Simplifying subtyping constraints

François Pottier
Ecole Normale Supérieure – INRIA[*]
Francois.Pottier@inria.fr

Abstract

This paper studies type inference for a functional, ML-style language with subtyping, and focuses on the issue of simplifying inferred constraint sets. We propose a powerful notion of entailment between constraint sets, as well as an algorithm to check it, which we prove to be sound. The algorithm, although very powerful in practice, is not complete. We also introduce two new typing rules which allow simplifying constraint sets. These rules give very good practical results.

1 Introduction

The concept of subtyping has been introduced by Cardelli [4] and by Mitchell [9]. It is of great importance in many record and object calculi. Subtyping has been extensively studied in the case of explicitly typed programs; ML-style type inference in the presence of subtyping is less understood. Fuh and Mishra [7] have studied type inference in the presence of subtyping and polymorphism. However, they consider only structural subtyping, i.e. their subtyping relation is entirely derived from subtyping on base types. More recently, Aiken and Wimmers [1] have proposed a general algorithm for solving systems of subtyping constraints. Their constraint language is rich; in particular, it includes $\cap$ and $\cup$ type operators.

The basis for our type system has been proposed by Eifrig, Smith and Trifonov [5]. It is based on *constrained types*, i.e. types of the form $\tau \mid C$, where τ is an ordinary type expression and C is a set of *subtyping constraints* of the form $\tau_1 \triangleright \tau_2$, meaning that τ_1 must be a subtype of τ_2. Palsberg [10] has introduced a similar system.

This constraint language is more restricted than Aiken and Wimmer's. It does not have $\bot$ and $\top$ types, nor intersection and union operators. However, $\bot$ and $\top$ are easily encoded with subtyping constraints. Besides, we introduce notions of *greatest lower bounds* and *least upper bounds* of ground types, and we will use them in a restricted way in constraints.

[*]François Pottier, Projet Cristal, INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France

To appear in the 1996 ACM SIGPLAN International Conference on Functional Programming.

The type inference algorithm analyzes the program and accumulates a set of subtyping constraints. Instead of trying to solve these constraints, one defines a notion of *consistency*, and the program is declared well-typed if and only if its constraint set is consistent. We prove that a constraint set is consistent if and only if it has a solution, so this is not a fundamental departure from other systems, only a difference in presentation.

This system is theoretically correct. However, because each function application node in the program causes a subtyping constraint to be generated, the size of inferred types is at best linear in the program size. In fact, it is even worse, because constraint sets are duplicated by every use of a `let`-bound variable. As a result, inferred types are much too large to be read by humans, and they are unwieldy for the type-checker itself. Hence, simplifying the inferred constraints is necessary for the system to be useable at all in practice. We provide formal tools to this end.

We strengthen Smith's subtyping rule by introducing a new, powerful notion of *entailment* between constraint sets. We define it formally and give an algorithm to verify it, which is unfortunately incomplete but useful in practice. Then, we introduce two new, independent typing rules to simplify constraint sets. The first one removes so-called *unreachable* constraints. The second one applies *substitutions* to the constraint set. Both are proved to be sound. These simplification methods have been implemented in a prototype type-checker, and give good practical results.

2 An overview of constraint-based type-checking

Before beginning a formal description of our framework, let us describe the principles of constraint-based type inference. We assume given a functional, ML-like language with subtyping.

During type-checking, a classical ML-style system generates *equations* between types: basically, every time a function is applied, the type of the supplied argument must be equal to the function's domain type. Unification is used as an efficient way to solve the resulting system of equalities. A constraint-based system works in a similar way, except that *subtyping constraints* are generated instead of equations: the supplied argument must be a subtype of the function's domain type.

How does subtyping make type inference more difficult?

In a language without subtyping, when faced with an equation $\alpha = \tau$, one can use unification to equate α to τ in a very efficient way. This does not work any more for a constraint $\alpha \triangleright \tau$.

Fuh and Mishra [7] have studied the problem in the case of *structural* subtyping, where all type inclusions are consequences of inclusions between atomic types. Their approach, when encoutering a constraint such as $\alpha \triangleright \tau_1 \rightarrow \tau_2$, is to identify α with a function type $\alpha_1 \rightarrow \alpha_2$, where the α_i are fresh type variables. This generates sub-constraints $\alpha_2 \triangleright \tau_2$ and $\tau_1 \triangleright \alpha_1$. The program is correct if all constraints involving atomic types are true.

In our case, however, the system has *non-structural* subtyping. For instance, the record types $\{f_1 : \alpha_1; f_2 : \alpha_2\}$ and $\{f_1 : \alpha_1; f_3 : \alpha_3\}$ are both subtypes of $\{f_1 : \alpha_1\}$, yet they have different shapes. Hence, when faced with the constraint $\alpha \triangleright \{f_1 : \alpha_1\}$, we do not know the precise structure of the type represented by α, so we cannot identify α with a more precise type.

Type-checkers for ML-style languages use unification to *solve* systems of equations, that is, find a principal mapping from type variables to ground types which satisfies all constraints. However, replacing equations with constraints makes resolution much more complex; it is no longer obvious that a principal type exists, and computing the set of all solutions becomes non-trivial. This is the approach taken by Aiken [1]. The other approach, proposed by Smith [5], consists in merely demanding the constraint set to be *consistent*; writing an algorithm to check consistency is trivial. A subject reduction theorem then guarantees soundness. Following Smith, we take the second approach. Palsberg [11] has proved that consistency and solvability are equivalent, and we adapt his proof to our system.

3 Examples

This section gives examples and motivations for the definitions we will give later. The reader might wish to have a look at section 5, which describes the basic type system, if certain notions are unclear.

3.1 A strong notion of entailment

Entailment between constraint sets is an important notion: the power of the subtyping rule (see figure 3) depends on it. $C_1 \Vdash C_2$ is defined in [5] as "elements of C_2 are elements of C_1's closure, or reflexive constraints". This notion is not powerful enough to prove that certain constraint sets are equivalent.

Let us examine the subtyping rule. Assume we have derived a certain constraint set C_2 for an expression, and we would like to replace it safely with another set C_1. We need to make sure that if the expression is part of a larger program which is invalid, this change will not cause the program to be accepted. The program is invalid if and only if the constraint set D generated for the rest of the program is such that $D \cup C_2$ is inconsistent. It will be declared valid, after replacing C_2 with C_1, if and only if $D \cup C_1$ is consistent. This must not happen. Hence, we define $C_1 \Vdash C_2$ as "for any constraint set D such that $C_1 \cup D$ is consistent, $C_2 \cup D$ is consistent". This definition seems to be the most powerful possible one. It can be qualified as "observational", because entailment between constraint sets depends on their behavior with respect to the "external world" D.

In section 6, we show that $C_1 \Vdash C_2$ is equivalent to the statement "every solution of C_1 is a solution of C_2", which proves that this definition is indeed natural.

Our definition of $C_1 \Vdash C_2$ is strictly more powerful than checking whether C_1's closure contains C_2. Let us give a few interesting examples. If F is a non-trivial, covariant context, then

$$\{\alpha \triangleright F(\alpha), F(\beta) \triangleright \beta\} \Vdash \alpha \triangleright \beta$$

Another typical example is

$$\{\alpha_1 * \alpha_2 \triangleright \gamma, \beta_1 * \beta_2 \triangleright \gamma\} \Vdash \alpha_1 * \beta_2 \triangleright \gamma$$

Both statements can be proved by using the definition of entailment. However, this definition involves a universally quantified constraint set D. This gives it great power, but also makes it difficult to verify automatically. In section 8, we introduce an algorithm, in the form of a set of inference rules, to verify entailment without recourse to its definition. Let us hint (informally) at how the algorithm copes with these two examples.

In the case of the first example, to show that $\alpha \triangleright \beta$, it is sufficient to show that $F(\alpha) \triangleright \beta$, because of the hypothesis $\alpha \triangleright F(\alpha)$. Symmetrically, it is sufficient to prove $F(\alpha) \triangleright F(\beta)$. Since F is a covariant context, after applying a certain number of propagation rules, we will realize that we need to show $\alpha \triangleright \beta$ again. This suggests building an inductive proof, and declaring success when the original goal is encountered again.

In the case of the second example, one uses a *least upper bound*, as follows. To show that $\alpha_1 * \beta_2$ is smaller than γ, it is sufficient to show that it is smaller than one of γ's lower bounds, i.e. to prove one of these assertions:

$$\begin{array}{lcl} \alpha_1 * \beta_2 & \triangleright & \alpha_1 * \alpha_2 \\ \alpha_1 * \beta_2 & \triangleright & \beta_1 * \beta_2 \end{array}$$

But none of them holds. Better yet, it is sufficient to prove that it is smaller than the *least upper bound* of γ's lower bounds, i.e.

$$\alpha_1 * \beta_2 \triangleright (\alpha_1 * \alpha_2) \sqcup (\beta_1 * \beta_2)$$

Now, because $\sqcup$ is distributive with respect to product, this rewrites to

$$\alpha_1 * \beta_2 \triangleright (\alpha_1 \sqcup \beta_1) * (\alpha_2 \sqcup \beta_2)$$

which is equivalent to $\alpha_1 \triangleright \alpha_1 \sqcup \beta_1$ and $\beta_2 \triangleright \alpha_2 \sqcup \beta_2$, both of which are immediate.

This notion of least upper bound will be formalized in section 7. Note that we need to allow $\sqcup$ symbols on the *right*[1] side of constraints, so defining least upper bounds by declaring "$\sigma_1 \sqcup \sigma_2 \triangleright \tau$ is equivalent to $\sigma_1 \triangleright \tau \wedge \sigma_2 \triangleright \tau$" would not be of any help here.

Both of these examples are of practical interest: they appear while simplifying the types of various classic functions.

3.2 Removing unreachable constraints

Consider the following expression:

```
let Y = fun f -> (fun g -> fun x -> f (g g) x)
                 (fun g -> fun x -> f (g g) x) in
let compute = Y (fun f -> fun x -> plus 1 (f x)) in
compute 1
```

[1]More precisely, in contravariant positions.

Here `Y` is the classic fix-point combinator, and `plus` is a primitive addition function. Therefore, `compute` is a function which takes an integer and returns an integer (it is of no significance here that the computation does not terminate). Now, the type obtained for this expression is `int` together with the constraints

$$\begin{array}{rcl} \alpha & \triangleright & (\beta \rightarrow \gamma) \rightarrow \delta \rightarrow \varepsilon \\ \alpha & \triangleright & (\beta \rightarrow \gamma) \rightarrow \beta \rightarrow \gamma \\ \varphi & = & \varphi \rightarrow \beta \rightarrow \gamma \\ \psi & = & \psi \rightarrow \rho \rightarrow \mathtt{int} \\ \sigma & = & \sigma \rightarrow \tau \rightarrow \mathtt{int} \\ \mathtt{int} & \triangleright & \tau \end{array}$$

The inference algorithm indeed reports that the expression has type `int`; but it also produces a series of inclusion constraints, making the result unexpectedly complex.

Assume that this expression is part of a larger program. Other constraints will be generated for the other parts of the program; then the constraint set will be closed and checked for consistency. However, the new constraints cannot possibly involve any of the type variables above, because these variables are unreachable: they are not part of the result type of the expression, nor are they part of the environment the expression was typed in. Hence, these constraints will not affect the consistency of the whole program; we might as well discard them now.

Section 9 formalizes the concept of *unreachable type variables* and shows that constraints involving those variables can be safely removed from the constraint set. Because the use of `let` constructs leads to duplication of constraint sets, removing unreachable constraints early is vital.

3.3 Simplifying constraint sets

We introduce a *substitution rule* to simplify constraint sets. Let us explain the idea. Assume we have written a function `f` from integers to integers. The type inference system might assign it type

$$\mathtt{f} : \forall \alpha\beta.\alpha \rightarrow \beta \mid \{\alpha \triangleright \mathtt{int}, \mathtt{int} \triangleright \beta\}$$

When we apply `f` to an argument of type σ, the function application rule will declare the result to be of type τ, together with constraints

$$\sigma \triangleright \alpha \triangleright \mathtt{int} \qquad \mathtt{int} \triangleright \beta \triangleright \tau$$

Clearly, variables α and β add no information to this constraint set. So we might just as well get rid of them immediately, and declare that `f` has type

$$\mathtt{int} \rightarrow \mathtt{int}$$

We have applied substitution $[\mathtt{int}/\alpha, \mathtt{int}/\beta]$ to the original type.

While doing type inference, we must be careful to choose substitutions which are not restrictive; otherwise, they will yield a less general type, possibly causing us to reject a correct program later. To make sure that the new type is as expressive as the previous one, we will require that it be possible to cancel the effect of the substitution by applying the subtyping rule. That is, before applying substitution γ to constrained type $\tau \mid C$, we will require that $C \Vdash \gamma(C)$ and $C \Vdash \gamma(\tau) \triangleright \tau$.

3.4 Practical results

Our simplification methods have been implemented into a small type-checker. It gives very good results on classic examples. For instance, we have coded the classic `quicksort` function on lists; we omit the code here for brevity. Its unsimplified type contains 300 constraints, only 60 of which are reachable; after simplification, the type is trimmed down to

$$(\alpha \rightarrow \beta \rightarrow \mathtt{bool}) \rightarrow [\mathtt{Nil} \mid \mathtt{Cons\ of}\ \beta * \gamma] \rightarrow \delta$$

together with the constraints

$$\begin{array}{rcl} \delta & = & [\mathtt{Nil} \mid \mathtt{Cons\ of}\ \beta * \delta] \\ \gamma & = & [\mathtt{Nil} \mid \mathtt{Cons\ of}\ \alpha * \gamma] \\ \alpha & \triangleright & \beta \end{array}$$

This is identical to the usual ML type of `quicksort`, except that the first element of the argument list has type β, whereas the remaining ones have type α. This is bewildering at first; it seems that a list can only be sorted if all of its elements have the same type. However, a closer look reveals that this type is correct, and slightly more general than the ML type. This difference comes from the fact that our version of `quicksort` always uses the *first* element of the list as a pivot.

This is of no practical interest, of course, but it is amusing to see that the inferred type is, unexpectedly, more general than the usual ML type. If we restrict the inferred type by applying substitution $[\alpha/\beta, \gamma/\delta]$, we obtain the ML type as a special case. Note that even then the function is more powerful than its ML counterpart, since it can sort heterogeneous lists, provided that each element's type is a subtype of the comparison function's domain type.

The time needed to infer and simplify `quicksort`'s type was slightly under 10 seconds. The prototype typechecker was compiled to native code with Caml Special Light, so the performance still leaves to be desired. However, the prototype uses naive data structures to represent constraints, so substantial speed improvements should be possible.

4 The expression language

The expression language is given by figure 1.

While the concrete syntax of our language has constructs for building and accessing pairs, records and variants, they are not necessary in the theoretical definition of the language. Instead, they can be regarded as an infinite collection of primitive functions.

Instead of type-checking programs in an empty environment, we will always assume a basic environment containing all of these primitives. This way, there is no need to introduce the primitives as constants; they are handled exactly in the same way as variables.

The advantage of introducing such constructs through primitive functions is to reduce the number of typing rules, thus shortening proofs and making it obvious that the typing rules for pairs, records and variants are derived from the function application rule (together with the type of a primitive function). Notice that the expression language only has four constructs: it is a λ-calculus with `let`.

In order to simplify reasoning on records and variants, we assume given a collection of label fields f_i, as well as a collection of constructors K_i, where i ranges over natural integers.

Note that pairs and variants do not add any essential complexity to the system with records; they have been added to allow writing classic examples, such as sorting lists. Classic examples based on records are less common, and often involve contrived pseudo-object-oriented message passing.

5 The type system

5.1 The type language

The type language is defined by figure 2. We denote the set of all type terms by $\mathcal{T}$.

5.2 The type inference rules

Type inference rules are given by figure 3. Judgements are of the form $A \vdash e : \tau \mid C$, where A is an environment, e is the expression to type-check, and $\tau \mid C$ is the inferred constrained type.

All constraint sets appearing in judgements are implicitly required to be *consistent*, as defined below. If a type inference rule yields an inconsistent set, its use is invalid.

The rules are the four classic typing rules for variable instantiation, λ-abstraction, application, and `let` binding, plus a subtyping rule. The latter is based on our *entailment* relation (see section 5.5) and is more powerful than the one introduced in [5], which is based on set containment.

Using properties of the entailment relation, one verifies that it is possible to rewrite any typing proof so that it uses the subtyping rule at most once at the bottom.

As a result, we have

Theorem 5.1 *The type system has subject reduction.*

Proof: This proof is based on Smith's proof of subject reduction[2]. Assume $A \vdash e : \tau \mid C$. Then there exists a typing proof which uses the subtyping rule at most once at the end. Thus, e is typable without using the subtyping rule (hence, in Smith's system) which yields a certain type $\tau_0 \mid C_0$. So, if e reduces to e', then according to Smith's subject reduction property, e' also has type $\tau_0 \mid C_0$ in Smith's system. Because Smith's subtyping rule is weaker than ours, e' also has type $\tau_0 \mid C_0$ within our system; and adding back the final subtyping rule shows that $A \vdash e' : \tau \mid C$ holds.$\Box$

5.3 Closure of a constraint set

A constraint set C is *closed* if and only if it is closed by *transitivity* and by *propagation*.

C is closed by transitivity if and only if

$$\tau_1 \triangleright \alpha \in C \wedge \alpha \triangleright \tau_3 \in C \Rightarrow \tau_1 \triangleright \tau_3 \in C$$

where α is a type variable[3].

C is closed by propagation if and only if it verifies the following conditions:

$$\sigma_1 \rightarrow \sigma_2 \triangleright \tau_1 \rightarrow \tau_2 \in C \Rightarrow \tau_1 \triangleright \sigma_1 \in C \wedge \sigma_2 \triangleright \tau_2 \in C$$

[2]Although we have a larger set of primitive functions, whose semantics should be defined and checked, and we have introduced variant types. A direct proof could also be easily written.

[3]One usually allows any type τ_2 instead of only type variables. We have shown the two definitions to be equivalent. In particular, although this modification alters the definition of a set's closure, it does not change the *consistency* of its closure.

$$\sigma_1 * \sigma_2 \triangleright \tau_1 * \tau_2 \in C \Rightarrow \sigma_1 \triangleright \tau_1 \in C \wedge \sigma_2 \triangleright \tau_2 \in C$$

$$\{f_i : \sigma_i\}_{i \in I} \triangleright \{f_j : \tau_j\}_{j \in J} \in C \Rightarrow \forall k \in I \cap J \quad \sigma_k \triangleright \tau_k \in C$$

$$[K_i \text{ of } \sigma_i]_{i \in I} \triangleright [K_j \text{ of } \tau_j]_{j \in J} \in C \Rightarrow \forall k \in I \cap J \quad \sigma_k \triangleright \tau_k \in C$$

The *closure* of C, denoted C^∞, is the smallest closed constraint set containing C.

5.4 Consistency of a constraint set

A constraint set is *consistent* if and only if all constraints in its closure are consistent. A constraint $\tau_1 \triangleright \tau_2$ is consistent if and only if one of the following conditions holds:

- τ_1 or τ_2 is a type variable
- τ_1 and τ_2 are the same atomic type
- τ_1 and τ_2 are function types
- τ_1 and τ_2 are product types
- τ_1 is $\{f_i : \sigma_i\}_{i \in I}$, τ_2 is $\{f_j : \tau_j\}_{j \in J}$ and $J \subseteq I$
- τ_1 is $[K_i \text{ of } \sigma_i]_{i \in I}$, τ_2 is $[K_j \text{ of } \sigma_j]_{j \in J}$ and $I \subseteq J$

5.5 The entailment relation

Let C_1 and C_2 be coercion sets. We say that C_1 *entails* C_2 (and we write $C_1 \Vdash C_2$) if and only if for any coercion set D such that $C_1 \cup D$ is consistent, $C_2 \cup D$ is also consistent.

This relation behaves like an implication, as demonstrated by the following properties:

$$\begin{array}{rcl} C \Vdash C_1 \text{ and } C \Vdash C_2 & \Longleftrightarrow & C \Vdash C_1 \cup C_2 \\ A \Vdash B \text{ and } B \Vdash C & \Rightarrow & A \Vdash C \\ C_1 \Vdash C \text{ and } C_1 \subseteq C_2 & \Rightarrow & C_2 \Vdash C \end{array}$$

If γ is a substitution,

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

6 Consistency versus solvability

After accumulating subtyping constraints, one has a choice between trying to solve them, as does Aiken [1], or merely verifying that they are consistent, as suggested by Smith [5]. In the former case, the existence of a solution guarantees that the program has a valid monomorphic type, and its soundness can be established from there; in the latter case, the program's soundness is guaranteed by a subject reduction theorem.

Palsberg [11] has established that both approaches are equivalent, in a system similar to ours, and we adapt his proof to our case.

Then, we further show that our notion of entailment, which is based on consistency, is equivalent to a natural notion of entailment based on solutions of constraint sets. Hence, it appears that both approaches are entirely equivalent, and many questions about constraint sets can be formulated in both manners.

Expressions:	
$e ::= x$	variable or constant
$\mid \lambda x.e$	function
$\mid e\, e$	function application
$\mid$ `let` $x = e$ `in` e	polymorphic let

Figure 1. The expression language

Types:	
$\tau ::= \alpha$	type variable
$\mid a$	atomic type: `bool`, `int`, etc.
$\mid \tau \rightarrow \tau$	function type
$\mid \tau * \tau$	product type
$\mid \{f_i : \tau_i\}_{i \in I}$	record type
$\mid [K_i \text{ of } \tau_i]_{i \in I}$	variant type
Constraints:	
$c ::= \tau \triangleright \tau$	
Type schemes:	
$\kappa ::= \forall \overline{\alpha}.\tau \mid C$	a quantified (type, constraint set) pair

Figure 2. The type language

$$\frac{A(x) = \forall \overline{\alpha}.\tau \mid C \qquad \varphi \text{ is a substitution of domain } \overline{\alpha}}{A \vdash x : \varphi(\tau \mid C)} \qquad \frac{A; x : \tau \vdash e : \tau' \mid C}{A \vdash \lambda x.e : \tau \rightarrow \tau' \mid C}$$

$$\frac{A \vdash e_1 : \tau_1 \mid C_1 \qquad A \vdash e_2 : \tau_2 \mid C_2}{A \vdash e_1\, e_2 : \tau \mid C_1 \cup C_2 \cup \{\tau_1 \triangleright \tau_2 \rightarrow \tau\}} \qquad \frac{\begin{array}{c} A \vdash e_1 : \tau_1 \mid C_1 \qquad A; x : \forall \overline{\alpha}.\tau_1 \mid C_1 \vdash e_2 : \tau_2 \mid C_2 \\ \overline{\alpha} = \mathrm{FV}(\tau_1 \mid C_1) \setminus \mathrm{FV}(A) \qquad \varphi \text{ is a substitution of domain } \overline{\alpha} \end{array}}{A \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \mid \varphi(C_1) \cup C_2}$$

$$\frac{A \vdash e : \tau \mid C \qquad C' \Vdash C \qquad C' \Vdash \tau \triangleright \tau'}{A \vdash e : \tau' \mid C'}$$

Figure 3. Type inference rules

6.1 Definition of solvability

We need to give a few definitions before we can introduce the notion of solvability. These definitions are essentially taken from [11], with a few adjustments to accomodate our richer type language, which in particular adds record types. Ground types are defined as regular trees and admit a least type $\bot$ and a greatest type $\top$. They are ordered by a subtyping relation adapted from Palsberg's [11], the latter being equivalent to Amadio and Cardelli's [3].

These definitions are straightforward and given in appendix A.

6.2 Consistency versus solvability

Theorem 6.1 *A constraint set is consistent if and only if it is solvable.*

6.3 Syntactic versus semantic entailment

We have proved that the notion of consistency can be regarded as a more algorithmic definition of solvability, which is a more semantic notion. Now, we would like to compare our syntactic notion of entailment, which is based on consistency, with a semantic notion of entailment based on solvability.

The natural way of defining a semantic notion of entailment between two constraint sets C_1 and C_2 (denoted $C_1 \models C_2$) would be to require that every solution of C_1 be a solution of C_2. It is actually slightly more complex than that, because C_2 might have more free variables than C_1. For instance, the statement

$$\{\alpha \triangleright \texttt{int}, \alpha \triangleright \texttt{bool}\} \models \alpha \triangleright \beta$$

should hold because the principal solution of the left-hand set is $\alpha \mapsto \bot$, and $\bot \triangleright \beta$ will hold for all β. However, $\alpha \mapsto \bot$ is not, strictly speaking, a solution of the right-hand set because it doesn't assign any value to β. Obviously, free variables such as β should be universally quantified. We achieve this effect by giving the following definition:

Definition 6.1 *Let C_1 and C_2 be two constraint sets. We define $C_1 \models C_2$ by*

$$\forall \varphi \in \mathcal{S}(C_1) \quad \forall \psi \in \mathcal{V} \to T_\Sigma \quad \psi\varphi \in \mathcal{S}(C_2)$$

Here, ψ ranges over maps from type variables to ground types and the quantification over ψ is, in effect, a universal quantification over the variables in $\mathrm{FV}(C_2) \setminus \mathrm{FV}(C_1)$.

We can now make the following statement:

Theorem 6.2 *Let C_1 and C_2 be two constraint sets. Then*

$$C_1 \Vdash C_2 \iff C_1 \models C_2$$

7 Introducing *lub*'s and *glb*'s

As explained in section 3, the entailment algorithm will need notions of *greatest lower bounds* and *least upper bounds* of sets of types (sometimes referred to as *glb*'s and *lub*'s for short), which we define now.

7.1 On ground terms

Theorem 7.1 *The set T_Σ of ground terms, ordered by relation $\leq$, forms a lattice. That is, there exist operations $\sqcup$ and $\sqcap$ which, when applied to a set of ground terms, yield its least upper bound and its greatest lower bound, respectively. They are commutative and associative.*

There is not enough room here to give the explicit definition of $\sqcup$ and $\sqcap$. It is very straightforward and yields a series of identities which can be used to compute least upper bounds and greatest upper bounds.

Proposition 7.1 *For any $s, s', t, t' \in T_\Sigma$, these equations hold:*

$$\begin{aligned} t \sqcup \top &= \top \\ t \sqcup \bot &= t \\ (s \to s') \sqcup (t \to t') &= (s \sqcap t) \to (s' \sqcup t') \\ (s * s') \sqcup (t * t') &= (s \sqcup t) * (s' \sqcup t') \end{aligned}$$

For the sake of brevity, we do not give all of them here. Note that in particular, $\sqcup$ and $\sqcap$ are distributive over type constructors.

7.2 On types

In order to be able to use *glb*'s and *lub*'s in constraints, we make the following definition:

Definition 7.1 *A generalized type is a type term possibly containing occurrences of the $\sqcup$ and $\sqcap$ constructors, as well as of $\bot$ and $\top$, as follows:*

Generalized types:

$$\begin{aligned} \tau ::= & \ \ldots \\ & \mid \sqcup\{\tau_i\}_{i\in I} \\ & \mid \sqcap\{\tau_i\}_{i\in I} \\ & \mid \bot \\ & \mid \top \end{aligned}$$

Generalized types are considered modulo the equations given in proposition 7.1[4].

A generalized constraint is a subtyping constraint involving generalized types.

This time, we have introduced $\sqcup$, $\sqcap$, $\bot$ and $\top$ as *constructors* into the syntax of our constraint language. We give them a semantics by defining how they are instantiated:

Definition 7.2 *For any mapping φ from type variables to ground terms, φ is extended to generalized type terms as a homomorphism, that is*

$$\begin{aligned} \varphi(\sqcup\{\tau_i\}) &= \sqcup\{\varphi(\tau_i)\} \\ \varphi(\sqcap\{\tau_i\}) &= \sqcap\{\varphi(\tau_i)\} \\ \varphi(\bot) &= \bot \\ \varphi(\top) &= \top \end{aligned}$$

Two generalized types which are equal up to the equations given in proposition 7.1 are mapped to the same ground term by φ, so this definition does make sense.

This is enough to extend the notions of solution, solvability and semantic entailment to generalized constraint sets.

[4]That is, these equations can be freely used inside a term. There is a slight abuse of language, as they originally applied to ground terms, and they are used for generalized types here.

8 An algorithm to verify entailment

8.1 Description

We now give an algorithm to verify the entailment relation; it is defined by a set of inference rules given in figure 4. Let us comment on these rules.

Judgements inferred by the algorithm are of the form $C, H_1, H_0 \vdash \tau_1 \triangleright \tau_2$ where C is a closed constraint set, H_1 and H_0 are generalized constraint sets, and $\tau_1 \triangleright \tau_2$ is a generalized constraint, which we call the *goal*. Each rule replaces the current goal with zero or more sub-goals.

In goals, $\sqcup$ and $\sqcap$ symbols always occur in contravariant and covariant positions, respectively. That is, each rule produces sub-goals which verify this property, provided that the original goal does.

$$\frac{\tau_1 \triangleright \tau_2 \in C}{C, H_1, H_0 \vdash \tau_1 \triangleright \tau_2}\ (1) \qquad \frac{\tau_1 \triangleright \tau_2 \in H_1}{C, H_1, H_0 \vdash \tau_1 \triangleright \tau_2}\ (2)$$

$$C, H_1, H_0 \vdash \sigma \sqcap \alpha \triangleright \alpha \sqcup \tau \quad (3)$$

$$\frac{\begin{array}{c} R = \{\rho \mid \rho \triangleright \alpha \in C \land \rho \neq \alpha\} \\ C, H_1, H_0 \cup \{\sigma \triangleright \alpha \sqcup \tau\} \vdash \sigma \triangleright (\sqcup R) \sqcup \tau \end{array}}{C, H_1, H_0 \vdash \sigma \triangleright \alpha \sqcup \tau}\ (4)$$

$$\frac{\begin{array}{c} R = \{\rho \mid \alpha \triangleright \rho \in C \land \rho \neq \alpha\} \\ C, H_1, H_0 \cup \{\sigma \sqcap \alpha \triangleright \tau\} \vdash \sigma \sqcap (\sqcap R) \triangleright \tau \end{array}}{C, H_1, H_0 \vdash \sigma \sqcap \alpha \triangleright \tau}\ (5)$$

$$C, H_1, H_0 \vdash \sigma \triangleright \top \quad (6) \qquad C, H_1, H_0 \vdash \bot \triangleright \tau \quad (7)$$

$$C, H_1, H_0 \vdash a \triangleright a \quad (8)$$

$$\frac{C, H_1 \cup H_0, \emptyset \vdash \tau \triangleright \sigma \qquad C, H_1 \cup H_0, \emptyset \vdash \sigma' \triangleright \tau'}{C, H_1, H_0 \vdash \sigma \to \sigma' \triangleright \tau \to \tau'}\ (9)$$

$$\frac{C, H_1 \cup H_0, \emptyset \vdash \sigma \triangleright \tau \qquad C, H_1 \cup H_0, \emptyset \vdash \sigma' \triangleright \tau'}{C, H_1, H_0 \vdash \sigma * \sigma' \triangleright \tau * \tau'}\ (10)$$

$$\frac{J \subseteq I \qquad \forall j \in J \quad C, H_1 \cup H_0, \emptyset \vdash \sigma_j \triangleright \tau_j}{C, H_1, H_0 \vdash \{f_i : \sigma_i\}_{i \in I} \triangleright \{f_j : \tau_j\}_{j \in J}}\ (11)$$

$$\frac{I \subseteq J \qquad \forall i \in I \quad C, H_1 \cup H_0, \emptyset \vdash \sigma_i \triangleright \tau_i}{C, H_1, H_0 \vdash [K_i \text{ of } \sigma_i]_{i \in I} \triangleright [K_j \text{ of } \tau_j]_{j \in J}}\ (12)$$

Figure 4. Algorithm definition

One of the most interesting features of this algorithm is its ability to reason by induction. The algorithm uses H_1 and H_0 as a *trace*, which contains the goals which have been encountered so far. More precisely, if a goal has been encountered and a propagation rule has been used since, then it belongs to H_1; if no propagation rule has been used since, the goal belongs to H_0. The trace is initially empty, i.e. we will be interested in proving statements of the form $C, \emptyset, \emptyset \vdash \tau_1 \triangleright \tau_2$.

Now, in addition to accepting elements of C as hypotheses (rule (1)), the algorithm also regards elements of H_1 as true, thanks to rule (2). This is reasoning by induction: the algorithm realizes that this goal has already been encountered before, so continuing to build a proof for it would cause an endless loop. Proving the goal can be thought of as comparing two infinite trees. However, because at least one propagation rule has been used since the goal was last encountered, we know that part of the trees have been successfully compared. Hence, by carrying on the comparison, we would be able to verify that the two trees can be successfully compared down to an arbitrary depth. So, it is valid to declare that the goal holds. The algorithm owes a lot of its power, and of its complexity, to this induction rule.

The reader might wonder why C and H_1 are kept separate, since constraints found in both sets are considered as true by the algorithm. One reason is that C contains hypotheses (i.e. regular constraints), while H_1 contains goals (i.e. generalized constraints), and merging the two would lead us to generate malformed goals (e.g. with *glb*'s in contravariant positions), which we cannot handle.

Let us now comment on the remaining rules. Which one is to be used depends on the structure of the goal. Distributivity of $\sqcup$ and $\sqcap$ over type constructors is used whenever possible to push down *lub*'s and *glb*'s as far as possible; they remain at top level only when one of their arguments is a variable, or when two arguments have incompatible head constructors.

The reflexivity rule (3) states that a goal holds if a single type variable appears on both sides of it.

Next come the variable elimination rules, (4) and (5), which are symmetrical. Let us consider rule (4), which can be used when the *lub* on the right side of the goal contains a type variable α. To prove that the left side of the goal, σ, is smaller than its right side, it suffices to show that it is smaller than α. To do this, it would be sufficient to find a type ρ such that $\rho \triangleright \alpha \in C$ and prove that σ is smaller than ρ. However, this is too restrictive; instead, it is still sufficient (and easier) to prove that σ is smaller than $\sqcup R$, where R is the set of all[5] lower bounds of α in C.

The rules discussed so far deal with goals which have at least one type variable at top level. The remaining rules deal with the cases where all types appearing at top level are constructed types.

Rules (6) and (7) are symmetrical. $\top$ and $\bot$ appear in goals when computing the *lub* or *glb* of types with incompatible head constructors, such as `int` $\sqcup$ `bool`.

The remaining rules deal with goals where both sides have the same head constructors. These goals can be decomposed into smaller sub-goals, so these rules are collectively called *propagation rules*. There is one per construct: atoms, arrows, pairs, records and variants.

Rule (8) is the propagation rule for atomic types; we do not allow subtyping between base types, so this rule has the look of a reflexivity rule.

The other propagation rules are similar to the propagation rules used when computing the closure of a constraint set, only they work in the opposite way.

[5] except α itself, otherwise the algorithm would not terminate.

8.2 Termination

To make sure that our rules actually define an algorithm, we need to verify that given a goal, determining whether it is provable is a finite process.

A closed coercion set C contains a *cycle* if and only if there exist n type variables ($n \geq 2$) $\alpha_1 \dots \alpha_n$ such that $\{\alpha_1 \triangleright \alpha_2 \triangleright \dots \triangleright \alpha_n \triangleright \alpha_1\} \subseteq C$.

Theorem 8.1 *Let C be a closed coercion set without cycles. Then checking whether there exists a proof of $C, \emptyset, \emptyset \vdash \tau_1 \triangleright \tau_2$ fails or succeeds in finite time.*

Hence, the algorithm works only on coercion sets without cycles. This is not a problem, because we will prove (using the substitution rule) that any cycle can be eliminated by identifying all of its variables[6].

8.3 Correctness

The algorithm is sound with respect to entailment:

Theorem 8.2 *Let C be a constraint set and τ_1, τ_2 be type terms. Then*

$$C, \emptyset, \emptyset \vdash \tau_1 \triangleright \tau_2 \Rightarrow C \Vdash \tau_1 \triangleright \tau_2$$

A sketch of the proof is given in appendix B.

8.4 Completeness

Although we have long believed the algorithm to be complete with respect to the definition of entailment, we have recently found evidence to the contrary.

Since we have shown the equivalence between solvability and consistence (see 6), we present the counter-example from the point of view of solvability. It is slightly simpler. Take $C = \{\alpha \to \mathtt{int} \triangleright \alpha\}$ and $\tau = (\bot \to \top) \to \mathtt{int}$. Then we have[7]

$$C \Vdash \tau \triangleright \alpha$$

but the following assertion does *not* hold:

$$C \Vdash \tau \triangleright \alpha \to \mathtt{int}$$

This means that rule (4) of the algorithm (and, symmetrically, rule (5)) is not complete, since it would replace the goal $\tau \triangleright \alpha$ with the sub-goal $\tau \triangleright \alpha \to \mathtt{int}$.

These are the only two incomplete rules — for all other rules, we have shown that the premises hold if and only if the conclusion holds. We have not yet grasped the full significance of this counter-example, and we do not know how wide a range of counter-examples could be produced. It should be emphasized that even though the algorithm is not complete, it remains powerful and useful in practice; the 60 reachable coercions in `quicksort`'s type are reduced to 3 meaningful ones.

[6] Alternatively, we could modify rules (3), (4) and (5) so that the algorithm still works with sets containing cycles.

[7] Actually, $\bot$ and$\top$ are not allowed in types, so one should define $\tau = (\beta \to \gamma) \to \mathtt{int}$ and add suitable constraints to C so that all solutions of C assign $\bot$ to β and $\top$ to γ. We omit it for the sake of simplicity.

9 Removing unreachable constraints

Consider a typing judgement $A \vdash e : \tau \mid C$. The set of *reachable variables* of this judgement, written as $\mathrm{RV}(A, \tau, C)$, is the smallest set V of type variables such that

$$V \supset \mathrm{FV}(A) \cup \mathrm{FV}(\tau)$$

$$\alpha \triangleright \sigma \in C^\infty \wedge \alpha \in V \Rightarrow \mathrm{FV}(\sigma) \subseteq V$$

$$\sigma \triangleright \alpha \in C^\infty \wedge \alpha \in V \Rightarrow \mathrm{FV}(\sigma) \subseteq V$$

A constraint belonging to C's closure is said to be *reachable* if it contains only reachable variables. That is, the set $\mathrm{RC}(A, \tau, C)$ of reachable constraints is defined as

$$\{\tau_1 \triangleright \tau_2 \in C^\infty \mid \mathrm{FV}(\tau_1) \cup \mathrm{FV}(\tau_2) \subseteq \mathrm{RV}(A, \tau, C)\}$$

We now prove a few lemmas which will be used in forthcoming proofs.

We then introduce the following rule, called the *connexity* rule, which allows removing all unreachable constraints:

$$\frac{A \vdash e : \tau \mid C \qquad C' = \mathrm{RC}(A, \tau, C)}{A \vdash e : \tau \mid C'}$$

From now on, this rule is part of the typing rule and the statement $A \vdash e : \tau \mid C$ represents a derivation which is allowed to make use of this rule.

The following proposition will guarantee the soundness of the new rule (see section 11):

Proposition 9.1 *The connexity rule commutes towards the bottom with the basic typing rules (i.e. all but the subtyping rule), possibly demanding that hidden variables above it be renamed, and yielding a new constraint set which is a subset of the original set's closure.*

10 Simplifying constraints

We prove the following substitution lemma:

Lemma 10.1 *If $A \vdash e : \tau \mid C$ and γ is a substitution such that $\gamma(C)$ is consistent, then $\gamma(A) \vdash e : \gamma(\tau) \mid \gamma(C)$.*

The proof is very similar to that of ML's substitution lemma. Only the case of the subtyping rule is new.

Thus, the type system is not affected by the addition of the following rule:

$$\frac{A \vdash e : \tau \mid C}{\gamma(A) \vdash e : \gamma(\tau) \mid \gamma(C)}$$

However, during type inference, this rule must not be applied blindly, because it restricts the generality of the type and can eventually lead to rejecting a valid program. Since we do not wish to have the type inference algorithm backtrack, we must ensure that it always applies the rule in such a way that the new typing is as general as the previous one. We do this by requiring that the old judgement be a consequence of the new one through the subtyping rule. That is, we will use the substitution rule only as follows:

$$\frac{A \vdash e : \tau \mid C \qquad \gamma(A) = A \qquad C \Vdash \gamma(C) \qquad C \Vdash \gamma(\tau) \triangleright \tau}{A \vdash e : \gamma(\tau) \mid \gamma(C)}$$

The extra hypotheses ensure that the effect of a substitution rule can always be cancelled by a subtyping rule. Consider a typing derivation. At any point in this derivation, we can insert a substitution rule, immediately followed by a subtyping rule which cancels it. Now, the subtyping rule can be pushed towards the bottom of the proof, and eventually discarded, yielding a proof which has the same skeleton as the original proof, except a substitution rule has been added to it. This shows that the inference algorithm can use the substitution rule at any point without fear or failing in the future.

11 Correctness of the new typing rules

Because the substitution rule is derived from the other rules, and because the connexity rule and the subtyping rule commute with the four basic typing rules, adding our two new rules to the type system does not affect the set of typable programs. What's more,

Theorem 11.1 *The extended type system has subject reduction.*

Proof: (Sketch) Consider a typing judgement $A \vdash e : \tau \mid C$. Since the substitution rule is derived from other rules, there exists a proof of this judgement which doesn't use the substitution rule. Since the connexity rule, as well as the subtyping rule, commutes with the four basic rules[8], the proof can be rewritten so as to use these two rules only at the bottom of the proof, and we have a typing $A \vdash e : \tau' \mid C'$ which uses only the four basic rules. Now, assume expression e reduces into e'. Because the original typing system has subject reduction, there exists a proof of $A \vdash e' : \tau' \mid C'$ in the original system (consisting of the five rules in figure 3). Now, the same sequence of rules which was used to derive $A \vdash e : \tau \mid C$ from $A \vdash e : \tau' \mid C'$ can be used to derive $A \vdash e' : \tau \mid C$. $\square$

12 Choosing adequate substitutions

Given a typing judgement and a substitution γ, we have defined how to formally verify whether γ can be safely applied to this judgement. Now, a practical simplification algorithm needs to be able to determine which substitutions are likely to succeed on a given constraint set. This is important both for speed and for effectiveness of the simplification process.

Here, we have only heuristics. We give a few of them below. More study is needed in this area, as we have no formal results.

12.1 Basic heuristics

One heuristic is to look for cycles and eliminate them by identifying all variables within a cycle.

Another idea is to look only for substitutions which affect one variable at a time (although this is not sufficient, see 12.3). Consider a variable α. Two likely candidates for its replacement are the least upper bound of α's lower bounds in C, and, symmetrically, the greatest lower bound of its upper bounds. For instance, if C contains the constraints $\alpha \triangleright \{l_1 : \tau_1\}$ and $\alpha \triangleright \{l_2 : \tau_2\}$, trying to replace α with $\{l_1 : \tau_1; l_2 : \tau_2\}$ is probably a good idea. However, the terms of "least upper bound" and "greatest lower bound" used above are only informal; unlike Aiken [1], we have no such notions when the types involved contain variables. When our heuristic encounters type variables, for instance when "computing" $\alpha \sqcup \beta$, it tries both choices, α and β, one after the other.

12.2 Turning recursive constraints into equations

We find it desirable, whenever possible, to replace recursive constraints with equations, because a type variable involved in a recursive equation is easily understood as a fixpoint, whereas the effect of recursive constraints is less clear. We have found that this is possible in many cases. For instance, if we write a function which computes the length of a list, the following type might be inferred for it:

$$[\texttt{Nil} \mid \texttt{Cons of } \alpha * \beta] \rightarrow \texttt{int} \mid \{\beta \triangleright [\texttt{Nil} \mid \texttt{Cons of } \alpha * \beta]\}$$

Here, the type variable β, which obviously represents lists of elements of type α, is bound by a recursive constraint, instead of an equation.

Using the subtyping rule, we can introduce a new equation into the constraint set

$$\gamma = [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma]$$

where γ is a fresh variable[9].

We can now apply the substitution $[\gamma/\beta]$. To verify that the premises of the substitution rule are met, we need to show

$$\beta \triangleright [\texttt{Nil} \mid \texttt{Cons of } \alpha * \beta], \gamma = [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma] \Vdash \beta \triangleright \gamma$$

which is immediate using the entailment algorithm. Hence, our function has type

$$[\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma] \rightarrow \texttt{int} \mid \{\gamma = [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma]\}$$

and we have effectively replaced the constraint with an equation.

12.3 One variable at a time is not enough

The suggestion to try only substitutions whose domain is a singleton, although efficient in most cases, is too restrictive. For instance, suppose we have inferred type $\beta \mid C$ for a certain expression e, where

$$C = \{\beta = [\texttt{Nil} \mid \texttt{Cons of } \alpha * \beta], \texttt{int} \triangleright \alpha\}$$

This intuitively represents a list of integers, so we would like to apply substitution $[\texttt{int}/\alpha]$. However, the substitution rule requires us to prove that

$$C \Vdash \beta = [\texttt{Nil} \mid \texttt{Cons of int} * \beta]$$

which is false.

In fact, the variable β has been introduced only to represent a recursive type. If the type language had μ-binders, we could write the type of e as

$$\mu\beta.[\texttt{Nil} \mid \texttt{Cons of } \alpha * \beta] \mid \{\texttt{int} \triangleright \alpha\}$$

[8] The connexity rule does not exactly commute: when pushing it towards the bottom, it yields a constraint set smaller than the original set. This is not a problem — this change has to be propagated down, and it obviously can: since there are fewer constraints than there used to be, no new inconsistencies can appear.

[9] This can be done without fear of making the inference algorithm fail, because the new constraints are unreachable and could be removed by the connexity rule.

which the substitution $[\texttt{int}/\alpha]$ would turn into

$$\mu\beta.[\texttt{Nil} \mid \texttt{Cons of int} * \beta]$$

Here, β represents a different fixpoint. This suggests that, in the language without μ-binders, the substitution should also affect β.

Indeed, we can add a fresh variable γ together with the equation

$$\gamma = [\texttt{Nil} \mid \texttt{Cons of int} * \gamma]$$

in the same way as in 12.2, and then apply substitution $[\texttt{int}/\alpha, \gamma/\beta]$. This time the conditions of the substitution rule are met and we end up with type

$$\gamma \mid \{\gamma = [\texttt{Nil} \mid \texttt{Cons of int} * \gamma]\}$$

as expected.

We have implemented a heuristic which looks for variables bound by recursive equations, such as β, and extends the scope of the substitution as described above. It is necessary in practice, for example when simplifying `quicksort`'s type.

13 Related work

Amadio and Cardelli [3] have defined an algorithm to check whether two types are in the subtype relation. This algorithm uses a set of subtyping hypotheses similar to our constraint sets, except that there is only one bound per type variable. Our algorithm presents interesting similarities with theirs. Their rule for dealing with recursive types, μ_A, reads

$$\Sigma \cup \{t \leq s\}, \varepsilon \supset \varepsilon(t) \leq \varepsilon(s) \Rightarrow \Sigma, \varepsilon \supset t \leq s$$

This is similar to our variable elimination rules: variables are replaced with their bounds and the current goal is added to the execution trace. Their requirement that types should in be in canonical form, i.e. the body of a μ should not be another μ, corresponds to our requirement that the constraint set should contain no cycles. Their algorithm is complete, whereas ours isn't. This is because their initial set of assumptions only has equations, while ours contains inclusion constraints; hence replacing a variable with its bound poses a problem.

Fuh and Mishra [7] have described several methods for simplifying constraint sets. Each of them is subsumed by our substitution rule. The conditions for each of their methods are sufficient to verify that our substitution rule applies; however, they do not deal with the more complex cases handled by our rule.

Aiken [2] has built a constraint simplification algorithm into Illyria, a Lisp implementation of his constraint solver. Aiken's type language has intersection and union types, so each type variable has at most one upper bound and one lower bound. To simplify constraint sets, variables which occur only in covariant (resp. contravariant) positions are identified with their upper (resp. lower) bound. This is a special case of our substitution rule, provided that the variable's bound can be expressed in our more restricted type language. Aiken does not seem to handle variables with mixed variance; our substitution rule does (and such cases are easily encountered in practice).

Jones [8] has established a general framework for inferring types together with sets of constraints, without specifying the semantics of constraints. Our system is a special case of his. Namely, his so-called "simplification" rule is a special case of our subtyping rule (Jones requires that the constraint sets be equivalent in order to ensure completeness of the type inference algorithm, otherwise the two rules would be identical), and our substitution rule is a special case of his "improvement" rule. Indeed, his definition of "γ improves C" is (informally) equivalent to our requirement that $C \Vdash \gamma(C)$. Of course, Jones' framework is very abstract and does not tackle the issues of verifying the $\Vdash$ relation and of finding adequate substitutions γ.

Smith [6] has defined a rule to remove unreachable constraints which is very similar to ours, as well as a series of substitution rules (replacing a variable by its bound, merging variables). His method for removing excess constraints is more powerful than ours, because it tests whether variables occur in covariant or contravariant positions, whereas ours only checks whether they appear at all. For instance, given an expression of type $\alpha \mid \{\texttt{int} \triangleright \alpha, \alpha \triangleright \texttt{int}\}$, Smith can remove the second constraint, whereas we cannot[10]. However, this does not seem to be a limitation in practice, because actual constraint sets correspond to a program's data flow, and variables tend to appear in "natural" positions, so that Smith's rule has few extra possibilities as compared to ours — and these cases are usually solved anyway by our simplification rule. Of course, this statement is only informal. Smith's substitution rules, on the other hand, are restricted cases of ours (they handle only variables with single variance and with a single bound).

14 Conclusion

We have evidenced that the constraint-based type inference system introduced in [5], although theoretically correct, depends on type simplification in order to be useable in practice. Smith has introduced some simplification rules [6]; we have shown that most of them can be understood as restricted cases of a general substitution rule. Smith's reachability rule is similar to our connexity rule, and slightly more powerful; they were developed independently.

We have defined a new notion of entailment, which gives great power to the subtyping rule. Our substitution rule makes use of entailment, so we have introduced an algorithm to verify entailment and proven its soundness. The algorithm, although incomplete, seems to be powerful and gives good results in actual use.

Building on work by Palsberg [11], we have related consistency to solvability and proved that our notion of entailment is equivalent to a more natural, semantic one.

Several directions for future work can be followed. First, determining whether it is possible to strengthen the algorithm so as to achieve completeness with respect to entailment is desirable. Then, as mentioned in section 12, comes the issue of choosing adequate substitutions, which has not been formally investigated yet.

It might desirable to find a better formulation for the typing rules. For instance, we often had the intuition that unreachable variables should be existentially quantified, thus easing some thorny renaming problems. Also, it might be possible to enhance the generalization rule. Currently it enters the whole constraint set into the environment and each instantiation rule duplicates the constraints; ideally, only

[10]Extending our connexity rule after Smith might be possible but is not trivial — it breaks our proof of the substitution lemma.

constraints which have an effect on the generalized variables should be duplicated and instantiated.

Finally, a desirable goal is to obtain principal types. Yet, even if one achieves it, the notion of "simplest" type remains subjective, because there seems to be no way to prove that no other type inference system will yield a "simpler" type for a given program.

Attaining these goals would allow a better understanding of subtyping constraints. The results obtained so far, both theoretical and practical, are promising and encourage us to continue investigating this field.

References

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

[2] Alexexander Aiken. Illyria system, 1994. Available online as `http://http.cs.berkeley.edu:80/~aiken/ftp/Illyria.tar.gz`.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 104–118, Orlando, FL, January 1991. Also available as DEC Systems Research Center Research Report number 62, August 1990.

[4] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer Verlag, 1984. Also in Information and Computation, 1988.

[5] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. To appear. Currently available as `ftp://ftp.cs.jhu.edu/pub/scott/ooinfer.ps.Z`.

[6] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA'95*, 1995. Available as `ftp://ftp.cs.jhu.edu/pub/scott/sptio.ps.Z`.

[7] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT'89*, 1989.

[8] Mark Jones. Simplifying and improving qualified types. Technical Report YALEU/DCS/RR-1040, Yale University, New Haven, Connecticut, USA, June 1994.

[9] John C. Mitchell. Coercion and type inference. In *Eleventh Annual Symposium on Principles Of Programming Languages*, 1984.

[10] Jens Palsberg. Efficient type inference of object types. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 186–195, Paris, France, July 1994. IEEE Computer Society Press. To appear in Information and Computation.

[11] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. To appear in Proc. POPL 95. Currently available as `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl95.ps.Z`, 1995.

A Definition of solvability

Definition A.1 *Let Σ be a ranked alphabet composed of:*

- $\bot$, $\top$ *with null arity*
- *Atomic types (*`int`, `bool`*, etc.) with null arity*
- $\rightarrow$, $*$ *with arity 2*
- $\{\}_I$ *with arity* $|\, I \,|$*, for each set of labels* I
- $[\ ]_I$ *with arity* $|\, I \,|$*, for each set of labels* I

A ground type *is a regular tree over* Σ*. A* path *from the root of such a tree is a string over* $\mathcal{P} = \{d, r, f, s\} \cup I\!N$*, where* d *and* r *stand for "domain" and "range" respectively,* f *and* s *stand for "first" and "second", and a natural integer indicates a record or variant field number.*

Definition A.2 *A ground type is represented by a* ground term*, that is, a partial function from* $\mathcal{P}^*$ *to* Σ *which maps each path to the symbol at the end of that path. The set of all such terms is denoted* T_Σ*.*

Ground types are ordered by the subtype relation $\leq$, as follows.

Definition A.3 *The* parity *of* $p \in \mathcal{P}^*$*, denoted* πp*, is the number of* d*'s in* p*, taken modulo 2. Let* $\leq_0$ *be the partial order on* Σ *given by*

$$\forall \sigma \in \Sigma \quad \bot \leq_0 \sigma \leq_0 \top$$
$$I \subseteq J \Rightarrow \{\}_J \leq_0 \{\}_I$$
$$J \subseteq I \Rightarrow [\ ]_J \leq_0 [\ ]_I$$

and let $\leq_1$ *be its reverse*

$$\forall \sigma \in \Sigma \quad \top \leq_1 \sigma \leq_1 \bot$$
$$J \subseteq I \Rightarrow \{\}_J \leq_1 \{\}_I$$
$$I \subseteq J \Rightarrow [\ ]_J \leq_1 [\ ]_I$$

Two trees $s, t \in T_\Sigma$ *are in the subtype relation* $s \leq t$ *if and only if*

$$\forall p \in Dom(s) \cap Dom(t) \quad s(p) \leq_{\pi p} t(p)$$

We can now define the notion of solvability:

Definition A.4 *Let* C *be a constraint set. Let* φ *be a map from* $FV(C)$ *to* T_Σ*.* φ *is said to be a* solution *of* C *if and only if*

$$\forall \tau_1 \triangleright \tau_2 \in C \quad \varphi(\tau_1) \leq \varphi(\tau_2)$$

The set of all solutions of C *is denoted* $\mathcal{S}(C)$*.* C *is* solvable *if and only if* $\mathcal{S}(C) \neq \emptyset$*.*

B Correctness of the entailment algorithm

This appendix contains a sketch of the proof of correctness for our entailment algorithm. Reading it is not necessary to gain an understanding of the paper.

B.1 Weaker notions of solvability and entailment

Le $k \geq 0$. We define a new ordering relation on ground terms, denoted $\preceq_k$, by saying that $s \preceq_k t$ if and only if

$$\forall p \in \mathrm{Dom}(s) \cap \mathrm{Dom}(t) \quad \mid p \mid < k \Rightarrow s(p) \leq_{\pi p} t(p)$$

$\preceq_k$ is a weaker version of $\leq$ which compares trees only up to depth k.

Now, by replacing $\leq$ with $\preceq_k$ in the definitions of $\mathcal{S}$ and $\models$, we obtain weaker notions of solutions of a constraint set, and of entailment, denoted $\mathcal{S}_k$ and $\models_k$.

These new notions of entailment are related to the regular entailment relation by the following implication:

$$(\forall k \geq 0 \quad C_1 \models_k C_2) \Rightarrow C_1 \models C_2$$

k-entailment shares the three basic properties of entailment given in section 5.5.

B.2 Sketch of the proof

We now want to prove that the algorithm is sound, i.e. if $C, \emptyset, \emptyset \vdash \tau_1 \rhd \tau_2$, then $C \models \tau_1 \rhd \tau_2$, where $\tau_1 \rhd \tau_2$ is a generalized constraint.

The proof is made difficult by the existence of the induction rule (2). We would like to prove directly that

$$C, H_1, H_0 \vdash \tau_1 \rhd \tau_2 \Rightarrow C \models \tau_1 \rhd \tau_2$$

but this is obviously false in the absence of any hypothesis on H_1. Rather than trying to express a complex invariant for H_1, we choose to build an inductive proof based on relation $\models_k$.

Let P be a proof of assertion $C, H_1, H_0 \vdash \tau_1 \rhd \tau_2$. One calls *depth* of P the number of propagation rules used in the shortest branch topped by an induction rule. If the proof uses no induction rule at all, the depth of P is said to be infinite.

The proof of correctness is cut into two parts. The hardest part is proving

Theorem B.1 *If $C, H_1, H_0 \vdash \tau_1 \rhd \tau_2$ has a proof of depth k, then*

$$\forall k' \leq k \quad C \models_{k'} \tau_1 \rhd \tau_2$$

The proof is by induction on the proof's structure. The idea here is that the induction rule is considered "unsafe", because it takes a hypothesis from H_1, and we know nothing about H_1. So, if all induction rules are used at depth k or deeper, we should be able to show $C \models_{k'} \tau_1 \rhd \tau_2$, which involves only constraints derived with a derivation of height $k-1$ at most. Note that in the proof of this theorem, the case of the induction rule is trivial, because then $k = 0$, and relation $\models_0$ holds for all constraint sets.

The second part of the proof consists in the following theorem:

Theorem B.2 *Let P be a proof of assertion $C, \emptyset, \emptyset \vdash \tau_1 \rhd \tau_2$ of finite depth k. Then there exists a proof P' of the same assertion, with depth strictly greater than k.*

This is rather easy to understand. Whenever the induction rule is used in a proof, the current goal is part of H_1, so it has been encountered before. Instead of using the induction rule immediately, one can modify the proof to use regular rules. Necessarily, the same goal will be encountered later, and only then will the induction rule be used. This causes at least one more propagation rule to be used (because the goal was in H_1, and only propagation rules add goals to H_1). So, doing this for every induction rule at depth k yields a proof of depth strictly greater than k.

The combination of these two theorems gives the soundness theorem:

Theorem B.3 *Let C be a constraint set and τ_1, τ_2 be type terms. Then*

$$C, \emptyset, \emptyset \vdash \tau_1 \rhd \tau_2 \Rightarrow C \models \tau_1 \rhd \tau_2$$

If a proof of $C, \emptyset, \emptyset \vdash \tau_1 \rhd \tau_2$ exists, then there exist proofs of arbitrary depth, thanks to theorem B.2. So, according to theorem B.1, $C \models_k \tau_1 \rhd \tau_2$ holds for arbitrarily large k. Hence, $C \models \tau_1 \rhd \tau_2$ holds.