Theoretical Computer Science 73 (1990) 155-175
North-Holland

TYPE INFERENCE WITH SUBTYPES *

You-Chin FUH** and Prateek MISHRA

Department of Computer Science, The State University of New York at Stony Brook, Stony Brook, New York 11794-4400, USA

Abstract. We extend polymorphic type inference with a very general notion of subtype based on the concept of type transformation. 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. As a consequence, it can be used as the basis for type inference with a broad class of subtype theories. For a particular "structural" theory of subtypes, those engendered by inclusions between type constants only, we show that principal types are compactly expressible. This suggests that type inference for the structured theory of subtypes is feasible. We describe algorithms necessary for such a system. The main algorithm we develop is called MATCH, an extension to the classical unification algorithm. A proof of correctness for MATCH is given.

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 [2, 4, 15] 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 [1, 5].

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 very general and is based on the notion of *type transformation*: type t_1 is a subtype of type t_2, written $t_1 \rhd t_2$, if there is a function that maps every value with type t_1 to a value of type t_2. Such a function may even be a many-one function which can be thought of as embedding one type in another.

Traditional subtype relationships are obviously subsumed by this framework: *int* $\rhd$ *real*, *char* $\rhd$ *string* etc. In addition, such a framework could also accommodate subtype relationships between user-defined types. For example, the following natural

* This work has been partially supported by NSF CCR-8706973.
** Present affiliation: Expertest Inc., 2101 Landings Drive, Mountain View, CA 94043, USA.

relationship, which might arise when defining an interpreter or denotational semantics for a programming language, is expressible:

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

In such a case, in addition to indicating the relationship between types, the user would need to provide type transformation functions that map values from the subtype to the supertype.

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 underlving theory of subtypes. As a consequence, it can be used as the basis for type inference with a broad class of subtype theories.
- For a particular "structural" theory of subtypes, those engendered by inclusions between type constants only, we show that principal types are compactly expressible.[1] This suggests that type inference for the structural theory of subtypes is feasible. We describe algorithms necessary for such a system. The main algorithm we develop is called MATCH, an extension to the classical unification algorithm. A proof of correctness for MATCH is given.

1.1. *What is the problem?*

In the absence of subtypes, the set of all typings possessed by a program may be represented by a principal type consisting of a type expression alone. We show that even in the presence of the simplest possible subtype relationship, this is no longer the case.

Let $int \rhd real$ and term $I \equiv \lambda x.x$. Clearly, any proposed principal type for I must be of the form $\alpha \to \alpha$, as I is a function and does not possess type $real \to int$. However, as $int \rhd real$, one type I possesses is $int \to real$, and this type is not a substitution instance of $\alpha \to \alpha$. 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 an instance of τ or *is a supertype of some instance of* τ. From the standard semantics for $\to$, we have

$$int \rhd real \Rightarrow int \to int \rhd int \to real.$$

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

$$int \rhd real \Rightarrow \tau' \rhd (real \to int) \to (real \to int).$$

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,

[1] Such a subtype theory was first studied by Mitchell [8].

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 a principal type associated with the program. For example, we would compute the coercion set-type pair (C, γ) for the identity function I, where $C \equiv \{\alpha \to \beta \rhd \gamma, \alpha \rhd \beta\}$. Any substitution S that satisfies every coercion in C yields a typing $S(\gamma)$ for I.

1.3. A structural 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 complex and would in many applications require further simplification.

A specific subtype theory, which we call *structural subtyping*, is based on the structure of type expressions in that *every* subtype relation is the consequence of subtype relations 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

$$(\{\alpha \rhd \beta\}, \alpha \to \beta).$$

1.4. Implementing structural subtype inference

In the remainder of the paper we discuss concepts and algorithms required for an implementation of structural subtype inference. Section 6.2 describes algorithm MATCH together with its correctness proof; Section 6.4 describes the problem of consistency checking and some possible solutions; Section 6.5 includes a discussion of polymorphism.

2. Related work

Discussion on the semantics of subtypes in a very general category-theoretic setting has appeared in [12]. In [13] 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 [8] it was first shown

that principal types for "structural" subtype theory can be represented using coercion sets consisting only of atomic types. A type inference procedure for the pure lambda-calculus was also outlined therein (in particular, we follow Mitchell in using the term MATCH), but algorithms were omitted. In [7] 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.

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

As examples, consider value expressions

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

and type expressions $\textit{int} \to (\alpha \to \textit{bool})$, $(\textit{int}, \alpha)$.

A coercion is an ordered pair of type 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. $[t_1/\alpha_1, \ldots, t_n/\alpha_n]$ 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 that considered as a relation on $\textit{Type} \times \textit{Type}$ 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.

Define the relation $\Vdash$ on coercions by

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

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 quadruple:

$$\text{Coercion Set} \times \text{Type Assumption} \times \text{Value Expression} \times \text{Type Expression}.$$

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. The FUN rule allows the specification of typings for composite expressions as a function of the typings of the list of bound names ($\bar{x}$) and subexpressions (N_i) that make up a composite expression.

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

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

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

For some instances of FUN consider the following:

$$\frac{C, A; x : t_1 \vdash N : t_2}{C, A \vdash \lambda^{[r}_{L} x](N) : t_1 \to t_2} \qquad \frac{C, A \vdash P : \mathit{bool},\ M : t,\ N : t}{C, A \vdash \text{if } P \text{ then } M \text{ else } N : t}$$

$$\frac{C, A \vdash M : t_1 \to t_2,\ N : t_1}{C, A \vdash MN : t_2} \qquad \frac{}{C, A \vdash \mathit{true} : \mathit{bool}}$$

$C, A \vdash N : p$ is a *typing* if ($N \equiv x$ and $C \Vdash \{A(x) \rhd p\}$) or $N \equiv f^m[\bar{x}](N_1, \ldots, N_m)$ and

(1) $(\bar{q}, v_i, u)$ is a substitution instance of $(\bar{s}^f, r_i^f, w^f)$,
(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 3.1 (*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 3.2. *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, a *principal type* for M. We assume the existence of function *new*: (Type Expression)$^k \to$ (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:**
  $(A, f^m[\bar{x}](e_1, \ldots, e_n), t)$:
    $(\bar{q}, v_1, \ldots, v_n, u) = new(\bar{s}, r_1, \ldots, r_n, w)$;
    $C \leftarrow C \cup \{u \rhd t\}$;
    $G \leftarrow (G - \{g\}) \cup \{(A; \bar{x} : \bar{q}, e_i, v_i)\}$;
  (A, x, t):
    $C \leftarrow C \cup \{A(x) \rhd t\}$;
    $G \leftarrow G - \{g\}$;
  end case;
Output: (α_0, C).

Example 3.3. Let $N \equiv \lambda f.\lambda x.fx$ and $A = \emptyset$. Then

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

It is interesting to compare our algorithm with that given in [8]. Mitchell's algorithm works for the structural subtype theory and interleaves the generation of

the constraint set with the process of simplifying it. In contrast, our algorithm is designed to work for a class of subtype theories and is concerned only with the generation of the relevant coercion set. In this way, we separate the "syntactic" aspects of type inference (traversal of the abstract syntax tree, generation of coercion set) from the details of processing the coercion 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, $\rhd$ and $\Vdash$, that we have presented above. Thus, we expect that in many applications we will be able to reuse the framework given above as the basis for type inference with different subtype theories. Further, as our algorithm does not commit itself to any particular method for processing coercion sets, it can serve as the basis for algorithms that utilize different methods. This is of importance as details of coercion set processing depend critically on the particulars of the subtype theory as well as the application area of interest.

Theorem 3.4. TYPE *is sound and (syntactically) complete.*

Proof. Proof is given below. □

3.3. TYPE *is sound and complete*

Our proof follows Wand's [15] 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 triple (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, we prove that the following invariants hold before and after execution of the main loop in TYPE. We will write C_i, G_i for the values of variables C, G *before* the ith execution of the main loop in TYPE; thus $C_0 = \emptyset$ and $G_0 = \{(A_0, e_0, \alpha_0)\}$. Proof is by induction on the total number of times the main loop in TYPE is entered.

- TYPE is sound:

$$(\forall(\bar{C}, \sigma))((\bar{C}, \sigma) \models (C_i, G_i) \Rightarrow \bar{C}, \sigma(A_0) \vdash e_0 : \sigma(\alpha_0)),$$

- TYPE is complete:

$$\bar{C}, \bar{A} \vdash e_0 : \bar{t} \wedge \exists \delta\, \bar{A}|_{FV(e_0)} = \delta(A_0)|_{FV(e_0)}$$

$$\Rightarrow (\exists \sigma)((\bar{C}, \sigma) \models (C_i, G_i) \wedge \bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)} \wedge \bar{t} = \sigma(\alpha_0)).$$

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: Assume theorem holds for ith step; show for $(i+1)$st step.

(A, x, t): By assumption

$$(\bar{C}, \sigma)\models(C_{i+1}, G_{i+1})$$
$$\Rightarrow (\bar{C}, \sigma)\models\{A(x)\rhd t\}$$
$$\Rightarrow (\bar{C}, \sigma)\models(A, x, t) \quad \text{(by typing rule)}$$
$$\Rightarrow (\bar{C}, \sigma)\models(C_i, G_i)$$
$$\Rightarrow \bar{C}, \sigma(A_0)\vdash e_0:\sigma(\alpha_0) \quad \text{(by hypothesis).}$$

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

$$(\bar{C}, \sigma)\models(C_{i+1}, G_{i+1})$$
$$\Rightarrow (\bar{C}, \sigma)\models\{u\rhd i\} \text{ and } (\bar{C}, \sigma)\models\{A; \bar{x}:\bar{q}, e_i, v_i\}$$
$$\Rightarrow \bar{C}\Vdash\sigma(\{u\rhd t\}) \text{ and } \bar{C}, \sigma(A[\bar{x}:\bar{q}])\vdash e_i:\sigma(v_i) \quad \text{(by definition of } \models)$$
$$\Rightarrow \bar{C}, \sigma(A)\vdash f^m[\bar{x}](e_1, \ldots, e_n):\sigma(t) \quad \text{(by typing rule)}$$
$$\Rightarrow (\bar{C}, \sigma)\models(C_i, G_i)$$
$$\Rightarrow \bar{C}, \sigma(A_0)\vdash e_0:\sigma(\alpha_0) \quad \text{(by hypothesis).}\ \square$$

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 is obvious that $\exists\sigma$ such that $\bar{A}|_{FV(e_0)}=\sigma(A_0)|_{FV(e_0)}$ and $\bar{t}=\sigma(\alpha_0)$.

Step: Assume theorem holds for ith step; show for $(i+1)$st iteration.

(A, x, t):

$$\bar{C}, \bar{A}\vdash e_0:\bar{t}$$
$$\Rightarrow (\exists\sigma)((\bar{C}, \sigma)\models(C_i, G_i)\wedge\bar{A}|_{FV(e_0)}=\sigma(A_0)|_{FV(e_0)}\wedge\bar{t}=\sigma(\alpha_0)) \quad \text{(by hypothesis)}$$
$$\Rightarrow \bar{C}\Vdash\sigma(\{A(x)\rhd t\}) \quad \text{(by typing rule)}$$
$$\Rightarrow (\bar{C}, \sigma)\models\{A(x)\rhd t\} \quad \text{(by definition of } \models)$$
$$\Rightarrow (\bar{C}, \sigma)\models(C_{i+1}, G_{i+1})\wedge\bar{A}|_{FV(e_0)}=\sigma(A_0)|_{FV(e_0)}\wedge\bar{t}=\sigma(\alpha_0).$$

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

$$\bar{C}, \bar{A}\vdash e_0:\bar{t}$$
$$\Rightarrow (\exists\sigma_0)((\bar{C}, \sigma_0)\models(C_{i+1}, G_{i+1})\wedge\bar{A}|_{FV(e_0)}=\sigma_0(A_0)|_{FV(e_0)}\wedge\bar{t}=\sigma_0(\alpha_0)) \quad \text{(by hypothesis)}$$

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

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

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

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

$\Rightarrow (\bar{C}, \sigma) \models (C_{i+1}, G_{i+1}) \wedge \bar{A}|_{FV(e_0)} = \sigma(A_0)|_{FV(e_0)} \wedge \bar{t} = \sigma(\alpha_0)$. □

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 $\{\alpha \rhd \mathit{real}\}$, $\emptyset \vdash N : \alpha$ are of the form $\{t' \rhd \mathit{real}\} \cup C, A \vdash N : t'$, provided the coercions in $\{t' \rhd \mathit{real}\} \cup C$ 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 $\{\mathit{bool} \rhd \alpha, \mathit{int} \rhd \alpha, \alpha \rhd \mathit{real}\}$. We cannot find any type t such that replacing α by t in C results in a valid coercion set.

A coercion set C is *consistent* iff there exists a substitution S such that every coercion in $S(C)$ is valid. 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 3.2 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 × Value Expression → Well Typing + {*fail*}
let $(p, C) = \mathrm{TYPE}(A, N)$ 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. Structural type inclusion

In this section, we develop type inference methods for the case where the underlying theory of type inclusion arises from the structure of type expressions. We study the simplest such case: all inclusions are the consequences of inclusions between type constants. The following rules define the relation $\Vdash$ for the theory of interest.

$$\text{[AXIOM]} \quad C \cup \{t_1 \rhd t_2\} \Vdash t_1 \rhd t_2, \qquad \text{[REFLEX]} \quad C \Vdash t \rhd t,$$

$$\text{[TRANS]} \quad \frac{C \Vdash t_1 \rhd t_2,\ t_2 \rhd t_3}{C \Vdash t_1 \rhd t_3}, \qquad \text{[CONST]} \quad C \Vdash g^0 \rhd h^0,$$

$$\text{[STRUCT]} \quad \frac{C \Vdash t_i \circledast t_i',\ i = 1, \ldots, n,\ \circledast = \rhd \text{ or } \lhd}{C \Vdash g^n(t_1, \ldots, t_n) \rhd g^n(t_1', \ldots, t_n')}.$$

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. Rule [STRUCT] is a "structural" rule that defines coercions between structured types in terms of coercions between their components. Observe that type expression t_i may either be $\rhd$ or $\lhd$ related to t_i'; in the former case we say g^n is *monotonic* in its ith argument and in the latter, *anti-monotonic* in its ith argument. For some concrete instances of the [STRUCT] rule consider the following:

$$\text{[ARROW]} \quad \frac{C \Vdash t_1' \rhd t_1,\ t_2 \rhd t_2'}{C \Vdash t_1 \to t_2 \rhd t_1' \to t_2'} \qquad \text{[TUPLE]} \quad \frac{C \Vdash t_1 \rhd t_1',\ t_2 \rhd t_2'}{C \Vdash (t_1, t_2) \rhd (t_1', t_2')}$$

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 $\text{IS}_1, \ldots, \text{IS}_k \equiv C \Vdash t_1 \rhd t_2$, where each IS_i is obtained by one of the following:

- IS_i is a substitution instance of [AXIOM], [CONST] or [REFLEX] rules.
- IS_i is derived by an application of the [TRANS] rule to some I_l and I_m, where $l, m < i$.
- IS_i is derived by the application of the [STRUCT] rule to $\text{IS}_{m_1}, \ldots, \text{IS}_{m_n}$, where $m_1, \ldots, m_n < i$.

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 based on the structure of type expressions in contrast to 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 a 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 5.1. 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 5.1, 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,

$Match(g^n(t_1, \ldots, t_n), g^n(t'_1, \ldots, t'_n))$ provided $Match(t_i, t'_i)$, $\quad i = 1 \ldots n.$

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 an 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 5.2. *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 Appendix A. □

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

Example 5.3. Let $N \equiv \lambda f.\lambda x.fx$, $A = \emptyset$ and let $\mathrm{TYPE}(A, N) = (t_N, C)$ as in Example 3.3.

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

$$(t_N)_* \equiv (\beta_3 \to \beta_4) \to (\beta_1 \to \beta_2).$$

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

Another way that two well-typings may have identical information content is that their coercion set components are equivalent; they entail each other.

Example 5.4

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

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.

$$\mathrm{SIMPLIFY}\ C = C, \quad \text{if all coercions in } C \text{ are atomic,}$$

$$\mathrm{SIMPLIFY}\ C \cup \{g^n(t_1, \ldots, t_n) \rhd g^n(r_1, \ldots, r_n)\}$$

$$= \mathrm{SIMPLIFY}\left(C \cup \bigcup_{i=1}^{n} \{t_i \mathbin{\ll\!\gg} r_i\} \right)$$

where $\ll\!\gg = \rhd$ if g^n is monotonic in its ith position, $\lhd$ otherwise.

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 affect in any way the type variables contained in a coercion set. It is also obvious that the most general well-typing property is also preserved.

Example 5.5. Let C_* be as in Example 5.3.

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

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

6. Implementing structural subtype inference

6.1. *Algorithm WTYPE revisited*

WTYPE: Type Assumption × Value Expression → Well Typing + {*fail*}
  let $(p, C) = \text{TYPE}(A, N)$ 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 → Substitution + {*fail*},
SIMPLIFY:	Coercion Set → Atomic Coercion Set,
CONSISTENT:	Atomic Coercion Set → 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 differs 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.

6.2. *Algorithm* MATCH

It is useful to consider MATCH as a variation on the classical unification algorithm [6]. 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 [6], we describe MATCH in terms of three transformations on the tuple (C, S, M): (i) Decomposition, (ii) Atomic elimination and (iii) Expansion.

We write $\{(a, a') | a\, M\, a'\}$ 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 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

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

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($v \to v * int$) $= \alpha \to \beta * \gamma$, where α, β and γ are new variables.

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

Definition 6.1. (*Decomposition*). $(C \cup \{t \rhd t'\}, S, M)$.
case $t \rhd t'$ of:

(1) $g^n(\alpha_1, \ldots, \alpha_n) \rhd g^n(\alpha'_1, \ldots, \alpha'_n)$:
Replace $C \cup \{t \rhd t'\}$ by $C \cup \bigcup_{i=1}^{n} \{\alpha_i \circledast \alpha'_i\}$,
where $\circledast = \rhd$ if g^n is monotonic in its ith argument, $\lhd$ otherwise.
(2) **else** *fail.*

Definition 6.2 (*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 6.3 (*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 a type constant **then** *fail* **else**
for $x \in [v]_M$ **do**
begin
$t' \leftarrow$ ALLNEW(t);
$\delta \leftarrow \{t'/x\}$;
$(C, S, M) \leftarrow (\delta(C), \delta \circ S, (M \cup \text{PAIR}(t, t'))^*)$;
end
$M \leftarrow M_v$.

procedure MATCH($\bar{C}$)
begin
$(C, S, M) \leftarrow (\bar{C}, \mathrm{Id}, \{(a, a) \mid a \in \bar{C}\})$;
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 6.4. 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	$\{\{\alpha, int\}, \{\beta\}, \{\gamma\}\}$	Atomic elimination
$\{\beta \rhd int * \gamma\}$	Id	$\{\{\alpha, int, \gamma\}, \{\beta\}\}$	Atomic elimination
$\emptyset$	$\{\beta' * \beta''/\beta\}$	$\{\{\alpha, int, \beta', \beta'', \gamma\}$	Expansion

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

Example 6.5. Let $\bar{C} = \{v \rhd \alpha \to \beta, v \rhd \alpha\}$, the coercion set associated with the expression $\lambda x.xx$.

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

6.3. 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. We say substitution R respects relation M, if for all

$\langle a, a'\rangle \in M$, $Match(R(a), R(a'))$ holds. The triple (C, S, M) is an *approximation* to a coercion set $\bar{C}$ iff the following conditions hold:

(1) $\forall\lambda$, $\lambda(C)$ is matching $\wedge\ \lambda$ respects $M \Rightarrow (\lambda \circ S)(\bar{C})$ is matching;

(2) $\forall\theta$, $\theta(\bar{C})$ is matching $\Rightarrow \lambda$ such that

- $\theta = \lambda \circ S$,
- $\lambda(C)$ is matching,
- λ respects M.

Intuitively, 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 $M = \{(a, a) \mid a \in \bar{C}\}$, $C = \bar{C}$, and $S = \mathrm{Id}$ 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 approximation $A_0, A_1, \ldots$ 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 6.6. *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 6.7. *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 6.8. *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 triple 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 6.9. 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 6.10 (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. □

6.4. Consistency checking

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 $\emptyset \models S(C)$. As we are not interested in any details of the substitution S, it is enough if we can determine the existence of substitution S without constructing it.

In [16] Wand has shown that, in general, determining consistency of a coercion set C must be exponential in the size of C. For the restricted case where the underlying subtype theory is tree-structured, Wand gives a polynomial-time procedure for consistency checking.

From a practical point of view, it would still be useful if a "goal-directed" algorithm with good average-case behaviour could be developed for consistency checking. Unfortunately, we are not aware of any such algorithm.

Our current line of research is to examine a *weaker* notion of consistency given by the following definition: C is consistent if $C \Vdash g^0 \rhd h^0$ implies that $\emptyset \Vdash g^0 \rhd h^0$. Such a definition of consistency is strictly weaker than the definition of consistency given above.

Example 6.11. Let $odd = \lambda f.(f\ 1, f\ true)$. The function *odd* is well-typed with respect to the weaker notion of *consistency* and

$$\{int \rhd \alpha, bool \rhd \alpha\}, \emptyset \vdash odd : (\alpha \to \beta) \to \beta$$

is a well-typing. If *odd* is applied to the successor function whose type is $int \to int$, the coercion set component is instantiated to $\{int \rhd int, bool \rhd int\}$ and is inconsistent with respect to the weaker notion of consistency.

As we can see in the example, with respect to the weaker notion of consistency, part of the consistency checking can be deferred to the future. One practical advantage with this is that consistency checking simply becomes transitive closure. The principal type result remains unaltered and in addition, such a definition supports type inference in the presence of varying subtype theories. For instance, the function *odd* can be successfully used in any context where *int* and *bool* possess a common supertype.

Whether such a definition is appropriate, especially in terms of its interaction with the basic purpose of type inference—the prevention of type errors at runtime—is at the present time not clear to us.

6.5. *Polymorphism*

A major practical goal in type 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, i.e. possess single 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 (and the power!) of type inference with general quantification [4, 9].

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 types are suitably instantiated in different contexts to permit the let-bound name to behave polymorphically.

A similar approach is possible in our system. If let-bound name x possesses type (C, τ) we quantify all the type variables occurring (C, τ) that do not occur ideas in the type assumption A. In different occurrences of x, we can instantiate the quantified type variables with "new" type variables to obtain polymorphic behaviour.

7. Conclusion

In this work we have developed a general framework for type inference with subtypes, and have subsequently carefully derived the important special case of structural subtyping which may be of importance in practice.

The motivation for our general result arose from our struggles with correctness proofs for type inference algorithms as found in the literature. It appeared to us that while the essential ideas underlying the algorithms were intuitively accessible, the algorithms made very specific choices regarding traversal of programs (top-down vs bottom-up), processing of type expressions etc. and as a consequence the correctness proofs were complex. In contrast, Algorithm TYPE as well as the definitions and correctness proofs in Section 3 attempt to capture the essential ideas in subtype inference without entering into details that are specific to particular subtype theories. One indication of the success of our approach is that the correctness proofs are both compact and accessible.

Our motivation in studying structural subtyping was to systematically expose the issues that arise in passing from the general subtype framework to dealing with a specific subtype theory. A number of recent papers have been able to profit from

this paradigm: these papers include Remy's [10] work on record subtyping, Thatte's [14] work on reconciling dynamic and static typing, as well as the work of Kuo [3] on viewing strictness analysis as a form of subtype inference.

Acknowledgment

We thank Esther Shilcrat for her careful reading of, and comments on, several versions of this paper. We have also received valuable criticism from Guy Cousineau, Dave MacQueen, Flemming Nielson, Didier Remy and Satish Thatte.

Appendix A

Proof of Theorem 5.2. We will need to view type expressions as a partially ordered set. The technical machinery required is taken from [11], to which we refer the reader for further details. Terms t_1, t_2 are α-variants of each other if there exists substitutions S, R with the property $t_1 = S(t_2)$ and $t_2 = R(t_1)$. By $Texp_\perp$ we mean the set of type expressions augmented with the element $\perp$ and with α-variants considered equivalent. The set $Texp_\perp$ comes equipped with a natural partial order based on instantiation: $t_1 \geqslant t_2$ if t_2 is an instance of t_1. $Texp_\perp$ has a natural "top" or least instantiated element: the term consisting of a single variable. We assume that $t \geqslant \perp$, $\forall t \in Texp_\perp$. $Texp_\perp$ is closed under a "least upper bound" operator, written $\sqcup$. The $\sqcup$ operator is known as the "anti-unifier" or "least general predecessor" of a finite set of terms.

Definitions A.1

$$x \sqcup y = \begin{cases} x = \perp \Rightarrow y, \\ y = \perp \Rightarrow x, \\ x, y \text{ atomic}, \quad x = y \Rightarrow x, \\ x, y \text{ atomic}, \quad x \neq y \Rightarrow new(\,), \\ x \equiv g^n(x_1, \ldots, x_n), \quad y \equiv g^n(y_1, \ldots, y_n) \\ \quad \Rightarrow g^n(x_1 \sqcup y_1, \ldots, x_n \sqcup y_n), \\ \text{otherwise} \Rightarrow new(\,). \end{cases}$$

For an example, $(\alpha \to (int, bool)) \sqcup (int \to \beta) = \gamma \to \delta$.

Definition A.2. Let $\langle S, \leqslant \rangle$ be a partially ordered set. The least upper bound of X, $X \subseteq S$, written $\bigsqcup X$, is defined by

- $\forall x \in X, \bigsqcup X \geqslant x$.
- $r \geqslant x, \forall x \in X \Rightarrow r \geqslant \bigsqcup X$.

Definition A.3. A (join) complete semi-lattice is a partially ordered set $\langle S, \leqslant \rangle$ such that every $X \subseteq S$ has a least upper bound $\bigsqcup X \in S$.

Theorem A.4 (Reynolds). *$\langle Texp, \leqslant \rangle$ is a (join) complete semi-lattice, with $\bigsqcup \{x_i\} = p_k$, for some k where $p_0 = \bot$ and $p_i = p_{i-1} \sqcup x_i$.*

Lemma A.5. *Let $\{s_i \rhd r_i\}_i$ be an indexed family of coercion sets. Then*

$$\forall i, \quad Match(s_i, r_i) \Rightarrow Match\left(\bigsqcup_i \{s_i\}, \bigsqcup_i \{r_i\}\right).$$

Proof. It is enough to show that $Match(x_1, y)$, $Match(x_2, y) \Rightarrow Match(x_1 \sqcup x_2, y)$. Proof is by structural induction on (x_1, x_2) and is omitted. □

Consider the set $\{S_i(C)\}$ consisting of all matching instances of coercion set C. If C is consistent, this set must always be non-empty.

$$C_* = \forall \{r \rhd s\} \in C, \quad \left\{ \left(\bigsqcup_i r_i\right) \rhd \left(\bigsqcup_i s_i\right) \middle| r_i \rhd s_i = S_i(r) \rhd S_i(s) \right\}.$$

By Theorem A.4, every matching instance of C is an instance of C_*; then every valid instance of C is an instance of C_*. Further, C_* must be an instance of C; hence every valid instance of C_* is also an instance of C. Finally, by Lemma A.5 C_* is a matching coercion set. We will speak of C_* as the minimal matching instance of C and extend this notion to well-typings, writing $C_*, A_* \vdash N : t_*$.

References

[1] L. Cardelli, Basic polymorphic typechecking, Manuscript, 1985.

[2] L. Damas and R. Milner, Principal type schemes for functional programs, in: *Proc. 9th Ann. ACM Symp. on Principles of Programming Languages* (1982) 207–212.

[3] T.-M. Kuo and P. Mishra, Strictness analysis: a new perspective based on type inference, Technical Report, SUNY, Stony Brook, Tech. Report 89-13, 1989.

[4] D. Leivant, Polymorphic type inference, in: *Proc. 10th Ann. ACM Symp. on Principles of Programming Languages* (1983) 88–98.

[5] J. Malhotra, Implementation issues for standard ML, Master's thesis, SUNY at Stony Brook, 1987.

[6] A. Martelli and U. Montanari, An efficient unification algorithm *TOPLAS* 4(2) (1982) 258–282.

[7] P. Mishra and U.S. Reddy, Declaration-free type inference, in: *Proc. 12th Ann. ACM Symp. on Principles of Programming Languages* (1985) 8–19.

[8] J.C. Mitchell, Coercion and type inference, in: *Proc. 11th Ann. ACM Symp. on Principles of Programming Languages* (1984) 175–185.

[9] J.C. Mitchell, Type inference and type containment, in: *Semantics of Data Types*, Lecture Notes in Computer Science 173 (Springer, Berlin, 1984) 257–278.

[10] D. Remy, Typechecking records and variants in a natural extension of ML, in: *Proc. 16th Ann. ACM Symp. on Programming Languages* (1989) 60–76.

[11] J.C. Reynolds, Transformational systems and the algebraic structure of atomic formulas, *Mach. Inteli.* 5 (1970) 135–151.

[12] J.C. Reynolds, Using category theory to design implicit conversions and generic operators, in: *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science **94** (Springer, Berlin, 1980) 211–258.

[13] J.C. Reynolds, Three approaches to type structure, in: *TAPSOFT 1985*, Lecture Notes in Computer Science **186** (Springer, Berlin, 1985) 97–138.

[14] S. Thatte, Type inference with partial types, in: *ICALP 87*, Lecture Notes in Computer Science **317** (Springer, Berlin, 1987) 615–629.

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

[16] M. Wand and P. O'Keefe, On the complexity of type inference with coercion, Manuscript, 1989.