} e_1' e_2' :?)$$

(app_{fc}) $$\frac{\Gamma \vdash e_1 : \rho \Rightarrow e_1' \quad \Gamma \vdash e_2 : \sigma \Rightarrow e_2' \quad \rho \xrightarrow{c} (\sigma \rightarrow \tau)}{\Gamma \vdash e_1 e_2 : \tau \Rightarrow (c e_1') e_2'} \quad (\Gamma \not\vdash_{HM} e_1' e_2' :?)$$

Fig. 1. Typing rules of HM$_c$

Proof. It suffices to notice that the lemma is true because of the side condition of the rules (app_{ac}) and (app_{fc}). □

Completions. The addition to the judgement form of 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}) and (app_{fc}), i.e. in the base language. In other words, a completion of a term is an expansion of the term in question; and this completion type checks in the base language. Furthermore, every well-typed term in the HM system has the same type and its completion is itself. This is formally captured by the lemma below, which uses the notion of expansion, defined as follows.

Definition 3.2 (expansion). The notion that a term e_2 expands a term e_1, in symbols $e_1 \sqsubseteq e_2$, is inductively defined by the rules in Figure 2. □

Lemma 3.3 (completion).

1 If $\Gamma \vdash e : \sigma \Rightarrow e'$, then $\Gamma \vdash_{HM} e' : \sigma$ and $e \sqsubseteq e'$.

2 If $\Gamma \vdash_{HM} e : \sigma$, then $\Gamma \vdash e : \sigma \Rightarrow e$.

Proof. Both are by induction on derivations. We only consider the first for the case that the last rule is (app_{ac}). By induction hypothesis,

$$\begin{aligned} \Gamma &\ \vdash_{HM}\ e_1' : \sigma \rightarrow \tau \\ \Gamma &\ \vdash_{HM}\ e_2' : \sigma_0 \end{aligned}$$

$$\frac{}{x \sqsubseteq x} \qquad \frac{e_1 \sqsubseteq e_2}{\lambda x.e_1 \sqsubseteq \lambda x.e_2} \qquad \frac{e_1 \sqsubseteq e_3 \quad e_2 \sqsubseteq e_4}{let\ x = e_1\ in\ e_2 \sqsubseteq let\ x = e_3\ in\ e_4}$$

$$\frac{}{e_1e_2 \sqsubseteq e_1(ee_2)} \qquad \frac{}{e_1e_2 \sqsubseteq (ee_1)e_2} \qquad \frac{e_1 \sqsubseteq e_3 \quad e_2 \sqsubseteq e_4}{e_1e_2 \sqsubseteq e_3e_4}$$

$$\frac{e_1 \sqsubseteq e_2 \quad e_2 \sqsubseteq e_3}{e_1 \sqsubseteq e_3}$$

Fig. 2. Rules for term expansion

Since $\sigma_0 \xrightarrow{c} \sigma$, we have that c is of type $\sigma_0 \to \sigma$ (its preassumption). Therefore, $\Gamma \vdash_{HM} ce_2' : \sigma$ and hence $\Gamma \vdash_{HM} e_1'(ce_2') : \tau$. Also, we have $e_1e_2 \sqsubseteq e_1(ce_2)$, and with transitivity, $e_1e_2 \sqsubseteq e_1'(ce_2')$. $\square$

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 the side condition $\Gamma \not\vdash_{HM} e_1'e_2' :?$, we mean that $e_1'e_2'$ is not typable in Γ in the base language, i.e., there is no type τ such that $\Gamma \vdash_{HM} e_1'e_2' : \tau$. This means that the original HM typing has a priority over its extensions with coercions in the sense that, if a term is typable in the HM system, then no coercion should be inserted into the term, for such insertions of coercions may change the meaning of the term. Such ambiguities of meaning are explained in the following example, where we use the construct of pairs, whose rule in the base language is given on page 8 and whose rule in our extended language would be:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \Rightarrow e_1' \quad \Gamma \vdash e_2 : \sigma_2 \Rightarrow e_2'}{\Gamma \vdash (e_1, e_2) : \sigma_1 \times \sigma_2 \Rightarrow (e_1', e_2')}$$

Example 3.4. We assume that A and B are base types inhabited by the constants $a : A$ and $b_1, b_2 : B$, that fst and $swap$ are defined as

$$fst = \lambda(x, y).x \quad \texttt{and} \quad swap = \lambda(x, y).(y, x),$$

and that the following term

$$c = \lambda(x, y).(b_1, b_2) : A \times A \to B \times B$$

is declared as a coercion from $A \times A$ to $B \times B$. We can then derive

$$\begin{aligned} fst &: B \times B \to B \Rightarrow fst \\ swap &: A \times A \to A \times A \Rightarrow swap \\ swap &: B \times B \to B \times B \Rightarrow swap \end{aligned}$$

Thus, using (app_{ac}) without the side condition, we would derive the following judgements:

$$fst(swap(a, a)) \quad : \quad B \Rightarrow fst(swap(c(a, a)))$$

$$fst(swap(a,a)) \quad : \quad B \Rightarrow fst(c(swap(a,a)))$$

However, $fst(swap(c(a,a)))$ computes to b_2 while $fst(c(swap(a,a)))$ to b_1; in other words, $fst(swap(a,a))$, typable already in HM (and computes to a), now can evaluate to different values! $\square$

The side condition prevents the above unpleasant ambiguity for terms already typable in HM, 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 the traditional algorithm $\mathcal{W}$. This side condition does not prevent all forms of ambiguities, however. See Section 5.2 for more discussions.

The above example shows an essential difference between our current setting of polymorphic typing and that of coercive subtyping in dependent type theories with its unique and explicit typing (in particular, the type of *swap* 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 dependent type theories, the question arises whether identity coercions (i.e., the identity functions declared as coercions) 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.

Generalisation and substitution lemmas. First, in the typing system, derivability is monotone w.r.t. the generalisation relation, as the following lemma shows.

Lemma 3.5 (generalisation lemma). Let Γ and Γ' be contexts such that $dom(\Gamma) = dom(\Gamma')$ and $\Gamma'(x) \geq \Gamma(x)$ for all $x \in dom(\Gamma)$. Then, $\Gamma \vdash e : \sigma \Rightarrow e'$ implies that $\Gamma' \vdash e : \sigma \Rightarrow e'$. $\square$

The derivable judgements are stable with respect to substitutions, as the following lemma shows.

Lemma 3.6 (substitution lemma). If $\Gamma \vdash e : \tau \Rightarrow e'$, then $S\Gamma \vdash e : S\tau \Rightarrow e'$, where S is a type substitution.

Proof. By induction on derivations. For the rule (var), use the fact that the relation $\preceq$ respects substitutions. In the following, we give the proof for the case of (let), i.e., $e \equiv let\ x = e_1\ in\ e_2$ and the last rule is (let) in Figure 1:

$$(let) \qquad \frac{\Gamma \vdash e_1 : \sigma \Rightarrow e_1' \quad \Gamma, x : \mathcal{C}_\Gamma(\sigma) \vdash e_2 : \tau \Rightarrow e_2'}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau \Rightarrow let\ x = e_1'\ in\ e_2'}$$

By definition, $\mathcal{C}_\Gamma(\sigma) = \forall\bar{\alpha}.\sigma$, where $\bar{\alpha} = \{\alpha_1, ..., \alpha_n\} = FTV(\sigma) \setminus FTV(\Gamma)$. Let $\beta_1, ..., \beta_n$ be type variables out of reach for S and not free in Γ and let

$$T = S \circ [\beta_1/\alpha_1, ..., \beta_n/\alpha_n].$$

Applying the induction hypothesis to the premises of the (*let*)-rule, with T to the left premise and S to the right premise, we get

$$T\Gamma \vdash e_1 : T\sigma \Rightarrow e_1' \tag{1}$$

$$S\Gamma, x : S\mathcal{C}_\Gamma(\sigma) \vdash e_2 : S\tau \Rightarrow e_2' \tag{2}$$

Since $T\Gamma \equiv S\Gamma$, because α_i is not free in Γ, we have

$$S\Gamma \vdash e_1 : T\sigma \Rightarrow e_1' \tag{3}$$

Now, we show that

$$FTV(T\sigma) \setminus FTV(T\Gamma) = \bar{\beta} \tag{4}$$

where $\bar{\beta} = \{\beta_1, ..., \beta_n\}$.

— ($\subseteq$) Let $\gamma \in FTV(T\sigma) \setminus \bar{\beta}$ and $\alpha \in FTV(\sigma)$ such that $\gamma \in FTV(T\alpha)$. Then, $\alpha \notin \bar{\alpha} = FTV(\sigma) \setminus FTV(\Gamma)$, for otherwise γ would be in $\bar{\beta}$. Therefore, $\alpha \in FTV(\Gamma)$ and $\gamma \in \bigcup_{\alpha \in FTV(\Gamma)} FTV(T\alpha) = FTV(T\Gamma)$.

— ($\supseteq$) By the definition of T, for any $\alpha_i \in \bar{\alpha}$, $T\alpha_i \equiv S\beta_i \equiv \beta_i$. Furthermore, for any $\alpha \notin \bar{\alpha}$, $T\alpha \equiv S\alpha$ does not contain any variable in $\bar{\beta}$. Since the variables in $\bar{\alpha}$ are free in σ but not in Γ, we have $\beta_i \in \bigcup_{\alpha \in FTV(\sigma)} FTV(T\alpha) = FTV(T\sigma)$ and $\beta_i \notin \bigcup_{\alpha \in FTV(\Gamma)} FTV(T\alpha) = FTV(T\Gamma)$.

From the above, we have

$$\mathcal{C}_{S\Gamma}(T\sigma) = S\mathcal{C}_\Gamma(\sigma) \tag{5}$$

because

$$\begin{array}{lll}
 & \mathcal{C}_{S\Gamma}(T\sigma) & \\
= & \mathcal{C}_{T\Gamma}(T\sigma) & (\textit{because } T\Gamma \equiv S\Gamma) \\
= & \forall\bar{\beta}.T\sigma & (\textit{by } (4) \textit{ and definition of closure}) \\
= & S(\forall\bar{\alpha}.\sigma) & (\textit{by definition of } T \textit{ and that } \beta_i \textit{ is out of reach for } S) \\
= & S\mathcal{C}_\Gamma(\sigma) & (\textit{definition of closure})
\end{array}$$

Now, from (2) and (5), we have

$$S\Gamma, x : \mathcal{C}_{S\Gamma}(T\sigma) \vdash e_2 : S\tau \Rightarrow e_2' \tag{6}$$

and, from (3), (6) and rule (*let*), we have the required result:

$$S\Gamma \vdash let\ x = e_1\ in\ e_2 : S\tau \Rightarrow let\ x = e_1'\ in\ e_2'.$$

□

4. Type inference: a sound and complete algorithm

The previous section describes our type system which adds coercions to the HM system. The rules of HM$_c$ describe well-typing, but they do not provide a decision procedure to verify well-typedness. This is mainly due to the application rules; for example, in (*app*), *Assumption, abstraction and let*

$$(Var) \quad \frac{\Gamma(x) = \forall\bar{\alpha}.\tau}{\Gamma \vdash x \rightsquigarrow \langle \emptyset, [\bar{\beta}/\bar{\alpha}]\tau, x\rangle} \quad (\bar{\beta}\ fresh)$$

$$(\Lambda) \quad \frac{\Gamma, x:\alpha \vdash e \rightsquigarrow \langle S, \tau, e'\rangle}{\Gamma \vdash \lambda x.e \rightsquigarrow \langle S, S\alpha \to \tau, \lambda x.e'\rangle} \quad (\alpha\ fresh)$$

$$(Let) \quad \frac{\Gamma \vdash e_1 \rightsquigarrow \langle S_1, \tau_1, e_1'\rangle \quad S_1\Gamma, x : \mathcal{C}_{S_1\Gamma}(\tau_1) \vdash e_2 \rightsquigarrow \langle S_2, \tau_2, e_2'\rangle}{\Gamma \vdash let\ x = e_1\ in\ e_2 \rightsquigarrow \langle S_2S_1, \tau_2, let\ x = e_1'\ in\ e_2'\rangle}$$

Applications

$$(App) \quad \frac{\Gamma \vdash e_1 \rightsquigarrow \langle S_1, \tau_1, e_1'\rangle \quad S_1\Gamma \vdash e_2 \rightsquigarrow \langle S_2, \tau_2, e_2'\rangle \quad T = \mathcal{U}(S_2\tau_1, \tau_2 \to \alpha)}{\Gamma \vdash e_1e_2 \rightsquigarrow \langle TS_2S_1, T\alpha, e_1'e_2'\rangle} \quad (\alpha\ fresh)$$

$$(App_{ac}) \quad \frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow \langle S_1, \tau_1, e_1'\rangle \quad S_1\Gamma \vdash e_2 \rightsquigarrow \langle S_2, \tau_2, e_2'\rangle \quad \mathcal{U}(S_2\tau_1, \tau_2 \to \beta)\uparrow \\ \sigma_1 \xrightarrow{c} \sigma_2 \in \mathcal{C} \quad T = \mathcal{U}(\tau_2 \to S_2\tau_1, \sigma_1 \to \sigma_2 \to \alpha)\end{array}}{\Gamma \vdash e_1e_2 \rightsquigarrow \langle TS_2S_1, T\alpha, e_1'(ce_2')\rangle} \quad (\alpha, \beta\ fresh)$$

$$(App_{fc}) \quad \frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow \langle S_1, \tau_1, e_1'\rangle \quad S_1\Gamma \vdash e_2 \rightsquigarrow \langle S_2, \tau_2, e_2'\rangle \quad \mathcal{U}(S_2\tau_1, \tau_2 \to \beta)\uparrow \\ \sigma_1 \xrightarrow{c} \sigma_2 \in \mathcal{C} \quad T = \mathcal{U}(S_2\tau_1 \to \tau_2 \to \alpha, \sigma_1 \to \sigma_2)\end{array}}{\Gamma \vdash e_1e_2 \rightsquigarrow \langle TS_2S_1, T\alpha, (ce_1')e_2'\rangle} \quad (\alpha, \beta\ fresh)$$

Fig. 3. Algorithm $\mathcal{W}_c$

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.

We shall present a type inference algorithm and prove that it is sound and complete with respect to the typing system HM$_c$.

4.1. *The type inference algorithm*

The rules of the type inference algorithm, $\mathcal{W}_c$, are given in Figure 3, where $\mathcal{C}$ is the finite set of initial coercions under consideration, $\mathcal{U}(\sigma, \tau)$ stands for the result (the most general unifier of σ and τ) of the standard unification algorithm [§] and $\mathcal{U}(\sigma, \tau) \uparrow$ for "the unification algorithm fails (and hence the most general unifier is undefined)".

The rules in Figure 3 can be read as an algorithm, giving non-deterministic answers to the question:

Given context Γ and term e, what are the type and completion of e?

The judgement

$$\Gamma \vdash e \rightsquigarrow \langle S, \tau, e'\rangle$$

can be read as "in context Γ, term e type checks with substitution S, type τ and comple-

[§] The existence of a unification algorithm was first shown to be the case in (Robinson 1965).

tion e'". The algorithm $\mathcal{W}_c$ extends the algorithm $\mathcal{W}$ (Damas and Milner 1982). It takes as inputs a context Γ ("initial context") and a term e and returns as the outputs a (most general) type τ, a completion e' of e, and a substitution S (representing the necessary instantiations for the typing to succeed).

Here are some informal explanations.

Some simple properties It is easy to see that the algorithm $\mathcal{W}_c$ has the following properties:

— *Recovery of the algorithm $\mathcal{W}$.* The traditional algorithm $\mathcal{W}$ can be recovered from the rules in Figure 3 by removing (App_{ac}) and (App_{fc}).
— *Non-determinism and decidability.* $\mathcal{W}_c$ is non-deterministic because multiple coercions can be found for a given pair of types. However, note that only finitely many initial coercions are declared and, therefore, the problem of finding initial coercions is decidable.

Explanations of the rules (App_{ac}) and (App_{fc}). These two rules play the role of deciding whether and how to insert coercions. First of all, if the application of the completions of e_1 and e_2 (i.e., $e_1'e_2'$) is already typable in the original HM system, no coercion will be inserted. This is made sure by the premise $\mathcal{U}(S_2\tau_1, \tau_2 \to \beta) \uparrow$ (the unification result is undefined), where β is a fresh type variable, and corresponds to the side-condition of the rules (app_{ac}) and (app_{fc}) in HM$_c$. Note that the side condition of those rules in HM$_c$ refers to the separate system of HM typing, while the implementation $\mathcal{W}_c$ uses a simple unification test and does not need to refer to a separate type checking algorithm.

When the unification of $S_2\tau_1$ and $\tau_2 \to \beta$ fails, which means that the application is not typable without inserting coercions, the rules (App_{ac}) and (App_{fc}) can be used to decide whether coercions exist for insertion to make the application e_1e_2 typable. In other words, they are used to find out whether there exists any argument coercion c or function coercion c' such that $e_1'(ce_2')$ or $(c'e_1')e_2'$ becomes typable, where e_i' is the completion of e_i $(i = 1, 2)$.

The existence of the coercions that can be inserted is tested by unification. For example, in (App_{ac}), to see whether there exist argument coercions to be inserted, the initial coercions $\sigma_1 \xrightarrow{c} \sigma_2 \in \mathcal{C}$ are considered in turn to see whether $\sigma_1 \to \sigma_2 \to \alpha$ can be unified with $\tau_2 \to S_2\tau_1$, where α is a fresh type variable. This has two effects. One is to test whether τ_2, the inferred type of e_2, is unifiable with σ_1, the domain of the coercion. The other is to test whether $S_2\tau_1$, the type of e_1, is unifiable with any arrow type with domain σ_2. If the unification succeeds, the coercion is insertable. Take the Monad example on page 6 in Section 2, where the following coercion is declared:

$$\gamma \xrightarrow{\texttt{return}} \mathtt{T}\gamma.$$

To type the application fa, where

$$\begin{aligned} f &= \texttt{if (iszero } x\texttt{) err} \\ a &= \texttt{sysdiv } 1.0\ x, \end{aligned}$$

the unification between $\texttt{Float} \to \texttt{T}\alpha' \to \texttt{T}\alpha'$ and $\gamma \to \texttt{T}\gamma \to \alpha$ would produce the unifier

$$[\texttt{T Float}/\alpha,\ \texttt{Float}/\alpha',\ \texttt{Float}/\gamma].$$

Therefore, the term fa is well-typed with type $\texttt{Float}$ according to (App_{ac}).

This is similar for function coercions. In (App_{fc}), unification tests are performed between $S_2\tau_1 \to \tau_2 \to \alpha$, for fresh α, and each of the arrow types $\sigma_1 \to \sigma_2$, for the initial coercions $\sigma_1 \xrightarrow{c} \sigma_2$, to decide whether coercions may be inserted as function coercions. This unification test has two effects as well: one is to see whether $S_2\tau_1$, the type of e_2, is unifiable with σ_1, the domain of the coercion, and the other whether the range σ_2 of the coercion is unifiable with an arrow type with domain being the inferred type of e_2 (i.e., τ_2). If both are the case, the coercion can be inserted as a function coercion.

When a unification succeeds with the most general unifier T, then T is incorporated into the resulting substitution and an inferred type of the application term e_1e_2 is $T\alpha$. Of course, if the unification tests all fail, there is no coercion that is insertable.

4.2. *Soundness and completeness*

The type inference algorithm $\mathcal{W}_c$ is a sound and complete implementation of HM$_c$. The following soundness theorem shows that $\mathcal{W}_c$ is correct.

Theorem 4.1 (Soundness). The algorithm $\mathcal{W}_c$ is sound w.r.t. HM$_c$, i.e., if $\Gamma \vdash e \rightsquigarrow \langle S, \tau, e' \rangle$, then $S\Gamma \vdash e : \tau \Rightarrow e'$.

Proof. By induction on derivations in $\mathcal{W}_c$. We consider the cases where the last rule is (Var), (Let) or (App_{ac}), and the others are similar.

— (Var). As $[\bar{\beta}/\bar{\alpha}]\tau \preceq \forall\bar{\alpha}.\tau$, we have $\Gamma \vdash x : [\bar{\beta}/\bar{\alpha}]\tau \Rightarrow x$, by rule ($var$) in HM$_c$.

— (Let). By induction hypothesis, $S_1\Gamma \vdash e_1 : \tau_1 \Rightarrow e_1'$ and $S_2S_1\Gamma,\ S_2\mathcal{C}_{S_1\Gamma}(\tau_1) \vdash e_2 : \tau_2 \Rightarrow e_2'$. Note that $S_2\mathcal{C}_{S_1\Gamma}(\tau_1) = \mathcal{C}_{S_2S_1\Gamma}(S_2\tau_1)$.¶ Therefore, $S_2S_1\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau_2 \Rightarrow let\ x = e_1'\ in\ e_2'$, by rule ($let$) in HM$_c$.

— (App_{ac}). By induction hypothesis,

$$S_1\Gamma \ \vdash\ e_1 : \tau_1 \Rightarrow e_1' \tag{7}$$
$$S_2S_1\Gamma \ \vdash\ e_2 : \tau_2 \Rightarrow e_2' \tag{8}$$

By Lemma 3.6,

$$TS_2S_1\Gamma \ \vdash\ e_1 : TS_2\tau_1 \Rightarrow e_1' \tag{9}$$
$$TS_2S_1\Gamma \ \vdash\ e_2 : T\tau_2 \Rightarrow e_2' \tag{10}$$

where $T = \mathcal{U}(\tau_2 \to S_2\tau_1, \sigma_1 \to \sigma_2 \to \alpha)$, with α fresh. So, $TS_2\tau_1 = T\sigma_2 \to T\alpha$ and $T\tau_2 = T\sigma_1$; therefore, from (9) and (10),

$$TS_2S_1\Gamma \ \vdash\ e_1 : T\sigma_2 \to T\alpha \Rightarrow e_1' \tag{11}$$

¶ This is the case because, if required, we rename the generalised variables in the derivation of $S_1\Gamma \vdash e_1 : \tau_1 \Rightarrow e_1'$ so that they are out of reach for S_2. After such renaming if necessary, the equation can be proved as in the proof of the let-case for Lemma 3.6.

$$TS_2S_1\Gamma \quad \vdash \quad e_2 : T\sigma_1 \Rightarrow e'_2 \tag{12}$$

Since $\sigma_1 \xrightarrow{c} \sigma_2$, we have by rule $(*)$ of coercions on page 8,

$$T\sigma_1 \xrightarrow{c} T\sigma_2 \tag{13}$$

Therefore, from (11,12,13) and by rule (app_{ac}) in HM$_c$, whose side condition is satisfied as $\mathcal{U}(S_2\tau_2, \tau_2 \rightarrow \beta)$ fails, we have

$$TS_2S_1\Gamma \vdash e_1e_2 : T\alpha \Rightarrow e'_1(ce'_2),$$

as required. □

The completeness theorem below shows that the algorithm $\mathcal{W}_c$ is complete with respect to HM$_c$. It uses the following definition of equalities between substitutions when applied to the variables outside a set of type variables V, from which the fresh variables are taken.

Definition 4.2. Let V be an infinite set of type variables and S and T substitutions. Then, S *is equal to* T *outside* V, notation $S =_{\bar{V}} T$, if $S(\alpha) = T(\alpha)$ for all $\alpha \notin V$. □

Theorem 4.3 (Completeness). For any context Γ and any infinite set of type variables V such that $FTV(\Gamma) \cap V = \emptyset$, if $S\Gamma \vdash e : \tau \Rightarrow e'$, then there exist substitution S' and type τ' such that $\Gamma \vdash e \rightsquigarrow \langle S', \tau', e' \rangle$, and furthermore, there exists a substitution U such that $\tau = U\tau'$ and $S =_{\bar{V}} US'$.

Proof. By structural induction on e, considering the derivations of $S\Gamma \vdash e : \tau \Rightarrow e'$. We only consider three cases here.

— $e \equiv x$ and the last rule is (var): $S\Gamma \vdash x : \tau \Rightarrow x$, where $\tau \preceq S\Gamma(x)$. Since x occurs in $S\Gamma$, so does it in Γ. Let $\Gamma(x) = \forall\bar{\alpha}.\sigma$, where the variables in $\bar{\alpha}$ are chosen from V and out of reach for S. By (Var) in $\mathcal{W}_c$, $\Gamma \vdash x \rightsquigarrow \langle \emptyset, [\bar{\beta}/\bar{\alpha}]\sigma, x \rangle$, where the fresh variables in $\bar{\beta}$ are chosen from $V \setminus \bar{\alpha}$. Take

$$U = TS[\bar{\alpha}/\bar{\beta}],$$

where T is a substitution over $\bar{\alpha}$ such that $\tau = TS\sigma$ (T exists because $\tau \preceq S\Gamma(x) = \forall\bar{\alpha}.S\sigma$). Then $\tau = TS\sigma = U[\bar{\beta}/\bar{\alpha}]\sigma$ and, for $\gamma \notin V$, $S(\gamma) = TS[\bar{\alpha}/\bar{\beta}](\gamma) = U(\gamma) = U\emptyset(\gamma)$ (because $\gamma \notin \bar{\alpha} \cup \bar{\beta}$ and T is over variables out of reach of S).

— $e \equiv let\ x = e_1\ in\ e_2$ and the last rule is:

$$(let) \qquad \frac{S\Gamma \vdash e_1 : \sigma \Rightarrow e'_1 \quad S\Gamma, x : \mathcal{C}_{S\Gamma}(\sigma) \vdash e_2 : \tau \Rightarrow e'_2}{S\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau \Rightarrow let\ x = e'_1\ in\ e'_2}$$

By induction hypothesis, applying induction to e_1 (and Γ, σ and S), we get, for some S_1, τ_1 and U_1,

$$\Gamma \vdash e_1 \rightsquigarrow \langle S_1, \tau_1, e'_1 \rangle \tag{14}$$

$$U_1(\tau_1) = \sigma \ \ \texttt{and} \ \ S =_{\bar{V}} U_1S_1 \tag{15}$$

where the fresh variables in deriving (14) are taken from V, resulting in $V_1 \subseteq V$. In particular, $S\Gamma = U_1S_1\Gamma$. It is easy to check that $U_1(\mathcal{C}_{S_1\Gamma}(\tau_1)) \geq \mathcal{C}_{U_1S_1\Gamma}(U_1(\tau_1)) = \mathcal{C}_{S\Gamma}(\sigma)$. From Lemma 3.5 and the second premise, we can derive

$$S\Gamma, x : U_1(\mathcal{C}_{S_1\Gamma}(\tau_1)) \vdash e_2 : \tau \Rightarrow e_2',$$

which is

$$U_1(S_1\Gamma, x : \mathcal{C}_{S_1\Gamma}(\tau_1)) \vdash e_2 : \tau \Rightarrow e_2'.$$

By induction hypothesis, applying the induction to e_2 (and $S_1\Gamma, x : \mathcal{C}_{S_1\Gamma}(\tau_1)$, τ, U_1 and V_1), we obtain, for some S_2, τ_2 and U_2,

$$S_1\Gamma, x : \mathcal{C}_{S_1\Gamma}(\tau_1) \vdash e_2 \leadsto \langle S_2, \tau_2, e_2' \rangle \tag{16}$$

$$U_2(\tau_2) = \tau \ \ \texttt{and} \ \ U_1 =_{\bar{V}_1} U_2S_2 \tag{17}$$

where the fresh variables in deriving (16) are taken from V_1. Therefore, by (14,16) and rule (*Let*),

$$\Gamma \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \leadsto \langle S_2S_1, \tau_2, \mathit{let}\ x = e_1'\ \mathit{in}\ e_2' \rangle.$$

Now, let $U = U_2$ and we then have $U\tau_2 = U_2\tau_2 = \tau$ and, for $\gamma \notin V$, $S(\gamma) = U_1S_1(\gamma) = U_2S_2S_1(\gamma)$, where the last equality is true because $S_1(\gamma) \notin V_1$ (note that γ is out of reach for S_1). This concludes the proof of the case.

— $e \equiv e_1e_2$ and the last rule is:

$$(\mathit{app}_{ac}) \qquad \frac{S\Gamma \vdash e_1 : \sigma \to \tau \Rightarrow e_1' \quad S\Gamma \vdash e_2 : \sigma_0 \Rightarrow e_2' \quad \sigma_0 \xrightarrow{c} \sigma}{S\Gamma \vdash e_1e_2 : \tau \Rightarrow e_1'(ce_2')} \quad (S\Gamma \not\vdash_{HM} e_1'e_2' :?)$$

By induction hypothesis, applying the induction to e_1 (and Γ, $\sigma \to \tau$, S and V), we have, for some S_1, τ_1 and U_1,

$$\Gamma \vdash e_1 \leadsto \langle S_1, \tau_1, e_1' \rangle \tag{18}$$

$$U_1(\tau_1) = \sigma \to \tau \ \ \texttt{and} \ \ S =_{\bar{V}} U_1S_1 \tag{19}$$

where the fresh variables in deriving (18) are taken from V, resulting in $V_1 \subseteq V$. Also by induction hypothesis, applying the induction to e_2 (and $S_1\Gamma$, σ_0, U_1 and V_1),[‖] we have, for some S_2, τ_2 and U_2,

$$S_1\Gamma \vdash e_2 \leadsto \langle S_2, \tau_2, e_2' \rangle \tag{20}$$

$$U_2(\tau_2) = \sigma_0 \ \ \texttt{and} \ \ U_1 =_{\bar{V}_1} U_2S_2 \tag{21}$$

where the fresh variables in deriving (20) are taken from V_1.

We also know that $\mathcal{U}(S_2\tau_1, \tau_2 \to \beta)$ fails, where β is a fresh type variable, for otherwise $e_1'e_2'$ would be typable in $S\Gamma = U_2S_2S_1\Gamma$ in the HM system, contradicting the side-condition of rule (app_{ac}). Furthermore, for any fresh type variable α, $U_2' = U_2[\tau/\alpha]$ unifies $\tau_2 \to S_2\tau_1$ and $\sigma_0 \to \sigma \to \alpha$ and, therefore, there is a most general unifier T:

$$T = \mathcal{U}(\tau_2 \to S_2\tau_1, \sigma_0 \to \sigma \to \alpha). \tag{22}$$

‖ Note that, because $S\Gamma = U_1S_1\Gamma$ by (19) as $FTV(\Gamma) \cap V = \emptyset$, the second premise of (app_{ac}) is actually $U_1S_1\Gamma \vdash e_2 : \sigma_0 \Rightarrow e_2'$. Therefore, the induction step goes through.

Because $\sigma_0 \xrightarrow{c} \sigma$, we have, from (18,20,22) and by rule (App_{ac}) in $\mathcal{W}_c$,

$$\Gamma \vdash e_1 e_2 \rightsquigarrow \langle TS_2S_1, T\alpha, e_1'(ce_2')\rangle.$$

We now only have to show that there exists a substitution U such that

$$\tau = UT\alpha$$
$$S =_{\bar{V}} UTS_2S_1$$

Let U be a substitution such that $U_2' = UT$ (U exists because T is more general than U_2'). Then, $\tau = U_2'\alpha = UT\alpha$ and, for any $\gamma \notin V$, $S(\gamma) = U_2'S_2S_1(\gamma) = UTS_2S_1(\gamma)$. This concludes the proof of the case. □

5. Discussions: further extensions and ambiguity

The system considered here can be extended further to become more powerful. Such an extension is considered below. We shall also discuss the issue of ambiguity when coercions are introduced.

5.1. *Simultaneous insertions of argument and function coercions*

Argument coercions and function coercions may be inserted simultaneously to make some term typable. For instance, assume that `n` be a natural number (of type `Nat`) and that we have the following coercions:

$$\texttt{Nat} \xrightarrow{\texttt{n2i}} \texttt{Int}$$
$$\texttt{Plus} \xrightarrow{\lambda x.\texttt{plusi}} \texttt{Int} \to \texttt{Int} \to \texttt{Int},$$

where, as in Section 2, `Plus` is the unit type with the only object `plus` and `plusi` is the plus function of integers. Now, consider the following term

$$\texttt{plus n}.$$

Of course, this term is not well-typed in the HM system. It is not well-typed in the above extension HM$_c$, either! The reason is that the rules in HM$_c$ do not allow *simultaneous* insertions of argument and function coercions to make untyped terms typable. Otherwise, we would be able to insert `n2i` and $\lambda x.$`plusi` simultaneously to complete the term as

$$((\lambda x.\texttt{plusi})\ \texttt{plus})\ (\texttt{n2i n}),$$

and `plus n` would become well-typed.

It is possible to extend the system described above by allowing simultaneous insertions of an argument coercion and a function coercion. The system HM$_c$ can be extended by the following rule:

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \Rightarrow e_1' \quad \Gamma \vdash e_2 : \sigma_0 \Rightarrow e_2' \quad \sigma_0 \xrightarrow{c} \sigma \quad \rho \xrightarrow{c'} \sigma \to \tau}{\Gamma \vdash e_1 e_2 : \tau \Rightarrow (c'e_1')(ce_2')} \quad (\Gamma \not\vdash_{HM} e_1'e_2' :?)$$

With this rule, simultaneous insertions of argument and function coercions are possible and, for instance in the example above, the term `plus n` is of type `Int` $\to$ `Int` with $((\lambda x.$`plusi`$)$ `plus`$)$ $($`n2i n`$)$ as its completion.

Correspondingly, the type inference algorithm $\mathcal{W}_c$ can be extended by the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \leadsto \langle S_1, \tau_1, e_1' \rangle \quad S_1\Gamma \vdash e_2 \leadsto \langle S_2, \tau_2, e_2' \rangle \quad \mathcal{U}(S_2\tau_1, \tau_2 \to \beta) \uparrow \\ \sigma_1 \xrightarrow{c} \sigma_2 \in \mathcal{C} \quad \sigma_1' \xrightarrow{c'} \sigma_2' \in \mathcal{C} \quad T = \mathcal{U}(\tau_2 \to S_2\tau_1 \to \sigma_2', \sigma_1 \to \sigma_1' \to \sigma_2 \to \alpha)\end{array}}{\Gamma \vdash e_1e_2 \leadsto \langle TS_2S_1, T\alpha, (c'e_1')(ce_2') \rangle} \quad (\alpha, \beta\ fresh)$$

where the unification between $\tau_2 \to S_2\tau_1 \to \sigma_2'$ and $\sigma_1 \to \sigma_1' \to \sigma_2 \to \alpha$ plays the following roles:

1 to test whether the inferred type of e_2 (i.e., τ_2) can be unified with the domain of the coercion c (i.e., σ_1);

2 to test whether the inferred type of e_1 (i.e., $S_2\tau_1$) can be unified with the domain of the coercion c' (i.e., σ_1');

3 to test whether the range of the coercion c' (i.e., σ_2') is an arrow type with domain being the range of the coercion c (i.e., σ_2).

If all of the above are the case with a most general unifier T, then the inferred type of e_1e_2 is $T\alpha$ with completion $(c'e_1')(ce_2')$, where both coercions c and c' have been inserted.

With the above extensions, the type inference algorithm is still sound and complete with respect to the extended type system; in other words, Theorems 4.1 and 4.3 still hold with the above extensions.

5.2. *Ambiguity*

With extensions of coercions, the typing system is not deterministic anymore. The rules for HM$_c$ and the corresponding algorithm $\mathcal{W}_c$ allow certain ambiguities. The side condition of the HM$_c$ rules (app_{ac}) and (app_{fc}) removes the ambiguity of the terms that are already typable in the original HM system. However, ambiguities may exist for those terms only typable after insertion of some coercions, when there is more than one matching coercion that may be inserted to complete a term. The following are two examples of such an ambiguity.

Example 5.1 (ambiguity). Here are two examples of derivational ambiguity.

1 This example shows that there may be more than one argument coercion that can be inserted in the same position of a term. Assume, for instance, that A and B are base types, $a : A$ and $f : \alpha \times \alpha \to \alpha$ and that

$$A \xrightarrow{c_1} A \times A \quad \texttt{and} \quad A \xrightarrow{c_2} B \times B.$$

Then, we have, ambiguously,

$$\begin{array}{rcl} fa & : & A \Rightarrow f(c_1a) \\ fa & : & B \Rightarrow f(c_2a) \end{array}$$

or equivalently,

$$fa \quad \leadsto \quad \langle \emptyset, A, f(c_1a) \rangle$$

$$fa \quad \rightsquigarrow \quad \langle \emptyset, B, f(c_2 a)\rangle$$

2 This example shows that there can be an argument coercion and a function coercion, both possible to be inserted to a term, causing ambiguities. Assume, for instance, that $e_1 : \sigma \to \tau$ and $e_2 : \sigma_0$ and that

$$\sigma_0 \xrightarrow{c} \sigma \quad \texttt{and} \quad (\sigma \to \tau) \xrightarrow{c'} (\sigma_0 \to \tau').$$

Now, we have in HM_c, ambiguously,

$$\begin{aligned} e_1 e_2 &\ :\ \tau \Rightarrow e_1(ce_2) \\ e_1 e_2 &\ :\ \tau' \Rightarrow (c' e_1) e_2 \end{aligned}$$

or equivalently, in $\mathcal{W}_c$,

$$\begin{aligned} e_1 e_2 &\ \rightsquigarrow\ \langle \emptyset, \tau, e_1(ce_2)\rangle \\ e_1 e_2 &\ \rightsquigarrow\ \langle \emptyset, \tau', (c' e_1) e_2\rangle \end{aligned}$$

Of course, here we have assumed that $\sigma_0 \neq \sigma$, for otherwise $e_1' e_2'$ would have been typable in HM and the coercions c_i could not be inserted. □

The derivational ambiguities as exemplified above are due to the fact that, in a single derivation of $\Gamma \vdash e : \sigma \Rightarrow e'$, there could be some subterm s of e such that s is assigned different completions. One may study those terms whose derivations do not contain more than one completion of a subterm, as considered in (Kießling and Luo 2004). More precisely, we call a term e *derivationally coherent* in context Γ if, for any derivation Δ of $\Gamma \vdash e : \sigma \Rightarrow e'$, if $\Gamma_1 \vdash s : \tau_1 \Rightarrow s_1'$ and $\Gamma_2 \vdash s : \tau_2 \Rightarrow s_2'$ both occur in Δ, then the two completions s_1' and s_2' are the same. With such a notion of derivational coherence, it may be possible to consider a sound and complete type inference algorithm for derivational coherent terms. Such an algorithm will not type those terms that admit ambiguous derivations. In particular, the unification algorithm has to be modified so that it only succeeds in one of the many cases, if at all. It is unclear to the author whether such a notion of derivational coherence would resolve the problem of ambiguity satisfactorily and further investigation is called for in this respect.

6. Conclusion

We have considered an extension of the Hindley-Milner type system with coercions to incorporate ideas of coercive subtyping into a polymorphic type system.

It is possible, as considered in (Kießling and Luo 2004), to incorporate coercion declarations and local coercions. For example, we may consider the following local coercion

$$cdec\ c : \forall\bar{\alpha}.\sigma \to \tau\ in\ e$$

as a term. Such a mechanism may bring some flexibility in practice, but it causes complications in meta-theoretical studies, as mentioned earlier. For instance, the incorporation of local coercions of the above form introduces types into term expressions and complicates the meta-theoretic proofs considerably. In this paper, we have treated coercions more or less independently of the typing derivations and this has simplified the system, and in particular, the meta-theoretic proofs.

The coercions considered in this paper are well-typed terms in the HM system. In other words, coercions are terms without "gaps" and coercions cannot be inserted in to coercion terms. We point out that, on the one hand, it may be possible to lift this restriction to allow coercions to be terms only typable when some other coercion terms are inserted but, on the other hand, the requirement that coercions be terms without gaps can be considered reasonable since coercions are supposed to be hidden and not seen by the user.

There have been studies on how the HM-style typing may be linked to the constraint-based approach (Odersky, Sulzmann, and Wehr 1999; Pot05). It might be interesting to pursue this line to describe an extension with coercions by means of a constraint-based specification.

Coercive subtyping was developed as a framework of abbreviation for dependent type theories, where it is typical to have the property of unique typing, which is just the opposite of polymorphic typing. It is interesting to see to what extent and how type dependency and polymorphism can be combined to provide satisfactory language features for programming and proof development. The work presented here may be regarded as a step in this direction. It is clear from this paper that, to incorporate polymorphism, the lack of uniqueness of types may cause considerable difficulties. Further investigations are needed to have a better understanding in this respect.

Acknowledgement Thanks go to the referees for their comments that have helped to improve the paper.

References

A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1999.

V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. *Information and Computation*, 93, 1991.

T. Brus, M. van Eekelen, M. van Leer, and M. Plasmeijer. Clean: a language for functional graph rewriting. In G. Kahn, editor, *Functional Programming and Computer Architecture, LNCS'274*, pages 364–384, 1987.

G. Chen. *Subtyping, Type Conversion and Transitivity Elimination.* PhD thesis, University of Paris VII, 1998.

P. Callaghan and Z. Luo. An implementation of LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.0), INRIA*, 2004.

L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. of the 9th Ann. Symp. on Principles of Programming Languages*, pages 207–212, 1982.

R. Hindley. The princial type-scheme of an object in combinatory logic. *Trans. of the American Math Society*, 146:29–60, 1969.

R. Kießling and Z. Luo. Coercions in Hindley-Milner systems. In *Types for Proofs and Programs, Proc. of Inter. Conf. of TYPES'03, LNCS'3085.*, 2004.

Z. Luo and P. Callaghan. Coercive subtyping and lexical semantics (extended abstract). *Proc. of Logical Aspects of Computational Linguistics (LACL'98)*, 1998.

X. Leroy. Polymorphic typing of an algorithmic language. Tech Report RR-1778, INRIA, Rocquencourt, 1992. (English version of his PhD thesis at University of Paris 7.).

X. Leroy. *The Objective Caml system: documentation and user's manual.* 2000. Available `http://caml.inria.fr/`

Z. Luo and Y. Luo. Transitivity in coercive subtyping. *Information and Computation*, 197:122–144, 2005.

Y. Luo, Z. Luo, and S. Soloviev. Weak transitivity in coercive subtyping. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 220–239. Springer-Verlag, 2002.

G. Longo, K. Milsted, and S. Soloviev. Coherence and transitivity of subtyping as entailment. *Journal of Logic and Computation*, 10(4):493–526, 1999.

Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Dept of Computer Science, Univ of Edinburgh, 1992.

Z. Luo and S. Soloviev. Dependent coercions. *The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh, Scotland. Electronic Notes in Theoretical Computer Science*, 29, 1999.

Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

Z. Luo. Coercive subtyping in type theory. *Proc. of CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht. LNCS 1258*, 1997.

Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

Y. Luo. *Coherence and Transitivity in Coercive Subtyping.* PhD thesis, University of Durham, 2005.

R. Milner. A theory of type polymorphism in programming. *J. of Computer Systems and Sciences*, 17:348–375, 1978.

J. C. Mitchell. Coercion and type inference. In *Proc. of Tenth Annual Symposium on Principles of Programming Languages (POPL)*, 1983.

J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(2):245–286, 1991.

P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

R. Milner, M. Tofts, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999.

S. Peyton-Jones. *Haskell 98 Language and Libraries: the Revised Report.* Cambridge University Press, 2003.

S. Peyton-Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In *Proceedings of the Haskell Workshop June 1997, Amsterdam*, 1997.

F. Pottier. A modern eye on ML type inference. Lecture notes for the APPSEM Summer School, 2005.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

A. Saïbi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.

S. Soloviev and Z. Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 113(1-3):297–322, 2002.

C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, 2000.

P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, LNCS'925*, pages 24–52, 1995.