Type Inference with Subtypes

You-Chin Fuh Prateek Mishra
Department of Computer Science
The State University of New York at Stony Brook
Stony Brook, New York 11794-4400
CSNET: yfuh@sbcs, mishra@sbcs

Abstract

We extend polymorphic type inference to include subtypes. This paper describes the following results:

- We prove the existence of (i) principal type property and (ii) syntactic completeness of the type checker, for type inference with subtypes. This result is developed with only minimal assumptions on the underlying theory of subtypes.
- For a particular "structured" theory of subtypes, those engendered by coercions between type constants only, we prove that principal types are compactly expressible. This suggests that a practical type checker for the structured theory of subtypes is feasible.
- We develop efficient algorithms for such a type checker. There are two main algorithms: MATCH and CONSISTENT. The first can be thought of as an extension to the unification algorithm. The second, which has no analogue in conventional type inference, determines whether a set of coercions is consistent.

Thus, an extension of polymorphic type inference that incorporates the "structured" theory of subtypes is practical and yields greater polymorphic flexibility. We have begun work on an implementation.

1 Introduction

Polymorphic type inference, as embodied in the type-checker for Standard ML, has attracted widespread interest in the programming language community. The main results therein [DM82] [Lei83] [Wan87b] are (i) the principal type property: type correct programs possess multiple types all of which are substitution instances of a unique principal type, and (ii) syntactic completeness of

the type checker, which always finds the principal type. This principal type may be instantiated, depending on the context of program use to yield an appropriate type. Thus, a program may be used in many different contexts, and yet be correctly type-checked. In addition, the type checker requires few type declarations, supports interactive programming and is efficiently implementable [Car85,Mal87].

In this work we extend type inference to include subtypes. This provides additional flexibility as a program with type t may be used wherever *any* supertype of type t is acceptable. Our subtype concept is based on type embedding or coercion: type t_1 is a subtype of type t_2, written $t_1 \rhd t_2$, if we have some way of mapping every value with type t_1 to a value of type t_2. Traditional subtype relationships are subsumed by this framework: $int \rhd real, char \rhd string$ etc. In addition, we accomodate subtype relationships between user-defined types. For example, the following natural relationship, which might arise when defining an interpreter or denotational semantics for a programming language, is expressible.

$$term \rhd expr, var \rhd term, const \rhd term, int \rhd expr, bool \rhd expr$$

Of course, in addition to indicating the relationship between types the user must also provide coercion functions that map values from the subtype to the supertype.

We have three main results:

- We prove the existence of (i) principal type property and (ii) syntactic completeness of the type checker, for type inference with subtypes. This result is developed with only minimal assumptions on the underlying theory of subtypes.
- For a particular "structured" theory of subtypes, those engendered by coercions between type constants only, we prove that principal types are compactly expressible. This suggests that a practical type checker for the structured theory of subtypes is feasible.
- We develop efficient algorithms for such a type checker. There are two main algorithms: MATCH and CONSISTENT. The first can be thought of as an extension to the unification algorithm. The second, which has no analogue in conventional type inference, determines whether a set of coercions is consistent.

Thus, an extension of polymorphic type inference that incorporates the "structured" theory of subtypes is practical and yields greater polymorphic flexibility. We have begun work on an implementation.

1.1 What's the problem?

What are the problems caused by adding subtypes into a polymorphic type inference system? Consider the term $I \equiv \lambda x.x$ with (conventional) principal type $\alpha \rightarrow \alpha$. Now, one type I possesses is $int \rightarrow real$, as $int \rhd real$. This type is not a substitution instance of $\alpha \rightarrow \alpha$. A first "solution" is to redefine the principal type property: type τ is a principal type of term t if any type τ' that

t possesses is either a instance of τ or *is a supertype of some instance of* τ. From the standard semantics for $\rightarrow$, we have:

$$int \rhd real \Rightarrow int \rightarrow int \rhd int \rightarrow real$$

Hence, with the new definition, it appears that $\alpha \rightarrow \alpha$ is the principal type for I. However, consider the term $twice \equiv \lambda f.\lambda x.f(f\ x)$ with principal type $\tau \equiv (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. One type *twice* possesses is $(real \rightarrow int) \rightarrow (real \rightarrow int)$. A simple case-analysis demonstrates that there is *no* substitution instance τ' of τ, such that

$$int \rhd real \Rightarrow \tau' \rhd (real \rightarrow int) \rightarrow (real \rightarrow int)$$

1.2 A General Solution

The example above demonstrates that in the presence of subtypes we cannot represent the set of all typings of a program by a type expression alone. Instead, types are represented by a *pair* consisting of a set of coercion statements $\{t_i \rhd t_j\}$ (called a coercion set) and a type expression. The idea is that any substitution that satisfies the coercion set can be applied to the type expression to yield an instance of the type. Given a program how do we compute such a type expression? Our first result, described in Section 3, states that it is enough to carry out type inference in the standard manner with the additional requirement that at each step during type inference we conclude we have inferred a supertype of the standard type. The collection of such conclusions yields the coercion set. Furthermore, with only minimal assumptions about the underlying structure of subtypes, we show that the resulting type is the principal type associated with the program. For example, we would compute the coercion set-type pair (C, γ) for the identity function I, where $C \equiv \{\alpha \rightarrow \beta \rhd \gamma, \alpha \rhd \beta\}$. Any substitution S that satisfies every coercion in C yields a typing $S(\gamma)$ for I.

1.3 A Structured Theory of Subtypes

While the results in Section 3 provide a general framework for type inference in the presence of subtypes, it should be clear that types of the form shown above for I are not practically useful.

In practice, we are interested in subtype theories with a great deal more "structure". One of the simplest such subtype theories is one in which *every* coercion is the consequence of coercions between type constants: $int \rhd real$, $term \rhd expr$ and so on. Any coercion between structured types, say $(t_1, t_2) \rhd (t_1', t_2')$, follows precisely from coercions between its components, $t_1 \rhd t_1', t_2 \rhd t_2'$. For such a subtype theory, in Section 5, we show that we can always transform the coercion set-type pair into a form where the coercion set consists only of *atomic* coercions: coercions between type constants and type variables. The typing for I would now take the form:

$$(\{\alpha \rhd \beta\}, \alpha \rightarrow \beta)$$

To build a type inference system based on this concept we need efficient implementations of two algorithms: MATCH and CONSISTENT. Both appear to be polynomial time algorithms and are described in Section 6.

1.4 Polymorphism

The framework described above does not deal with the problem of polymorphism: permitting user-defined names to possess multiple types. Our approach is to follow ML, and use the "let" syntax combined with a limited form of "bounded" type quantification. This provides precisely as much flexibility as in ML and avoids the complexities of type inference with general quantification [Lei83][Mit84b]. This system is described in section 7.

2 Related Work

Discussion on the semantics of subtypes in a very general category-theoretic setting has appeared in [Rey80]. In [Rey85] inference rules for type inference with subtypes are given. However, strong assumptions are made about the subtype structure: every pair of subtypes must possess a unique least supertype. In [Mit84a] it was first shown that types with a "structured" subtype theory can be represented using coercion sets consisting only of atomic types. A type inference procedure was also outlined therein (in particular, we follow Mitchell in using the term MATCH), but algorithms were omitted. In [MR85] type inference methods for a theory of types with a general "type union" operator were developed, but the issue of completeness of the type checker was not addressed.

In a different direction, Cardelli [Car84] has suggested that inheritance be modelled by subtyping through the use of record structures with named fields. Recently, Wand [Wan87a] has given a type inference algorithm that models a form of inheritance based on record structures. In contrast to our work, Wand's algorithm is based on an extension of unification. We plan to investigate the possibility of extending our system to include some form of record-based inheritance.

3 Preliminary Definitions

There are two basic components in any type inference system, the language of *value* expressions and the language of *type* expressions. Value and type expressions are defined by the following abstract syntax.

$N \in$ Value Expressions	$t \in$ Type Expressions
$x \in$ Value Variables	$\alpha \in$ Type Variables
$f^m \in$ Value Constructors	$g^n \in$ Type Constructors

$$N ::= x \mid f^m[\bar{x}](N_1, \ldots, N_m) \quad t ::= \alpha \mid g^n(t_1, \ldots, t_n)$$

As examples, consider value expressions

$$\lambda[x](N),\ ifthenelse[](N,N_1,N_2),\quad \textbf{fix}\ [x](N),\ 7$$

and type expressions $int \to (\alpha \to bool)$, (int, α).

A coercion is an ordered pair of types written $t_1 \rhd t_2$. A coercion set $C = \{t_i \rhd r_i\}$ is a set of coercions. A type assumption A is a finite mapping from value variables to type expressions, often written $\bar{x} : \bar{t}$. Let Z be a set of value variables; by $A|_Z$ we mean A restricted to domain Z. A substitution S is a mapping from type variables to type expressions that is not equal to the identity function at only finitely many type variables. $[t_1/\alpha_1, \cdots, t_n/\alpha_n]$ is the substitution that maps α_i to t_i and is otherwise equal to the identity function. If t is a type expression, by $S(t)$ we mean the simultaneous replacement of every variable in t by its image under S. The meanings of $S(C)$ and $S(A)$ are defined in the standard fashion.

We will often consider some distinguished set of coercions as *valid* or *true* coercions. The only restriction we place on the set of valid coercions, is considered as a relation on $Type \times Type$ it should be (i) reflexive (ii) transitive and (iii) closed under substitution. Our intention here is that the set of valid coercions $\{t \rhd r\}$ consists of those pairs of types such that the first component may reasonably be transformed into the second. The three conditions on the set of valid coercions indicate that any type should be transformable to itself, that transformations between types be composable and that the transformation be unaffected by instantiation of type variables.

We say coercion $t \rhd r$ is *solvable* or *consistent* if there exists some substitution S such that $S(t) \rhd S(r)$ is valid. Define the relation $\Vdash$ on coercions by:

$$a \rhd b \Vdash c \rhd d \iff S(a) \rhd S(b) \text{ valid entails } S(c) \rhd S(d) \text{ valid}$$

We lift the relation $\Vdash$ to coercion sets by considering a coercion set to be a conjunction of all its coercions. Informally $C_1 \Vdash C_2$ should be read as saying that substitution S renders the coercions in C_2 valid, whenever it renders the coercions in C_1 valid. Observe that $\Vdash$ as a relation on coercion sets is (i) reflexive (ii) transitive and (iii) closed under substitution. Observe that $a \rhd b$ is valid iff $\emptyset \Vdash \{a \rhd b\}$.

3.1 Type Inference System

A type inference system is a system of rules that defines a relation, called a *typing*, over the four-tuple:

Coercion Set × Type Assumption × Value Expression × Type Expression

A *typing statement* is written $C, A \vdash M : t$. By $A\ ;\ \bar{x} : \bar{s}$ we mean a type assumption identical to A, except that it maps value variable x_i to type expression s_i. In the rules below, we would expect to have an instance of a FUN rule for each value constructor symbol f^m. FUN rules for

zero-ary constants may have empty antecedents.

$$VAR \qquad C, A \vdash x : A(x)$$

$$FUN_{f^m} \qquad \frac{C, A\,;\, \bar{x} : \bar{s}^f \vdash N_1 : r_1^f, \ldots, N_m : r_m^f}{C, A \vdash f^m[\bar{x}](N_1, \ldots, N_m) : \alpha^f}$$

$$COERCE \qquad \frac{C, A \vdash e : t, C \Vdash \{t \rhd p\}}{C, A \vdash e : p}$$

For some instances of FUN consider the following:

$$\frac{C, A\,;\, x : t_1 \vdash N : t_2}{C, A \vdash \lambda^1[x](N) : t_1 \to t_2}$$

$$\frac{C, A \vdash P : bool, M : t, N : t}{C, A \vdash \text{if } P \text{ then } M \text{ else } N \; : t}$$

$$C, A \vdash true : bool$$

$\boxed{C, A \vdash N : p}$ is a *typing* if

$\boxed{N \equiv x}$ and $C \Vdash \{A(x) \rhd p\}$.

$\boxed{N \equiv f^m[\bar{x}](N_1, \ldots, N_m)}$
and

(1) $(\bar{q}, v_i, u)$ is a substitution instance of $(\bar{s}^f, r_i^f, \alpha^f)$,

(2) $C, A\,;\, \bar{x} : \bar{q} \vdash N_i : v_i$ are typings,

(3) $C \Vdash \{u \rhd p\}$.

Observe that it is an immediate consequence of the transitivity of $\rhd$ that in any typing we need at most a single COERCE step after an application of a VAR or FUN step.

Definition 1 (Instance) *Typing statement $C', A' \vdash N : t'$ is an instance of typing statement $C, A \vdash N : t$, if there exists substitution S such that:*

1. $t' = S(t)$.
2. $A'|_{FV(N)} = S(A)|_{FV(N)}$.
3. $C' \Vdash S(C)$.

Lemma 1 *Typings are closed under the instance relation; i.e. if* $C, A \vdash N : t$ *is a typing then so is every instance* $C', A' \vdash N : t'$.

Proof: Proof is by induction on structure of term N. The main property required is that $C_1 \Vdash C_2 \Rightarrow S(C_1) \Vdash S(C_2)$. □

3.2 Algorithm TYPE

In this section, we describe an algorithm that constructs a distinguished representative of all typings for a term M, the *principal type* for M. We assume the existence of function *new* : $(Type\ Expression)^k \rightarrow (Type\ Expression)^k$, such that $new(t_1, \ldots, t_k)$ is obtained by consistently replacing all the type variables in types t_i by "new" type variables.

Algorithm TYPE:Type Assumption × Value Expression → Type Expression × Coercion Set

Input: (A_0, e_0), where $FV(e_0) \subseteq domain(A_0)$.

Initially:

$C = \emptyset$, $G = \{(A_0, e_0, \alpha_0)\}$, where α_0 is a new variable.

While G **is not empty do:**

Choose any g from G;

case g of:

$(A, f^m[\bar{x}](e_1, ..., e_n), t)$:

$(\bar{q}, v_i, u) = new(\bar{s}, r_i, \alpha)$;

$C \leftarrow C \cup \{u \rhd t\}$;

$G \leftarrow (G - \{g\}) \cup \{(A[\bar{x} : \bar{q}], e_i, v_i)\}$;

(A, x, t) :

$C \leftarrow C \cup \{A(x) \rhd t\}$;

$G \leftarrow G - \{g\}$;

end case;

Output: (α_0, C).

Example 1 *Let* $N \equiv \lambda f.\lambda x.fx$ *and* $A = \emptyset$. *Then*

$$TYPE(A, N) = (t_N, \left\{ \begin{array}{c} t_f \rightarrow t_{\lambda x.fx} \rhd t_N \\ t_f \rhd t_1 \rightarrow t_2 \\ t_x \rightarrow t_{fx} \rhd t_{\lambda x.fx} \\ t_2 \rhd t_{fx} \\ t_x \rhd t_1 \end{array} \right\})$$

It is interesting to compare our algorithm with that given by Mitchell [Mit84a]. Mitchell's algorithm works for a particular "structured" subtype theory and interleaves the generation of the constraint set with the process of simplifying it and checking it for consistency. In contrast, our algorithm is designed to work for a class of subtype theories and is concerned only with the generation of the relevant constraint set. In this way, we separate the "syntactic" aspects of type inference (traversal of the abstract syntax tree, generation of constraint set) from the details of processing the constraint set. One consequence is that we are able to give a general proof of soundness and syntactic completeness that makes use only of the assumptions about the relations: typing, $\triangleright$ and $\vdash$, that we have presented above. Further, as our algorithm does not commit itself to any particular method for processing constraint sets, it can serve as the basis for algorithms that utilize a variety of different methods. This is of importance as details of constraint set processing depend critically on the particulars of the subtype theory as well as on the particular application area of interest.

Theorem 1 *TYPE is sound and (syntactically) complete.*

Proof: Proof is given below. □

3.3 TYPE is sound and complete

Our proof follows Wand's [Wan87b] concise proof of soundness and completeness of parametric type inference. One difference between his proof and ours is that we need to reason about coercion sets instead of sets of equations used in his work.

A tri-tuple (A, e, t) is a *goal* if A is a type assumption, e is a value expression, and t is a type expression. The pair of coercion set and substitution, (C, σ), solves the goal (A, e, t), denoted by $(C, \sigma) \models (A, e, t)$, if $C, \sigma(A) \vdash e : \sigma(t)$. Let G be a set of goals: $(C, \sigma) \models G$, if $\forall g \in G$, $(C, \sigma) \models g$. We also say that $(C, \sigma) \models C_0$ if $C \vdash \sigma(C_0)$. We can extend the notion of solvability to pairs of coercion set and set of goals: $(C, \sigma) \models (C_0, G)$ if $(C, \sigma) \models C_0$ and $(C, \sigma) \models G$.

To prove soundness and completeness, it's sufficient to prove the following invariants:

TYPE is Sound: $(\forall(\bar{C}, \sigma))\,((\bar{C}, \sigma) \models (C, G) \implies \bar{C}, \sigma(A_0) \vdash e_0 : \sigma(\alpha_0))$

TYPE is Complete: $\bar{C}, \bar{A} \vdash e_0 : \bar{t} \wedge \exists \delta\ \bar{A}|_{FV(e_0)} = \delta(A_0)|_{FV(e_0)}$
$\implies (\exists \sigma)((\bar{C}, \sigma) \models (C, G) \wedge \bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)} \wedge \bar{t} = \sigma(\alpha_0))$

proof: (TYPE is sound)

Basis: Since $G = \{(A_0, e_0, \alpha_0)\}$, by definition of $\models$, $\bar{C}, \sigma(A_0) \vdash e_0 : \sigma(\alpha_0)$.

step: (Let C_0 and G_0 denote the values of C and G before the iteration)

(A, x, t) :

$(\bar{C}, \sigma) \models (C, G)$

$\Longrightarrow$ $(\bar{C}, \sigma) \models \{A(x) \rhd t\}$

$\Longrightarrow$ $(\bar{C}, \sigma) \models (A, x, t)$ (by typing rule)

$\Longrightarrow$ $(\bar{C}, \sigma) \models (C_0, G_0)$

$\Longrightarrow$ $\bar{C}, \sigma(A_0) \vdash e_0 : \sigma(\alpha_0)$ (By hypothesis)

$(A, f^m[\bar{x}](e_1, ..., e_n), t)$:

$(\bar{C}, \sigma) \models (C, G)$

$\Longrightarrow$ $(\bar{C}, \sigma) \models \{u \rhd t\}$ and $(\bar{C}, \sigma) \models \{A[\bar{x} : \bar{q}], e_i, v_i\}$

$\Longrightarrow$ $\bar{C} \Vdash \sigma(\{u \rhd t\})$ and $\bar{C}, \sigma(A[\bar{x} : \bar{q}]) \vdash e_i : \sigma(v_i)$ (by definition of $\models$)

$\Longrightarrow$ $\bar{C}, \sigma(A) \vdash f^m[\bar{x}](e_1, ..., e_n) : \sigma(t)$ (By typing rule)

$\Longrightarrow$ $(\bar{C}, \sigma) \models (C_0, G_0)$

$\Longrightarrow$ $\bar{C}, \sigma(A_0) \vdash e_0 : \sigma(\alpha_0)$ (By hypothesis)

proof: (TYPE is complete)

Basis: $\bar{C}, \bar{A} \vdash e_0 : \bar{t}$. Since α_0 is a new variable and $\exists \delta\ \bar{A}|_{FV(e_0)} = \delta(A_0)|_{FV(e_0)}$, it's obvious that $\exists \sigma$ such that $\bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)}$ and $\bar{t} = \sigma(\alpha_0)$.

step: (Let C_0 and G_0 denote the values of C and G before the iteration)

(A, x, t) :

$\bar{C}, \bar{A} \vdash e_0 : \bar{t}$

$\Longrightarrow$ $(\exists \sigma)\ ((\bar{C}, \sigma) \models (C_0, G_0) \wedge \bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)} \wedge \bar{t} = \sigma(\alpha_0))$ (By hypothesis)

$\Longrightarrow$ $\bar{C} \Vdash \sigma(\{A(x) \rhd t\})$ (By typing rule)

$\Longrightarrow$ $(\bar{C}, \sigma) \models \{A(x) \rhd t\}$ (by definition of $\models$)

$\Longrightarrow$ $(\bar{C}, \sigma) \models (C, G) \wedge \bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)} \wedge \bar{t} = \sigma(\alpha_0)$

$(A, f^m[\bar{x}](e_1, ..., e_n), t)$:

$\bar{C}, \bar{A} \vdash e_0 : \bar{t}$

$\Longrightarrow$ $(\exists \sigma_0)\ ((\bar{C}, \sigma_0) \models (C_0, G_0) \wedge \bar{A}|_{FV(e_0)} = \sigma_0(A_0)|_{FV(e_0)} \wedge \bar{t} = \sigma_0(\alpha_0))$
(By hypothesis)

$\Longrightarrow$ $\bar{C}, \sigma_0(A) \vdash f^m[\bar{x}](e_1, ..., e_n) : \sigma_0(t)$ (hence $(\bar{C}, \sigma_0) \models (A, f^m[\bar{x}](e_1, ..., e_n), t)$)

$\Longrightarrow$ $(\exists \gamma)(\bar{C}, (\sigma_0(A))[\bar{x} : \gamma(\bar{q})] \vdash e_i : \gamma(v_i) \wedge \bar{C} \Vdash \{\gamma(u) \rhd \sigma_0(t)\}$ (by typing rule)

$\Longrightarrow$ Since γ and σ_0 have disjoint domain, we can choose σ to be $\gamma \cup \sigma_0$

$\Longrightarrow$ $\bar{C}, \sigma(A[\bar{x} : \bar{q}]) \vdash e_i : \sigma(v_i) \wedge \bar{C} \Vdash \sigma(\{u \rhd t\}) \wedge \bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)} \wedge \bar{t} =$
$\sigma(\alpha_0)$

$\Longrightarrow$ $(\bar{C}, \sigma) \models (C, G) \wedge \bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)} \wedge \bar{t} = \sigma(\alpha_0)$

□

4 Well-typings

A typing $C, A \vdash N : t$ should be viewed as standing for a set of all possible instances $C', A' \vdash N : t'$, where C' is valid. Informally, the set of all *valid instances* of a typing expresses the "information content" of a typing, in that it describes all possible "correct" ways of using a typing. For an example, all valid instances of $\{\alpha \rhd real\}, \emptyset \vdash N : \alpha$ are of the form $\{t' \rhd real\} \cup C, A \vdash N : t'$, provided the coercions in $\{t' \rhd real\} \cup C$ are valid.

Typings of the form $C, A \vdash N : t$ which possess no valid instances are of no interest to us. This is the case when C contains contradictory information; for an example take C to be $\{bool \rhd \alpha, int \rhd \alpha, \alpha \rhd real\}$. We cannot find any type t such that replacing α by t in C results in a valid coercion set.

Define a *well-typing* to be a typing $C, A \vdash N : t$, where C is consistent. Immediately, the question arises whether the theory developed in previous sections carries over to well-typings (instance, principal type property, algorithm TYPE). Lemma 1 holds with the following obvious modification: If $C, A \vdash N : t$ is a well-typing, then so is every instance $C', A' \vdash N : t'$, whenever C' is consistent.

How do we compute well-typings? We simply run algorithm TYPE and check the final coercion set for consistency. If it is consistent, we are done; otherwise, we fail and conclude that no well-typing exists. That this method is sound is obvious; its completeness is argued below:

WTYPE :Type Assumption $\times$ Value Expression $\rightarrow$ Well Typing $+\{fail\}$
let $(p, C) = TYPE(A, N)$ in
if C is consistent then $C, A \vdash N : p$
else fail

To see that algorithm WTYPE is complete, we need to consider two cases. For the first, we have that C is consistent. But then, the syntactic completeness of TYPE ensures the syntactic completeness of WTYPE. For the second case, let C be inconsistent. We will argue that no well-typing $C', A' \vdash N : p'$ exists. Assume otherwise; as TYPE is syntactically complete we can find substitution S with $C' \Vdash S(C)$. Now, since C' is consistent we must have that $S(C)$ is consistent. But then C must be consistent as well and we have arrived at a contradiction.

5 Structured Type Inclusion

In this section, we develop type inference methods for the case where the underlying theory of type inclusion is structured. We study the simplest such case: all inclusions are the consequences of inclusions between type constants.

The abstract syntax of types t, is given by:

$$t ::= g_c \mid t_1 \rightarrow t_2 \mid (t_1, t_2)$$

The type inference rules are given by:

$C, A \vdash x : A(x)$	$\dfrac{C, A \vdash N : t_1 \to t_2, M : t_1}{C, A \vdash NM : t_2}$
$\dfrac{C, A\,;\{x : t_1\} \vdash M : t_2}{C, A \vdash \lambda x.M : t_1 \to t_2}$	$\dfrac{C, A \vdash M : boolean, N : t, O : t}{C, A \vdash \text{if } M \text{ then } N \text{ else } O : t}$
$\dfrac{C, A\,;\{x : t_1\} \vdash M : t_1}{C, A \vdash \textbf{fix}\ x.M : t_1}$	$C, A \vdash c : g_c$
$\dfrac{C, A \vdash M : t_1, C \Vdash t_1 \rhd t_2}{C, A \vdash M : t_2}$	

The following rules define the relation $\Vdash$ for the theory of interest.

$[AXIOM]$	$M \cup \{t_1 \rhd t_2\} \Vdash t_1 \rhd t_2$	$[CONST]$	$M \Vdash g_{c_i} \rhd g_{c_k}$
$[TRANS]$	$\dfrac{M \Vdash t_1 \rhd t_2, t_2 \rhd t_3}{M \Vdash t_1 \rhd t_3}$	$[REFLEX]$	$M \Vdash t \rhd t$
$[ARROW-I]$	$\dfrac{M \Vdash t_1 \rhd t_1', t_2' \rhd t_2}{M \Vdash t_1' \to t_2' \rhd t_1 \to t_2}$	$[PROD-I]$	$\dfrac{M \Vdash t_1 \rhd t_1', t_2 \rhd t_2'}{M \Vdash (t_1, t_2) \rhd (t_1', t_2')}$
$[ARROW-II]$	$\dfrac{M \Vdash t_1' \to t_2' \rhd t_1 \to t_2}{M \Vdash t_1 \rhd t_1', t_2' \rhd t_2}$	$[PROD-II]$	$\dfrac{M \Vdash (t_1, t_2) \rhd (t_1', t_2')}{M \Vdash t_1 \rhd t_1', t_2 \rhd t_2'}$

Rule [CONST] is the only means of introducing "new" statements about type inclusion in the system. Such statements are restricted to be relationships between type constants. Rules [TRANS] and [REFLEX] indicate that the relation $\Vdash$ is transitive and reflexive. Rules [ARROW-I], [PROD-I], [ARROW-II] and [PROD-II] are "structural" rules that define coercions between structured types in terms of coercions between their components.

An *inclusion statement* is written $C \Vdash t_1 \rhd t_2$. A proof for $C \Vdash t_1 \rhd t_2$ is a sequence of inclusion statements $IS_1, \cdots, IS_k \equiv C \Vdash t_1 \rhd t_2$, where each IS_i is a substitution instance of (i) [AXIOM], [CONSTANT] or [REFLEX] rules or (ii) is derived by an application of a substitution instance of one of the remaining rules to some finite subset of $IS_1, \cdots, IS_{i-1}$. In such a case, we say that the statement $C \Vdash t_1 \rhd t_2$ is true. We say $C_1 \Vdash C_2$ if $C_1 \Vdash t_i \rhd t_j$ for each $\{t_i \rhd t_j\} \in C_2$. It is decidable whether $C_1 \Vdash C_2$, where C_1 and C_2 are finite sets of coercions.

The description of $\Vdash$ given above indicates that the inclusion relationship between types is much more "structured" than the very general notion of $\Vdash$ studied in section 3. As a consequence, we will show that we need only consider coercion sets restricted to be of the format $\{t_i \rhd t_j\}$ where each t_i, t_j is an *atomic* type: either a type variable or a type constant.

5.1 Instantiating Coercion Sets

In section 4 we have defined the information content of a typing to be set of its valid instances. This suggests that there may exist distinct typings with identical information content. One way this might occur is if well-typing $C, A \vdash N : t$ and some instantiation $C', A' \vdash N : t'$, have identical information content. Further, it seems reasonable to consider the second typing preferable to the first as it contains more "structural" information than the first and therefore reveals more information to the user.

Example 2 *Consider the typing* $C, \emptyset \vdash N : \alpha$, *where*

$$C = \{\alpha \rhd \beta \to \gamma\}$$

Every valid instance of C *requires* α *to be instantiated to an "arrow" type; hence in place of the above typing we can use one of the form* $C', \emptyset \vdash N : \delta \to \rho$ *where*

$$C' = \{\delta \to \rho \rhd \beta \to \gamma\}$$

Both typings have identical information content but the second has more explicit "structural" information.

The notion of information content is essential in defining the equivalence of typings under instantiation. It is not the case, in Example 2 above, that $C \equiv C'$. Neither $C \Vdash C'$ holds, nor is it the case that $C' \Vdash C$. Further, arbitrary instantiation of coercion sets does not preserve information content: for example, if we instantiate $\alpha \rhd \beta \to \gamma$ to $\delta \to (\sigma \to \tau) \rhd \rho \to (\psi \to \mu)$, we are missing out on the possibility that α can be instantiated to a type with a "single" arrow type in some valid instance of $\alpha \rhd \beta \to \gamma$.

To see the general form of information content preserving instantiations we first need to characterize the "shape" of valid coercion sets. Define the relation $Match$ over pairs of type expression by:

$Match(t_1, t_2)$ is true, if both t_1, t_2 are atomic types.
$Match(t_1 \to t_2, t_1' \to t_2')$ whenever $Match(t_1, t_1'), Match(t_2, t_2')$.
$Match((t_1, t_2), (t_1', t_2'))$ whenever $Match(t_1, t_1'), Match(t_2, t_2')$.

We say C is a matching coercion set if every coercion $r \rhd s \in C$ is matching. Matching is a necessary condition for coercion sets to be valid; whenever coercion set C is valid every coercion contained in C must be matching. Proof is by induction on the number of proof steps needed to show $\emptyset \Vdash C$ and is straightforward.

Given that valid coercion sets are always matching, what kinds of instantiation preserve information content? If $t_1 \rhd t_2$ is a consistent coercion we will argue that we can always find a information-content preserving substitution S such that $S(t_1) \rhd S(t_2)$ is matching. Further, S is the least such substitution, in that any other matching instance of $t_1 \rhd t_2$ must also be an instance of $S(t_1) \rhd S(t_2)$.

Theorem 2 *Let $C, A \vdash N : t$ be a well-typing. There exists well-typing $C_*, A_* \vdash N : t_*$ with the property that:*

1. *$C_*, A_* \vdash N : t_*$ is a matching instance of $C, A \vdash N : t$.*
2. *$C_*, A_* \vdash N : t_*$ and $C, A \vdash N : t$ have identical information content.*
3. *If $C', A' \vdash N : t'$ is any other typing that satisfies property (1); then $C', A' \vdash N : t'$ is an instance of $C_*, A_* \vdash N : t_*$.*

Proof: See technical report # 87-25, SUNY Stony Brook. □

We will speak of $C_*, A_* \vdash N : t_*$ as the minimal matching instance of $C, A \vdash N : t$.

Example 3 *Let $N \equiv \lambda f.\lambda x.fx$, $A = \emptyset$ and let $TYPE(A, N) = (t_N, C)$ as in Example 1.*

$$C_* \equiv \left\{ \begin{array}{c} (\alpha_3 \to \alpha_4) \to (\alpha_1 \to \alpha_2) \rhd (\beta_3 \to \beta_4) \to (\beta_1 \to \beta_2) \\ \alpha_3 \to \alpha_4 \rhd t_1 \to t_2 \\ t_x \to t_{fx} \rhd \alpha_1 \to \alpha_2 \\ t_2 \rhd t_{fx} \\ t_x \rhd t_1 \end{array} \right\}, (t_N)_* \equiv (\beta_3 \to \beta_4) \to (\beta_1 \to \beta_2)$$

Finally, we need to ensure that representing a well-typing by its minimal matching instance does not perturb the most general type property. We need to show the following: let well-typing $C', A' \vdash N : t'$ be an instance of $C, A \vdash N : t$; then, we must have that $C'_*, A'_* \vdash N : t'_*$ is an instance of $C_*, A_* \vdash N : t_*$. To see this, observe that $C'_*, A'_* \vdash N : t'_*$ is a matching instance of $C, A \vdash N : t$; hence, it must be an instance of the minimal matching instance $C_*, A_* \vdash N : t_*$.

5.2 Simplifying Coercion Sets

Example 4

$$C_1 \equiv \{(\alpha \to \beta) \rhd (\delta \to \gamma)\}$$
$$C_2 \equiv \{\delta \rhd \alpha, \beta \rhd \gamma\}$$

C_1, C_2 are equivalent in that each entails the other: $C_1 \Vdash C_2$ and $C_2 \Vdash C_1$. As

$$C_1 \Vdash C_2 \Rightarrow S(C_1) \Vdash S(C_2)$$

both have identical information content. Finally, C_2 contains less redundant information and therefore seems preferable to C_1.

SIMPLIFY $C = C$, if all coercions in C are atomic
SIMPLIFY $C \cup \{t_1 \to t_2 \rhd r_1 \to r_2\}$ = SIMPLIFY $C \cup \{r_1 \rhd t_1, t_2 \rhd r_2\}$
SIMPLIFY $C \cup \{(t_1, t_2) \rhd (r_1, r_2)\}$ = SIMPLIFY $C \cup \{t_1 \rhd r_1, t_2 \rhd r_2\}$

Function SIMPLIFY maps matching coercion sets into the maximally simplified set of *atomic* coercions. It is trivially clear that $SIMPLIFY$ preserves information content; as $SIMPLIFY$ does not effect in any way the type variables contained in an coercion set it is also obvious that the most general well-typing property is also preserved.

Example 5 *Let C_* be as in Example 3 above.*

$SIMPLIFY(C_*) = \{\alpha_3 \rhd \beta_3,\ \beta_4 \rhd \alpha_4,\ \beta_1 \rhd \alpha_1,\ \alpha_2 \rhd \beta_2,\ \alpha_1 \rhd t_x,\ t_{fx} \rhd \alpha_2,\ t_1 \rhd \alpha_3,\ \alpha_4 \rhd t_2,\ t_2 \rhd t_{fx},\ t_x \rhd t_1\}$

In a practical implementation, we would only be concerned with $\{\beta_1 \rhd \beta_3, \beta_4 \rhd \beta_2\}$.

6 Algorithm WTYPE Revisited

WTYPE :Type Assumption $\times$ Value Expression $\to$ Well Typing $+\{fail\}$
let $(p, C) = TYPE(A, N)$ in
if C is consistent then
let $C_*, A_* \vdash N : p_*$ in
$SIMPLIFY(C_*), A_* \vdash N : p_*$ else fail

In practice, determining consistency is overlapped with computing the minimal matching instance for a typing. Instead of the "consistency check-instantiate-simplify" sequence given above, we use the following sequence of tests and reductions:

MATCH : Coercion Set $\to$ Substitution $+\{fail\}$
SIMPLIFY : Coercion Set $\to$ Atomic Coercion Set
CONSISTENT : Atomic Coercion Set $\to$ Boolean $+\{fail\}$

If $MATCH(C)$ succeeds and returns substitution S, then $S(C)$ is the minimal matching instance of C. If it fails, C is *structurally* inconsistent: it either entails a cyclic inclusion ($\alpha \rhd \alpha \to \beta$) or an inclusion where the type constructors differ at the top-level ($\alpha \to \beta \rhd (\gamma, \delta)$). $CONSISTENT(C)$ determines whether C is consistent: whether there exists some S such that $S(C)$ is valid.

6.1 Algorithm MATCH

It is useful to consider MATCH as a variation on the classical unification algorithm [MM82]. A unification algorithm can be modelled as the process of transforming a set of equations E into a substitution S, such that S unifies E. Similarly, MATCH should be viewed as transforming a coercion set C into a substitution S, such that S(C) is the minimal matching instance of C. In addition to S and C, MATCH maintains a third data-structure M which represents an equivalence relation over atomic types occurring in C and M. The main idea is that if atomic types a, a' belong to the same equivalence class in M, we must ensure that $S(a), S(a')$ are matching. Following the description of unification in [MM82] we describe MATCH in terms of three transformations on the tuple (C, S, M): (i) Decomposition (ii) Atomic Elimination and (iii) Expansion.

We write $\{(a, a') \mid \ a \ M \ a'\}$ for M. The following conventions are followed below: (1) v denotes type variables (2) a, a' denote atomic type expressions (3) α, β, α', and β' denote type expressions (4) t, t' denote non atomic type expressions. Let M be an equivalence relation defined as before, M_v is the equivalence relation obtained from M by deleting the equivalence class containing v and :

$$[a]_M \stackrel{\text{def}}{=} \{a' \mid (a, a') \in M\}$$
$$[t]^M \stackrel{\text{def}}{=} \{[a]_M \mid a \text{ occurs in } t\}$$

If A is a set of pairs of atomic types then A^* is the reflexive, symmetric, and transitive closure of the relation represented by A. $ALLNEW(t)$ is the type expression obtained from t by substituting "new" variables for every occurrence of variable or constant in t. $ALLNEW(v \rightarrow v * int) = \alpha \rightarrow \beta * \gamma$, where α, β, and γ are new variables.

$$PAIR(t, t') \stackrel{\text{def}}{=} \{(a, a') \mid \ a \text{ occurs in t at the same position as } a' \text{ occurs in } t'\}$$

Definition 2 Decomposition
$(C \cup \{t \rhd t'\}, S, M)$:

case $t \rhd t'$ of:

(1) $\alpha \rightarrow \beta \ \rhd \ \alpha' \rightarrow \beta'$:
Replace $C \cup \{t \rhd t'\}$ by $C \cup \{\alpha' \rhd \alpha, \ \beta \rhd \beta'\}$
(2) $\alpha * \beta \ \rhd \ \alpha' * \beta'$:
Replace $C \cup \{t \rhd t'\}$ by $C \cup \{\alpha \rhd \alpha', \ \beta \rhd \beta'\}$
(3) **else fail**

Definition 3 Atomic elimination
$(C \cup \{a \rhd a'\}, S, M)$:
Replace M *by* $(M \cup \{(a, a')\})^*$ *and delete the coercion* $a \rhd a'$ *from* C.

Definition 4 Expansion
$(C \cup \{e\}, S, M)$ where e is either $v \rhd t$ or $t \rhd v$.

if $[v]_M \in [t]^M \ \vee \ [v]_M$ contains type constant **then fail else**
for $x \in [v]_M$ **do**
begin
$t' \longleftarrow ALLNEW(t);$
$\delta \longleftarrow \{t'/x\};$
$(C, S, M) \longleftarrow (\delta(C), \delta \circ S, (M \cup PAIR(t, t'))^*);$
end
$M \longleftarrow M_v$

procedure $MATCH(\bar{C})$
begin
$(C, S, M) \longleftarrow (\bar{C}, Id, \{(a, a) \mid a \in \bar{C}\});$
while $C \neq \emptyset$ **do**
begin
choose any $e \in C$;
case e of:
(1) $t \rhd t'$: perform **Decomposition**
(2) $a \rhd a'$: perform **Atomic elimination**
(3) $v \rhd t \ \vee \ t \rhd v$: perform **Expansion**
end case
end
return S
end

Example 6 *Let* $\bar{C} = \{\alpha \to \beta \ \rhd \ int \to int * \gamma, \ int \ \rhd \gamma\}$.

C	S	M	Action
$\bar{C}$	Id	$\{\{\alpha\},\{\beta\},\{\gamma\},\{int\}\}$	–
$\{int \ \rhd \ \alpha, \beta \ \rhd \ int * \gamma, int \ \rhd \ \gamma\}$	Id	$\{\{\alpha\},\{\beta\},\{\gamma\},\{int\}\}$	*Decomposition*
$\{\beta \ \rhd \ int * \gamma, int \ \rhd \ \gamma\}$	Id	$\{\{\alpha, int\},\{\beta\},\{\gamma\}\}$	*AtomicElim.*
$\{\beta \ \rhd \ int * \gamma\}$	Id	$\{\{\alpha, int, \gamma\},\{\beta\}\}$	*AtomicElim.*
$\emptyset$	$\{\beta' * \beta''/\beta\}$	$\{\{\alpha, int, \beta', \beta'', \gamma\}$	*Expansion*

As expected, $\{\beta' * \beta''/\beta\}$ *is the minimal matching substitution for* $\bar{C}$.

Example 7 *Let* $\bar{C} = \{v \ \rhd \ \alpha \to \beta, v \ \rhd \ \alpha\}$, *the coercion set associated with the expression* $\lambda \, x. \ x \ x$.

C	S	M	Action
$\bar{C}$	Id	$\{\{\alpha\},\{\beta\},\{v\}\}$	–
$\{v' \to v'' \ \rhd \ \alpha\}$	$\{v' \to v''/v\}$	$\{\{\alpha, v'\},\{\beta, v''\}\}$	*Expansion*

Now let $e \equiv v' \rightarrow v'' \rhd \alpha$. *As* $[\alpha]_M = \{\alpha, v'\}$ *and* $[v' \rightarrow v'']^M = \{\{\alpha, v'\}, \{\beta, v''\}\}$, *Therefore the Expansion step fails, causing MATCH to fail and indicating that the coercion set* $\bar{C}$ *is inconsistent.*

The correctness of MATCH is proved below.

6.2 MATCH is correct

We first introduce the concept of "**approximation**" to model our algorithm. Let C be a coercion set, S be a substitution, and M be an equivalence relation on atomic types occurring in C or S. The tri-tuple (C, S, M) is an **approximation** to a coercion set $\bar{C}$ iff the following conditions hold:

1. $\forall \lambda, \quad \lambda(C)$ is matching $\wedge$ λ respects $M \quad \supset \quad (\lambda \circ S)(\bar{C})$ is matching.
2. $\forall \theta, \quad \theta(\bar{C})$ is matching $\quad \supset \quad \exists \lambda$ such that:
   - $\theta = \lambda \circ S$
   - $\lambda(C)$ is matching
   - λ respects M

where λ respects M iff $\forall\ a, \quad a\ M\ a', \quad \Rightarrow \lambda(a)$ matches $\lambda(a')$. Intutively, the first condition should be read as: if substitution λ "solves" C and M then we can solve $\bar{C}$ by $\lambda \circ S$. The second condition should be read as: any solution to $\bar{C}$ can be obtained from S by composing it with some solution to C and M. Therefore, S is the partial solution of $\bar{C}$ and C and M correspond to the unsolved part of $\bar{C}$. In particular, let $M = \{(a, a')\}$ iff $a = a' \ \in \ \bar{C}$, $C = \bar{C}$, and $S = Id$ then $A_0 \stackrel{\text{def}}{=} (C, S, M)$ is an **approximation** to $\bar{C}$ and further any substitution that makes $\bar{C}$ match must make C match and respect M. Moreover, if there exists an **approximation** $(\emptyset, S, M)$ to $\bar{C}$ then, by choosing λ to be the identity substitution Id in (2) above, we can show that S is the most general matching substitution for C. As shown above, MATCH starting from A_0, generates a sequence of **approximations** $A_0, A_1, ...$ by nondeterministically executing Decomposition, Atomic Elimination and Expansion. If $\bar{C}$ is matchable the algorithm terminates with $A_n = (\emptyset, S_n, M_n)$. Otherwise it fails.

Lemma 2 Let $(C \cup \{t \rhd t'\}, S, M)$ be an **approximation** to $\bar{C}$. If **Decomposition** fails then $\bar{C}$ is not matchable else the resulting (C, S, M) is still an **approximation** to $\bar{C}$.

proof: Trivial. □

Lemma 3 Let $(C \cup \{a \rhd a'\}, S, M)$ be an **approximation** to $\bar{C}$. The result of applying **Atomic elimination** is still an **approximation** to $\bar{C}$.

proof: Trivial. □

Lemma 4 Let $(C \cup \{e\}, S, M)$ be an **approximation** to $\bar{C}$, where e is either $v \rhd t$ or $t \rhd v$. If **Expansion** fails then $\bar{C}$ is not matchable else the resulting tri-tuple is still an **approximation** to $\bar{C}$.

proof: By the fact that $[v]_M$ is finite and left unchanged during the execution of the loop, the for loop must terminate. The rest of the proof is by induction on the number of times the for loop is executed. □

We now prove the termination of this algorithm. Let $|M|$ be the number of equivalent classes in M and $|C|$ be the number of occurrences of symbols in C. We define the lexicographic order $<$ between pairs of M and C in the natural way. More precisely, $(M_1, C_1) < (M_2, C_2)$ iff either $| M_1 | < | M_2 |$ or $| M_1 | = | M_2 |$ and $| C_1 | < | C_2 |$. Obviously, the set of M, C pairs is well founded under $<$. In the following Lemma, we show that "$MATCH$" always terminates.

Lemma 5 "$MATCH$" always terminates.

proof: Let M_1, C_1 and M_2, C_2 denote the values of M, C before and after any pass of the while loop. No matter what transformation is made, we have $(M_2, C_2) < (M_1, C_1)$. By the well-founded property, the algorithm must terminate. □

Theorem 3 (Correctness of $MATCH$)
If $\bar{C}$ is not matchable then $MATCH$ fails else S is returned where $S(C)$ is the minimal matching instance of $\bar{C}$.

proof: By previous lemmas.

6.3 Algorithm CONSISTENT

Let C be an atomic coercion set. C is consistent if we can find some substitution S, mapping type variables in C to type constants, such that $\emptyset \vdash S(C)$. As we are not interested in any details of the substitution S, CONSISTENT determines whether there is anyway C can be consistent.

Let T be a finite set of type constants, $T\uparrow = \{t \mid \exists t' \in T, \vdash t' \rhd t\}$ and $T\downarrow = \{t \mid \exists t' \in T, \vdash t \rhd t'\}$. With each $a \in C$ we associate I_a to stand for the set of types that a can be instantiated to. We set $I_a = \{*\}$ to indicate there are no constraints on a. Let a be an atomic type expression, $var(a)$ is $True$, if a is a variable, $False$, otherwise. Let T_1 and T_2 be finite sets of type constants.

$$COMPRESS(T_1, T2) \stackrel{\text{def}}{=} \text{if } T_1 \cap T_2 = \emptyset \text{ then } fail$$
$$\text{else if } T_1 \cap T_2 = T_1 \text{ then } (True, T_1) \text{ else } (False, T_1 \cap T_2)$$

procedure CONSISTENT(C);
begin
for each $a \in C$ **do**
if $var(a)$ **then** $I_a \longleftarrow \{*\}$ **else** $I_a \longleftarrow \{a\}$;

```
    do
      stable ⟵ True;
      for a ▷ a' ∈ C do
          begin
            (stable, I_a') ⟵ let (flag, I) = COMPRESS(I_a', I_a ↑) in (stable ∧ flag, I);
            (stable, I_a) ⟵ let (flag, I) = COMPRESS(I_a, I_a' ↓) in (stable ∧ flag, I);
          end
    until stable
    return True
end;
```

In CONSISTENT, we start by initializing I_a to $\{*\}$, if $var(a)$, $\{a\}$, otherwise. During each pass of the loop, if some I_a converges to $\emptyset$ then C is obviously inconsistent and the algorithm fails. Otherwise either all I_a's are left unchanged, causing the algorithm to terminate and return $True$, or at least one of them is decreased, causing the do loop to be executed again. Since there are only finitely many type constants and the assignment to I_a is finite therefore the algorithm must terminate. Moreover, when the algorithm terminates successfully, the following condition holds:

$$\forall a \in C,\ I_a = (\bigcap_{a' \in above(a)} I_{a'}\downarrow) \cap (\bigcap_{a'' \in below(a)} I_{a''}\uparrow)$$

where $above(a) = \{a' \mid a \rhd a' \in C\}$ and $below(a) = \{a'' \mid a'' \rhd a \in C\}$. We conjecture that this condition guarantees the consistency of C. However, how to prove this is still an open question.

Example 8 *Let posint* $\rhd int, int \rhd real$ *be the valid atomic coercions and let* $C = \{int \rhd v_1,\ v_1 \rhd v_2,\ v_2 \rhd v_3,\ v_3 \rhd int\}$.

iteration	*edge*	I_{int}	I_{v_1}	I_{v_2}	I_{v_3}
0		$\{int\}$	$\{*\}$	$\{*\}$	$\{*\}$
1	$int \rhd v_1$	{ *int* }	{ *int,real* }	$\{*\}$	$\{*\}$
1	$v_1 \rhd v_2$	{ *int* }	{ *int,real* }	$\{int, real\}$	$\{*\}$
1	$v_2 \rhd v_3$	{ *int* }	{ *int,real* }	$\{int, real\}$	$\{int, real\}$
1	$v_3 \rhd int$	{ *int* }	{ *int,real* }	$\{int, real\}$	$\{int\}$
2	$int \rhd v_1$	{ *int* }	{ *int,real* }	$\{int, real\}$	$\{int\}$
2	$v_1 \rhd v_2$	{ *int* }	{ *int,real* }	$\{int, real\}$	$\{int\}$
2	$v_2 \rhd v_3$	{ *int* }	{ *int,real* }	$\{int\}$	$\{int\}$
2	$v_3 \rhd int$	{ *int* }	{ *int,real* }	$\{int\}$	$\{int\}$
3	$int \rhd v_1$	{ *int* }	{ *int,real* }	$\{int\}$	$\{int\}$
3	$v_1 \rhd v_2$	{ *int* }	{ *int* }	$\{int\}$	$\{int\}$

The last change occurs in the third iteration; the algorithm will go through a fourth step and find that no I assignment has changed.

Example 9 Let $C = \{int \rhd v_1,\ v_2 \rhd v_1,\ v_2 \rhd bool\}$. Let the valid atomic coercions be as before.

iteration	edge	I_{int}	I_{v_1}	I_{v_2}	I_{bool}
0	-	{ int }	$\{*\}$	$\{*\}$	$\{bool\}$
1	$int \rhd v_1$	{ int }	{ int,real }	$\{*\}$	$\{bool\}$
1	$v_2 \rhd v_1$	{ int }	{ int,real }	$\{int, real\}$	$\{bool\}$
1	$v_2 \rhd bool$	{ int }	{ int,real }	$\emptyset$	$\{bool\}$

In the first iteration we find that there is no consistent assignment to type variable v_2.

7 Polymorphism

A major practical goal in type inference systems is to permit programmer-defined names to possess multiple types. This phenomenon has been given the name *polymorphism.* In the system described above, expressions may possess multiple typings but in any individual typing programmer-defined names can only behave monomorphically – possess single types.

In ML this problem is resolved by the use of syntactic device: the *let* expression. Names defined using "let" are permitted to possess type-schemes (quantified types) instead of a type. Names defined in lambda-expression continue to behave monomorphically, and may only possess types. Quantified type is suitably instantiated in different contexts to permit the let-bound name to behave polymorphically.

We take a similar approach in our system. We distinguish between types, written τ, and type-schemes, written σ. In contrast to ML the notion of a simple quantified type, as in $\forall\alpha.\sigma$, is *not* adequate for our purposes. Instead, the relevant concept is that of a *conditionally* quantified type $\forall\alpha|_C.\sigma$. The quantified variable α is conditioned by the constraints appearing in C.

$$\sigma \ ::= \tau \mid \forall\alpha|_C.\sigma$$

By a generic instance of a type scheme $\forall\alpha|_C.\sigma$ we mean the pair $(C', \sigma') = [t/\alpha](C, \sigma)$, provided no capture of free variables in t occurs. We also write $(C', \sigma') \in geninst(\forall\alpha|_C.\sigma)$.

$$C \downarrow \alpha = C', \text{ where } (\alpha \text{ subterm of } t_i \text{ or } t_j \ \wedge\ C \Vdash t_i \rhd t_j) \implies C' \Vdash t_i \rhd t_j$$

$$\frac{C, A \vdash N : \forall\alpha|_{C_1}.\sigma, (C_1', \sigma') \in geninst(\forall\alpha|_{C_1}.\sigma), C \Vdash C_1'}{C, A \vdash N : \sigma'}$$

$$\frac{C, A \vdash N : \sigma, \alpha \notin FV(A)}{C, A \vdash N : \forall\alpha|_{C\downarrow\alpha}.\sigma}$$

$$\frac{C, A \vdash N : \sigma, \quad C, A\,;\, x : \sigma \vdash M : \tau}{C, A \vdash \textbf{let x = N in M} : \tau}$$

References

[Car84] L Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types: LNCS 173*, 1984.

[Car85] L Cardelli. Basic polymorphic typechecking. 1985. Manuscript.

[DM82] L Damas and R. Milner. Principal type schemes for functional programs. In *POPL IX*, 1982.

[Lei83] D Leivant. Polymorphic type inference. In *POPL X*, 1983.

[Mal87] J Malhotra. *Implementation Issues for Standard ML.* Master's thesis, SUNY at Stony Brook, August 1987.

[Mit84a] J. C. Mitchell. Coercion and type inference. In *POPL XI*, 1984.

[Mit84b] J. C. Mitchell. Type inference and type containment. In *Semantics of Data Types: LNCS 173*, 1984.

[MM82] A Martelli and U Montanari. An efficient unification algorithm. *TOPLAS*, 4(2), 1982.

[MR85] P Mishra and U.S Reddy. Declaration-free type inference. In *POPL XII*, 1985.

[Rey70] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence 5*, 1970.

[Rey80] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation: LNCS 94*, 1980.

[Rey85] J. C. Reynolds. Three approaches to type structure. In *TAPSOFT 1985: LNCS 186*, 1985.

[Wan87a] M. Wand. Complete type inference for simple objects. In *LICS II*, 1987.

[Wan87b] M Wand. A simple algorithm and proof for type inference. 1987. Manuscript.