Type inference and simplification for recursively constrained types

François Pottier

1 Introduction

This paper studies type inference for a functional language with subtyping, and focuses on the issue of simplifying inferred types. It does not attempt to give a fully detailed, formal framework; this shall be the topic of a forthcoming paper.

The type system itself was devised by Eifrig, Smith and Trifonov [4]. It is based on *constrained types*, i.e. types of the form $\tau \mid C$, where τ is a regular type expression and C is a set of *subtyping constraints* of the form $\tau_1 \triangleright \tau_2$.

The language offers *polymorphic* record (and variant) types, i.e. they are in a non-structural subtyping relation. For instance, $\{a : \tau_a ; b : \tau_b\} \triangleright \{a : \tau'_a\}$ if and only if $\tau_a \triangleright \tau'_a$; even though type τ_b appears in the first constraint, it is not affected by it. This form of subtyping is more complex than usual structural subtyping, where $\tau_1 \triangleright \tau_2$ can hold only if terms τ_1 and τ_2 have the same shape. Thanks to this subtyping relation, it is possible to "forget" about the presence of certain fields of a record. This mechanism can be used as a basis to encode object-oriented messaging and inheritance.

The type inference algorithm analyzes the program and gathers a set of subtyping constraints; the program is accepted if this set is *consistent*. One does not try to actually *solve* the constraint set.

The main drawback of this system is the huge number of generated constraints: proportional to program size. For real-world programs, types cannot be understood by humans. So, while the type system is theoretically correct, type simplification is still a prerequisite for practical use. It is vital, given a type, to be able to reduce it into a smaller, equivalent type.

To this end, we have devised several methods. They are explained in this paper; they are not fully formalized or detailed here, but examples are given to help understand their design.

2 A quick overview of the type system

The type language is defined by figure 1. The language supports atomic types such as `int`, `bool`, etc. but does not allow subtyping between atomic types. Although it should be possible to support it, it was not deemed useful enough. Subtyping mainly takes place between records (and, dually, between variants).

The expression language is given by figure 2. It is a λ-calculus with `let`. 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 viewed as a (denumerable) collection of primitive functions (for instance, pairs can be built using `(_, _)` and accessed using `fst` and `snd`). Type-checking takes place in a "built-in" environment containing these primitives, so they are handled in the same way as variables.

Journées du GDR Programmation.
22, 23 et 24 novembre 1995.
Grenoble.

Types:

$$
\begin{array}{rll}
\tau ::= & \alpha & \text{type variable} \\
| & A & \text{atomic type: } \texttt{bool}, \texttt{int}, \text{etc.} \\
| & \tau \to \tau & \text{function type} \\
| & \tau * \tau & \text{product type} \\
| & \{f_i : \tau_i\}_{i \in I} & \text{record type} \\
| & [K_i \text{ of } \tau_i]_{i \in I} & \text{variant type}
\end{array}
$$

Constraints:

$$c ::= \tau \triangleright \tau$$

Type schemes:

$$\kappa ::= \forall \overline{\alpha}.\tau \mid \{c_i\} \qquad \text{a quantified (type, constraint set) pair}$$

Figure 1: The type language

Expressions:

$$
\begin{array}{rll}
e ::= & x & \text{variable or constant} \\
| & \lambda x.e & \text{function} \\
| & e\, e & \text{function application} \\
| & \texttt{let } x = e \texttt{ in } e & \text{polymorphic let}
\end{array}
$$

Figure 2: The expression language

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

$$\frac{A \vdash e_1 : \tau_1 \mid C_1 \qquad A \vdash e_2 : \tau_2 \mid C_2}{A \vdash e_1\, e_2 : \tau \mid C_1 \cup C_2 \cup \{\tau_1 \triangleright \tau_2 \to \tau\}}$$

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

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

Figure 3: Type inference rules

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 *closed* and *consistent*, as defined below. When taking the union of several sets, one implicitly takes the closure of this union. If a type inference rule yields an inconsistent set, its use is invalid.

Each function application generates a subtyping constraint: the type of the supplied argument must be a subtype of the type expected by the function.

The natural thing to do after accumulating a system of constraints is trying to solve it (that is, finding a substitution from variables to ground types which satisfies all constraints), and declaring that the program is well-typed if and only if the constraints have a solution. However, solving a constraint set is not easy; it involves at least introducing recursive types (μ-binders), as well as $\top$ and $\bot$ types. So, instead of solving the constraint set, one only verifies that it is *consistent*, as defined below. The approach of solving constraints is detailed in [1], while the approach of checking consistency is introduced in [4]. A formal equivalence between the two has not yet been established, but seems to be an interesting research topic.

Checking consistency of a constraint set is done in two steps: computing its *closure* and checking that the closure contains only consistent constraints. Computing the closure is done using two kinds of rules: transitivity

$$\frac{C \vdash \tau_1 \triangleright \alpha \qquad \alpha \triangleright \tau_3}{C \vdash \tau_1 \triangleright \tau_3}$$

and structural propagation. There are different propagation rules for each type construct, but all of them have a similar appearance. For instance, here is the rule for record types:

$$\frac{C \vdash \{f_i : \sigma_i\}_{i \in I} \triangleright \{f_j : \tau_j\}_{j \in J} \qquad k \in J \subset I}{C \vdash \sigma_k \triangleright \tau_k}$$

Once the closure has been computed, the algorithm checks that it is *consistent*. A constraint set is consistent if it does not contain any ill-formed constraints, that is, constraints of the form $\tau_1 \triangleright \tau_2$ where τ_1 and τ_2 are terms with different head constructors. For instance, typing application `(0 1)` would generate constraint $\texttt{Int} \triangleright \texttt{Int} \rightarrow \alpha$, which is inconsistent.

The policy of checking the consistency of a constraint set instead of trying to solve it could be qualified as "lazy". It should be emphasized that any constraint with a variable on the left or right side is automatically declared consistent. For instance, $\tau_1 \rightarrow \tau_2 \triangleright \alpha$ is a consistent constraint and requires no further attention. One would be tempted to replace variable α with expression $\alpha_1 \rightarrow \alpha_2$ (where α_1 and α_2 are fresh variables), and to break the constraint into more elementary constraints: $\alpha_1 \triangleright \tau_1$ and $\tau_2 \triangleright \alpha_2$. However, this "greedy" approach does not terminate when faced with a recursive constraint such as $\tau_1 \rightarrow \alpha \triangleright \alpha$. Choosing the lazy policy eliminates this problem.

The inference system described above is simple, elegant and can easily be extended to new language features. Its main disadvantage is the size of the coercion sets. Since a constraint is generated for every application node in the program, the size of the constraint set is directly proportional to that of the program. Thus, it can get out of hand very quickly. Indeed, even simple primitive functions yield large coercion sets. This situation hampers readability: understanding the inferred type becomes not just a challenge, but an utopia. Also, it raises the issue of speed: computing the closure of a large coercion set is slow; also, coercions are duplicated when a `let` construct is encountered; hence, the smaller the constraint sets involved, the faster the type inference process completes.

We have devised several methods to reduce the size of a constraint set, while conserving its meaning. The first one works by removing so-called "disconnected" constraints from the set. The other works by applying a substitution to the type and to the coercion set; determining which substitution to use and proving that the substitution yields an equivalent type are two interesting problems. The following sections give an overview of the simplification algorithms we have developed, as well as examples using our lightweight typechecker implementation.

3 Removing disconnected constraints

Consider the following computation:

```
let y = function f -> (function g -> function x -> f (g g) x)
                      (function g -> function x -> f (g g) x) in

let compute = y (function f -> function x -> plus 1 (f x)) in
compute 1;;
```

The syntax used here is that of our prototype implementation of the typechecker, and is therefore somewhat awkward. Here `y` is the classic fix-point combinator. Therefore, `compute` is a function which takes an integer and returns an integer (it is of no significance here that the function does not terminate), and the computation as a whole should have an integer result.

Now, let's look at the output produced by a (hypothetical) typechecker which does not remove disconnected constraints:

```
int where
  'a |> ('b -> 'c) -> 'd -> 'e
  'a |> ('b -> 'c) -> 'b -> 'c
  'f -> 'b -> 'c = 'f
  'g -> 'h -> int = 'g
  'i -> 'j -> int = 'i
  int |> 'j
```

Here `'a`, `'b`, etc. should be read α, β, etc. and stand for type variables. 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.

Each of these constraints is, of course, consistent, otherwise the typechecker would have rejected the expression. Also, each of them has a variable on at least one side; otherwise, we would have broken it into smaller, equivalent constraints.

What is their meaning? Assume that the above 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 (by transitivity and structural decomposition) and its consistency will be verified. Since the coercions above are consistent and cannot be decomposed into more elementary constraints, the only influence they could have is through transitivity. But the variables involved in these constraints will be found nowhere else in the constraint set, because the typing algorithm uses fresh variables to type the rest of the program. Hence, transitivity cannot be used with these constraints. To sum things up, these constraints have no effect whatsoever on the consistency of the coercion set associated to the whole program. Hence, we can safely remove them now, and report that this expression has type `Int`, with no associated coercions.

More generally speaking, assume we have inferred $A \vdash e : \tau \mid C$ (where A is an environment, e is a language expression, τ a type expression and C a coercion set). We define the set of *reachable variables* as the smallest set R of type variables such that: first, $\mathrm{FV}(A) \cup \mathrm{FV}(\tau) \subset R$, i.e. variables which appear free in the environment or in τ are reachable; and second, if $\tau_1 \triangleright \tau_2 \in C$, then $\mathrm{FV}(\tau_1) \subset R \iff \mathrm{FV}(\tau_2) \subset R$, i.e. if all variables on one side of a coercion are reachable, all variables on the other side become reachable too.

Now, each coercion in C either involves only reachable variables, or has at least one unreachable variable on each side. In the latter case, the coercion is said to be *disconnected*. It is now rather straightforward to prove that disconnected constraints can be removed from the coercion set without affecting the rest of the type inference process. This property holds if we only work with canonical typing proofs, that is, if the typing algorithm uses fresh variables wherever possible, so that a single type variable cannot appear in two different branches of the typing proof's tree.

Formally speaking, we introduce the following typing rule (called the connexity rule):

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

Here $\mathrm{RC}(C, \mathrm{FV}(A) \cup \mathrm{FV}(\tau))$ stands for the set of reachable coercions in C. Hence, the rule states that any disconnected coercions can be removed. Now, the following property holds:

Theorem 3.1 *Consider a canonical proof of the typing judgement $A \vdash e : \tau \mid C$, possibly making use of the connexity rule. Then there exists a canonical proof of this same judgement such that the connexity rule is used at most once at the end of the proof.*

This theorem is proved by showing that the connexity rule commutes with all other rules. As a consequence, adding the connexity rule to the set of typing rules doesn't affect the set of well-typed programs.

In conclusion, removing disconnected constraints is easy, and it is mandatory to avoid keeping constraints which no longer have any effect.

4 Using substitutions to simplify coercion sets

4.1 The substitution rule

The substitution which maps variable α to type τ is written as $[\tau/\alpha]$.

One possible type for the addition primitive `+` is

$$\forall\alpha\beta\gamma.\alpha \rightarrow \beta \rightarrow \gamma \mid \alpha \triangleright \texttt{Int}, \beta \triangleright \texttt{Int}, \texttt{Int} \triangleright \gamma$$

This type expresses that `+` accepts arguments of any types α and β, provided that α and β are both subtypes of `Int`. But it would be equivalent to say that `+` expects arguments of type `Int`, because then, when `+` is applied, the typechecker would generate constraints expressing that the types of the actual arguments are subtypes of `Int`. We can reason in a similar way regarding γ: if we said that `+` returns a result of type `Int`, we would be able to pass this result to any function which accepts a supertype of `Int`, because applying the function would generate the proper constraint. Hence, the above type is equivalent to

$$\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$$

Intuitively speaking, we have applied substitution $[\texttt{Int}/\alpha, \texttt{Int}/\beta, \texttt{Int}/\gamma]$ to the original type. It remains to be seen how applying a substitution to a type can be proven to be valid in general.

Assume a typing judgement $A \vdash e : \tau \mid C$ has been derived. Let σ be a substitution which leaves the environment untouched, i.e. $\sigma(A) = A$. Since the typing rules require all coercion sets appearing in a judgement to be consistent, we are assured that C is consistent. However, this is not necessarily the case for $\sigma(C)$: replacing variables with terms can introduce inconsistencies. For instance, coercion $\alpha \triangleright \beta \rightarrow \gamma$ is consistent, but replacing α with a product makes it inconsistent. So, we will further assume that $\sigma(C)$ is consistent. Under this hypothesis, it is easy to rewrite the proof of $A \vdash e : \tau \mid C$ into a proof of $A \vdash e : \sigma(\tau) \mid \sigma(C)$. This means that applying σ to an inferred type is sound: it always yields another correct typing judgement.

However, we have to be more cautious if the substituted type is to be equivalent to the original one. Generally, it is less general than the original type, so applying a badly chosen substitution might lead us to reject an otherwise correct program. We show that substitution σ yields an equivalent type if the following two requirements are met:

$$C \Vdash \sigma(C)$$

$$C \Vdash \sigma(\tau) \triangleright \tau$$

Here, symbol $\Vdash$ reads "entails" and will be formally introduced shortly thereafter. It is interesting to note that [4] uses the following subsumption rule between judgements[1]:

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

[1]Except that the authors use plain containment ($\supset$) instead of our more powerful entailment ($\Vdash$) relation.

Hence, the two requirements introduced above mean exactly that it is possible to revert from $A \vdash e : \sigma(\tau) \mid \sigma(C)$ to $A \vdash e : \tau \mid C$ by applying the subsumption rule. The substituted type must therefore be as general as the original one.

To summarize, the substitution rule reads

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

We have shown that adding this typing rule to the system does not alter the set of well-typed programs. Also, we have shown that it is safe for the type inference algorithm to apply this rule at any time, i.e. applying the rule will not lead the algorithm into a dead end, causing it to reject an otherwise valid program.

4.2 The entailment relation

When reasoning about coercion sets, the need to define an entailment relation arises naturally; that is, we need to be able to formally define and prove that a certain set of coercions are "consequences" of another set.

Given two coercion sets C_1 and C_2, a possibility is to compute their closures (by transitivity and structural decomposition) $\overline{C_1}$ and $\overline{C_2}$, and to define "C_1 entails C_2" by $\overline{C_1} \supset \overline{C_2}$, as in [4]. However, this definition has limited power. Closing the set using full structural congruence (yielding infinite closures) would give slightly better results, but some desirable deductions remain out of reach (see the example below).

We realized that any entailment relation should verify the following natural property: if C_1 entails C_2, then for any coercion set D such that $C_1 \cup D$ is consistent, $C_2 \cup D$ is consistent. This property expresses that since C_2 is a consequence of C_1, it cannot possibly cause an inconsistency in an environment where C_1 didn't. We noted that most proofs pertaining to the entailment relation can be written so as to use only this property, and no knowledge of how the relation is defined.

This remark suggested that we define $\Vdash$ precisely by

$$C_1 \Vdash C_2 \iff (\forall D \quad C_1 \cup D \text{ consistent} \Rightarrow C_2 \cup D \text{ consistent})$$

This way, $\Vdash$ seems to be as powerful an entailment relation as possible.

Let us give an interesting example of non-trivial entailment between two coercion sets. Let

$$C_1 = \{\alpha \triangleright F(\alpha), \beta = F(\beta)\}$$

$$C_2 = \{\alpha \triangleright \beta\}$$

where F is a unary, covariant context. We claim that $C_1 \Vdash C_2$ (showing it takes a quick induction which we can't develop here). This kind of deduction is impossible if one simply compares closures; yet it is desirable and useful in practice: it is needed when trying to simplify most functions dealing with recursive types, such as lists or objects coded as records.

4.3 Deciding the entailment relation

It is not clear whether relation $\Vdash$ is decidable. For now, we have not tried to address this issue directly; instead, we have concentrated on designing an algorithm that be as close to completeness as possible. Even if the relation is not decidable, we hope to define an algorithm which is powerful enough in practice.

This section describes the steps we have taken in designing this algorithm. We do not know yet whether it is complete or incomplete, but it is already powerful enough to give very good practical results.

We want to define an algorithm capable of deciding whether $C \Vdash \tau_1 \triangleright \tau_2$ (then, deciding whether $C_1 \Vdash C_2$ is a straightforward extension). From now on, we assume C to be closed (by transitivity and structural decomposition).

4.3.1 A basic algorithm

Let us start by giving a simplified version of the algorithm. It tries to prove entailment using the following rules (where $\vdash$ stands for a proof by the algorithm):

- (AXIOM):

$$\frac{\tau_1 \triangleright \tau_2 \in C}{C \vdash \tau_1 \triangleright \tau_2}$$

- (DOWN): A series of structural decomposition rules of the form:

$$\frac{\forall i \in \{1 \dots n\} \quad C \vdash c_i}{C \vdash \tau_1 \triangleright \tau_2}$$

  where the c_i are the elementary constraints which make up constraint $\tau_1 \triangleright \tau_2$. There is one such rule per construct; for instance, the one for products reads

$$\frac{C \vdash \sigma_1 \triangleright \sigma_2 \qquad C \vdash \tau_1 \triangleright \tau_2}{C \vdash \sigma_1 * \tau_1 \triangleright \sigma_2 * \tau_2}$$

  The reflexivity rule can also be seen as a special case of the (DOWN) rules:

$$C \vdash \alpha \triangleright \alpha$$

- (TRANS):

$$\frac{\alpha \triangleright \zeta \in C \qquad C \vdash \zeta \triangleright \tau}{C \vdash \alpha \triangleright \tau}$$

  (and the symmetric rule), based on transitivity.

Verifying soundness of this simple algorithm is straightforward; all deductions made by the algorithm yield coercions belonging to the closure $\overline{C}$.

The (TRANS) rule is interesting: when a variable appears on one side of the goal, it is impossible to use a structural decomposition rule, hence it attempts to use transitivity. Here ζ can be any supertype of α in C, so the algorithm has to try them in sequence.

4.3.2 Making the algorithm reason by induction

How does this algorithm work when faced with the non-trivial example mentioned above: that is, when we ask it to decide whether $\{\alpha \triangleright F(\alpha), \beta = F(\beta)\} \Vdash \alpha \triangleright \beta$? Since both sides of the goal are variables, the algorithm must start by trying to apply rule (TRANS) twice. Here there is only one possible candidate for ζ; hence, the goal becomes $F(\alpha) \triangleright F(\beta)$. Now, since we have assumed constructor F to be covariant, the (DOWN) rule for F transforms the goal into $\alpha \triangleright \beta$. We have found the original goal again.

The simplest reaction would be to reject the goal, in order to prevent the algorithm from looping. But (intuitively speaking) since we have applied rule (DOWN) once before stumbling upon the original question, we have verified that "α and β are compatible at depth 1", that is, any instantiations of α and β have the same head constructor. If we chose to let the algorithm run a little longer before failing, we would be able to verify that they are compatible at any given finite depth k, i.e. that any instantiations of α and β have the same structure above depth k. Therefore, the conclusion $\alpha \triangleright \beta$ is sound.

Formalizing this idea, we define the following generalization of the algorithm:

- (AXIOM):

$$\frac{\tau_1 \triangleright \tau_2 \in C}{C, H_1, H_0 \vdash \tau_1 \triangleright \tau_2}$$

- (REC):

$$\frac{\tau_1 \triangleright \tau_2 \in H_1}{C, H_1, H_0 \vdash \tau_1 \triangleright \tau_2}$$

- (DOWN):

$$\frac{\forall i \in \{1 \ldots n\} \quad C, H_1 \cup H_0, \emptyset \vdash c_i}{C, H_1, H_0 \vdash \tau_1 \triangleright \tau_2}$$

- (TRANS):

$$\frac{\alpha \triangleright \zeta \in C \qquad C, H_1, H_0 \cup \{\alpha \triangleright \tau\} \vdash \zeta \triangleright \tau}{C, H_1, H_0 \vdash \alpha \triangleright \tau}$$

Here H_0 and H_1 are sets of coercions; they contain a history of the questions which have been asked since the start of the algorithm. Questions are accumulated into H_0 (see (TRANS) rule). When a (DOWN) rule is applied, the content of H_0 is moved into H_1. Hence, every coercion in H_1 is a question which has already been asked before applying the last (DOWN) rule; if we encounter this question again, we will be able to give it a positive answer. That's what the new (REC) rule is for. Initially, H_0 and H_1 are empty, so the algorithm's answer is positive if and only if $C, \emptyset, \emptyset \vdash \tau_1 \triangleright \tau_2$.

We have shown this extended algorithm to be sound with respect to the entailment relation, that is,

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

4.3.3 Reasoning with upper and lower bounds

The algorithm described above already yields acceptable results on basic cases (provided that suitable substitutions are used; see section 4.4). For instance, it is powerful enough to reduce the type of the `map` primitive to its natural, ML-like form (see section 5). However, we quickly realized that it is not complete with respect to $\Vdash$.

For instance, the following assertion holds for any type τ:

$$\{\alpha \triangleright \mathtt{Int}, \alpha \triangleright \mathtt{Bool}\} \Vdash \alpha \triangleright \tau$$

Indeed, let D be a coercion set such that $D \cup \{\alpha \triangleright \mathtt{Int}, \alpha \triangleright \mathtt{Bool}\}$ is consistent (i.e. the closure of this set is consistent). Suppose D contains a coercion of the form $\sigma \triangleright \alpha$. Then, the consistency hypothesis implies that coercions $\sigma \triangleright \mathtt{Int}$ and $\sigma \triangleright \mathtt{Bool}$ are both consistent. This forces σ to be a variable. Hence, coercion $\sigma \triangleright \tau$ is also consistent, and $D \cup \{\alpha \triangleright \tau\}$ is consistent. This proves the assertion.

However, the algorithm, when asked whether $\alpha \triangleright \tau$, uses the (TRANS) rule and will give a positive answer only if τ is `Int` or `Bool`. Hence, it is not complete.

One might formulate the objection that, even though this case can arise in practice, it is of marginal interest. Let us give a more crucial example. Consider the following assertion:

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

Verifying it is straightforward, using the definition of $\Vdash$. But the algorithm cannot prove it, because it can only use the (TRANS) rule. Hence, it will successively try to show

$$\alpha_1 * \beta_2 \triangleright \alpha_1 * \alpha_2$$

and

$$\alpha_1 * \beta_2 \triangleright \beta_1 * \beta_2$$

none of which holds. The (TRANS) rule alone is too restrictive, for it forces us to choose too early which hypothesis to use.

The reader might wonder why the algorithm does not identify γ with a product, say $\gamma_1 * \gamma_2$ (where γ_1 and γ_2 are fresh type variables), and break the assumptions into more elementary coercions. The algorithm would then have to decide whether

$$\{\alpha_1 \triangleright \gamma_1, \alpha_2 \triangleright \gamma_2, \beta_1 \triangleright \gamma_1, \beta_2 \triangleright \gamma_2\} \Vdash \{\alpha_1 \triangleright \gamma_1, \beta_2 \triangleright \gamma_2\}$$

which would immediately be shown to be true using the (AXIOM) rule. However, recall that we have decided not to "add structure" to the coercion set in such a way, because it would cause us to loop in the presence of recursive constraints.

The problem here seems to stem from the fact that, when asked to prove that a certain type τ is smaller than variable γ, the algorithm tries to prove that it is smaller than one of γ's lower bounds. Instead, it would be sufficient to prove that τ is smaller than *γ's greatest lower bound*. When asked whether $C \Vdash \tau \triangleright \gamma$, we could turn this goal into

$$C \Vdash \tau \triangleright \bigsqcup_i \gamma_i$$

where the γ_i's are γ's lower bounds. To do this, we need to allow upper bounds on the right side of goals, and symmetrically, lower bounds on the left side (coarsely speaking; the contravariance of arrow constructors makes things more complex). However, no lower or upper bounds are needed in the hypothesis set C, which helps keep things simple.

For now, we won't formalize the definition of upper and lower bounds; instead, let us show how they help solve the previous problem. The original goal is

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

This time, the (TRANS) rules have been replaced by the upper bound introduction rule. Hence the goal becomes

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

We give ourselves a set of rules to compute upper and lower bounds. In particular, they are distributive with respect to product, hence this goal can be rewritten as

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

Now, the usual (DOWN) rule breaks this into two sub-goals:

$$C \Vdash \alpha_1 \triangleright \alpha_1 \sqcup \beta_1$$

$$C \Vdash \beta_2 \triangleright \alpha_2 \sqcup \beta_2$$

both of which are accepted if we introduce $\alpha \triangleright \alpha \sqcup \beta$ as an axiom.

This idea also lifts the first limitation discovered above: when asked whether

$$C = \{\alpha \triangleright \texttt{Int}, \alpha \triangleright \texttt{Bool}\} \Vdash \alpha \triangleright \tau$$

we turn this goal into

$$C \Vdash \texttt{Int} \sqcap \texttt{Bool} \triangleright \tau$$

Since `Int` and `Bool` are incompatible atomic types, the lower bound evaluates to

$$C \Vdash \bot \triangleright \tau$$

which is a tautology. Hence the assertion is accepted.

There isn't room here to formalize this concept further, but the formalization exists and we have proven an extended algorithm to be sound with respect to entailment.

4.4 Choosing the right substitutions

Assuming that we are able to decide whether it is valid to apply a certain substitution σ to a type, it remains to choose σ itself. This can be a difficult problem; if we build substitutions using terms which exist in the coercion set, we obtain a finite number of substitutions, but it is huge nonetheless.

4.4.1 Multi-point substitutions are necessary

A seemingly reasonable restriction would be to only try one-point substitutions, i.e. substitutions of the form $[\gamma/\alpha]$ where α is a variable and γ is a term occurring in the coercion set. However, we find that this is too restrictive in some cases: it is sometimes necessary to substitute several variables *at once*. For instance, suppose we have inferred type

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

for a certain expression e. This type says that e evaluates to a list of α's, for any α which is a supertype of `Int`. It would be simpler, and equivalent, to just say that we have a list of integers; hence, we would like to apply substitution $[\texttt{Int}/\alpha]$. The substitution rule requires us to prove that

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

We find that

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

holds, thanks to the hypothesis $\texttt{Int} \triangleright \alpha$. However, the other inclusion:

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

doesn't.

This situation is easy to understand: variable γ, together with the equation $\gamma = [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma]$, is a mute variable, i.e. it has been introduced only to express a fixpoint and would not be necessary if the type language offered μ-binders. Indeed, if μ-binders were available, the type of e would read

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

Substitution $[\texttt{Int}/\alpha]$ would turn it into

$$\mu\gamma.[\texttt{Nil} \mid \texttt{Cons of } \texttt{Int} * \gamma]$$

In the latter type, γ represents a different fixpoint. This seems to indicate that in the language without μ-binders, the substitution should also have an effect on γ.

Consider again the original type inferred for e:

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

Let us introduce a fresh type variable γ', together with the following equation (an equation stands for two symmetrical inclusions):

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

Adding this equation to the inferred set of constraints yields an equivalent type, because the added equation is consistent and disconnected from the other constraints (see section 3). Now, instead of substitution $[\texttt{Int}/\alpha]$, let us try $[\texttt{Int}/\alpha, \gamma'/\gamma]$. This time, according to the substitution rule, all we have to show is:

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

which our algorithm (see section 4.3.2) is able to show using hypothesis $\texttt{Int} \triangleright \alpha$ and the (REC) rule.

We have now shown that the original type is equivalent to type

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

as expected. However, a substitution affecting two variables at once was needed. Our implementation of the typechecker has a heuristic to try detecting "mute variables" like γ and build multi-variable substitutions accordingly.

4.4.2 Turning coercions into equations

Let us write a `list_length` function which computes the length of a list:

```
rec list_length in function
  Nil -> 0
| Cons(_, rest) -> plus 1 (list_length rest)
;;
```

Running it through a typechecker might produce this type:

$$[\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma] \to \texttt{Int} \mid \{\gamma \triangleright [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma]\}$$

Obviously, this type means that the function accepts a list of elements of type α (α is implicitly universally quantified) and returns an integer. It seems natural to raise the following issue: why is γ constrained by a (one-way) coercion instead of a full equation? An equation would help us better understand γ's nature as a fixpoint; also, if this coercion can be replaced with an equation, then it means that the `list_length` function has the same type in our system as in ML.

Indeed, we are able to show that using a coercion or an equation here is indifferent. First, we introduce a fresh variable γ' together with the equation

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

Adding this constraint to the coercion set yields an equivalent type, because it is consistent and disconnected (see section 3). Now, we can attempt to apply substitution $[\gamma'/\gamma]$. The substitution rule requires us to show

$$C \Vdash [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma'] \to \texttt{Int} \triangleright [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma] \to \texttt{Int}$$

where $C = \{\gamma \triangleright [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma], \gamma' = [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma']\}$. Since γ' and γ appear in contravariant position here, this is equivalent to

$$C \Vdash \gamma \triangleright \gamma'$$

This assertion can easily be shown to be true (actually, it is a particular case of the non-trivial entailment example mentioned in section 4.2). Hence, the substitution is valid, and the `list_length` function also has type

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

which (using the equation) we can also write as

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

Neglecting the fact that γ has been renamed to γ', we have shown that a coercion or an equation have the same meaning.

Replacing coercions with equations can be important not only for readability, but also because it might allow further simplifications: substitutions which were previously invalid can become valid, since there is one more hypothesis in the constraint set. Therefore, we deem it useful in practice, and have implemented a heuristic to introduce equations in the same manner as above.

5 An example: the `map` function

We define `map` as follows:

```
rec map in function f -> function
  Nil -> Nil
| Cons(element, rest) -> Cons(f element, map f rest)
;;
```

The syntax used here is that of our prototype implementation of the typechecker.

5.1 Typechecking

5.1.1 A quick sketch of the typing process

The typechecker first encounters the `rec map in` construct and introduces a fresh type variable α for `map`. The inference rule for recursive constructs states that a constraint will be added on α after the body of the construct has been typed; see later on.

Then, it sees the λ-binder for `f` and introduces a fresh variable β for `f`.

It then starts building a type for the next function. This function uses pattern matching. The typechecker processes each pattern; as output, it produces the type of the pattern and the environment which should be used to type the result expression.

For the first pattern, the pattern type is $[\texttt{Nil}]$ and the result environment is $[\texttt{map}: \ \alpha; \texttt{f}: \ \beta]$. For the second pattern, the type is $[\texttt{Cons of } \gamma * \delta]$, where γ and δ are fresh variables, and the result environment is $[\texttt{map}: \ \alpha; \texttt{f}: \beta; \texttt{element}: \gamma; \texttt{rest}: \ \delta]$.

Then the algorithm computes a type for the result expressions. For the first one, it is simply $[\texttt{Nil}]$. For the second one, things are slightly more complex; each function application introduces a new type variable as well as a new type constraint. Subexpression `(f element)` has type ε and the constraint $\beta \triangleright \gamma \to \varepsilon$ is added. Subexpression `(map f)` has type φ where $\alpha \triangleright \beta \to \varphi$; and subexpression `(map f rest)` has type ψ where $\varphi \triangleright \delta \to \psi$. Finally the result type for the whole line is $[\texttt{Cons of } \varepsilon * \psi]$.

The rule for functions with pattern matching can now compute the type of the innermost function: $[\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho$, where ρ is a fresh type variable, together with the constraints $[\texttt{Nil}] \triangleright \rho$ and $[\texttt{Cons of } \varepsilon * \psi] \triangleright \rho$.

Finally, the recursive constraint on `map` mentioned above is added; it reads $\beta \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \alpha$.

5.1.2 The unsimplified type

To sum things up, the typechecker outputs the following type for the whole expression:

$$\beta \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho$$

together with the constraint set $\begin{cases} \beta \triangleright \gamma \to \varepsilon \\ \alpha \triangleright \beta \to \varphi \\ \varphi \triangleright \delta \to \psi \\ [\texttt{Nil}] \triangleright \rho \\ [\texttt{Cons of } \varepsilon * \psi] \triangleright \rho \\ \beta \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \alpha \end{cases}$

Also, for the sake of brevity, we have yet omitted to mention that after each step, the typechecker computes the closure of the constraint set (by transitivity and structural decomposition) and checks its consistency. Let us compute the closure of the above constraint set.

Transitivity on α gives

$$\beta \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \beta \to \varphi$$

which a structural propagation rule breaks into

$$[\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \varphi$$

and the trivial constraint $\beta \triangleright \beta$, which is dropped.

Then, transitivity on φ yields

$$[\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \delta \to \psi$$

which is in turn broken into

$$\rho \triangleright \psi$$

and

$$\delta \triangleright [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta]$$

The closure operation is now complete. All constraints are consistent so the term is found to be well-typed. Without any simplifications, the type of map is found to be:

$$\beta \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho$$

$$\text{where } \begin{cases} \beta \triangleright \gamma \to \varepsilon \\ \alpha \triangleright \beta \to \varphi \\ \varphi \triangleright \delta \to \psi \\ [\texttt{Nil}] \triangleright \rho \\ [\texttt{Cons of } \varepsilon * \psi] \triangleright \rho \\ \beta \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \alpha \\ [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \varphi \\ \rho \triangleright \psi \\ \delta \triangleright [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \end{cases}$$

This type is obviously much too complex to be readily understood by a human being. Yet it is the type of a simple, well-known primitive function. Hence it is necessary to simplify it by showing that it is equivalent to a smaller type. Actually we will show that this type is, in a certain sense, equivalent to the usual ML type for map.

5.2 Simplifying the inferred type

Only the substitutions and their results are shown here. Verifications are left to the reader. Recall that the substitution rule reads

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

The algorithm described in section 4.3.2 is powerful enough to verify that the substitutions applied below are valid. Hence, the full power of $\Vdash$ is not needed in the case of map. However, the (REC) rule is central to many proofs; therefore, the definition of entailment used in [4] would not suffice to effect these reductions.

First, apply $[\delta \to \psi / \varphi]$. The coercion set becomes

$$\begin{cases} \beta \triangleright \gamma \to \varepsilon \\ \alpha \triangleright \beta \to \delta \to \psi \\ [\texttt{Nil}] \triangleright \rho \\ [\texttt{Cons of } \varepsilon * \psi] \triangleright \rho \\ \beta \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho \triangleright \alpha \\ \rho \triangleright \psi \\ \delta \triangleright [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \end{cases}$$

Next, apply $[\beta \to \delta \to \psi / \alpha]$. We obtain

$$\begin{cases} \beta \triangleright \gamma \to \varepsilon \\ [\texttt{Nil}] \triangleright \rho \\ [\texttt{Cons of } \varepsilon * \psi] \triangleright \rho \\ \rho \triangleright \psi \\ \delta \triangleright [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \end{cases}$$

Applying $[\gamma \to \varepsilon / \beta]$ then yields type:

$$(\gamma \to \varepsilon) \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \to \rho$$

$$\text{where } \begin{cases} [\texttt{Nil}] \triangleright \rho \\ [\texttt{Cons of } \varepsilon * \psi] \triangleright \rho \\ \rho \triangleright \psi \\ \delta \triangleright [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta] \end{cases}$$

Using $[\rho / \psi]$, we reduce the coercion set to

$$\left\{\begin{array}{l}[\texttt{Nil}] \rhd \rho \\ [\texttt{Cons of } \varepsilon * \rho] \rhd \rho \\ \delta \rhd [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta]\end{array}\right.$$

Now, as explained in section 4.4.2, we turn the coercion concerning δ into an equation. This is done by introducing a fresh variable, δ', together with the equation

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

Now, it is possible to apply substitution $[\delta'/\delta]$. Hence the type becomes

$$(\gamma \to \varepsilon) \to [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta'] \to \rho$$

$$\text{where} \left\{\begin{array}{l}[\texttt{Nil}] \rhd \rho \\ [\texttt{Cons of } \varepsilon * \rho] \rhd \rho \\ \delta' = [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta']\end{array}\right.$$

Note that the body of the type can now be reduced to $(\gamma \to \varepsilon) \to \delta' \to \rho$.

In the same way, we can introduce a fresh variable ρ' together with the equation

$$\rho' = [\texttt{Nil} \mid \texttt{Cons of } \varepsilon * \rho']$$

and apply substitution $[\rho'/\rho]$. This yields

$$(\gamma \to \varepsilon) \to \delta' \to \rho'$$

$$\text{where} \left\{\begin{array}{l}\rho' = [\texttt{Nil} \mid \texttt{Cons of } \varepsilon * \rho'] \\ \delta' = [\texttt{Nil} \mid \texttt{Cons of } \gamma * \delta']\end{array}\right.$$

This is, in a sense, equivalent to the usual ML type for `map`. Indeed, the two equations above can be read as meaning $\delta' = \gamma\ \texttt{list}$ and $\rho' = \varepsilon\ \texttt{list}$. The type itself then reads

$$(\gamma \to \varepsilon) \to \gamma\ \texttt{list} \to \varepsilon\ \texttt{list}$$

One must not forget, however, that the type system will generate other inclusion constraints when `map` is applied. As a result, `(map f)` can be fed a heterogeneous list, provided that the types of its elements are subtypes of `f`'s argument type. Even though `map`'s type reads the same as its ML equivalent, it has more intrinsic power.

6 A more elaborate example: `quicksort`

Here is the code for a more complex function, an implementation of the QuickSort algorithm. It is written in a slightly awkward way to circumvent some limitations of our implementation: functions accept only one argument, and `let` constructs don't incorporate pattern matching.

```
let concat = rec concat in function
  Nil ->
    (function list2 -> list2)
| Cons(one, rest) ->
    (function list2 -> Cons(one, concat rest list2))

in let split = rec split in
  function compare -> function pivot -> function
    Nil ->
      (Nil, Nil)
  | Cons(one, rest) ->
      (function (smaller, greater) -> if compare one pivot
         then (Cons(one, smaller), greater)
```

```
      else (smaller, Cons(one, greater)))
      (split compare pivot rest)

in let quicksort = rec quicksort in
  function compare -> function
    Nil ->
      Nil
  | Cons(pivot, list) ->
      (function (smaller, greater) ->
        concat (quicksort compare smaller)
               (Cons(pivot, quicksort compare greater)))

      (split compare pivot list)

in quicksort
;;
```

With no simplifications at all, the inferred type contains 300 coercions, totalling 100 type variables. This number might seem huge, as there aren't 300 function applications in the above code; it is explained by the fact that `let` constructs duplicate coercions. If we remove disconnected constraints (but perform no substitutions), 60 coercions, totalling 20 variables, are left. Needless to say, this complexity makes the inferred type unreadable.

Our implementation of the typechecker is somewhat behind schedule and cannot yet simplify this type completely. However, the theoretical results described in this paper are sufficient to reduce the type automatically.

We finished the job by hand and obtained the following type for `quicksort`:

$$(\alpha \to \beta \to \texttt{Bool}) \to [\texttt{Nil} \mid \texttt{Cons of } \beta * \gamma] \to \delta$$

where $\begin{cases} \delta = [\texttt{Nil} \mid \texttt{Cons of } \beta * \delta] \\ \gamma = [\texttt{Nil} \mid \texttt{Cons of } \alpha * \gamma] \\ \alpha \triangleright \beta \end{cases}$

Note 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 usual ML type. If one looks at the code above, one will notice that it always uses the first element of the list as pivot, and the pivot is always passed as second argument to the comparison function. This allows for a small generalization of the type. For instance, let's assume for a second that coercion $\texttt{Int} \triangleright \texttt{Float}$ is valid in our system. Suppose our comparison function has type $\texttt{Int} \to \texttt{Float} \to \texttt{Bool}$, that is, it knows how to compare an integer with a floating point number (but not two floating point numbers). Then we are still able to sort a list whose first element only is real, for instance $[1.3; 2; 4]$.

This is of no practical interest, of course, but it was amusing to see that the inferred type was, 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.

7 Conclusion

We have evidenced that the constraint-based type inference system described in [4] cannot be used in practice without performing type simplification: large coercion sets need to be reduced to smaller, equivalent ones. We give two main methods to simplify coercion sets: the first one consists in eliminating disconnected constraints and is well understood. The second one consists in applying substitutions to lower the number of type variables. We have given a rule to check the validity of a substitution which we believe to be very powerful. This rule relies on an *entailment* relation ($\Vdash$) between coercion sets; deciding this relation – if it is at all decidable – seems to be a difficult problem. At present we have devised a powerful algorithm and proven it to be sound with respect to entailment. The issue of completeness is still open at the time of this writing.

If we are able to decide relation $\Vdash$, then we hope to be able to simplify inferred types completely, that is, to reach a point where the remaining coercions are necessary to express the subtyping properties of the program. Obtaining an efficient implementation also seems within reach. Attaining these goals would show that type inference using inclusion constraints is indeed practical.

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] 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.

[3] 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.

[4] 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`.

[5] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

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

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

[8] 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.