Type reconstruction for coercion polymorphism*

Ryan Stansifer
University of North Texas

Dan Wetklow
University of Pittsburgh at Johnstown

Abstract

The problem addressed in this paper is type reconstruction for programming languages that permit implicit coercions between primitive (base) types. We present an approach to performing type reconstruction in such a situation. If elements of type τ_1 can be implicitly coerced to elements of type τ_2, then τ_1 can be viewed as a subtype of τ_2, denoted $\tau_1 \leq \tau_2$. We assume that the subtype relations between the primitive types allow the collection of these types to be arranged in a lattice, although in some cases a more general partial order is permitted.

Our approach to performing type reconstruction in the presence of subtype relations defined on the collection of primitive types is to translate the problem to the well-studied problem of performing type reconstruction in the presence of subtype polymorphism where the subtype relation is defined on record types. We define a translation from primitive types to record types, perform type reconstruction (using the standard type reconstruction algorithm), and then translate the types obtained back to types in the original type system.

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.

The approach we take to type reconstruction in the presence of subtype relations defined on the set of primitive types is to translate the problem to that of well-studied problem of performing type reconstruction in the presence of subtype relations defined on records. This is accomplished by representing each primitive type as a record type. The translation preserves the subtype relations between primitive types, i.e., if p_1 and p_2 are primitive types, then $\mathcal{T}(p_1)$ is a subtype of $\mathcal{T}(p_2)$ if and only if p_1 is a subtype of p_2. Then type reconstruction is performed on the translated version of the problem and the resulting record types are then translated back to primitive types.

Aside from having to define translations between the two type systems, we are also faced with the problem of performing type reconstruction in the presence of subtype polymorphism defined on records. For this problem we choose to use the approach used by Rémy [6, 7]. His approach was to use a multisorted type system which was defined in such a way as to capture (some) subtype polymorphism with parametric polymorphism. There was no subtype relation defined on records in this type system. This technique has been used by Wand [12, 11] to model object-oriented languages and it has the advantage of making use of the well-understood mechanisms and algorithms that are used for performing type reconstruction for languages that permit only parametric polymorphism.

To summarize our approach, given a type environment A (a mapping of variables to types) 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, `TypeOf`, in the other type system, and

*Appeared in *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*, 1994, pages 420–425.

3. map the types obtained back to the primitive type system.

While our approach has the advantage of using an existing type reconstruction algorithm, which we call `TypeOf`, 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 the multisorted type system succeeds in deriving a type τ 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.
2. If given e and a translated version of A, `TypeOf` succeeds in typing e, then it must be possible to derive some type for e in the primitive type system.

The types in the multisorted type system are more expressive than those in the primitive type system, but the typing rules are not as powerful (due to the lack of a subtype relation in the multisorted type system). 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, we sacrifice some of the expressiveness of the primitive type system in order to gain the simplicity that the multisorted type system offers for performing type reconstruction. The translation back and forth explains some of the reasons why embedding subtype polymorphism in parametric polymorphism fails to capture all subtype polymorphism.

Language and type systems

In this paper we will deal exclusively with the simple language of lambda expressions. The following table presents the language.

expression	description
v	variable
$e_1(e_2)$	function application
$\lambda v.e$	function definition
$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditional

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:

$$\tau \quad ::= \quad \alpha \mid p \mid \tau_1 \rightarrow \tau_2$$

Functions have the usual function space types denoted using $\rightarrow$. Each primitive type is denoted by its own special constant represented by the metavariable p in the grammar above. Universal parametric polymorphism is achieved by allowing type variables, represented by the metavariable α. 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.

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.

Since the goal is to take advantage of the lattice structure formed by the subtype relations on records to perform type reconstruction in the presence of subtype relations on primitive types, we use a type system with record types. Due to the nature of the translation, there are only a finite number of fields (one field for each primitive type) and we can assume that all fields have type u, a type with just one element. Only the presence or absence of a field will be important in this paper; we are not actually interested in records as data structures. Unfortunately type reconstruction in this system is difficult [2, 4, 8, 10].

The type system r, inspired by the work of Rémy [6, 7], is now displayed. This is the system in which we will perform type reconstruction. Type system r has two syntactic categories, or sorts—the usual one for "types," and an additional sort for fields of records.

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \{l_1 : F_1, \ldots, l_n : F_n\} \mid \tau_1 \rightarrow \tau_2 \\ F & ::= & \mathcal{X} \mid + \mid - \end{array}$$

Each record has exactly n fields, where n is the number of primitive types. The field value $+$ means that the field must be present in the record. A $-$ 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, r, a substitution may map both type variables and field variables to their respective sort. We denote a substitution S in type system r restricted to field variables by $S|_F$.

$$\frac{}{A \vdash_r v : A(v)} \quad \text{if } v \in \mathrm{Dom}(A)$$

$$\frac{A[v \mapsto \tau_1] \vdash_r e : \tau_2}{A \vdash_r \lambda\, v \,.\, e' : \tau_1 \to \tau_2}$$

$$\frac{A \vdash_r e_1 : \tau_2 \to \tau \quad A \vdash_r e_2 : \tau_2}{A \vdash_r e_1(e_2) : \tau}$$

$$\frac{A \vdash_r e_1 : \mathit{bool} \quad A \vdash_r e_2 : \tau \quad A \vdash_r e_3 : \tau}{A \vdash_r \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

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

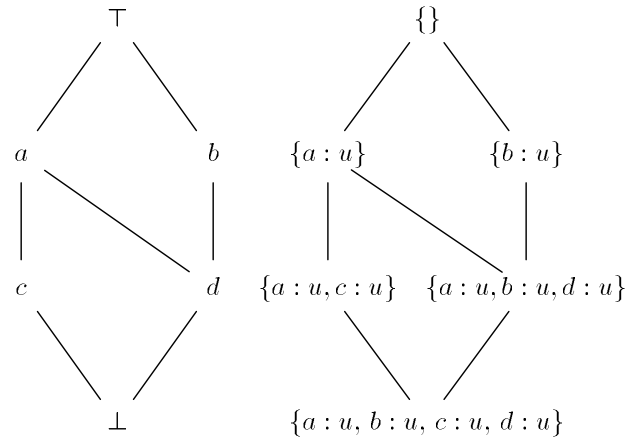

Figure 2: A simple lattice (u is a type with one element).

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

Definition 4 A *typing* is a triple of the form $A \vdash e : \tau$ where A is a type assignment, e is an expression, and τ is a type.

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

Theorem 5 (Syntactic subtyping, Cardelli) *For all expressions e, type assignments A, and types τ, σ and σ', if $A[v \mapsto \sigma] \vdash_p e : \tau$ is derivable and $\sigma' \leq \sigma$, then $A[v \mapsto \sigma'] \vdash_p e : \tau$ is derivable.*

Type system r is a generalization of the system studied by Milner [5] to the case of a multi-sorted algebra of type terms. All the desirable properties of that system carry through to the multi-sorted case. So there is a type reconstruction algorithm for type system r and the next two theorems hold.

Theorem 6 (Syntactic soundness) *For all expressions e, type assignments A, and type ρ, if* `TypeOf(`A`,`e`)` *succeeds with $\langle S, \rho \rangle$, then $SA \vdash_r e : \rho$ is derivable from the typing rules.*

Theorem 7 (Syntactic completeness) *For all expressions e, type assignments A, and type ρ, ρ' if $A \vdash_r e : \rho$ is derivable from the typing rules, then* `TypeOf(`A`,`e`)` *succeeds with $\langle S, \rho' \rangle$, and there exists a substitution T such that $T\rho' = \rho$.*

Translation

In this section we define a translation from the primitive type system to the record type system. It is easiest to consider the translation as two steps: first from primitive types to simple record types, then from simple record types to record types in type system r. Both steps of the translation can be seen in figures 3 and 4. This translation preserves the lattice structure in type system p.

The table in figure 3 illustrates the translation for the lattice depicted in figure 2.

Definition 8 The *translation* of a primitive type τ, denoted $\mathcal{T}(\tau)$, 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 σ is given by the following rule: if $\tau \leq \sigma$, then the field is a new field variable $\mathcal{X}$; otherwise, the field is $-$.

Definition 9 The *dual translation* of a primitive type τ, denoted $\mathcal{T}^D(\tau)$, 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 σ is given by the following rule: if $\tau \leq \sigma$, then the field is a $+$; otherwise, the field is a new field variable $\mathcal{X}$.

Definition 10 We extend $\mathcal{T}$ to all types in type system p. We define $\hat{\mathcal{T}}$ and $\hat{\mathcal{T}}^D$ by the following mutually recursive definitions.

$$\hat{\mathcal{T}}(\tau) = \begin{cases} \tau & \text{if } \tau \text{ is a type variable} \\ \mathcal{T}(\tau) & \text{if } \tau \text{ is a primitive type} \\ \hat{\mathcal{T}}^D(\tau_1) \to \hat{\mathcal{T}}(\tau_2) & \text{if } \tau = \tau_1 \to \tau_2 \end{cases}$$

$$\hat{\mathcal{T}}^D(\tau) = \begin{cases} \tau & \text{if } \tau \text{ is a type variable} \\ \mathcal{T}^D(\tau) & \text{if } \tau \text{ is a primitive type} \\ \hat{\mathcal{T}}(\tau_1) \to \hat{\mathcal{T}}^D(\tau_2) & \text{if } \tau = \tau_1 \to \tau_2 \end{cases}$$

The important property of the translation $\hat{\mathcal{T}}$ is that it captures the subtype relation in type system p by the substitution mechanism of parametric polymorphism.

Theorem 11 *For all types τ and σ if $\tau \leq_p \sigma$ then $\hat{\mathcal{T}}(\tau) \leq_r \hat{\mathcal{T}}(\sigma)$ and $\hat{\mathcal{T}}^D(\sigma) \leq_r \hat{\mathcal{T}}^D(\tau)$.*

Proof. By mutual induction on the structure of τ. ∎

Interpretation

Now we formalize the reverse translation $\hat{\mathcal{I}}$, i.e., the translation from record types to primitive types. It is crucial to the proof of theorem 18 that $\hat{\mathcal{I}}$ not treat the domain of the function space differently than the range, as $\hat{\mathcal{T}}$ did. This is the cause of a great loss of precision on the part of $\hat{\mathcal{I}}$.

Definition 12 For any record type τ in type system r, $\mathcal{I}(\tau)$ is the greatest lower bound of all primitive types whose field is $+$ or a field variable in τ.

Definition 13 We extend $\mathcal{I}$ to all types in type system r. We define $\hat{\mathcal{I}}$ by the following definition.

$$\hat{\mathcal{I}}(\rho) = \begin{cases} \rho & \text{if } \rho \text{ is a type variable} \\ \mathcal{I}(\rho) & \text{if } \rho \text{ is a record} \\ \hat{\mathcal{I}}(\rho_1) \to \hat{\mathcal{I}}(\rho_2) & \text{if } \rho = \rho_1 \to \rho_2 \end{cases}$$

The function $\hat{\mathcal{I}}$ respects the subtype ordering. We think of $\hat{\mathcal{I}}$ as an inverse of $\mathcal{T}$ and have the following two facts.

Theorem 14 *For all primitive types τ in type system p we have $\mathcal{I}(\mathcal{T}(\tau)) = \tau$.*

Proof. Immediate from the definition of $\mathcal{T}$ and $\mathcal{I}$ ∎

Theorem 15 *For all primitive types τ in type system p we have $\mathcal{I}(\mathcal{T}^D(\tau)) = \bot$.*

Proof. Immediate from the definitions of $\mathcal{T}^D$ and $\mathcal{I}$. ∎

The function $\hat{\mathcal{I}}$ is a pretty poor inverse. For example, $\hat{\mathcal{I}}(\hat{\mathcal{T}}(\top \to \bot)) = \mathcal{I}(\mathcal{T}^D(\top)) \to \mathcal{I}(\mathcal{T}(\bot)) = \bot \to \bot \neq \top \to \bot$. However $\bot \to \bot \geq_p \top \to \bot$. And even $\hat{\mathcal{I}}(\hat{\mathcal{T}}(c \to b)) = \mathcal{I}(\mathcal{T}^D(c)) \to \mathcal{I}(\mathcal{T}(b)) = \bot \to b \neq c \to b$. However $\bot \to b \geq_p c \to b$. So, although $\hat{\mathcal{I}}(\hat{\mathcal{T}}(\tau)) \neq \tau$ for some types, we have the following theorem.

Theorem 16 *For all types τ in type system p we have $\tau \leq_p \hat{\mathcal{I}}(\hat{\mathcal{T}}(\tau))$.*

Proof. From Theorems 11, 13 and induction on the structure of τ. ∎

Correctness

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

Definition 17 For all type assignments A in the type system p, we write $\hat{\mathcal{T}}(A)$ for that assignment which maps every variable v in the domain of A to $\hat{\mathcal{T}}(A(v))$. Also, for all type assignments A in the type system r, we write $\hat{\mathcal{I}}(A)$ for that assignment which maps every variable v in the domain of A to $\hat{\mathcal{I}}(A(v))$.

The next theorem is a crucial element in linking derivations in type system r with those in p. The proof of theorem 18 requires that $\hat{\mathcal{I}}(\rho_1 \to \rho_2)$ be equal to $\hat{\mathcal{I}}(\rho_1) \to \hat{\mathcal{I}}(\rho_2)$.

Theorem 18 *For all expressions e, all type assignments A in type system r and all types ρ in type system r, if $A \vdash_r e : \rho$, then $\hat{\mathcal{I}}(A) \vdash_p e : \hat{\mathcal{I}}(\rho)$.*

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 the rule for variables. Then we are given $A \vdash_r v : A(v)$. Then $\hat{\mathcal{I}}(\rho)$ is derivable in type system p as $\hat{\mathcal{I}}(A)(v) = \hat{\mathcal{I}}(A(v)) = \hat{\mathcal{I}}(\rho)$.

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 $\rho = \rho_1 \to \rho_2$ for some ρ_1 and ρ_2 and that $A[v \mapsto \rho_1] \vdash_r e' : \rho_2$ is derivable. By the induction hypothesis $\mathcal{I}(A)[v \mapsto \mathcal{I}(\rho_1)] \vdash_p e' : \mathcal{I}(\rho_2)$ is derivable. Hence $\mathcal{I}(A) \vdash_p \lambda v\,.\,e' : \mathcal{I}(\rho_1) \to \mathcal{I}(\rho_2)$ is derivable. From the definition of $\hat{\mathcal{I}}$ we have $\hat{\mathcal{I}}(\rho_1) \to \hat{\mathcal{I}}(\rho_2) = \hat{\mathcal{I}}(\rho_1 \to \rho_2)$. 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 $A \vdash_r e_1 : \rho_2 \to \rho$ and $A \vdash_r e_1 : \rho_2$ must also be derivable for some ρ_2. By the induction hypothesis we have $\mathcal{I}(A) \vdash_p e_1 : \mathcal{I}(\rho_2 \to \rho)$ is derivable, and we have $\mathcal{I}(A) \vdash_p e_2 : \mathcal{I}(\rho_2)$ is derivable. From the definition of $\hat{\mathcal{I}}$ we have $\hat{\mathcal{I}}(\rho_2 \to \rho) = \hat{\mathcal{I}}(\rho_2) \to \hat{\mathcal{I}}(\rho)$. Therefore $\hat{\mathcal{I}}(A) \vdash_p e_1(e_2) : \hat{\mathcal{I}}(\rho)$. ∎

The next theorem demonstrates the close connection between proofs in type systems p and r and shows that our first requirement of correctness, stated in the introduction, is satisfied.

Theorem 19 *For all expressions e, all type assignments A in the type system p, if there is some type ρ in type system r such that $\hat{\mathcal{T}}(A) \vdash_r e : \rho$, then $A \vdash_p e : \hat{\mathcal{I}}(\rho)$.*

τ	record	$\mathcal{T}(\tau)$	$\mathcal{T}^D(\tau)$
$\top$	$\{\}$	$\{a:-,\, b:-,\, c:-,\, d:-\}$	$\{a:\mathcal{X},\, b:\mathcal{Y},\, c:\mathcal{Z},\, d:\mathcal{W}\}$
a	$\{a:u\}$	$\{a:\mathcal{X},\, b:-,\, c:-,\, d:-\}$	$\{a:+,\, b:\mathcal{X},\, c:\mathcal{Y},\, d:\mathcal{Z}\}$
b	$\{b:u\}$	$\{a:-,\, b:\mathcal{X},\, c:-,\, d:-\}$	$\{a:\mathcal{X},\, b:+,\, c:\mathcal{Y},\, d:\mathcal{Z}\}$
c	$\{a:u,\, c:u\}$	$\{a:\mathcal{X},\, b:-,\, c:\mathcal{Y},\, d:-\}$	$\{a:+,\, b:\mathcal{X},\, c:+,\, d:\mathcal{Y}\}$
d	$\{a:u,\, b:u,\, d:u\}$	$\{a:\mathcal{X},\, b:\mathcal{Y},\, c:-,\, d:\mathcal{Z}\}$	$\{a:+,\, b:+,\, c:\mathcal{X},\, d:+\}$
$\bot$	$\{a:u,\, b:u,\, c:u,\, d:u\}$	$\{a:\mathcal{X},\, b:\mathcal{Y},\, c:\mathcal{Z},\, d:\mathcal{W}\}$	$\{a:+,\, b:+,\, c:+,\, d:+\}$

Figure 3: Translation of some types

τ		$\hat{\mathcal{T}}(\tau)$
real	$\{c:u,\, r:u\}$	$\{c:\mathcal{X},\, r:\mathcal{Y},\, i:-\}$
int	$\{c:u,\, r:u,\, i:u\}$	$\{c:\mathcal{X},\, r:\mathcal{Y},\, i:\mathcal{Z}\}$
$\mathit{real} \to \mathit{int}$	$\{c:u, r:u\} \to \{c:u, r:u, i:u\}$	$\{c:+,\, r:+,\, i:\mathcal{Z}'\} \to \{c:\mathcal{X}'',\, r:\mathcal{Y}'',\, i:\mathcal{Z}''\}$

Figure 4: Encoding of $\mathit{real} \to \mathit{int}$ in type system r

Proof. We assume that $\hat{\mathcal{T}}(A) \vdash_r e : \rho$. By the previous theorem $\hat{\mathcal{I}}(\hat{\mathcal{T}}(A)) \vdash_r e : \hat{\mathcal{I}}(\rho)$. Since $A(v) \leq_p \hat{\mathcal{I}}(\hat{\mathcal{T}}(A(v)))$ for all v, we have $A \vdash_p e : \hat{\mathcal{I}}(\rho)$ by the syntactic subtyping theorem (theorem 5). ∎

The following lemma is required in the proof of theorem 21. Theorem 21 shows that our second requirement for correctness is satisfied.

Lemma 20 *For all "relevant" substitutions S, and types τ in type system p, there is a substitution S' such that $S'(\tau) \leq_p \hat{\mathcal{I}}(S\hat{\mathcal{T}}(\tau))$.*
Proof. From the definitions of $\mathcal{T}$ and $\mathcal{I}$ for primitive types and induction on the structure of τ. ∎

Theorem 21 *For all expressions e, all type assignments A in type system p, all substitutions S, and all types ρ, if* `TypeOf(`$\hat{\mathcal{T}}(A)$`,`e`)` *succeeds with $\langle S, \rho\rangle$, then there is an S' such that $S'A \vdash_p e : \hat{\mathcal{I}}(\rho)$.*
Proof. By soundness of the type system r we know that $S\hat{\mathcal{T}}(A) \vdash_r e : \rho$. By theorem 18 we have $\hat{\mathcal{I}}(S\hat{\mathcal{T}}(A)) \vdash_p e : \hat{\mathcal{I}}(\rho)$. By lemma 20 there is an S' such that $S'A(v) \leq_p \hat{\mathcal{I}}(S\hat{\mathcal{T}}(A(v)))$ for all v. By lemma 5 we can "lower" every type in A and still derive the same type of e. Therefore, $S'A \vdash_p e : \hat{\mathcal{I}}(\rho)$. ∎

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

Corollary 22 *For all expressions e, all type assignments A in type system p, if there is no S' and τ such that $S'A \vdash_p e : \tau$, then* `TypeOf(`$\hat{\mathcal{T}}(A)$`,`e`)` *fails, i.e., there is no S and ρ such that* `TypeOf(`$\hat{\mathcal{T}}(A)$`,`e`)` *succeeds with $\langle S, \rho\rangle$.*
Proof. We prove the contrapositive: if there is some S and ρ such that `TypeOf(`$\hat{\mathcal{T}}(A)$`,`e`)` succeeds with $\langle S, \rho\rangle$, then there is some S' and τ such that $S'A \vdash_p e : \tau$. The result is immediate by the previous theorem. ∎

Example

In this section we look at an example of coercions in a lattice with $\mathit{int} \leq \mathit{real} \leq \mathit{comp}$. The most basic sort of coercion between primitive types is illustrated by the expression $\mathit{trunc}(5)$, where *trunc* is a function from *real* to *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 $A \vdash \mathit{trunc}(5) : \mathit{int}$ in type system p.

$$\frac{A \vdash \mathit{trunc} : \mathit{real} \to \mathit{int} \qquad \dfrac{A \vdash 5 : \mathit{int}}{A \vdash 5 : \mathit{real}}\ (\mathit{int} \leq \mathit{real})}{A \vdash \mathit{trunc}(5) : \mathit{int}}$$

In viewing types as records we think of *int* as the record with three fields $\{c:u, r:u, i:u\}$, *real* as $\{c:u, r:u\}$, and the type of *trunc* as $\{c:u, r:u\} \to \{c:u, r:u, i:u\}$. The table in figure 4 shows how this is encoded in type system r. At the key point in running `TypeOf` on $\hat{\mathcal{T}}(A)$ and e, the algorithm unifies the type of *trunc* with $\mathit{int} \to \alpha$. The domains unify (both are mapped to $\rho = \{c:+,\, r:+,\, i:\mathcal{Z}\}$) and the type returned for $\mathit{trunc}(5)$ is $\rho = \{c:\mathcal{X},\, r:\mathcal{Y},\, i:\mathcal{Z}\}$ and $\hat{\mathcal{I}}(\rho) = \mathit{int}$.

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 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 emphasized simplicity over getting the "best" translation possible. It is possible to define a reverse translation that treats the domain differently than the range in many cases. 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.

References

[1] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[2] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In H. Ganzinger, editor, *ESOP '88: 2nd European Symposium on Programming, Nancy, France, March 21–24*, pages 94–130, Berlin, 1988. Springer-Verlag.

[3] A. V. Hense and G. Smolka. Principal types for object-oriented languages. Technical Report A 02/93, Fachbereich Informatik, Universität des Saarlandes, June 1993.

[4] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Conference record of the 1988 ACM symposium on LISP and functional programming*. ACM, 1988.

[5] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, December 1978.

[6] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–88. ACM, 1989.

[7] D. Rémy. Type inference for records in a natural extension of ML. Technical Report MS-CIS-90–73, Department of Computer and Information Science, University of Pennsylvania, October 1990.

[8] R. Stansifer. Type inference with subtypes. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–97. ACM, January 1988.

[9] R. Stansifer and D. Wetklow. Method reuse in typed object-oriented languages. In *Proceedings of the ACM/SIGAPP Sympoisum on Applied Computing*, pages 63–70, 1993.

[10] M. Wand. Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44. 1987.

[11] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 92–97. 1989.

[12] M. Wand. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89–2, College of Computer Science, Northeastern University, November 88.

[13] D. Wetklow. *Type reconstruction algorithms for object-oriented languages.* PhD thesis, Purdue University, 1990.