1991

Type reconstruction for coercion polymorphism (Technical summary)

Ryan Stansifer

Dan Wetklow

Report Number: 91-054


http://docs.lib.purdue.edu/cstech/894

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
TYPE RECONSTRUCTION FOR COERCION POLYMORPHISM (TECHNICAL SUMMARY)

Ryan Stansifer
Dan Wetklow

CSD-TR-91-054
July 1991
Type reconstruction for coercion polymorphism
(Technical summary)

Ryan Stausifer
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Dan Wetklow
Department of Computer Science
University of Pittsburgh at Johnstown
Johnstown, PA 15956

July 23, 1991

Abstract
In this paper we examine coercion between primitive (or unstructured) types. We assume that the collection of primitive types in a language can be arranged in some lattice, although in some cases a more general partial order is permitted. Type \( T_1 \leq T_2 \) if an element of type \( T_1 \) can be coerced into an element of type \( T_2 \). We show how type reconstruction for subtype polymorphism can be used to obtain coercion polymorphism.

The approach we use to subtype polymorphism is the one which embeds subtype polymorphism into parametric polymorphism. This paper illuminates the trade-offs involved in this approach by revealing some of the nature of the embedding.

1 Introduction
Recent results in type reconstruction for subtype polymorphism inspire a new approach to handling implicit coercions between primitive types. We call any type without substructure a primitive type. All languages have these types: int, bool, real, etc. Often, these primitive data types can be converted from one to another. For example, an integer value can be converted to a real value. Therefore, programmers are inclined to think that an integer is a real number, and should be permitted to be used anywhere a real number is expected. In this way the relationship between an integer and a real number is the same as the subtype relation in subtype polymorphism. While it is mathematically true that it is "type" correct to substitute an integer for a real number, it is not type correct in programming languages, because integers are not represented in the same manner that real number are. Integer values, however, can be coerced to real numbers by executing some operation to change their representation. So implicitly coercing values is different from subtype polymorphism and universal polymorphism, because operations, like addition, do not act uniformly on values of all the types.

The problem of type reconstruction in the presence of subtype relations is a difficult one. One difficulty is that all derivable types of an expression cannot be expressed as a single type in the type system. The usual approach to type reconstruction in the presence of coercion polymorphism is to rely on various ad-hoc methods. A more systematic solution, developed by Fuh and Mishra [2] generated a set of subtype assertions.
and then checked this set for consistency. Any type that did not create an inconsistency in the set of subtype assertions was a derivable type of the expression.

One approach due to Rémy [5, 6] is to capture some inclusion polymorphism in a multisorted type system. This technique has been used by Wand [9, 10] to model object-oriented languages. This approach to subtype polymorphism has the advantage of integrating well with parametric polymorphism and of making use of the same well-understood mechanisms and algorithms.

We translate the coercion problem to a setting with multisorted types designed to encode coercion polymorphism, and solve the translated version of the problem. The solution to the translated problem is then translated back into the original type system. In other words, given a type environment \( A \) and an expression \( e \) in the primitive type system,

1. translate \( A \) to a type environment in the other type system,
2. run the type reconstruction algorithm, \( \text{TypeOf} \), in the other type system, and
3. map the type back to the primitive type system.

While our approach has the advantage that an existing type reconstruction algorithm, we call \( \text{TypeOf} \), can be used, it must be shown that the solutions obtained using this approach are "correct." The criteria for correctness that we use is that for any type environment \( A \) and expression \( e \):

1. If a type for \( c \) cannot be derived in the primitive type system, then a type should not be derivable for \( e \) in the other type system.
2. If the multisorted type system succeeds in deriving a type \( \tau \) for \( e \) then \( I(\tau) \) must be derivable as a type of \( e \) in the primitive type system, where \( I \) is the function that translates types in the other type system to types in the primitive type system.
3. If given \( c \) and a translated version of \( A \), \( \text{TypeOf} \) succeeds in typing \( e \), then it must be possible to derive some type for \( e \) in the primitive type system.

While the types in the multisorted type system are more expressive than those in the primitive type system, the typing rules are not as powerful. The final result of this is that while some subtype polymorphism is obtained using this translation scheme, there are expressions that can be typed in the primitive type system that are not typable in the multisorted type system. Essentially, some power has been sacrificed for the simplicity approach using parametric polymorphism. The translation back and forth explains some of the reasons why embedding subtype polymorphism in parametric polymorphism fails to capture all subtype polymorphism.

2 Language and type systems

In this paper we will deal exclusively with the simple language of lambda expressions. The following table presents the language.
We will be interested in several type systems for this language. One system, which we call type system $p$, is the one we consider the most natural. Here is type system $p$:

$$
\tau ::= \alpha \mid \beta \mid \tau_1 \to \tau_2
$$

Functions have the usual function space types denoted using $\to$. Each primitive type is denoted by its own special constant represented by the metavariable $p$ in the grammar above. Universal polymorphism is achieved by allowing type variables, represented by the metavariable $\alpha$. We have in mind here Milner let-style polymorphism. The results reported here do carry through in this case, but we eschew complicating the presentation by the additional mechanisms that would be necessary.

Of central importance is the structure of the primitive types.

Definition 1 A subtype relation for the type system $p$ is a partial order between primitive types, denoted $\tau \leq_p \sigma$.

We assume that there is some particular subtype relation for a given system of primitive types, and that for every pair of primitive types the least upper bound and the greatest lower bound exist and are unique. In other words, the primitive types form a lattice. Later we show how this restriction can be relaxed somewhat.

We now introduce a collection of types with record types. The goal is to take advantage of the lattice structure formed by subtyping on records to solve the type reconstruction problem in the original type system $p$. We assume that records have some fixed number of available field labels known in advance, as we do not require anything more general. We first give the obvious type system.

$$
\tau ::= \alpha \mid \left\{ l_1 : \tau, \ldots, l_n : \tau \right\} \mid \tau_1 \to \tau_2
$$

For the most part we are only interested in records whose fields all have type $u$, a type with just one element. Just the presence or absence of a field will be important in this paper; we are not actually interested in records as data structures. Unfortunately we cannot do type reconstruction very well with this system [2, 3, 7, 8].

The type system $r$ is shown next. It is inspired by the work of Rémy [5, 6]. Type system $r$ has two syntactic categories, or sorts. The usual one for "types," and an additional sort for fields of records.

$$
\tau ::= \alpha \mid \left\{ l_1 : F_1, \ldots, l_n : F_n \right\} \mid \tau_1 \to \tau_2
$$

$$
F ::= \mathcal{X} \mid + \mid -
$$

Each record has exactly $n$ fields. The field value $+$ means that the field must be present in the record. $-$ means that the field is absent. Field variables, represented by the metavariable $\mathcal{X}$ in the grammar above, provide the opportunity for a record to gain or lose fields.
Definition 2 A substitution is a finite mapping of type variables to types.

In the case of the two-sorted type system $\tau$ a substitution may map both type variables and field variables to their respective sort. An instance of a type is the result of applying any substitution to a type. We denote a substitution $S$ in type system $\tau$ restricted to field variables by $S |_{\sigma}$. We extend, without comment or notional device, the application of substitutions to all types in general and also to type assignments (which will be defined shortly).

Definition 3 We define a partial order on types in the type system $\tau$ by $\tau \leq_\sigma \sigma$ if there exists a substitution $S$ such that $S |_{\sigma} (\tau) = \sigma$.

Definition 4 A type assignment is a mapping of expression variables to types.

Definition 5 A typing is a triple of the form $A \vdash e : \tau$ where $A$ is a type assignment, $e$ is an expression, and $\tau$ is a type.

Typings are used to capture the definition of "$e$ has type $\tau". The rules display in figures 1 and 2 are the rules to derive typing judgements in the respective type systems. The typing rules in system $p$ are more powerful than those of system $\tau$, because of the rule permitting the derivation of supertypes.

The following result about type system $p$ was proved by Cardelli [1]. It is important in proving the correctness of our approach.

Theorem 6 (Syntactic subtyping, Cardelli) For all expressions $e$, type assignments $A$, and types $\tau$, $\sigma$, and $\sigma'$, if $A \vdash e : \tau$ is derivable and $\sigma' \leq \sigma$, then $A [u \mapsto \sigma'] \vdash : \tau$ is derivable.

Type system $\tau$ has not been used before, but it is a generalization of the system studied by Milner [4] to the case of a multi-sorted algebra of type terms. All the nice properties of that system carry through to the multi-sorted case. So there is a type reconstruction algorithm for type system $\tau$. It is shown in figure 3. And the next two theorems hold as well.

Theorem 7 (Syntactic soundness) For all expressions $e$, type assignments $A$, and type $\rho$, if $\text{TypeOf}(A, e)$ succeeds with $(S, \rho)$, then $SA \vdash e : \rho$ is derivable from the typing rules.

Theorem 8 (Syntactic completeness) For all expressions $e$, type assignments $A$, and type $\rho$, $\rho'$ if $A \vdash e : \rho$ is derivable from the typing rules, then $\text{TypeOf}(A, e)$ succeeds with $(S, \rho')$, and there exists a substitution $T$ such that $T \rho' = \rho$.

3 Translation

In this section we define a translation from the primitive type system to the record type system. This translation preserves the lattice structure in type system $p$ by translating it to the subtype relation between records. Consider the lattice of primitive types shown in figure 4. It can be put in a one-to-one correspondence with records.

Definition 9 The translation of a primitive type $\tau$, denoted $T(\tau)$, is a record with $n$ fields, where $n$ is the number of primitive types in the lattice excluding $T$ and $\bot$. Each field corresponds to one of the types in the lattice excluding $T$ and $\bot$. The value of the field corresponding to $\sigma$ is given by the following rule:
\[
\frac{\Gamma \vdash p \ v : A(v)}{\text{if } v \in \text{Dom}(A)}
\]

\[
\frac{\Theta[u \mapsto \tau] \vdash p \ e : \tau_2}{\Gamma \vdash \lambda v. e' : \tau_1 \rightarrow \tau_2}
\]

\[
\frac{\Gamma \vdash p \ e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash p \ e_2 : \tau_2}{\Gamma \vdash p \ e_1(e_2) : \tau}
\]

\[
\frac{\Gamma \vdash p \ e_1 : \text{bool} \quad \Gamma \vdash p \ e_2 : \tau \quad \Gamma \vdash p \ e_3 : \tau}{\Gamma \vdash p \ \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 : \tau}
\]

\[
\frac{\Gamma \vdash p \ \sigma}{\Gamma \vdash p \ \tau \quad \text{if } \sigma \preceq \tau}
\]

Figure 1: The typing rules for the primitive type system.

\[
\frac{\Gamma \vdash r \ v : A(v)}{\text{if } v \in \text{Dom}(A)}
\]

\[
\frac{\Theta[u \mapsto \tau] \vdash r \ e : \tau_2}{\Gamma \vdash \lambda v. e' : \tau_1 \rightarrow \tau_2}
\]

\[
\frac{\Gamma \vdash r \ e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash r \ e_2 : \tau_2}{\Gamma \vdash r \ e_1(e_2) : \tau}
\]

\[
\frac{\Gamma \vdash r \ e_1 : \text{bool} \quad \Gamma \vdash r \ e_2 : \tau \quad \Gamma \vdash r \ e_3 : \tau}{\Gamma \vdash r \ \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 : \tau}
\]

Figure 2: The typing rules for the record type system.
\[ \text{TypeOf} (A, e) = \]
\[
\text{case } e \text{ of } x \\
\quad \text{if } v \in \text{Dom}(A) \text{ then } (\emptyset, A(v)) \text{ else fail } \\
\quad (\lambda x.e_1) \Rightarrow (\text{function definition}) \\
\] 
\[
\text{let } \\
\alpha \text{ be a new type variable } \\
(S_1, \tau_1) = \text{TypeOf} (A[x \rightarrow \alpha], e_1) \\
\text{in } \\
(S_1, S_1 \alpha \rightarrow \tau_1) \\
\text{end} \\
\] 
\[
e_1(e_2) \Rightarrow (\text{function application}) \\
\text{let } \\
(S_1, \tau_1) = \text{TypeOf} (A, e_1) \\
(S_2, \tau_2) = \text{TypeOf} (S_1 A, e_2) \\
\alpha \text{ be a new type variable } \\
U = \text{Unify} (S_2 \tau_1, \tau_2 \rightarrow \alpha) \\
\text{in } \\
(U S_2 \tau_1, U \alpha) \\
\text{end} \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow (\text{conditional}) \\
(\ast \ast \ast \ast)
\]

Figure 3: Type reconstruction algorithm.

If \( \tau \leq \sigma \) then the field is a new field variable \( \mathcal{X} \). Otherwise, the field is \(-\).

**Definition 10** The dual translation of a primitive type \( \tau \), denoted \( T^D(\tau) \), is a record with \( n \) fields, where \( n \) is the number of primitive types in the lattice excluding \( \top \) and \( \bot \). Each field corresponds to one of the types in the lattice excluding \( \top \) and \( \bot \). The value of the field corresponding to \( \sigma \) is given by the following rule:

If \( \tau \leq \sigma \) then the field is a \(+\). Otherwise, the field is a new field variable \( \mathcal{X} \).

The following table illustrates the translation for the lattice depicted in figure 4. If \( T(\tau) \)'s \( \sigma \) field is a \(+\), then \( \tau \leq \sigma \). If its \( \sigma \) field is a \(-\), then it is not the case that \( \tau \leq \sigma \).

<table>
<thead>
<tr>
<th>( \tau )</th>
<th>( T(\tau) )</th>
<th>( T^D(\tau) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \top )</td>
<td>( { } )</td>
<td>( { a : - , b : - , c : - , d : - } )</td>
</tr>
<tr>
<td>( a )</td>
<td>( { a : u } )</td>
<td>( { a : \mathcal{X} , b : - , c : - , d : - } )</td>
</tr>
<tr>
<td>( b )</td>
<td>( { b : u } )</td>
<td>( { a : - , b : \mathcal{X} , c : - , d : - } )</td>
</tr>
<tr>
<td>( c )</td>
<td>( { a : u , c : u } )</td>
<td>( { a : \mathcal{X} , b : - , c : - , d : - } )</td>
</tr>
<tr>
<td>( d )</td>
<td>( { a : u , b : u , d : u } )</td>
<td>( { a : \mathcal{X} , b : \mathcal{Y} , c : - , d : \mathcal{Z} } )</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( { a : u , b : u , c : u , d : u } )</td>
<td>( { a : \mathcal{X} , b : \mathcal{Y} , c : \mathcal{Z} , d : W } )</td>
</tr>
</tbody>
</table>

**Definition 11** We extend \( T \) to all types in type system \( p \). We define \( \hat{T} \) and \( \hat{T}^D \) by the following mutually recursive definitions.

\[
\hat{T}(\tau) = \begin{cases} 
\tau & \text{if } \tau \text{ is a type variable} \\
T(\tau) & \text{if } \tau \text{ is a primitive type} \\
\hat{T}^D(\tau_1) \rightarrow \hat{T}(\tau_2) & \text{if } \tau = \tau_1 \rightarrow \tau_2 
\end{cases}
\]
The important property of the translation $T$ is that captures the subtype relation in type system $p$ by the substitution mechanism of parametric polymorphism.

**Theorem 12** For all types $\tau$ and $\sigma$ if $\tau \leq_p \sigma$ then $T(\tau) \leq_r T(\sigma)$.

### 4 Interpretation

Now we formalize the reverse translation $\hat{T}$, i.e., the translation from records to primitive types. It is crucial to the proof of theorem 21 that $\hat{T}$ not treat the domain of the function space differently than the range, as $T$ did. This is the cause of a great loss of precision on the part of $T$.

We need to first define some auxiliary notation.

**Definition 13** For any record $\rho$ in type system $r$ we define $\Gamma_\rho$ to be the set of primitive types $\{\tau \mid \tau \leq \rho\}$, if $\rho$'s label in $\rho$ is +. We define $\Gamma_\rho$ to be the set $\{\tau \mid \neg \tau \leq \rho\}$, if $\rho$'s label in $\rho$ is -. Otherwise $\Gamma_\rho$ is the entire set of primitive types.

**Definition 14** The interpretation of a record $\rho$ in type system $p$, denoted $I(\rho)$, is a primitive type $\tau$. The type $\tau$ is determined by picking a type from the intersection of $\Gamma_\rho$ for all labels $\sigma$ in $\rho$. We can choose any type from the intersection, a minimal element will do. (There may be more than one minimal element.) If the set $\Gamma_\rho$ is empty just pick $\bot$.

**Definition 15** We extend $I$ to all types in type system $r$. We define $\hat{I}$ by the following definition.

\[
\hat{I}(\rho) = \begin{cases} 
\rho & \text{if } \rho \text{ is a type variable} \\
I(\rho) & \text{if } \rho \text{ is a record} \\
\hat{I}(\rho_1) \rightarrow \hat{I}(\rho_2) & \text{if } \rho = \rho_1 \rightarrow \rho_2
\end{cases}
\]

The function $\hat{I}$ respects the subtype ordering.
Theorem 16 For all types $\rho$ and $\rho'$ in type system $r$, if $\rho \leq_r \rho'$ then $\tilde{T}(\rho) \leq_r \tilde{T}(\rho)$.

We think of $\tilde{T}$ as an inverse of $T$. We do have the following fact.

Theorem 17 For all primitive types $\tau$ in type system $p$ we have $\mathcal{I}(\mathcal{T}(\tau)) = \tau$.

But we also have the following.

Theorem 18 For all primitive types $\tau$ in type system $p$ we have $\mathcal{I}(\mathcal{T}(\tau)) = \perp$.

The function $\tilde{T}$ is a pretty poor inverse. For example, $\tilde{T}(\mathcal{T}(c \rightarrow b)) = \mathcal{I}(\mathcal{T}(c)) \rightarrow \mathcal{I}(\mathcal{T}(b)) = \perp \rightarrow b \neq c \rightarrow b$.

5 Correctness

In this section we prove the main theorems that justify the approach taken to obtain coercion polymorphism.

Definition 20 For all type assignments $A$ in the type system $p$, we write $\tilde{T}(A)$ for that assignment which maps every variable $v$ in the domain of $A$ to $\tilde{T}(A(v))$. Also, for all type assignments $A$ in the type system $r$, we write $\tilde{I}(A)$ for that assignment which maps every variable $v$ in the domain of $A$ to $\tilde{I}(A(v))$.

The next theorem is a crucial element in linking derivations in type system $r$ with those in $p$. The proof of theorem 21 requires that $\tilde{T}(\mathcal{T}(T \rightarrow \perp)) = \mathcal{I}(\mathcal{T}(\perp)) = \perp \rightarrow \perp \neq T \rightarrow \perp$. However $\perp \rightarrow \perp \geq_r T \rightarrow \perp$. And even $\tilde{T}(\mathcal{T}(c \rightarrow b)) = \mathcal{I}(\mathcal{T}(c)) \rightarrow \mathcal{I}(\mathcal{T}(b)) = \perp \rightarrow b \neq c \rightarrow b$.

Theorem 19 For all types $\tau$ in type system $p$ we have $\tau \leq_r \tilde{T}(\mathcal{T}(\tau))$.

Theorem 21 For all expressions $e$, all type assignments $A$ in type system $r$ and all types $\rho$ in type system $r$, if $A \vdash_r e : \rho$, then $\tilde{T}(A) \vdash_p e : \tilde{T}(\rho)$.

Proof. The proof is by induction on the derivation in type system $r$.

Suppose that the last rule (the only rule) used in the derivation was for variables. Then we are given $A \vdash_r v : A(v)$. Then $\tilde{T}(A)$ is derivable in type system $p$ as $\tilde{T}(A) = \tilde{T}(A(v)) = \tilde{T}(\rho)$.

Suppose the last rule used in the derivation was the rule for abstraction. Then we are given $A \vdash_r \lambda v. e' : \rho$. We must have $\rho = \rho_1 \rightarrow \rho_2$ for some $\rho_1$ and $\rho_2$ and that $A[v \mapsto \rho_1] \vdash_p e' : \rho_2$ is derivable. By the induction hypothesis $\tilde{I}(A) = \tilde{I}(A[v \mapsto \rho_1]) \vdash_p e' : \tilde{I}(\rho_2)$ is derivable. Hence $\tilde{I}(A) \vdash_p \lambda v. e' : \tilde{I}(\rho_1) \rightarrow \tilde{I}(\rho_2)$ is derivable. From the definition of $\tilde{T}$ we have $\tilde{T}(A) = \tilde{T}(A(v)) = \tilde{T}(A(v)) = \tilde{T}(\rho_1 \rightarrow \rho_2)$. This is the desired conclusion.

Suppose that the last rule used in the derivation was the rule for application. Then we are given $A \vdash_r e_1(e_2) : \rho$. Hence $A \vdash_r e_1 : \rho_2 \rightarrow \rho$ and $A \vdash_r e_2 : \rho_2$ must also be derivable for some $\rho_2$. By the induction hypothesis we have $\tilde{I}(A) \vdash_p e_1 : \tilde{I}(\rho_2) \rightarrow \tilde{I}(\rho)$ and we have $\tilde{I}(A) \vdash_p e_2 : \tilde{I}(\rho_2)$ is derivable. From the definition of $\tilde{T}$ we have $\tilde{T}(\rho_2 \rightarrow \rho) = \tilde{T}(\rho_2) \rightarrow \tilde{T}(\rho)$. Therefore $\tilde{T}(A) \vdash_p e_1(e_2) : \tilde{T}(\rho)$.

Suppose that the last rule used in the derivation was the conditional rule. This case follows immediately by the induction hypothesis.

The next theorem demonstrates the close connection between proofs in type system $p$ and $r$.

Theorem 22 For all expressions $e$, all type assignments $A$ in the type system $p$, if there is some type $\rho$ in type system $r$ such that $\tilde{T}(A) \vdash_r e : \rho$, then $A \vdash_p e : \tilde{T}(\rho)$.
Proof. We assume that \( T(A) \vdash r \ e : \hat{\eta} \). By the previous theorem \( \hat{\iota}(T(A)) \vdash \ e : \hat{\iota}(\rho) \). Since \( A(v) \leq \rho \), we have \( A \vdash \ e : \hat{\iota}(\rho) \) by the syntactic subtyping theorem (theorem 6).

The following lemma is required in the proof of theorem 24. The lemma only works for substitutions obtained from running TypeOf on translated type assignments. We omit the proof. A slightly different approach to this is proved in [11].

Lemma 23 For all "relevant" substitutions \( S \), and types \( r \) in type system \( p \), there is a substitution \( S' \) such that \( S'(r) \leq \hat{\iota}(\rho) \).

Theorem 24 For all expressions \( e \), all type assignments \( A \) in type system \( p \), all substitutions \( S \), and all types \( \rho \), if TypeOf(\( T(A) \),\( e \)) succeeds with \( (S,\rho) \), then there is an \( S' \) such that \( S'A \vdash \ e : \hat{\iota}(\rho) \).

Proof. By soundness of the type system \( r \) we know that \( \hat{\iota}(T(A)) \vdash \ e : \hat{\iota}(\rho) \). By theorem 21 we have \( \hat{\iota}(S(T(A))) \vdash \ e : \hat{\iota}(\rho) \). By lemma 23 there is an \( S' \) such that \( S'A(v) \leq \hat{\iota}(S(T(A(v)))) \) for all \( v \). By lemma 6 we can "lower" every type in \( A \) and still derive the same type of \( e \). Therefore, \( S'A \vdash \ e : \hat{\iota}(\rho) \).

The converse of this last theorem does not hold. We are not able to achieve all of the subtype polymorphism in type system \( p \) in type system \( r \). Here is an example of a deduction in type system \( p \). Suppose we have an environment \( A \) in which \( f \) has type \( c \rightarrow \text{bool} \) and \( g \) has type \( a \).

\[
\begin{align*}
A' & \vdash f : c \rightarrow \text{bool} & A' & \vdash v : c \\
A' & \vdash f(v) : \text{bool} & A' & \vdash v : a \\
A' & \vdash (\lambda v. \text{if } f(v) \text{ then } v \text{ else } g) : a \\
A & \vdash (\lambda v. \text{if } f(v) \text{ then } v \text{ else } g) : c \rightarrow a
\end{align*}
\]

The proof tree is not one that we can mimic in type system \( r \), because \( T^D(c) \) does not unify with \( T(a) \). If we knew what the argument to the function was going to be, say some element of type \( c \), the last statement could be used to give \( v \) different versions of type \( c \) represented using different field variables. Thus \( v \) could be unified once with \( T^D(c) \) and once with \( T(a) \).

The next result guarantees that we do not claim \( e \) has a type unless it has one in type system \( p \).

Corollary 25 For all expressions \( e \), all type assignments \( A \) in type system \( p \), if there is no \( S' \) and \( \tau \) such that \( S'A \vdash \ e : \tau \), then TypeOf(\( T(A) \),\( e \)) fails, i.e., there is no \( S \) and \( \rho \) such that TypeOf(\( T(A) \),\( e \)) succeeds with \( (S,\rho) \).

Proof. We prove the contrapositive: if there is some \( S \) and \( \rho \) such that TypeOf(\( T(A) \),\( e \)) succeeds with \( (S,\rho) \), then there is some \( S' \) and \( \tau \) such that \( S'A \vdash \ e : \tau \). The result is immediate by the previous theorem.

This completes the justification of our approach. The remaining sections comment on some of the ramifications.

6 Non-lattices

While it is appealing to assume that the system of coercions arranges the primitive types in a lattice, it is hard to come up with a practical example where this is the case. A more realistic situation is depicted in
figure 5. Here an integer can be coerced to a real number which can be coerced to a complex number, and a character can be coerced to a string.

The labels c, r, i, s, and ch stand for the types comp, real, int, string and char respectively. In the figure we have added T and ⊥ which are necessary if the structure is to be a lattice. Yet it is not realistic to assume that there is some primitive type T to which elements of both complex numbers and strings can be coerced. We would like to excise the phantom types T and ⊥.

Figure 5: A realistic ordering of primitive types.

In some cases, like the one in figure 5, it is possible to remove T and ⊥ easily. If, after cutting off T and ⊥, each of the remaining pieces are lattices, then we can translate each lattice to its own distinct record type. The approach described in this paper can be applied to each lattice individually.

7 Code generation

In this section we look briefly at translating an expression with implicit coercions to an expression with explicit coercions suitable for a compiler.

Definition 26 We denote the function which coerces the type τ₁ to τ₂ by \(c_{τ₁←τ₂}\) for all \(τ₁ ≤ τ₂\).

We permit use of the notation \(c_{τ₁←τ₂}\) in cases where \(τ₁\) and \(τ₂\) are syntactically identical types. Of course, there always is such a coercion function, namely the identity function. For the purposes of this paper we are interested in the case where \(τ₁\) and \(τ₂\) are primitive types. We have the following to define coercions between types which do not have a direct coercion function:

\[
\begin{align*}
c_{τ₁←τ₁} &= \lambda x . x \\
c_{τ₁←τ₂} &= \lambda x . c_{τ₁←τ₃} (c_{τ₃←τ₂} x)
\end{align*}
\]

where \(τ₁ ≤ τ₂ ≤ τ₃\), and

\[
\begin{align*}
c_{(τ₁→τ₂)→(τ₁→τ₃)} &= \lambda f . \lambda x . c_{τ₂→τ₃} (f (c_{τ₁→τ₁} x))
\end{align*}
\]
This last coercion function takes a function as input and returns a function which applies a coercion in the input beforehand and another to the output afterward.

The `TypeOf` algorithm can be used to translate expressions to ones with the coercions explicitly appearing in the expression. This is necessary to generate the appropriate code for expressions of this language. This emphasizes the difference between coercion polymorphism and universal polymorphism. With universal polymorphism the code generated is exactly the same for, say, a polymorphic list function, regardless if it is for a list of real numbers or a list of integer numbers. With coercion polymorphism we know that a function for, say, complex numbers, is appropriate for integers, if the representation is converted.

1. Modify `TypeOf` to translate every expression $e$ to $(e : r)$ where $r$ is the type the algorithm finds for the expression $e$.

2. Recursively descend through the expression again (applying the substitution $S$ returned by `TypeOf` at the top level) and insert the coercion functions. Every time a variable does not match the type of the context, then supply a coercion.

8 Example

In this section we look at an example of coercions in a lattice with $\text{int} \leq \text{real} \leq \text{comp}$. The most basic sort of coercion between primitive types is illustrated by the expression $\text{trunc}(5)$, where $\text{trunc}$ is a function from $\text{real}$ to $\text{int}$. (We have in mind the function that truncates a real number, but, of course, only its type is important here.) Here is a deduction of the typing $A \vdash \text{trunc}(5) : \text{int}$ in type system $p$.

$$
\frac{
A \vdash 5 : \text{int} \\
A \vdash \text{trunc} : \text{real} \rightarrow \text{int} \\
\text{real} \leq \text{comp}}{
A \vdash \text{trunc}(5) : \text{int}}
$$

In viewing types as records we think of $\text{int}$ as the record with three fields $\{c : u, r : u, i : u\}$, $\text{real}$ as $\{c : u, r : u\}$, and $\text{trunc}$ as $\{c : u, r : u \rightarrow c : u, r : u, i : u\}$. The follow table shows how this is encoded in type system $r$.

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>${c : u, r : u}$</th>
<th>${c : X, r : -, i : -}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{int}$</td>
<td>${c : u, r : u, i : u}$</td>
<td>${c : X, r : Y, i : -}$</td>
</tr>
<tr>
<td>$\text{real} \rightarrow \text{int}$</td>
<td>${c : u, r : u \rightarrow c : u, r : u, i : u}$</td>
<td>${c : X, r : Y, i : -}$</td>
</tr>
</tbody>
</table>

At the key point in running `TypeOf` on $\hat{T}(A)$ and $e$, the algorithm unifies the domain of $\text{trunc}$ with $\text{int}$. This succeeds in mapping both representation to $\rho = \{c : +, r : +, i : -\}$. This is a representation of $\text{int}$, as it should be, because $\hat{T}(\rho) = \text{int}$.

There is another derivation in type system $p$ of the typing $A \vdash \text{trunc}(5) : \text{int}$ is possible.

$$
\frac{
A \vdash \text{trunc} : \text{real} \rightarrow \text{int} \\
\text{real} \leq (\text{int} \rightarrow \text{int}) \\
A \vdash 5 : \text{int}}{
A \vdash \text{trunc}(5) : \text{int}}
$$

Both derivations are modeled by the same same process in type system $r$. For code generation either choice is possible, although coercing the argument is less expensive in this case.
9 Conclusion

We have contributed to understanding the nature of subtype polymorphism when obtained by means of parametric polymorphism. We have applied subtype polymorphism to the problem of coercion polymorphism. It appears possible that better translations of subtype polymorphism to parametric polymorphism are possible. The translation used here is by no means the only way to encode the lattice in records. And the reverse translation has some degree of latitude. It is possible to secretly distinguish the domain from the range by detecting the existence of any + fields or any - fields. Only mixed records, which have been used in both the domain and the range, would have to be translated rigidly to any of a number of unsatisfactory primitive types. Exactly what amount of subtype polymorphism can be obtained by means of parametric polymorphism should be characterized.

References


