Type inference in systems of recursive types with subtyping

Trevor Jim[*] Jens Palsberg[†]

June 1999

Abstract

We present general methods for performing type inference and deciding subtyping in languages with recursive types. Our type inference algorithm generalizes a common idea of previous work: type inference is reduced to a constraint satisfaction problem, whose satisfiability can be decided by a process of closure and consistency checking. We prove a general correctness theorem for this style of type inference. We define subtyping co-inductively, and we prove by co-induction that a closed and consistent constraint set has a solution. Our theorem makes it easier to find new type inference algorithms. For example, we provide definitions of closure and consistency for recursive types with a greatest type, but not a least type; we show that the definitions satisfy the conditions of our theorem; and the theorem immediately provides a type inference algorithm, thereby solving an open problem.

1 Introduction

Subtyping and recursive types are common in modern programming languages. For example, Java [14] has a notion of subtyping by name based on explicit subtype declarations, and it allows interfaces to be mutually recursive, although there is no unfolding rule. In theoretical studies and experimental languages, one more often finds subtyping by structure without subtype declarations, and full-fledged recursive types which can be unfolded, e.g., [1, 3]. Some languages rely on explicit type annotations and static type checking, e.g., Java [14] and C++ [11]. Others do not require type annotations and rely on dynamic type checking, e.g., Smalltalk [13] and Self [38].

What is not common is *type inference* for real languages with subtyping and recursive types. Although type inference is known for some systems combining both

[*]Department of Computer and Information Science, University of Pennsylvania, 200 S. 33rd Street, Philadelphia, PA 19104–6389, tjim@saul.cis.upenn.edu.

[†]Purdue University, Dept of Computer Science, W Lafayette, IN 47907, palsberg@cs.purdue.edu.

features, these systems tend to lack many features of full-fledged languages. We believe that by studying the problem in general, for a large class of languages, we can learn how to extend type inference to production languages. Our method has already given us algorithms for some languages for which type inference was not previously known.

We formulate recursive types as possibly infinite trees. This is more general than alternatives such as μ notation, because it includes nonregular types. All of our constructions work on both regular and nonregular trees, and result in *algorithms* when types are restricted to be regular.

We consider two kinds of subtyping. In *atomic subtyping*, a subtype order is given for a set of base types. For example, the base types might be **nat**, **int**, and **bool**, where **nat** is a subtype of **int**, but **bool** is unrelated to the other types. Subtyping with a least type $\bot$ and greatest type $\top$ has a distinctly different character than atomic subtyping. In atomic subtyping it is possible to have a least base type, but no base type is a subtype of any function type. In contrast, $\bot$ is a subtype of every type, including both atomic and function types, and $\top$ is a supertype of every type. Thus subtyping with $\top$ and $\bot$ is *nonstructural*: subtyping does not follow the tree structure of types.

Our method applies to systems with both atomic subtyping and nonstructural subtyping, or to systems with one but not the other. A nice consequence of the generality of our method is that it provides a common framework for expressing algorithms for all of these systems. This makes it easy to analyse why difficulties inherent in one system do not arise in another. In particular, we will see that having $\top$ and $\bot$ actually makes type inference easier, and we give the first algorithms for the system with $\top$ but not $\bot$, and the system with $\bot$ but not $\top$. Well-known languages with $\top$ but not $\bot$ include $F_{<:}$ [6] and O–1 [1].

Our algorithm uses well-known ideas, starting with the equivalence between type inference and finding solutions to sets of constraints. A constraint set is simply a relation R on types, and a solution to R is a substitution S such that $S(\sigma) \leq S(\tau)$ for all $(\sigma, \tau) \in R$. So, to find a typing for a term M, we first construct a constraint set R_M; M is typable if and only if R_M has a solution, and a typing for M is easily obtained from any solution.

The constraints R_M are "closed" by certain rules, obtaining an equivalent set R of constraints—that is, R and R_M have exactly the same solutions. A "consistency" test then determines whether R (and therefore, R_M) has a solution. Again, this is a standard technique. However, known methods for constructing solutions, or showing that solutions exist, are complicated. For example, Aiken and Wimmers [2] show that their constraints have solutions by transforming them to contractive equations, a class that MacQueen et al. [20] demonstrated to be solvable by Banach's Fixed Point Theorem. Tiuryn and Wand [36] reduce solvability to the emptiness problem for Büchi automata. Other work [18, 28] transforms R into a language of graphs, and then to a nondeterministic automaton whose language represents the solution.

Our method of constructing a solution is much simpler. To find the solution for a variable t, we start with the "closed" R, and consider the upper and lower bounds of t in R. The bounds constitute a *state* corresponding to a node in a type. A *symbol function* maps a state to a symbol ($\rightarrow$, $\perp$, etc.), and if that symbol is $\rightarrow$, a *transition function* maps the state to two new states which represent the argument and result types.

Our definition of the subtyping relation is based on *simulations*, an idea from concurrency theory [21, 29]. This leads to a very simple algorithm for deciding subtyping, based on co-induction. Co-induction is also used to prove the correctness of our method of finding solutions to constraints.

Brandt and Henglein also use co-inductive techniques to decide subtyping for recursive types [5]. They use μ notation, so their definition applies only to regular types. We show that it applies as well to nonregular types. Pierce and Sangiorgi [30] used simulations to define a subtyping relation, however, they did not base their subtyping algorithm on co-induction. Simulations have been used to compare *elements* of recursive domains, for example, to compare infinite lists, where the domain of lists is recursively defined.

The main observation of this paper is that co-induction can be used to prove a general correctness theorem for a widely applicable approach to type inference.

Overview. In §2 we define the subtyping relation using simulations, and give the co-inductive subtyping algorithm. In §3, we define the type system. In §4 we show how to construct solutions for constraints, and in §5, we show how our method applies to some examples. In §6 we discuss related work.

2 Recursive types and subtyping

We work with a countably infinite set, **Tv**, of *type variables*, ranged over by s, t. A *signature* is a set, Σ, of symbols including at least **Tv** and a distinguished symbol, $\rightarrow$. A signature may also contain distinguished symbols $\top$ and $\perp$, and symbols for base types, e.g., the natural numbers **nat**, the integers **int**, the booleans **bool**, and so on.

We assume that any signature Σ has an associated partial order, $\leq_\Sigma$, on Σ, satisfying the following conditions.

- If $\top \in \Sigma$, then $\top$ is the greatest element in $\leq_\Sigma$.
- If $\perp \in \Sigma$, then $\perp$ is the least element in $\leq_\Sigma$.
- Any type variable is $\leq_\Sigma$-comparable only to itself, and to $\top$ and $\perp$ if they are members of Σ.
- The symbol $\rightarrow$ is $\leq_\Sigma$-comparable only to itself, and to $\top$ and $\perp$ if they are members of Σ.

A *path* is a finite sequence, α, of 0's and 1's; ϵ denotes the empty path. We use ℓ to range over $\{0,1\}$. A Σ *type*, σ, is a partial function from paths into Σ, whose domain is nonempty and prefix closed, and such that $\sigma(\alpha\ell)$ is defined if and only if $\sigma(\alpha) = \rightarrow$. We omit Σ when it can be recovered from context, and we use σ, τ, ρ to range over types. A type is *finite* if its domain has finite cardinality. A *subtree* of a type σ is a type τ such that for some path α, we have $\sigma(\alpha\beta) = \tau(\beta)$ for all paths β. A type is *regular* if it has only a finite number of subtrees. All regular types are finitely representable (for example, by finite state automata, or μ notation).

We now introduce some convenient notation. We write $\sigma(\alpha) = \uparrow$ if σ is undefined on α. We abuse notation and write s for the type σ such that $\sigma(\epsilon)$ is the type variable s and $\sigma(\alpha) = \uparrow$ for all $\alpha \neq \epsilon$. Similarly, we define $\top$, $\bot$, **nat**, and so on, as Σ types (provided they are members of Σ). We define $\sigma_0 \rightarrow \sigma_1$ to be the type σ such that $\sigma(\epsilon) = \rightarrow$, $\sigma(0\alpha) = \sigma_0(\alpha)$, and $\sigma(1\alpha) = \sigma_1(\alpha)$.

Our definition of the subtyping relation generated by a signature is based on the *simulations* familiar from concurrency theory.

Definition 1 *A relation R on Σ types is called a Σ* simulation *if it satisfies the following conditions.*

(C1) *If $(\sigma,\tau) \in R$, then $\sigma(\epsilon) \leq_\Sigma \tau(\epsilon)$.*

(P1) *If $(\sigma_0 \rightarrow \sigma_1, \tau_0 \rightarrow \tau_1) \in R$, then $(\tau_0, \sigma_0) \in R$ and $(\sigma_1, \tau_1) \in R$.*

For example, the empty relation on Σ types and the identity relation on Σ types are both Σ simulations. Σ simulations are closed under union, so there is a largest Σ simulation. We define our subtyping relation, $\leq$, to be this largest simulation:

$$\leq = \bigcup \{ R \mid R \text{ is a simulation } \}.$$

The subtyping relation $\leq$ should properly be annotated with Σ, as we will be considering a number of signatures. However, we wish to avoid possible confusion with the ordering $\leq_\Sigma$ on symbols, so we assume that Σ can be recovered from context.

Alternately, $\leq$ can be seen as the maximal fixpoint of a monotone function on relations between Σ types. Then we immediately have the following result.

Lemma 2 $\sigma \leq \tau$ *if and only if*

- $\sigma(\epsilon) \leq_\Sigma \tau(\epsilon)$*; and*
- *if $\sigma = \sigma_0 \rightarrow \sigma_1$ and $\tau = \tau_0 \rightarrow \tau_1$, then $\tau_0 \leq \sigma_0$ and $\sigma_1 \leq \tau_1$.*

This result is standard in concurrency theory, and has an easy proof, cf. [22]. Similarly, it is easy to show that $\leq$ is a preorder, and that all simulations are antisymmetric. Therefore we have the following result.

Lemma 3 $\leq$ *is a partial order.*

A principle advantage of using simulations to define our subtyping relation is that we may apply the principle of *co-induction* to prove that one type is a subtype of another:

> **Co-induction**: To show $\sigma \leq \tau$, it is sufficient to find a simulation R such that $(\sigma, \tau) \in R$.

The co-induction principal gives us an easy algorithm for subtyping on regular types. Suppose R is a relation on types, and we want to know whether $\sigma \leq \tau$ for every $(\sigma, \tau) \in R$. By co-induction this is equivalent to the existence of a simulation containing R. And since simulations are closed under intersection, this is equivalent to the existence of a *smallest* simulation containing R. This smallest simulation, if it exists, can be found by "closing" R under property P1. If the "closure" is "consistent" (i.e., satisfies property C1), then it is the smallest simulation; and if not, no such simulation exists.

More formally, suppose R' is a relation on types. The *P1-closure* of R' is the least relation R containing R' and satisfying property P1 above. If R satisfies P1, we say *R is P1-closed*. The P1-closure of a relation is well defined, because the P1-closed relations are closed under intersection. Every simulation is P1-closed, and P1-closure is a monotone operation.

We say R is *C1-consistent* if it satisfies property C1 above. Note that any subset of a C1-consistent set is C1-consistent.

Lemma 4 *Let R be a relation on types. The following statements are equivalent.*

1. *The P1-closure of R is C1-consistent.*
2. *The P1-closure of R is a simulation.*
3. *$\sigma \leq \tau$ for every $(\sigma, \tau) \in R$.*

Proof:

- (1) $\Rightarrow$ (2): Immediate by the definition of simulation.
- (2) $\Rightarrow$ (3): Immediate by co-induction.
- (3) $\Rightarrow$ (1): R is a subset of $\leq$, so by the monotonicity of P1-closure and the fact that $\leq$ is P1-closed, the P1-closure of R is a subset of $\leq$. Then since $\leq$ is C1-consistent, its subset, the P1-closure of R, is C1-consistent. ■

This immediately suggests an *algorithm* for testing whether $\sigma \leq \tau$ when σ and τ are regular: construct the P1-closure of $\{(\sigma, \tau)\}$ and test whether it is C1-consistent.

$$(\textsc{const}) \quad A \vdash c^\sigma : \sigma$$

$$(\textsc{var}) \quad A \vdash x : A(x)$$

$$(\textsc{abs}) \quad \frac{A \backslash x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash \lambda x M : \sigma \to \tau}$$

$$(\textsc{app}) \quad \frac{A \vdash M : \sigma \to \tau, \quad A \vdash N : \sigma}{A \vdash MN : \tau}$$

$$(\textsc{sub}) \quad \frac{A \vdash M : \sigma}{A \vdash M : \tau} \quad \sigma \leq \tau$$

Figure 1: Typing rules.

If n is the number of distinct subtrees of σ and τ, then the P1-closure of $\{(\sigma, \tau)\}$ has at most n^2 pairs, and can be constructed in $O(n^2)$ time. The only remaining task is to check C1-consistency. Thus we have the following theorem.

Theorem 5 *If $\leq_\Sigma$ can be checked in constant time, then subtyping for regular Σ types is decidable in $O(n^2)$ time.*

3 Lambda calculi with subtyping

Let Σ be a signature, and let $\mathbf{C}$ be a set of constants, each of which has the form c^σ for a closed Σ type σ. We now define the language $\Lambda(\Sigma, \mathbf{C})$, or Λ_Σ for short.

There is a countably infinite set of *(term) variables*, ranged over by x, y. The *terms of* Λ_Σ are defined by the following grammar.

$$M, N ::= x \mid (\lambda x M) \mid (MN) \mid c^\sigma$$

A Σ *type environment* is a finite set $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of (variable, Σ type) pairs, where the variables $x_1, \ldots, x_n$ are distinct. We use A to range over type environments. We write $A(x)$ for the type paired with x in A, and $A \backslash x$ for the type environment A with any pair for the variable x removed. We write $A_1 \cup A_2$ for the union of two type environments whose variables are disjoint.

We write $\Lambda_\Sigma \triangleright A \vdash M : \sigma$ if the judgment $A \vdash M : \sigma$ follows by the rules of Figure 1, where types are restricted to Σ types, terms are restricted to Λ_Σ terms, and $\leq$ is the subtyping relation generated by $\leq_\Sigma$. The rules (CONST), (VAR), (ABS), and (APP) are the usual typing rules of the lambda calculus with constants, and rule (SUB) is the rule of subsumption.

Example 6

- The system Λ_{Σ_1}, where $\Sigma_1 = \mathbf{Tv} \cup \{\bot, \top, \to, \mathbf{nat}\}$, is the system of recursive types studied by Amadio and Cardelli [3].
- The system Λ_{Σ_2}, where $\Sigma_2 = \mathbf{Tv} \cup \{\top, \to, \mathbf{nat}\}$, is sometimes called the system of *partial types*, after the work of Thatte [32, 33, 34], who studied a nonrecursive version. Other well-known calculi with $\top$ but not $\bot$ include $F_{<:}$ [6] and O–1 [1].
- The system Λ_{Σ_3}, where $\Sigma_3 = \mathbf{Tv} \cup \{\bot, \to, \mathbf{nat}\}$ may be viewed as a dual of Λ_{Σ_2}.
- The system Λ_{Σ_0}, where $\Sigma_0 = \mathbf{Tv} \cup \{\to, \mathbf{nat}\}$, is the system of recursive types without subtyping ($\leq$ is just syntactic equality of types).

As we will see, type inference algorithms for all of these systems are simple applications of our general result.

4 Type inference

It is well known that type inference can be reduced to the problem of finding solutions to constraint sets (in fact, the problems are equivalent for the languages we consider).

Recall that a constraint set is simply a relation, R, on types. To see whether R has a solution, we will use a strategy similar to that used in our subtyping algorithm: first, R is closed by certain rules, then, a consistency check establishes whether a solution exists. A solution to R can be easily "read off" from the closure of R.

A difference from the subtyping algorithm is that for certain signatures, our closure conditions will add "fresh" type variables to the constraints. Therefore, with any relation R, we implicitly associate a set of "fresh" type variables appearing in R. This is a standard technique from unification theory, and is used to ensure that our closure conditions do not change the set of solutions of a relation.

Definition 7 *A Σ* substitution *is a partial function from type variables to Σ types. As usual, any Σ substitution can be extended to a total function from Σ types to Σ types. If R is a relation on Σ types with fresh variables V, then a Σ* solution *to R is a Σ substitution S, such that there is a Σ substitution S', such that for every $(\sigma, \tau) \in R$, we have $S'(\sigma) \leq S'(\tau)$, and $S(s) = S'(s)$ for every $s \notin V$. We say two relations are* equivalent *if they have the same Σ solutions.*

The closure and consistency conditions for finding solutions will be different than those of our subtyping algorithm. And, unlike the subtyping algorithm, the closure and consistency conditions will differ depending on the signature Σ.

Our closure conditions for R will always be *stronger* than the closure condition P1 used for subtyping. For example, in addition to P1, we will include

(P2) R is transitive.

The Property P2, like P1, makes some inconsistencies "immediately apparent." For example, consider a relation R_1 consisting of the pairs

$$(\sigma_0 \to s, \sigma_0 \to \sigma_1 \to \sigma_2) \qquad (\mathbf{nat}, s)$$

If R is the closure of R_1 under P1 and P2, then R contains the pair $(\mathbf{nat}, \sigma_1 \to \sigma_2)$. It is immediately apparent that R has no solution, and we will see that this implies that R_1 has no solution.

Our consistency conditions will always be *weaker* than the condition C1 we used for subtyping. A typical condition is

(C2) $\sigma(\epsilon) \leq_\Sigma \tau(\epsilon)$ whenever $(\sigma, \tau) \in R$ and σ and τ are not type variables.

For example, suppose R_2 is a relation consisting of the pairs

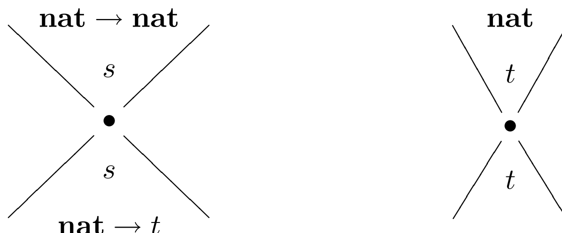
$$(\mathbf{nat} \to t, s) \qquad (s, \mathbf{nat} \to \mathbf{nat})$$

Although R_2 does not satisfy C1, it has a solution—so C1 is too strong. R_2 does satisfy C2, though, as does its closure under P1 and P2:

$$(\mathbf{nat} \to t, s) \quad (s, \mathbf{nat} \to \mathbf{nat}) \quad (\mathbf{nat} \to t, \mathbf{nat} \to \mathbf{nat}) \quad (t, \mathbf{nat}) \quad (\mathbf{nat}, \mathbf{nat})$$

Once we have obtained a closed, consistent relation R, we use the following idea to construct a solution. We must assign each type variable appearing in R a type, while satisfying the constraints imposed by R. The constraints relevant to a particular type variable are just its lower and upper bounds in R. The bounds constitute a *state* that contains all the information needed to construct the type for the variable.

Consider the closure of R_2, above. We will show how to construct types for the variables s and t. The states of s and t are drawn suggestively below. (For technical reasons, s is considered to be its own lower bound even though (s, s) does not appear in the closed relation.)



The idea is to construct a type that will lie between the upper and lower bounds, at the '•'. Assume we are working in the signature $\Sigma_1 = \mathbf{Tv} \cup \{\perp, \top, \to, \mathbf{nat}\}$. Because the state for s, on the left, contains function types in both the upper and

lower bounds, the type we construct for s must have $\rightarrow$ as its root symbol. On the other hand, the state for t gives us a choice: we may use either $\bot$ or **nat** without violating any constraints.

Having chosen $\rightarrow$ as the root symbol for the type of s, we must construct its argument and result types. These types correspond to states that can be derived from the state for s. By the rules for function subtyping, we obtain the following states.

The state on the left forces us to make **nat** the argument type of s (the only type lying between **nat** and **nat** is **nat**). The state on the right indicates that the result type of s should lie between the type of t and **nat**. This is a tricky point. Recall that we had a choice for the type of t: either $\bot$ or **nat**. If we chose **nat** for t, we must choose **nat** as the return type of s. But if we chose $\bot$ for the type of t, then we may choose either $\bot$ or **nat** for the return type. This means that in order to construct a type for a state, we must be aware of how types will be constructed for other states. Our contribution is to give general conditions that say when a choice of symbols will result in a solution.

The three solutions to our example are given below.

$s = \mathbf{nat} \rightarrow \mathbf{nat}$	$s = \mathbf{nat} \rightarrow \bot$	$s = \mathbf{nat} \rightarrow \mathbf{nat}$
$t = \mathbf{nat}$	$t = \bot$	$t = \bot$

We begin with some routine definitions. For any type σ we use the following notation for lower and upper bounds:

$$\begin{aligned} \sigma{\downarrow_R} &= \{\sigma\} \cup \{\tau \mid (\tau, \sigma) \in R\}, \\ \sigma{\uparrow_R} &= \{\sigma\} \cup \{\tau \mid (\sigma, \tau) \in R\}. \end{aligned}$$

If $\sigma = \sigma_0 \rightarrow \sigma_1$, we define $\sigma.0$ to be σ_0 and $\sigma.1$ to be σ_1. This is extended to arbitrary paths: if $\alpha = \ell_1 \ell_2 \cdots \ell_n$, $n \geq 0$, then $\sigma.\alpha = (\cdots((\sigma.\ell_1).\ell_2)\cdots).\ell_n$. For sets of types we define

$$\begin{aligned} \mathbf{T}.0 &= \{\, \sigma_0 \mid \exists \sigma_1.\ \sigma_0 \rightarrow \sigma_1 \in \mathbf{T} \,\} \\ \mathbf{T}.1 &= \{\, \sigma_1 \mid \exists \sigma_0.\ \sigma_0 \rightarrow \sigma_1 \in \mathbf{T} \,\} \\ \mathbf{T}{\uparrow_R} &= \bigcup_{\sigma \in \mathbf{T}} \sigma{\uparrow_R} \\ \mathbf{T}{\downarrow_R} &= \bigcup_{\sigma \in \mathbf{T}} \sigma{\downarrow_R} \end{aligned}$$

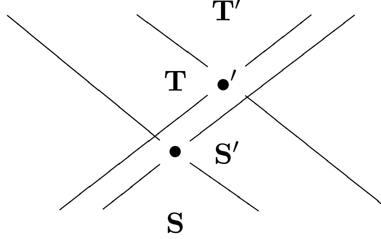
Note that $\mathbf{T}.0$ and $\mathbf{T}.1$ are defined even when $\mathbf{T}$ contains types that are not function types, e.g., $\{\mathbf{nat}, \sigma_0 \to \sigma_1, s\}.0 = \{\sigma_0\}$.

We say σ *appears in* R if for some $(\tau, \tau') \in R$ and path α, $\sigma = \tau.\alpha$ or $\sigma = \tau'.\alpha$.

A Σ *pre-state* is a pair $(\mathbf{S}, \mathbf{T})$ where $\mathbf{S}$ and $\mathbf{T}$ are sets of Σ types. We use g, h to range over pre-states, and we define a partial order, $\leq_{\mathrm{s}}$, on pre-states:

$$(\mathbf{S}, \mathbf{T}) \leq_{\mathrm{s}} (\mathbf{S}', \mathbf{T}') \text{ iff } \mathbf{T}' \subseteq \mathbf{T} \text{ and } \mathbf{S} \subseteq \mathbf{S}'.$$

Intuitively, if $(\mathbf{S}, \mathbf{T}) \leq_{\mathrm{s}} (\mathbf{S}', \mathbf{T}')$, then $(\mathbf{S}, \mathbf{T})$ represents a subtype of the type represented by $(\mathbf{S}', \mathbf{T}')$:



If R is a relation on types, then $(\sigma, \tau){\downharpoonleft\!\upharpoonleft}_R$ is defined to be the pre-state $(\sigma{\downarrow}_R, \tau{\uparrow}_R)$, and similarly, $(\mathbf{S}, \mathbf{T}){\downharpoonleft\!\upharpoonleft}_R$ is the pre-state $(\mathbf{S}{\downarrow}_R, \mathbf{T}{\uparrow}_R)$. If R and R' are relations on types, then $R'{\downharpoonleft\!\upharpoonleft}_R$ is the set of pre-states $\{\, (\sigma, \tau){\downharpoonleft\!\upharpoonleft}_R \mid (\sigma, \tau) \in R' \,\}$.

If R is a relation on Σ types, and $\mathbf{Sym}$ is a partial function from Σ pre-states to Σ, then $\delta_{\mathbf{Sym},R}$ is a partial function on pre-states defined as follows.

$$\delta_{\mathbf{Sym},R}(\mathbf{S}, \mathbf{T})(\ell) = \begin{cases} \uparrow & \text{if } \mathbf{Sym}(\mathbf{S}, \mathbf{T}) \neq \to, \\ (\mathbf{T}.0, \mathbf{S}.0){\downharpoonleft\!\upharpoonleft}_R & \text{if } \mathbf{Sym}(\mathbf{S}, \mathbf{T}) = \to \text{ and } \ell = 0, \\ (\mathbf{S}.1, \mathbf{T}.1){\downharpoonleft\!\upharpoonleft}_R & \text{if } \mathbf{Sym}(\mathbf{S}, \mathbf{T}) = \to \text{ and } \ell = 1. \end{cases}$$

When we decide that a pre-state $(\mathbf{S}, \mathbf{T})$ corresponds to a function type (that is, $\mathbf{Sym}(\mathbf{S}, \mathbf{T}) = \to$), the $\delta_{\mathbf{Sym},R}$ function is used to find the states corresponding to the argument and result types. We gave an example of its use above.

We also define a function mapping pre-states to partial functions from paths to Σ:

$$\begin{aligned} \mathbf{Type}_{\mathbf{Sym},R}(g)(\epsilon) &= \mathbf{Sym}(g), \\ \mathbf{Type}_{\mathbf{Sym},R}(g)(\ell\alpha) &= \mathbf{Type}_{\mathbf{Sym},R}(\delta_{\mathbf{Sym},R}(g)(\ell))(\alpha). \end{aligned}$$

Finally, we define S_R to be the least partial function such that for every type variable s appearing in R, we have

$$S_R(s) = \mathbf{Type}_{\mathbf{Sym},R}((s, s){\downharpoonleft\!\upharpoonleft}_R).$$

Our idea is that S_R will be a solution to R. However, we have not yet given enough conditions to guarantee this. For example, $\mathbf{Type}_{\mathbf{Sym},R}(g)$ is not necessarily a type. We may have $\mathbf{Type}_{\mathbf{Sym},R}(g)(\epsilon) = \uparrow$, or $\mathbf{Type}_{\mathbf{Sym},R}(g)(\alpha) = \to$ but $\mathbf{Type}_{\mathbf{Sym},R}(g)(\alpha 0) = \uparrow$. Below, we will give sufficient conditions to ensure that $\mathbf{Type}_{\mathbf{Sym},R}(g)$ is a type and S_R is a solution.

Definition 8 (Structures) *A* Σ pre-structure, Γ, *consists of:*

- *a predicate on relations on* Σ *types called* Γ-consistency*;*
- *a predicate on relations on* Σ *types called* Γ-closure*;*
- *a partial function from pre-states to* Σ *called* $\mathbf{Sym}_\Gamma$*;*
- *and, for every* Γ*-closed and* Γ*-consistent* R*, a set of* Σ *pre-states called the* (Γ, R) states.

We write $\delta_{\Gamma,R}$ *for* $\delta_{\mathbf{Sym}_\Gamma,R}$*, and* $\mathbf{Type}_{\Gamma,R}$ *for* $\mathbf{Type}_{\mathbf{Sym}_\Gamma,R}$*, and we omit* Γ, R *when they can be recovered from context.*

A Σ *pre-structure is a* Σ structure *if it satisfies the following requirements.*

(R1) *If* R *has a* Σ *solution, then* R *is* Γ*-consistent.*

(R2) *Every relation has an equivalent* Γ*-closure.*

(R3) $\mathbf{Sym}_\Gamma$ *is a total function from states to* Σ*.*

(R4) $\delta_{\Gamma,R}$ *is a partial function from states to states.*

(R5) *If* R *is* Γ*-closed and* Γ*-consistent, and* σ *appears in* R*, then* $(\sigma, \sigma){\downharpoonright}{\upharpoonright}_R$ *is a state.*

(R6) *If* R *is* Γ*-closed, then* R *satisfies Properties P1 and P2.*

(R7) *If* $(\mathbf{S}, \mathbf{T})$ *is a state, then* $\sigma(\epsilon) \leq_\Sigma \mathbf{Sym}_\Gamma(\mathbf{S}, \mathbf{T})$ *for every* $\sigma \in \mathbf{S} - \mathbf{Tv}$*, and* $\mathbf{Sym}_\Gamma(\mathbf{S}, \mathbf{T}) \leq_\Sigma \tau(\epsilon)$ *for every* $\tau \in \mathbf{T} - \mathbf{Tv}$*.*

(R8) $\mathbf{Sym}_\Gamma$ *is a monotone function: if* $g \leq_{\mathrm{s}} h$ *then* $\mathbf{Sym}_\Gamma(g) \leq_\Sigma \mathbf{Sym}_\Gamma(h)$*.*

Intuitively, R1 ensures that consistency does not contradict solvability, R2 ensures that a Γ-closure exists for every relation R on Σ types, R3 and R4 ensure that $\mathbf{Type}_{\Gamma,R}(g)$ is in fact a type for every state g, R5 ensures that there is a state for every type appearing in a closed and consistent R, and R6, R7, and R8 ensure that S_R is a solution.

Our two main theorems show that in any structure, typability can be answered by constructing the closure of a relation and checking consistency, and a type inference solution can be constructed from any closed, consistent relation.

Theorem 9 *Suppose* Γ *is a* Σ *structure and* R *is a relation on* Σ *types. Then* R *has a* Σ *solution iff there is a* Γ*-closure of* R *that is* Γ*-consistent.*

Theorem 10 *If* Γ *is a* Σ *structure,* R *is a relation on* Σ *types, and* R *is* Γ*-closed and* Γ*-consistent, then* S_R *is a* Σ *solution to* R*.*

The Theorems are proved in an appendix. The proof of Theorem 10 is interesting in that it uses co-induction.

5 Structures

In this section we give particular examples of structures for a variety of signatures. We will also show that the operations of closure and consistency are decidable for these structures when we restrict attention to regular types, and, similarly, that the solution S_R given by our construction above is finitely representable for regular types.

5.1 Type inference with $\top$ and $\bot$

We first consider the signature $\Sigma_1 = \mathbf{Tv} \cup \{\bot, \top, \rightarrow\} \cup \mathbf{B}$, where $\mathbf{B}$ is a finite set. We assume that Σ_1 is a complete lattice. A type inference algorithm for the regular subset of Λ_{Σ_1} in the special case of $\mathbf{B} = \{\mathbf{nat}\}$ was first given by Palsberg and O'Keefe [26]. We will need the following notation: if R is a relation on Σ types, then $R^{=}$ is defined to be the relation $R \cup \{(\sigma, \sigma) \mid \sigma \text{ is a } \Sigma \text{ type}\}$.

Definition 11 *The structure Γ_1 is defined as follows.*

- $$\mathbf{Sym}_{\Gamma_1}(\mathbf{S}, \mathbf{T}) = \begin{cases} \top & \textit{if } \mathbf{T}(\epsilon) - \mathbf{Tv} = \varnothing, \\ \bot & \textit{if } \mathbf{T}(\epsilon) - \mathbf{Tv} \neq \varnothing \textit{ and } \mathbf{S}(\epsilon) - \mathbf{Tv} = \varnothing, \\ \mathbf{glb}_{\Sigma_1}(\mathbf{T}(\epsilon) - \mathbf{Tv}) & \textit{otherwise.} \end{cases}$$

- *A relation R on Σ_1 types is* Γ_1-closed *if it satisfies Properties P1 and P2.*

- *A relation R on Σ_1 types is* Γ_1-consistent *if it satisfies Condition C2 (with $\Sigma = \Sigma_1$).*

- *If R is Γ_1-closed and Γ_1-consistent, then a Σ_1 pre-state $(\mathbf{S}, \mathbf{T})$ is a (Γ_1, R)* state *if $\mathbf{S} \times \mathbf{T} \subseteq R^{=}$.*

An alternative definition of $\mathbf{Sym}_{\Gamma_1}(\mathbf{S}, \mathbf{T})$ is $\mathbf{glb}_{\Sigma_1}(\mathbf{T}(\epsilon) - \mathbf{Tv})$. We have chosen the more complicated definition because it can lead to types of smaller shape, in cases where it chooses $\bot$.

Lemma 12 *Γ_1 is a Σ_1 structure.*

Proof: Since C2 is strictly weaker than C1, Γ_1 satisfies R1: if R has a Σ_1 solution, then R satisfies C1, and therefore, C2; thus R is Γ_1 consistent.

It is not hard to see that for every relation R, there is a smallest relation satisfying Properties P1 and P2, and including R, and this smallest relation has exactly the same solutions as R; therefore, R2 is satisfied.

To show that $\mathbf{Sym}_{\Gamma_1}$ is a total function on states (R3), it is sufficient to note that by assumption, $\mathbf{glb}_{\Sigma_1}(\mathbf{T}(\epsilon) - \mathbf{Tv})$ exists.

Clearly Γ_1 satisfies R5, R6, and R8. To prove R7, we need C2 and the fact that $\mathbf{S} \times \mathbf{T} \subseteq R^{=}$.

Finally, we must establish that δ maps states to states (condition R4). This is proved by the lemma below. ■

Lemma 13 (δ preserves $R^=$) *If R is Γ-closed, $\mathbf{S} \times \mathbf{T} \subseteq R^=$, and $\delta_{\mathbf{Sym},R}(\mathbf{S},\mathbf{T})(\ell)$ is defined, then $(\times\, \delta_{\mathbf{Sym},R}(\mathbf{S},\mathbf{T})(\ell)) \subseteq R^=$.*

Proof: Let $(\mathbf{S}',\mathbf{T}') = \delta_{\mathbf{Sym},R}(\mathbf{S},\mathbf{T})(\ell)$.

- Suppose $\ell = 0$, so that $(\mathbf{S}',\mathbf{T}') = (\mathbf{T}.0, \mathbf{S}.0)\rfloor\lceil_R$, and $(\sigma,\tau) \in \mathbf{S}' \times \mathbf{T}'$. We must show that $(\sigma,\tau) \in R^=$.

  Since $\sigma \in \mathbf{S}'$, there must be types σ_0, σ_1 such that $\sigma_0 \to \sigma_1 \in \mathbf{T}$, and $(\sigma,\sigma_0) \in R^=$.

  Since $\tau \in \mathbf{T}'$, there must be types τ_0, τ_1 such that $\tau_0 \to \tau_1 \in \mathbf{S}$, and $(\tau_0,\tau) \in R^=$.

  Since $\mathbf{S} \times \mathbf{T} \subseteq R^=$, we have $(\tau_0 \to \tau_1, \sigma_0 \to \sigma_1) \in R^=$.

  Then by P1, we have $(\sigma_0,\tau_0) \in R^=$, and by P2, we have $(\sigma,\tau) \in R$, as desired.

- If $\ell = 1$ the result follows similarly. ■

To show that typability and type inference are decidable when restricted to regular types, note that the P1 and P2 closure of a relation on regular Σ_1 types has size at most n^2, where n is the number of subtrees of the relation, and can be constructed in $O(n^3)$ time (transitive closure requires cubic time). Checking that a relation satisfies C2 takes time linear in the number of pairs of the relation, $O(n^2)$. Hence typability is $O(n^3)$.

The complexity of type inference is harder to measure, and depends on the representation we choose for the solution. In the worst case, there are an exponential number of states reachable from $R\rfloor\lceil_R$, which forms the solution. If we choose this representation for the solution, then, the complexity of type inference is exponential. However, we could also say that a closed, consistent R is tantamount to a solution, so that the complexity of type inference is cubic. Such a representation is arguably more cryptic than the exponential representation, however, we feel that this is no worse than what is required to obtain the linear lower bound for the well-understood problem of unification.

In unification, when types are represented naively as strings, the size of the solution to a unification problem can be exponential in the size of the problem. When a dag representation is used for types, the size of the unification solution is quadratic in the size of the problem. In order to achieve the linear lower bound for unification, a more cryptic representation must be used, similar to our closed, consistent relations.

5.2 Type inference with $\top$ but not $\bot$

We now consider the signature $\Sigma_2 = \mathbf{Tv} \cup \{\top, \rightarrow\} \cup \mathbf{B}$, where $\mathbf{B}$ is a finite set. We will use the terminology that a lattice is *downwards conditionally complete* if every subset with a lower bound has a greatest lower bound. Similarly, we say that a lattice is *upwards conditionally complete* if every subset with an upper bound has a least upper bound. We assume that Σ_2 is downwards conditionally complete.

The case of $\mathbf{B} = \{\mathbf{nat}\}$ is interesting both historically and technically. Type inference has not previously been solved for this case. When recursive types and the base type **nat** are omitted, we have the system of *partial types* introduced by Thatte [34]. Thatte gave a semi-decision procedure for type inference in his system. O'Keefe and Wand [25] gave the first type inference algorithm for Thatte's system. Kozen, Palsberg, and Schwartzbach [18, 19] improved on O'Keefe and Wand's algorithm, and extended it to recursive types.

The addition of the base type **nat** to the type system makes type inference more difficult. The signature seems simpler than Σ_1, since it omits $\bot$, but we will see that type inference is actually more complicated. The difference is illustrated by the following example.

$$(s, \sigma_0 \rightarrow \sigma_1) \quad (s, \tau_0 \rightarrow \tau_1)$$

Here we have two pairs of a relation; suppose s does not appear in $\sigma_0, \sigma_1, \tau_0, \tau_1$. If S is a solution to the relation, then $S(s)$ is a lower bound of $\sigma_0 \rightarrow \sigma_1$ and $\tau_0 \rightarrow \tau_1$. In Σ_1, an obvious solution maps s to $\bot$. This is not possible in Σ_2: s must be mapped to a function type, say, $\rho_0 \rightarrow \rho_1$. Then ρ_1 must be a lower bound of $S(\sigma_1)$ and $S(\tau_1)$. The existence of such a lower bound was not implied in Σ_1.

This is exactly where **nat** causes difficulties. The type inference algorithm in [18] for the system without **nat** constructs a *closed* solution, that is, a solution containing no type variables. The possible types in the solution are thus generated by the signature $\{\rightarrow, \top\}$. This class of types has a special property: *every nonempty set of such types has a lower bound.* Therefore, in constructing a closed solution S to the relation above, the constraint that $S(\sigma_1)$ and $S(\tau_1)$ have a lower bound is satisfied vacuously.

When **nat** is added to the system, it is no longer possible to find a lower bound for every set of closed types. Therefore, the implied lower bounds must be checked for. This affects both the closure and consistency conditions.

We now return to the more general case of $\Sigma_2 = \mathbf{Tv} \cup \{\top, \rightarrow\} \cup \mathbf{B}$, where $\mathbf{B}$ is a finite set and Σ_2 is assumed to be downwards conditionally complete. Before we define our structure, we need an auxilliary definition: A set $\mathbf{S}$ of types is *pairwise bounded below (PBB) in* R if for all types $\sigma, \tau \in \mathbf{S}$, if $\sigma \neq \tau$, then there exists a type ρ such that $(\rho, \sigma), (\rho, \tau) \in R$.

Definition 14 *The structure* Γ_2 *is defined as follows.*

- $\mathbf{Sym}_{\Gamma_2}(\mathbf{S}, \mathbf{T}) = \mathbf{glb}_{\Sigma_2}(\mathbf{T}(\epsilon) - \mathbf{Tv})$.

(Note that $\mathbf{glb}_{\Sigma_2}(\varnothing) = \top$*.)*

- *A relation* R *on* Σ_2 *types is* Γ_2-closed *if it satisfies Properties P1 and P2, and the following additional property.*

  (P3) *If* $(s, \sigma_0 \to \sigma_1) \in R$ *and* $(s, \tau_0 \to \tau_1) \in R$*, then there is some* σ *such that* $(\sigma, \sigma_1) \in R^=$ *and* $(\sigma, \tau_1) \in R^=$*.*

- *A relation* R *on* Σ_2 *types is* Γ_2-consistent *if it satisfies C2 (with* $\Sigma = \Sigma_2$*) and the following condition.*

  (C3) *For all types* σ*,* $(\sigma{\uparrow}_R)(\epsilon) - \mathbf{Tv}$ *has a* $\leq_{\Sigma_2}$*-lower bound.*

- *A* Σ_2 *pre-state* $(\mathbf{S}, \mathbf{T})$ *is a* (Γ_2, R) state *if* $\mathbf{S} \times \mathbf{T} \subseteq R^=$ *and* $\mathbf{T}$ *is PBB in* $R^=$*.*

Lemma 15 Γ_2 *is a* Σ_2 *structure.*

Proof: Not surprisingly, the proof is quite similar to that for the structure Γ_1. The differences show up in conditions R2, R3, and R4, so we concentrate on those here.

For R2, we must show that there is a Γ_2-closure for every R. The only rule that causes difficulty is P3. P3 asserts that a lower bound exists for certain types in $R^=$. In order to close arbitrary relations under P3 we will pick a fresh type variable to be this lower bound if a lower bound does not already exist. For example, the closure of a relation containing the pairs

$$\begin{array}{ll} (s, \sigma_0 \to \sigma_1) & (t, \tau_0 \to \tau_1) \\ (s, \tau_0 \to \tau_1) & (t, \rho_0 \to \rho_1) \end{array}$$

will include in addition the pairs

$$\begin{array}{ll} (s', \sigma_1) & (t', \tau_1) \\ (s', \tau_1) & (t', \rho_1) \end{array}$$

where s' and t' are marked as fresh. It is easy to show that this always results in a closure, and the closure will have exactly the same solutions as the original relation. Moreover, when working solely with regular types, we will need to add at most n^2 fresh type variables, where n is the number of subtrees in the original relation: one for each pair of subtrees whose root symbol is $\to$.

To prove R3 ($\mathbf{Sym}_{\Gamma_2}$ is a total function on states), we use Lemma 16, below.

That leaves only R4 (δ maps states to states). By Lemma 13, δ preserves $R^=$, so we only need to show that δ preserves the PBB property.

Suppose $(\mathbf{S}, \mathbf{T})$ is a (Γ_2, R) state, and $\mathbf{glb}_{\Sigma_2}(\mathbf{T}(\epsilon) - \mathbf{Tv}) = \to$, so that $\delta_{\Gamma_2,R}(\mathbf{S}, \mathbf{T})(0)$ and $\delta_{\Gamma_2,R}(\mathbf{S}, \mathbf{T})(1)$ are defined.

Since $\delta_{\Gamma_2,R}(\mathbf{S}, \mathbf{T})(0) = (\mathbf{T}.0, \mathbf{S}.0){\Downarrow}{\upharpoonleft}_R$, we must show that $\mathbf{S}.0{\uparrow}_R$ is PBB in $R^=$. By transitivity (P2), it is sufficient to show that $\mathbf{S}.0$ is PBB in $R^=$, so assume

$\sigma_0, \tau_0 \in \mathbf{S}.0$. Then there must exist types σ_1, τ_1 such that $\sigma_0 \to \sigma_1, \tau_0 \to \tau_1 \in \mathbf{S}$. And there must be a type $\rho_0 \to \rho_1 \in \mathbf{T}$, because $\mathbf{glb}_{\Sigma_2}(\mathbf{T}(\epsilon) - \mathbf{Tv}) = \to$. Since $\mathbf{S} \times \mathbf{T} \subseteq R^=$, we have $(\sigma_0 \to \sigma_1, \rho_0 \to \rho_1) \in R^=$ and $(\tau_0 \to \tau_1, \rho_0 \to \rho_1) \in R^=$. Then by P1, we have $(\rho_0, \sigma_0) \in R^=$ and $(\rho_0, \tau_0) \in R^=$ as desired.

Since $\delta_{\Gamma_2,R}(\mathbf{S}, \mathbf{T})(1) = (\mathbf{S}.1, \mathbf{T}.1)\Downarrow\uparrow_R$, we must show that $\mathbf{T}.1\uparrow_R$ is PBB in $R^=$. By transitivity (P2), it is sufficient to show that $\mathbf{T}.1$ is PBB in $R^=$, so assume $\sigma_1, \tau_1 \in \mathbf{T}.1$. Then there must exist types σ_0, τ_0 such that $\sigma_0 \to \sigma_1, \tau_0 \to \tau_1 \in \mathbf{T}$. Since $(\mathbf{S}, \mathbf{T})$ is a state, $\mathbf{T}$ is PBB in $R^=$. Therefore there is a type ρ such that $(\rho, \sigma_0 \to \sigma_1) \in R^=$ and $(\rho, \tau_0 \to \tau_1) \in R^=$. If ρ is a type variable, then by P3, σ_1 and τ_1 have a lower bound in $R^=$. If $\rho = \rho_0 \to \rho_1$, then by P1, ρ_1 is a lower bound of σ_1 and τ_1 in $R^=$. And ρ cannot be other than a type variable or of the form $\rho_0 \to \rho_1$ by C2. Thus in all cases, σ_1 and τ_1 have a lower bound in $R^=$, so $\mathbf{T}.1$ is PBB in $R^=$, as desired. ■

Lemma 16 *If* $\mathbf{T}$ *is PBB in* $R^=$ *and* R *satisfies C3 then* $\mathbf{glb}_{\Sigma_2}(\mathbf{T}(\epsilon) - \mathbf{Tv})$ *exists.*

Proof: Since Σ_2 is assumed to be downwards conditionally complete, it is sufficient to show that $(\mathbf{T}(\epsilon) - \mathbf{Tv})$ has a $\leq_{\Sigma_2}$-lower bound. Since $\Sigma_2 - \mathbf{Tv}$ is finite, it sufficient to show that any two elements of $(\mathbf{T}(\epsilon) - \mathbf{Tv})$ have a $\leq_{\Sigma_2}$-lower bound. Let $\sigma, \tau \in (\mathbf{T} - \mathbf{Tv})$ be such that $\sigma(\epsilon) \neq \tau(\epsilon)$. Since $\mathbf{T}$ is PBB in $R^=$, we can choose ρ such that $(\rho, \sigma), (\rho, \tau) \in R^=$. Since R satisfies C3, we have that $R^=$ satisfies C3, so $\sigma(\epsilon)$ and $\tau(\epsilon)$ have a $\leq_{\Sigma_2}$-lower bound. ■

5.3 Type inference with $\bot$ but not $\top$

Consider $\Sigma_3 = \mathbf{Tv} \cup \{\bot, \to\} \cup \mathbf{B}$, where $\mathbf{B}$ is a finite set. We assume that Σ_3 is upwards conditionally complete. The Σ_3 types have a least type $\bot$, but not a greatest type $\top$. The problems arising in constructing a structure for Σ_3 are dual to those encountered for Σ_2, and we leave to the reader to construct an appropriate structure. The result is the first type inference algorithm for this system.

6 Related work

Huet [16] gave the first unification algorithm for recursive types; see also the papers by Cardone and Coppo [7, 8]. Mitchell [23, 24] gave the first inference algorithm for atomic subtyping, without recursive types. With no further assumptions about the partial order, this problem is PSPACE-complete [35, 15, 12], and if the partial order is a disjoint union of lattices or trees, then type inference is in polynomial time [35, 4]. Tiuryn and Wand [36] gave the first inference algorithm for atomic subtyping with recursive types: this problem is in EXPTIME. Thatte [34] introduced the problem of type inference with partial types (simple types plus $\top$), and showed that it was semi-decidable. O'Keefe and Wand [25] gave the first type inference algorithm for

Thatte's system. Kozen, Palsberg, and Schwartzbach [18, 19] improved on O'Keefe and Wand's algorithm, and extended it to recursive types: these problems are P-complete. Palsberg, Wand, and O'Keefe [28] gave an $O(n^3)$ time algorithm for simple types plus $\top$ and $\bot$. Amadio and Cardelli [3] gave the first subtyping algorithm for the system of recursive types with $\top$, $\bot$, and discretely-ordered base types. Later, Kozen, Palsberg, and Schwartzbach showed that the subtype ordering can be decided in $O(n^2)$ time [19]. Palsberg and O'Keefe [26] gave a type inference algorithm for the Amadio-Cardelli system: this problem is in $O(n^3)$ time.

The type system with recursive types and $\top$ and $\bot$ is equivalent in expressive power to a type system with constrained types [27]. A constrained type is a combination of a type and a constraint set. Recursive types corresponds to allowing cycles in the constraint set; $\top$ correspond to allowing constructs of the form, say, $\mathbf{nat} \leq s$, $\mathbf{nat} \rightarrow \mathbf{nat} \leq s$, where s is a type variable; and $\bot$ corresponds to allowing constraints of the form, say, $s \leq \mathbf{nat}$, $s \leq \mathbf{nat} \rightarrow \mathbf{nat}$. Constrained types without cycles in the constraint set were studied by Kaes [17] and Smith [31]. Type inference for constrained types extended with let-polymorphism was studied for functional languages by Aiken and Wimmers [2], and later for object-oriented languages by Eifrig, Smith, and Trifonov [10, 9]. An algorithm for deciding a subtyping relation for constrained types, including type quantifiers, was presented by Trifonov and Smith [37].

The first use of simulations to define a subtyping relation that we are aware of is Pierce and Sangiorgi [30], who used it for a process calculus. Later, Brandt and Henglein [5] used co-inductive methods to obtain complete axiomatizations of type equality and subtyping, and to obtain an easy proof of the $O(n^2)$ algorithm for deciding the subtyping relation.

Acknowledgment: Palsberg is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265.

A Proof of the main theorems

Recall that $\mathbf{Type}_{\mathbf{Sym},R}(g)$ is not always a type. Whenever $\mathbf{Type}_{\mathbf{Sym},R}(g)$ is in fact a type, we have the following result.

Lemma 17 *If* $\sigma = \mathbf{Type}_{\mathbf{Sym},R}(g)$, *then* $\sigma.\ell = \mathbf{Type}_{\mathbf{Sym},R}(\delta_{\mathbf{Sym},R}(g)(\ell))$.

Proof: Immediate from the definition of $\mathbf{Type}_{\mathbf{Sym},R}$. ■

We use $\mathbf{R}$ to range over relations on states, and we extend the function $\mathbf{Type}_{\mathbf{Sym},R}$ to relations on states as follows:

$$\mathbf{Type}_{\mathbf{Sym},R}(\mathbf{R}) = \{(\mathbf{Type}_{\mathbf{Sym},R}(g), \mathbf{Type}_{\mathbf{Sym},R}(h)) \mid (g,h) \in \mathbf{R}\}$$

Thus $\mathbf{Type}_{\mathbf{Sym},R}$ maps relations on states to relations on types.

Lemma 18 *Suppose Γ is a Σ structure, and R is Γ-closed and Γ-consistent. If $\sigma \notin \mathbf{Tv}$ appears in R, then $\mathbf{Sym}_\Gamma((\sigma,\sigma)\downharpoonleft\upharpoonright_R) = \sigma(\epsilon)$.*

Proof: Let $(\mathbf{S},\mathbf{T}) = (\sigma,\sigma)\downharpoonleft\upharpoonright_R$. By R5, $(\mathbf{S},\mathbf{T})$ is a state, and then by R3, $\mathbf{Sym}_\Gamma(\mathbf{S},\mathbf{T})$ is defined. Note, $\sigma \in \mathbf{S}$ and $\sigma \in \mathbf{T}$. By R7, $\sigma(\epsilon) \leq_\Sigma \mathbf{Sym}_\Gamma(\mathbf{S},\mathbf{T}) \leq_\Sigma \sigma(\epsilon)$, so $\mathbf{Sym}_\Gamma(\mathbf{S},\mathbf{T}) = \sigma(\epsilon)$ as desired. ■

Lemma 19 *Suppose Γ is a Σ structure, and R is Γ-closed and Γ-consistent.*

1. *If $(\sigma,\tau) \in R$, then $(\sigma,\sigma)\downharpoonleft\upharpoonright_R \leq_\mathrm{s} (\tau,\tau)\downharpoonleft\upharpoonright_R$.*
2. *If g is a (Γ,R) state, then $\mathbf{Type}_{\Gamma,R}(g)$ is a type.*
3. *If $\mathbf{R}$ is a relation on (Γ,R) states, and $\mathbf{R} \subseteq \leq_\mathrm{s}$, then $\mathbf{Type}_{\Gamma,R}(\mathbf{R})$ is C1-consistent.*

Proof:

1. By P2.
2. By R3 and R4.
3. By (2), $\mathbf{Type}_{\Gamma,R}(\mathbf{R})$ is a relation on Σ types, so it makes sense to ask whether it is C1-consistent. And C1-consistency follows from R8. ■

Proof of Theorem 9: ($\Rightarrow$) Suppose R has a Σ solution. By R2, R has a Γ-closure, R', that is equivalent to R. Hence, R' has a Σ solution. Then by R1, R' is Γ-consistent.

($\Leftarrow$) By Theorem 10. ■

Proof of Theorem 10: We want to show that $S_R(R) \subseteq \leq$. By Lemma 4, it is sufficient to show that the P1-closure of $S_R(R)$ is C1-consistent.

First, we define a sequence R_0, R_1, R_2, ..., of relations on types:

$$\begin{aligned} R_0 &= S_R(R) \\ R_{i+1} &= R_i \cup \{(\tau_0,\sigma_0) \mid \exists \sigma_1,\tau_1.\ (\sigma_0 \to \sigma_1, \tau_0 \to \tau_1) \in R_i\} \\ &\quad \cup \{(\sigma_1,\tau_1) \mid \exists \sigma_0,\tau_0.\ (\sigma_0 \to \sigma_1, \tau_0 \to \tau_1) \in R_i\} \end{aligned}$$

The P1-closure of $S_R(R)$ is just $\bigcup_{0 \leq i < \infty} R_i$. Since the union of C1-consistent relations is C1-consistent, it is sufficient to show that each relation R_i is C1-consistent.

Define a sequence $\mathbf{R}_0$, $\mathbf{R}_1$, $\mathbf{R}_2$, ..., of relations on (Γ,R) states:

$$\begin{aligned} \mathbf{R}_0 &= R\downharpoonleft\upharpoonright_R \\ \mathbf{R}_{i+1} &= \mathbf{R}_i \cup \{(\delta_{\Gamma,R}(h)(0), \delta_{\Gamma,R}(g)(0)) \mid (g,h) \in \mathbf{R}_i\} \\ &\quad \cup \{(\delta_{\Gamma,R}(g)(1), \delta_{\Gamma,R}(h)(1)) \mid (g,h) \in \mathbf{R}_i\} \end{aligned}$$

That $\mathbf{R}_0$ is a relation on states follows from R5, and that $\mathbf{R}_{i+1}$ is a relation on states follows from R4.

By induction on i, we will show that $R_i = \mathbf{Type}_{\Gamma,R}(\mathbf{R}_i)$, and that $\mathbf{R}_i \subseteq \leq_{\mathrm{s}}$. Then by Lemma 19(3), every R_i is C1-consistent.

For the base case, by Lemma 20, $\mathbf{Type}_{\Gamma,R}(\mathbf{R}_0) = \mathbf{Type}_{\Gamma,R}(R{\downharpoonleft}{\upharpoonleft}_R) = S_R(R) = R_0$, and by Lemma 19(1), $\mathbf{R}_0 = R{\downharpoonleft}{\upharpoonleft}_R \subseteq \leq_{\mathrm{s}}$.

For the inductive case, assume $R_i = \mathbf{Type}_{\Gamma,R}(\mathbf{R}_i)$ and $\mathbf{R}_i \subseteq \leq_{\mathrm{s}}$. Then by Lemma 17, $R_{i+1} = \mathbf{Type}_{\Gamma,R}(\mathbf{R}_{i+1})$, and by Lemma 22, $\mathbf{R}_{i+1} \subseteq \leq_{\mathrm{s}}$. ■

Lemma 20 *If Γ is a Σ structure, and R is Γ-closed and Γ-consistent, then for any type σ appearing in R,*

$$S_R(\sigma) = \mathbf{Type}_{\Gamma,R}((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R).$$

Proof: If σ is a type variable, then the result is immediate by the definition of S_R. The interesting case is when σ is not a type variable.

For all α and σ, we show that $S_R(\sigma)(\alpha) = \mathbf{Type}_{\Gamma,R}((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R)(\alpha)$. The proof is by induction on α.

- If $\alpha = \epsilon$ and σ is a type variable, then the result follows immediately from the definition of S_R.

- If $\alpha = \epsilon$ and $\sigma(\epsilon) \notin (\mathbf{Tv} \cup \{\rightarrow\})$, then $S_R(\sigma) = \sigma$, and thus, $S_R(\sigma)(\epsilon) = \sigma(\epsilon)$.
  By Lemma 18, $\mathbf{Sym}_\Gamma((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R) = \sigma(\epsilon)$. Therefore, $\mathbf{Type}_{\Gamma,R}((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R)(\epsilon) = S_R(\sigma)(\epsilon)$, as desired.

- If $\alpha = \epsilon$ and $\sigma = \sigma_0 \rightarrow \sigma_1$, then $S_R(\sigma)(\epsilon) = \rightarrow$.
  And $\mathbf{Type}_{\Gamma,R}((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R)(\epsilon) = \mathbf{Sym}_\Gamma((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R)$, which is $\rightarrow$ by Lemma 18.

- If $\alpha = \ell\alpha'$ and σ is a type variable, then the result follows immediately from the definition of S_R.

- If $\alpha = \ell\alpha'$ and $\sigma(\epsilon) \notin (\mathbf{Tv} \cup \{\rightarrow\})$, then both $S_R(\sigma)(\alpha)$ and $\mathbf{Type}_{\Gamma,R}((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R)(\alpha)$ are undefined.

- If $\alpha = 0\alpha'$ and $\sigma = \sigma_0 \rightarrow \sigma_1$, then $S_R(\sigma)(\alpha) = S_R(\sigma_0)(\alpha')$. By induction, $S_R(\sigma)(\alpha) = \mathbf{Type}_{\Gamma,R}((\sigma_0,\sigma_0){\downharpoonleft}{\upharpoonleft}_R)(\alpha')$. Finally by Lemma 21 below, $S_R(\sigma)(\alpha) = \mathbf{Type}_{\Gamma,R}((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R)(0\alpha')$.

- If $\alpha = 1\alpha'$ and $\sigma = \sigma_0 \rightarrow \sigma_1$, we proceed just as in the previous case. ■

Lemma 21 *If Γ is a Σ structure, R is Γ-closed and Γ-consistent, and $\sigma = \sigma_0 \rightarrow \sigma_1$ appears in R, then*

$$\delta_{\Gamma,R}((\sigma,\sigma){\downharpoonleft}{\upharpoonleft}_R)(\ell) = (\sigma.\ell,\sigma.\ell){\downharpoonleft}{\upharpoonleft}_R.$$

Proof: First note that the left-hand side is defined: by Lemma 18, it must be $\to$. The right-hand side is always defined.

We will show only the case $\ell = 0$; the case $\ell = 1$ is proved similarly. We must show $((\sigma{\uparrow_R}).0, (\sigma{\downarrow_R}).0){\downharpoonleft\upharpoonright_R} = (\sigma.0, \sigma.0){\downharpoonleft\upharpoonright_R}$, that is,

$$\sigma{\uparrow_R}.0{\downarrow_R} = \sigma.0{\downarrow_R} \quad \text{and} \quad \sigma{\downarrow_R}.0{\uparrow_R} = \sigma.0{\uparrow_R}.$$

Note, $\{\sigma\} \subseteq \sigma{\uparrow_R}$ and $\{\sigma\} \subseteq \sigma{\downarrow_R}$. Since .0 is monotone with respect to set inclusion, $\{\sigma\}.0 = \{\sigma.0\} \subseteq \sigma{\uparrow_R}.0$ and $\{\sigma\}.0 = \{\sigma.0\} \subseteq \sigma{\downarrow_R}.0$.

And since $\cdot{\downarrow_R}$ is monotone with respect to set inclusion, $\sigma.0{\downarrow_R} = \{\sigma.0\}{\downarrow_R} \subseteq (\sigma{\uparrow_R}).0{\downarrow_R}$ and $\sigma.0{\uparrow_R} = \{\sigma.0\}{\uparrow_R} \subseteq (\sigma{\downarrow_R}).0{\uparrow_R}$.

Now suppose $\tau \in \sigma{\downarrow_R}.0{\uparrow_R}$. Then there must be types $\tau' \in \sigma{\downarrow_R}$, $\tau_0 \in \sigma{\downarrow_R}.0$, and τ_1 such that $(\tau_0, \tau) \in R$, $\tau' = \tau_0 \to \tau_1$, and $(\tau', \sigma) \in R$.

Since $(\tau', \sigma) \in R$, by Property P1 we have $(\sigma_0, \tau_0) \in R$. Then since $(\tau_0, \tau) \in R$, by Property P2 we have $(\sigma_0, \tau) \in R$. Therefore $\tau \in \sigma_0{\uparrow_R}$ as desired.

We can similarly show that $\tau \in \sigma{\uparrow_R}.0{\downarrow_R} \Rightarrow \tau \in \sigma_0{\downarrow_R}$, as desired. ■

Lemma 22 *If $g \leq_s h$ and $\mathbf{Sym}(g) = \mathbf{Sym}(h) = \to$, then:*

1. *$\delta_{\mathbf{Sym},R}(h)(0) \leq_s \delta_{\mathbf{Sym},R}(g)(0)$; and*
2. *$\delta_{\mathbf{Sym},R}(g)(1) \leq_s \delta_{\mathbf{Sym},R}(h)(1)$.*

Proof: Suppose $g = (\mathbf{S}_g, \mathbf{T}_g)$ and $h = (\mathbf{S}_h, \mathbf{T}_h)$. Since $g \leq_s h$ we know $\mathbf{T}_h \subseteq \mathbf{T}_g$ and $\mathbf{S}_g \subseteq \mathbf{S}_h$.

1. Let $(\mathbf{S}'_g, \mathbf{T}'_g) = \delta_{\mathbf{Sym},R}(g)(0)$ and $(\mathbf{S}'_h, \mathbf{T}'_h) = \delta_{\mathbf{Sym},R}(h)(0)$.

   Since $\mathbf{T}_h \subseteq \mathbf{T}_g$ and the .0 operation is monotone, we have $\mathbf{T}_h.0 \subseteq \mathbf{T}_g.0$. Since the $\cdot{\downarrow_R}$ operation is monotone, we have we have $(\mathbf{T}_h.0){\downarrow_R} \subseteq (\mathbf{T}_g.0){\downarrow_R}$. This says exactly that $\mathbf{S}'_h \subseteq \mathbf{S}'_g$.

   Similarly, from $\mathbf{S}_g \subseteq \mathbf{S}_h$ we have $(\mathbf{S}_g.0){\uparrow_R} \subseteq (\mathbf{S}_h.0){\uparrow_R}$. This says exactly that $\mathbf{T}'_g \subseteq \mathbf{T}'_h$.

   Thus we have $\delta_{\mathbf{Sym},R}(h)(0) \leq_s \delta_{\mathbf{Sym},R}(g)(0)$ as desired.

2. This case is proved similarly to the previous case. ■

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Alexander Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[4] Marcin Benke. Efficient type reconstruction in the presence of inheritance. Manuscript, 1994.

[5] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proceedings, 3rd International Conference on Typed Lambda Calculi and Applications*, 2–4 April 1997.

[6] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1/2):4–56, 15 February/March 1994.

[7] Felice Cardone and Mario Coppo. Two extensions of Curry's type inference system. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, volume 31 of *APIC Studies in Data Processing*, pages 19–75. Academic Press, 1990.

[8] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

[9] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA'95, ACM SIGPLAN Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 169–184, 1995.

[10] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and it application to OOP. In *Proc. Mathematical Foundations of Programming Semantics*, 1995. To appear.

[11] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[12] Alexandre Frey. Satisfying systems of subtype inequalities in polynomial space. In *Proc. SAS'97, International Static Analysis Symposium*. Springer-Verlag (*LNCS*), 1997.

[13] Adele Goldberg and David Robson. *Smalltalk-80—The Language and its Implementation*. Addison-Wesley, 1983.

[14] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[15] My Hoang and John Mitchell. Lower bounds on type inference with subtypes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

[16] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*. Thèse de Doctorat d'Etat, Université de Paris VII, 1976.

[17] Stefan Kaes. Typing in the presence of overloading, subtyping, and recursive types. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 193–204, 1992.

[18] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[19] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995.

[20] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, October/November 1986.

[21] R. Milner. A calculus of communicating systems. volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[22] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[23] John Mitchell. Coercion and type inference. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[24] John Mitchell. Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–285, July 1991.

[25] Patrick M. O'Keefe and Mitchell Wand. Type inference for partial types is decidable. In B. Krieg-Brückner, editor, *4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 408–417. Springer-Verlag, February 1992.

[26] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[27] Jens Palsberg and Scott F. Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, September 1996.

[28] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[29] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 15–32. Springer-Verlag, 1981.

[30] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 376–385, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.

[31] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[32] Satish Thatte. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium*, volume 317 of *Lecture Notes in Computer Science*, pages 615–629. Springer-Verlag, 1988.

[33] Satish R. Thatte. Quasi-static typing. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 367–381, 1990.

[34] Satish R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124:127–148, 1994.

[35] Jerzy Tiuryn. Subtype inequalities. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 308–315, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

[36] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *CAAP '93: 18th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science, pages 686–701. Springer-Verlag, July 1993.

[37] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proc. SAS'96, International Static Analysis Symposium*. Springer-Verlag (*LNCS* 1145), September 1996.

[38] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proc. OOPSLA '87, Object-Oriented Programming Systems, Languages and Applications*, pages 227–241, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Acadamic Publishers, June 1991.