# A Framework for Type Inference with Subtyping

François Pottier*
Francois.Pottier@inria.fr

## Introduction

In type systems based on subtyping, type equality is replaced with subtyping, which is a less restrictive relationship. The idea is, if $\tau_1$ is a subtype of $\tau_2$, then a value of type $\tau_1$ can be transparently supplied wherever a value of type $\tau_2$ is expected.

Subtyping has been used as a key concept to create formal type systems for object-oriented languages. These systems often require the programs to be annotated with user-supplied type information. Being able to omit this information—or at least part of it—provides the programmer with a greater degree of freedom; hence, the desire arises to do type inference in the presence of subtyping.

This issue has been extensively studied in the past few years. Many type systems have been proposed, with varying degrees of richness and complexity. Possible features are the existence of a least type $\bot$ and a greatest type $\top$, the presence of contravariant type constructors such as $\rightarrow$, the existence of union and intersection types, the existence of recursive types, the ability for the user to extend the type language through generative type declarations, etc. Virtually all of these systems base their type inference algorithms upon the same principle. Each function application node in the program generates a subtyping constraint, which requires that the actual argument's type be a subtype of the function parameter's type. Type inference consists in gathering these constraints and checking that they admit a solution.

The type system we present here is quite general. It has a fixed type language, which is the set of regular types formed with $\bot$, $\top$ and $\rightarrow$. It is not as powerful as that of [4, 5], which has much more general union and intersection types; still, it is general enough to easily support the addition of a wide class of type constructs, such as extensible records and variants. Type inference is done as explained above; determining whether a conjunction of constraints admits a solution is done through a *closure* computation [7]. In theory, the issue is settled. In practice, however, things only begin here. The number of constraints accumulated by type inference is large (at best, linear in the program size; at worst, exponential, because `let` constructs duplicate them). This slows down type inference (the closure algorithm takes time cubic in the number of type variables) and makes types illegible (constraints are part of the type information given to the user).

Therefore, algorithms are needed to simplify sets of subtyping constraints, without affecting their meaning. In [14], we introduced two such methods. One removed so-called unreachable constraints; the other used heuristics to come up with substitutions which could be applied to the inferred type schemes without lessening their power. Smith and Trifonov [18] refine the former into a concept called *garbage collection*. Besides, they describe a process called *canonization*, which consists in rewriting a type scheme so that each variable has at most one constructed lower (resp. upper) bound. Other substitution methods can be found in [1].

So, at this point, various simplification methods are known, some of which are very effective, such as garbage collection. However, these are not sufficient to obtain an efficient, well-integrated type inference algorithm; several problems remain. First, substitution heuristics are inherently inefficient. Many possibilities have to be tried out, and each try is costly, since it involves proving that the substitution is legal. We solve this problem by eliminating heuristics altogether and replacing them with a very efficient *minimization* algorithm, a close cousin to the "Hopcroft" algorithm introduced in [9]. Second, although garbage collection, as described by Smith and Trifonov, works well, it does not preserve the closure property. This is a problem, since we would like the type inference algorithm to work with closed constraint sets at all times, so it can do incremental closure computations. We solve it by showing that if the type inference rules are properly formulated, then no *bipolar* type variables are generated, which ensures that garbage collection preserves closure. Third, we give a precise formal description of the *canonization* algorithm, and we combine it with garbage collection, which makes it more efficient. We also show that a natural generalization of this algorithm can be used, if desired, to eliminate bipolar variables. Fourth and finally, we draw a distinction between *internal* and *external* simplification methods. The former help efficiency, and can be used throughout the type inference process; the latter help readability, and must be used only when submitting type information to the user. The former conflict with the latter, which is why the distinction is important; trying to achieve efficiency and readability at the same time is a design mistake.

*François Pottier, Projet Cristal, INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France. Phone: +33 1 39 63 53 16. Fax: +33 1 39 63 56 84.

This paper appeared at the International Conference on Functional Programming, Baltimore, September 1998.

To sum up, we describe a full framework for type inference and simplification in the presence of subtyping. It consists of a few key components: a set of type inference rules, designed to preserve a couple of desirable invariants, some internal simplification methods (canonization, garbage collection and minimization), and some (classic) external methods. Each of them integrates smoothly into the whole. As a result, the system is theoretically simple and leads to an efficient implementation.

This paper is organized as follows. Section 1 describes the type system, while section 2 recalls how to compute polarities and use them to do garbage collection. Section 3 shows that it is enough (and best) to work with "small" terms. Section 4 introduces canonization and combines it with garbage collection to gain efficiency. Section 5 explains why and how to eliminate bipolar variables. Section 6 talks about the difference between internal and external strategies. Section 7 defines the minimization algorithm. Section 8 describes the implementation and gives some performance figures. Section 9 discusses the remaining problems. Finally, related work is surveyed in section 10. An appendix shows the type inference engine at work on a typical example. Proofs, as well as several non-essential definitions, were omitted by lack of space; all can be found in [15].

Sections 3 to 7 discuss separate points and are largely independent from one another.

## 1 The setting

The type system described in this section is a close cousin to Smith and Trifonov's [18]. Our type inference rules have been modified so as to preserve some additional invariants, discussed in sections 3 and 5. Also, our definition of constraint graphs allows occurrences of $\sqcup$ and $\sqcap$ at certain positions in types, as a purely syntactic convenience.

The language under study is a $\lambda$-calculus with `let`, where $\lambda$-bound and `let`-bound identifiers form two distinct syntactic categories.

The set of *ground types* is the set of regular trees built over $\bot$, $\top$ (both with arity 0) and $\rightarrow$ (with arity 2). A partial order on ground types, called *subtyping*, is defined in the classic manner.

**Definition 1.1** *Subtyping is the greatest relation such that $\tau \leq \tau'$ holds iff at least one of the following is true:*

- $\tau = \bot$.
- $\tau' = \top$.
- $\exists \tau_0 \tau_1 \tau_0' \tau_1' \quad \tau = \tau_0 \rightarrow \tau_1,\ \tau' = \tau_0' \rightarrow \tau_1',\ \tau_0' \leq \tau_0$ *and* $\tau_1 \leq \tau_1'$.

Equipped with this ordering, the set of ground types becomes a lattice. We denote its least upper bound and greatest lower bound operators by $\sqcup$ and $\sqcap$, respectively. As in any lattice, $\sqcup$ and $\sqcap$ are associative and commutative. In addition, they are fully characterized as follows:

**Proposition 1.1** *The following are identities:*

$$\bot \sqcup \tau = \tau \qquad \bot \sqcap \tau = \bot$$
$$\top \sqcup \tau = \top \qquad \top \sqcap \tau = \tau$$
$$(\tau_1 \rightarrow \tau_2) \sqcup (\tau_1' \rightarrow \tau_2') = (\tau_1 \sqcap \tau_1') \rightarrow (\tau_2 \sqcup \tau_2')$$
$$(\tau_1 \rightarrow \tau_2) \sqcap (\tau_1' \rightarrow \tau_2') = (\tau_1 \sqcup \tau_1') \rightarrow (\tau_2 \sqcap \tau_2')$$

The definition of *types* resembles that of ground types; however, types are finite terms, and they may contain *type variables*. Furthermore, we shall permit $\sqcup$ and $\sqcap$ constructs to appear at specific positions in types. This allows encoding multiple bounds into a single one, e.g. by writing $(\tau \leq \alpha \wedge \tau' \leq \alpha)$ as $(\tau \sqcup \tau') \leq \alpha$. This decision affects only the syntax of our constraint language, not its power; it shall mainly allow a simpler description of canonization. Thus, we define two kinds of types, *pos-types* and *neg-types*:

$$\tau^+ ::= \alpha \mid \bot \mid \top \mid \tau^- \rightarrow \tau^+ \mid \sqcup\{\tau^+, \ldots, \tau^+\}$$
$$\tau^- ::= \alpha \mid \bot \mid \top \mid \tau^+ \rightarrow \tau^- \mid \sqcap\{\tau^-, \ldots, \tau^-\}$$

(Note that $\sqcup$ (resp. $\sqcap$) has two distinct meanings; it is an operation on ground types and a type constructor.) Types are considered equal modulo the equations of proposition 1.1. Types have *normal forms* such that whenever $\sqcup$ or $\sqcap$ appears with $n$ arguments, then $n$ is at least 2, at most one argument is not a type variable, and if such an argument exists, then its head constructor is $\rightarrow$.

A type is said to be *constructed* iff it is neither a variable nor a $\sqcup$- or $\sqcap$-construct.

To keep track of the subtyping constraints created by analyzing an expression, we introduce *constraint graphs.* A constraint graph represents a conjunction of constraints, but allows only one constructed bound per type variable. (This requirement is made possible by the availability of $\sqcup$ and $\sqcap$ constructs.) A *constraint graph $C$* is made up of a relation between type variables, denoted by $\leq_C$, and two maps $C^\downarrow$ and $C^\uparrow$, which map each $\alpha \in V$ to a its lower bound (a constructed pos-type) and to its upper bound (a constructed neg-type), respectively.

A ground substitution $\rho$ is a *solution* of $C$ (which we denote by $\rho \vdash C$) iff the following conditions hold:

$$\forall \alpha\beta \in V \quad \alpha \leq_C \beta \Rightarrow \rho(\alpha) \leq \rho(\beta)$$
$$\forall \alpha \in V \quad \rho(C^\downarrow(\alpha)) \leq \rho(\alpha) \leq \rho(C^\uparrow(\alpha))$$

To verify that a constraint graph admits a solution, one needs to put it in some kind of *solved* form, off which a solution can be read straightforwardly. There exist several notions of solved form. The simplest, which is also the strongest, we call *closure*; it consists in requiring that the constraint graph be closed by transitivity on variables and by structural decomposition. It was originally introduced in [7][1]. Its advantage is that there exists a simple procedure which turns any constraint graph into an equivalent closed graph, if one exists. This allows deciding whether any given constraint graph has a solution. The weakest known notion of solved form, which we call *weak closure*, is similar in principle, but is defined using a notion of provable entailment. It was introduced in [18][2]. Its advantage is that it imposes less drastic requirements on the constraint graph, which is useful in some delicate proofs, like that of Smith and Trifonov's subsumption algorithm. Here, we need to introduce a third notion, called *simple closure*, which is intermediate between weak closure and closure. It is used to formalize the canonization process; it is introduced in detail in section 4.1.

A *context* $A = \langle x : \tau_x \rangle_{x \in I}$ is a finite map from $\lambda$-identifiers to neg-types. A *type scheme* $\sigma = A \Rightarrow \tau \mid C$ is a triple of a context $A$, a pos-type $\tau$, and a constraint graph $C$. All type variables in a type scheme should be understood

[1] where it is called *closed and consistent.*

[2] where it is called *consistent and canonical.*

as universally quantified. However, we do not introduce any quantifiers, and we deal with $\alpha$-conversion explicitly.

The subtyping relation on ground types is extended to a so-called *subsumption* relation on type schemes. One says that $\sigma_1$ is *subsumed* by $\sigma_2$, and one writes $\sigma_1 \leq^\forall \sigma_2$, iff for any ground instance of $\sigma_2$, there exists a ground instance of $\sigma_1$ which is smaller. This can be written

$$A_1 \Rightarrow \tau_1 \mid C_1 \leq^\forall A_2 \Rightarrow \tau_2 \mid C_2 \quad \text{iff}$$
$$\forall \rho_2 \vdash C_2 \quad \exists \rho_1 \vdash C_1 \quad \rho_1(A_1 \Rightarrow \tau_1) \leq \rho_2(A_2 \Rightarrow \tau_2)$$

*Equivalence* of type schemes, denoted by $=^\forall$, is defined as subsumption in both directions.

The typing rules are given in figure 1. They are essentially identical to those given in [18], only slightly simplified. Note that environments contain only `let`-identifiers, while $\lambda$-identifiers appear in the inferred contexts.

Introducing contexts in type schemes and in the typing rules is not required by the advent of subtyping. Rather, this technique, called $\lambda$-lifting, is a technical trick which allows us to deal solely with closed (i.e. fully quantified) type schemes, and thus, to define scheme subsumption without taking environments into account. Since subsumption is at the very core of the theory, the whole system is made significantly simpler. Concerning subsumption, the presence of a context does not add any complexity. It behaves essentially as a record type to the left of an arrow type, hence the arrow notation $A \Rightarrow \tau \mid C$.

These typing rules are not syntax-directed, because rule (Sub) can be applied at any time. Furthermore, rule (App) places sharing constraints on its premises. In figure 2, we present a set of type inference rules, which do not have these problems, and thus can be directly turned into an algorithm. One shows that they are correct and complete with respect to the typing rules. That is, they find a typing if and only if one exists, and in that case, it is a most general typing with respect to subsumption.

A type inference judgment is of the form $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma$. $F$ and $F'$ are sets of type variables, used to ensure that any newly introduced type variable is "fresh"; thus, two unrelated branches of a type inference tree are guaranteed to produce type schemes which share no variables. These annotations should otherwise be ignored.

Our type inference rules differ from those of [18] in several aspects. Modifications have been made to ensure that they comply with the small terms invariant (see section 3) and with the mono-polarity invariant (see section 5). Incidentally, these rules do describe an algorithm, since no sharing constraints remain, and fresh variables are handled explicitly.

Note that both sets of rules implicitly require all constraint graphs to have a solution. This condition is enforced in the implementation by working with closed constraint graphs at all times. The $+$ operation, which adds a constraint to a constraint graph, performs an incremental closure computation in practice.

## 2 Polarities and garbage collection

Trifonov and Smith [18] explain how to annotate each type variable in a type scheme with its *polarity*, and use this notion to define *garbage collection*. We recall their definitions here.

If $\sigma = A \Rightarrow \tau \mid C$ is a type scheme, where $C$ is a weakly closed constraint graph, then the sets $\mathrm{dom}^+(\sigma)$ and $\mathrm{dom}^-(\sigma)$ of positive (resp. negative) variables of $\sigma$ are the smallest sets such that for any $\epsilon \in \{-,+\}$,

- $\mathrm{fv}^\epsilon(\tau) \subset \mathrm{dom}^\epsilon(\sigma)$;
- $\forall x \in \mathrm{dom}(A) \quad \mathrm{fv}^\epsilon(A(x)) \subset \mathrm{dom}^{\bar\epsilon}(\sigma)$;
- $\forall \alpha \in \mathrm{dom}^+(\sigma) \quad \mathrm{fv}^\epsilon(C^\downarrow(\alpha)) \subset \mathrm{dom}^\epsilon(\sigma)$;
- $\forall \alpha \in \mathrm{dom}^-(\sigma) \quad \mathrm{fv}^\epsilon(C^\uparrow(\alpha)) \subset \mathrm{dom}^{\bar\epsilon}(\sigma)$.

The *polarity* of a variable $\alpha$ is defined as $\{\epsilon \in \{-,+\}\,;\, \alpha \in \mathrm{dom}^\epsilon(\sigma)\}$. Informally speaking, the $+$ (resp. $-$) sign indicates that $\alpha$ might receive a new upper (resp. lower) bound in the future. Polarities carry information about the direction of the data flow: positive (resp. negative) variables represent an output (resp. input) of the code being typed. Note that a variable can also carry both signs at once (we call such a variable *bipolar*), or no sign at all (we call such a variable *neutral*).

Garbage collection uses this information to throw away meaningless constraints. A constraint is only useful if it actually *constrains* the type scheme, i.e. if it can potentially cause a future closure computation to fail. If $\leq_C$ is transitive, then $\mathrm{GC}(\sigma)$ is defined as $A \Rightarrow \tau \mid D$, where

- $\alpha \leq_D \beta$ iff $\alpha \leq_C \beta$, $\alpha \in \mathrm{dom}^-(\sigma)$ and $\beta \in \mathrm{dom}^+(\sigma)$;
- $D^\downarrow(\alpha)$ is $C^\downarrow(\alpha)$ when $\alpha \in \mathrm{dom}^+(\sigma)$, and $\bot$ otherwise;
- $D^\uparrow(\alpha)$ is $C^\uparrow(\alpha)$ when $\alpha \in \mathrm{dom}^-(\sigma)$, and $\top$ otherwise.

In section 5, we show that it is possible to prohibit bipolar variables; this gives better theoretical properties to garbage collection. The minimization algorithm described in section 7 also makes fundamental use of polarities.

## 3 Work with small terms

We are now done recalling definitions, and can start discussing this paper's contributions. In this section, we show that it is possible to enforce quite drastic restrictions on the height and shape of the type terms we manipulate. In addition to simplifying the theory, these restrictions are actually beneficial from an implementor's point of view.

A type term is said to be a *leaf term* iff it has height 0 and it is not constructed; equivalently, iff it is a combination using $\sqcup$ or $\sqcap$ of one or more type variables. A type term is said to be a *small term* iff it it is constructed and its subterms are leaf terms. A type scheme $A \Rightarrow \tau \mid C$ is said to *verify the small terms invariant* iff every $A(x)$ is a leaf term, $\tau$ is a leaf term, and for each $\alpha$, $C^\downarrow(\alpha)$ and $C^\uparrow(\alpha)$ are small terms.

It is quite easy to show that any type scheme admits an equivalent form which verifies the small terms invariant. Whenever a type term violates the invariant, it suffices to break it down by introducing fresh variables (together with appropriate subtype inequations) to stand for its sub-terms. Actually, this is not even necessary in practice, because our type inference rules are designed to always produce compliant type schemes. In the typing rules, (Abs) is the only rule which builds up new terms. So, rules (Abs$_\mathrm{I}$) and (Abs'$_\mathrm{I}$) introduce one extra fresh variable, namely $\alpha$, to avoid breaking the small terms invariant.

This invariant simplifies both theory and implementation of subsequent developments. Additionally, it helps simplify inferred type schemes. Indeed, large terms no longer

$$\frac{A(x) = \tau}{\Gamma \vdash x : A \Rightarrow \tau \mid C} \ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : (A + [x \mapsto \tau]) \Rightarrow \tau' \mid C}{\Gamma \vdash \lambda x.e : A \Rightarrow \tau \to \tau' \mid C} \ (\textsc{Abs})$$

$$\frac{\Gamma \vdash e_1 : A \Rightarrow \tau_2 \to \tau \mid C \qquad \Gamma \vdash e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash e_1\, e_2 : A \Rightarrow \tau \mid C} \ (\textsc{App}) \qquad \frac{\Gamma(X) = \sigma}{\Gamma \vdash X : \sigma} \ (\textsc{LetVar})$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma + [X \mapsto \sigma_1] \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let}\ X = e_1\ \texttt{in}\ e_2 : \sigma_2} \ (\textsc{Let}) \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma \leq^{\forall} \sigma'}{\Gamma \vdash e : \sigma'} \ (\textsc{Sub})$$

Figure 1: Typing rules

$$\frac{\alpha, \beta \notin F}{[F]\ \Gamma \vdash_{\mathrm{I}} x : [F \cup \{\alpha, \beta\}]\ \langle x : \alpha \rangle \Rightarrow \beta \mid \varnothing + (\alpha \leq \beta)} \qquad (\textsc{Var}_{\mathrm{I}})$$

$$\frac{[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ A \Rightarrow \tau' \mid C \qquad A(x) = \tau \qquad \alpha \notin F'}{[F]\ \Gamma \vdash_{\mathrm{I}} \lambda x.e : [F' \cup \{\alpha\}]\ (A \setminus x) \Rightarrow \alpha \mid C + (\tau \to \tau' \leq \alpha)} \qquad (\textsc{Abs}_{\mathrm{I}})$$

$$\frac{[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ A \Rightarrow \tau' \mid C \qquad x \notin \mathrm{dom}(A) \qquad \alpha, \beta \notin F'}{[F]\ \Gamma \vdash_{\mathrm{I}} \lambda x.e : [F' \cup \{\alpha, \beta\}]\ A \Rightarrow \alpha \mid C + (\beta \to \tau' \leq \alpha)} \qquad (\textsc{Abs'}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\ \Gamma \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2 \qquad \alpha, \beta \notin F''\end{array}}{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F'' \cup \{\alpha, \beta\}]\ (A_1 \sqcap A_2) \Rightarrow \beta \mid C \\ \text{where } C = (C_1 \cup C_2) + (\alpha \leq \beta) + (\tau_1 \leq \tau_2 \to \alpha)\end{array}} \qquad (\textsc{App}_{\mathrm{I}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renaming of } \sigma \qquad \mathrm{rng}(\rho) \cap F = \varnothing}{[F]\ \Gamma \vdash_{\mathrm{I}} X : [F \cup \mathrm{rng}(\rho)]\ \rho(\sigma)} \qquad (\textsc{LetVar}_{\mathrm{I}})$$

$$\frac{[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ \sigma_1 \qquad [F]\ \Gamma + [X \mapsto \sigma_1] \vdash_{\mathrm{I}} e_2 : [F'']\ \sigma_2}{[F]\ \Gamma \vdash_{\mathrm{I}} \texttt{let}\ X = e_1\ \texttt{in}\ e_2 : [F'']\ \sigma_2} \qquad (\textsc{Let}_{\mathrm{I}})$$

Figure 2: Type inference rules

exist—informally speaking, each sub-term of a large term is labeled with its own type variable. Thus, the minimization algorithm (see section 7), whose occupation is to merge variables, is able to find new opportunities—which essentially means *share* sub-terms of large terms.

The idea of working solely with small terms is not new, at least from a theoretical point of view. It is to be found, for instance, in the theory of unification [12], where problems are presented as sets of multi-equations of depth 1 at most, rather than as equations between arbitrary terms. Among works more closely related to ours, those of Aiken and Wimmers [2] and of Palsberg [13] use a similar convention. However, it is often a mere theoretical convenience. Here, the invariant shall also help improve sharing between nodes, through the minimization algorithm, and is thus essential from an implementor's point of view.

## 4 Canonization

When dealing with arbitrary sets of constraints, it quickly appears desirable that each type variable have exactly one lower (resp. upper) constructed bound. There are two main reasons for this. First, combining several bounds into a single one is a simplification *per se*, because some structure is shared (or even eliminated). Second, this property is essential in the design of some simplification algorithms, such as minimization (see section 7).

To this end, we have introduced $\sqcup$ and $\sqcap$ constructs in type expressions. The closure algorithm can easily handle them, and they are a cheap way of enforcing uniqueness of the constructed bound. However, in doing so, we have made the type language more complex, and these new constructs cannot easily be dealt with by some algorithms (again, minimization).

So, we now wish to eliminate them, in a process called *canonization*. The small terms invariant ensures that, once types have been put in normal form, arguments of $\sqcup$ and $\sqcap$ can only be type variables. To do away with these constructs, we can use the same principle as in section 3: adding fresh variables with appropriate constraints. An occurrence of, say, $\alpha \sqcup \beta$ can be replaced with a fresh variable $\gamma$, provided $\alpha \leq \gamma$ and $\beta \leq \gamma$ are added.

$$
\begin{aligned}
r^+(\alpha) &= \alpha \quad \text{when } \{\alpha\} \notin \mathcal{E} \\
r^+(\sqcup S) &= \lambda_S \text{ when } S \in \mathcal{E} \\
r^+(\bot) &= \bot \\
r^+(\top) &= \top \\
r^+(\tau_0 \to \tau_1) &= r^-(\tau_0) \to r^+(\tau_1)
\end{aligned}
\qquad
\begin{aligned}
r^-(\alpha) &= \alpha \quad \text{when } \{\alpha\} \notin \mathcal{E} \\
r^-(\sqcap S) &= \gamma_S \text{ when } S \in \mathcal{E} \\
r^-(\bot) &= \bot \\
r^-(\top) &= \top \\
r^-(\tau_0 \to \tau_1) &= r^+(\tau_0) \to r^-(\tau_1)
\end{aligned}
$$

Figure 3: Definition of the rewriting functions

$$
\begin{array}{lll}
D^\downarrow(\alpha) = r^+(C^\downarrow(\alpha)) & D^\uparrow(\alpha) = r^-(C^\uparrow(\alpha)) & D^\downarrow(\gamma_S) = r^+(\bigsqcup_{\alpha \leq_C \sqcap S} C^\downarrow(\alpha)) \\
D^\uparrow(\gamma_S) = r^-(\bigsqcap_{\alpha \in S} C^\uparrow(\alpha)) & D^\downarrow(\lambda_S) = r^+(\bigsqcup_{\alpha \in S} C^\downarrow(\alpha)) & D^\uparrow(\lambda_S) = r^-(\bigsqcap_{\sqcup S \leq_C \alpha} C^\uparrow(\alpha))
\end{array}
$$

Figure 4: Definition of $D^\downarrow$ and $D^\uparrow$

$$
\begin{array}{rcll}
\alpha & \leq_D & \beta & \text{when } \alpha \leq_C \beta \\
\gamma_S & \leq_D & \alpha & \text{when } \alpha \in S \\
\alpha & \leq_D & \lambda_S & \text{when } \alpha \in S \\
r^-(\sqcap S) & \leq_D & \gamma_T & \text{when } |S| \geq 1 \text{ and } \sqcap S \leq_C \sqcap T \\
\lambda_S & \leq_D & r^+(\sqcup T) & \text{when } |T| \geq 1 \text{ and } \sqcup S \leq_C \sqcup T \\
r^+(\sqcup S) & \leq_D & r^-(\sqcap T) & \text{when } |S| \geq 1,\ |T| \geq 1 \text{ and } \sqcup S \leq_C \sqcap T
\end{array}
$$

Figure 5: Definition of $\leq_D$, modulo transitive closure

This process was first introduced by Smith and Trifonov [18]. We improve their results in several ways. First, we give a more precise formal description of it, and prove that it preserves *simple closure*. This allows running the garbage collection algorithm on the output type scheme. We simulate its execution and show that many of the constraints generated by canonization are then dropped. We can thus define an efficient combined algorithm, which avoids generating any constraints which would provably be dropped immediately afterwards by garbage collection. Second, we show that the same algorithm can be used to eliminate bipolar variables (more about this in section 5).

### 4.1 Simple closure

As explained above, we wish to run the type scheme output by canonization through garbage collection, which requires it to be at least *weakly closed*. This notion, defined using provable entailment, is rather complex, so we prefer to use something coarser, but simpler. Plain *closure* is too crude: we cannot, in general, require that the output constraint graph be closed. So, we define an intermediate notion, called *simple closure*.

**Definition 4.1** *Let $C$ be a constraint graph. The relation $\leq_C$ is extended to leaf terms by*

$$
\begin{array}{lcl}
\alpha \leq_C \sqcup V & \iff & \exists \beta \in V \quad \alpha \leq_C \beta \\
\sqcap V \leq_C \alpha & \iff & \exists \beta \in V \quad \beta \leq_C \alpha \\
\sqcup V \leq_C \tau & \iff & \forall \beta \in V \quad \beta \leq_C \tau \\
\tau \leq_C \sqcap V & \iff & \forall \beta \in V \quad \tau \leq_C \beta
\end{array}
$$

*Then, $\leq_C$ is straightforwardly extended to small terms, as follows:*

$$\bot \leq_C \tau \leq_C \top$$

$$\tau_0 \to \tau_1 \leq_C \tau_0' \to \tau_1' \iff \tau_0' \leq_C \tau_0 \land \tau_1 \leq_C \tau_1'$$

**Definition 4.2** *A constraint graph $C$ of domain $V$ is* simply closed *iff for all $\alpha, \beta \in V$,*

- *$\leq_C$ is transitive on type variables;*
- *$\alpha \leq_C \beta$ implies $C^\downarrow(\alpha) \leq_C C^\downarrow(\beta) \land C^\uparrow(\alpha) \leq_C C^\uparrow(\beta)$;*
- *$C^\downarrow(\alpha) \leq_C C^\uparrow(\alpha)$.*

*A type scheme $\sigma = A \Rightarrow \tau \mid C$ is said to be simply closed iff $C$ is.*

### 4.2 Canonization

**Theorem 4.1** *Let $\sigma = A \Rightarrow \tau \mid C$ be a simply closed type scheme of domain $V$. Let $\mathcal{E}$ be a subset of $2^V$, that is, a family of subsets of $V$. $\mathcal{E}$ must contain all subsets of $V$ of cardinality strictly greater than 1, and must not contain the empty set. (Each singleton $\{\alpha\}$ can be left out or made part of $\mathcal{E}$ at will.)*

*For each $S$ in $\mathcal{E}$, pick two fresh variables $\lambda_S$ and $\gamma_S$. The* rewriting functions *$r^+$ and $r^-$ are defined in figure 3. The* output constraint graph *$D$ is defined by its components $D^\downarrow$ and $D^\uparrow$, given in figure 4, and by the relation $\leq_D$, which is* the transitive closure *of the constraints given in figure 5.*

*Let $\sigma' = r^-(A) \Rightarrow r^+(\tau) \mid D$. Then $\sigma'$ is simply closed, contains no $\sqcup$ or $\sqcap$ constructs, and $\sigma =^\forall \sigma'$.*

$$E^{\downarrow}(\alpha) = r^+(C^{\downarrow}(\alpha)) \qquad E^{\uparrow}(\alpha) = r^-(C^{\uparrow}(\alpha)) \qquad E^{\downarrow}(\gamma_S) = \bot$$
$$E^{\uparrow}(\gamma_S) = r^-(\bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha)) \qquad E^{\downarrow}(\lambda_S) = r^+(\bigsqcup_{\alpha \in S} C^{\downarrow}(\alpha)) \qquad E^{\uparrow}(\lambda_S) = \top$$

Figure 6: Definition of $E^{\downarrow}$ and $E^{\uparrow}$

$$\begin{array}{rcll} \alpha & \leq_E & \beta & \text{when } \alpha \leq_C \beta \\ \gamma_S & \leq_E & \alpha & \text{when } \sqcap S \leq_C \alpha \\ \alpha & \leq_E & \lambda_S & \text{when } \alpha \leq_C \sqcup S \\ \gamma_S & \leq_E & \lambda_T & \text{when } \exists \alpha \in S \quad \exists \beta \in T \quad \alpha \leq_C \beta \end{array}$$

Figure 7: Definition of $\leq_E$

This definition might seem complex and non-intuitive. In particular, one might wonder why figure 4 assigns a constructed lower (resp. upper) bound to $\gamma_S$ (resp. $\lambda_S$), or why the last line of figure 5 creates constraints of the form $\lambda_S \leq \gamma_T$. These constraints are only necessary to guarantee that simple closure is preserved; they will be dropped immediately by garbage collection, as we shall see below.

$\sigma'$ is simply closed; this allows computing polarities and performing garbage collection. We now simulate these phases. It turns out that the newly introduced $\lambda_S$ are (at most) positive, while the $\gamma_S$ are (at most) negative. As for the existing variables $\alpha$, their polarities must decrease. In particular, if $\{\alpha\} \in \mathcal{E}$, then $\alpha$ becomes neutral. These results are formalized below.

**Lemma 4.1** *We have*

$$\begin{aligned} \mathrm{dom}^+(\sigma') &\subset \{\lambda_S \,;\, S \in \mathcal{E} \wedge S \subset \mathrm{dom}^+(\sigma)\} \\ &\quad \cup \{\alpha \,;\, \{\alpha\} \notin \mathcal{E} \wedge \alpha \in \mathrm{dom}^+(\sigma)\} \\ \mathrm{dom}^-(\sigma') &\subset \{\gamma_S \,;\, S \in \mathcal{E} \wedge S \subset \mathrm{dom}^-(\sigma)\} \\ &\quad \cup \{\alpha \,;\, \{\alpha\} \notin \mathcal{E} \wedge \alpha \in \mathrm{dom}^-(\sigma)\} \end{aligned}$$

A first consequence of this lemma is that this transformation can be used to eliminate bipolar variables. It suffices to choose $\mathcal{E}$ such that whenever $\alpha$ is bipolar, then it is scheduled for elimination, i.e. $\{\alpha\} \in \mathcal{E}$. The lemma shows that if this condition holds, then there are no bipolar variables in the output type scheme.

The second consequence is to allow garbage collection on $\sigma'$. Because we only have an approximation of the polarities, we can only do a partial garbage collection. So, the type scheme $\sigma''$ which we are going to define is not $\mathrm{GC}(\sigma')$; it contains more constraints than $\mathrm{GC}(\sigma')$, but fewer than $\sigma'$. Thus, in practice, we still have to complete the job by running the garbage collection algorithm, but we have saved time by not generating superfluous constraints.

**Theorem 4.2** *Let $E$ be the constraint graph given in figures 6 and 7. Let $\sigma'' = r^-(A) \Rightarrow r^+(\tau) \mid E$. Then $\sigma''$ is a partially garbage-collected version of $\sigma'$. That is,* $\mathrm{GC}(\sigma'') = \mathrm{GC}(\sigma')$.

## 5 Outlaw bipolar variables

We want our implementation of the type inference engine to work with closed constraint graphs, which allows performing the closure computations incrementally. As a consequence, we must ensure that all simplification phases preserve closure.

However, it turns out that in general, garbage collection does not preserve closure. For instance, consider a bipolar variable $\alpha^{\pm}$ whose constructed bounds are

$$C^{\downarrow}(\alpha) = \delta^{\pm} \to \beta^+ \qquad C^{\uparrow}(\alpha) = \delta^{\pm} \to \gamma^-$$

Then, if the constraint graph is to be closed, it must contain $\beta^+ \leq \gamma^-$. However, such a constraint shall be thrown away by garbage collection, thus breaking the invariant.

Fortunately, this problem has a simple solution. One easily verifies that if the input type scheme has no bipolar variables, then garbage collection does preserve closure. Such a type scheme is said to verify the *mono-polarity invariant.*

Furthermore, we prove that any type scheme produced by our type inference rules complies with this invariant. The principle of the proof is two-fold. First, no freshly introduced variables are bipolar. Rules ($\mathrm{V{\scriptsize AR}_I}$) and ($\mathrm{A{\scriptsize PP}_I}$) expressly introduce two variables $\alpha \leq \beta$, where $\alpha$ is non-positive and $\beta$ is non-negative. If a single fresh variable were used, it might be bipolar. Second, polarities decrease with time. That is, if a variable is non-positive (resp. non-negative) when it is first introduced, then it will never become positive (resp. negative) at a later stage of the type inference process. This property is consistent with the intuition that a non-positive (resp. non-negative) variable will never receive new upper (resp. lower) bounds in the future.

As shown in section 4, the canonization algorithm can be used to transform any type scheme into an equivalent one which verifies the invariant. It replaces each bipolar variable with two fresh variables, a negative one and a positive one. Thus, the mono-polarity invariant will at worst double the number of type variables in a type scheme.

This invariant also has beneficial effects in other areas. For instance, it opens new opportunities to the minimization algorithm described in section 7. Indeed, a bipolar variable cannot be merged with any other variable; but once it is split, each half can potentially be merged with some other variable. Another effect is that any cycles in the constraint graph are now automatically removed by garbage collection. Indeed, it is easy to verify that if a cycle survives garbage collection, then it contains a bipolar variable. Thus, removing cycles in a separate phase (as done in [6, 14, 1]) is no longer necessary.

## 6 Sugar before display

There is a well-known "simplification" strategy which we have not discussed so far: namely, replacing a non-negative (resp. non-positive) variable with its unique lower (resp. upper) bound, if it exists. This strategy has been proposed in numerous papers [6, 1, 3, 14].

However, it becomes illegal in our setting! Indeed, suppose we attempt to replace a variable $\alpha$ with its unique bound. If this bound is a constructed term, then the substitution violates the small terms invariant. If, on the other hand, it is a type variable $\beta$, then $\alpha$ and $\beta$ must have opposite signs, because garbage collection would otherwise have thrown away the constraint which links them. Thus, identifying them creates a bipolar variable and violates the mono-polarity invariant.

One should not be particularly upset about this fact. In our eyes, the fact that this strategy breaks some desirable invariants only shows that it conflicts with the efficiency goal. So, we forbid its use during the type inference process. The strategy still remains useful *after* the type inference process is complete, that is, immediately before displaying the result to the user. Indeed, though our invariants are internally useful, they make type schemes illegible by creating many intermediate nodes. So, we do allow replacing variables with their unique bounds prior to display. We believe that this distinction between internal and external representations is quite important, and failure to recognize it can lead a designer to set up conflicting "simplification" strategies.

This distinction also exists in ML typecheckers, where type terms are internally represented by graphs, but displayed as trees. Preserving sharing internally is crucial to avoid an exponential efficiency loss.

## 7 Minimization

In [14], we proposed a two-step method to simplify a type scheme $\sigma$. First, come up with some substitution $\rho$; then, check whether $\rho(\sigma)$ is equivalent to $\sigma$. However, the number of possible $\rho$ is huge, and the entailment algorithm used in the second step is rather costly, so it was necessary to devise heuristics to select appropriate substitutions. This solution was unsatisfactory, because these heuristics were rather *ad hoc* and still extremely inefficient.

Thus, a systematic algorithm, which directly builds as powerful a substitution as possible, is needed. Such an algorithm was proposed by Felleisen and Flanagan [9] in the case of set-based analysis; the same principle can be applied here. Start with the largest conceivable substitution, i.e. one which merges all negative (resp. positive) variables together. (We cannot merge a variable with a constructed term, or a negative variable with a positive one, as explained in section 6.) We can, equivalently, consider it as a partition of the variables into two classes. Then, simulate a run of the subsumption algorithm which checks whether this substitution is acceptable. If a failure occurs, determine why and refine the partition by splitting an appropriate class so as to eliminate the cause of the failure. Repeat this process until no more failures are detected.

A failure typically occurs when we try to merge two variables which do not play the same role in the type scheme. A variable's "role" is determined by the way it is linked to other variables through subtyping constraints. Thus, another way to (informally) present the algorithm is to say that two variables can be merged if each one carries links of the same kind as the other one and these links lead to variables which can themselves be merged.

Both descriptions ring a bell—similar ideas are used to minimize finite state automata.

**Theorem 7.1** *Let $\sigma = A \Rightarrow \tau \mid C$ be a garbage collected type scheme. A partition $\equiv$ of $\sigma$'s variables is said to be* compatible *iff $\alpha \equiv \beta$ implies*

- *$pred_C(\alpha) = pred_C(\beta)$ and $succ_C(\alpha) = succ_C(\beta)$;*
- *$\mathrm{polarity}(\alpha) = \mathrm{polarity}(\beta)$;*
- *$C^{\downarrow}(\alpha) \equiv C^{\downarrow}(\beta)$ and $C^{\uparrow}(\alpha) \equiv C^{\uparrow}(\beta)$.*

*Then, the type scheme $\sigma/_{\equiv}$ (obtained by collapsing classes) is equivalent to $\sigma$.*

From this result, we deduce the so-called *minimization* algorithm. The algorithm computes the coarsest compatible partition. (As explained above, this involves computing an initial partition, and running Hopcroft's minimization algorithm [11] to refine it.) It then collapses each class down to a single type variable. The whole process takes time $O(dn \log n)$, where $d$ is the degree of the graph $\leq_C$, and $n$ is the number of type variables in $\sigma$. We conjecture that it is also $O(N \log N)$, where $N$ is some measure of the size of $\sigma$. Actual tests show that its running time is approximately linear in the size of its input.

It is natural to ask whether the algorithm is complete, i.e. whether the coarsest compatible substitution is really the coarsest substitution allowed by our definition of scheme subsumption. The answer is negative; the problem is that our definition of predecessor and successor sets rely on constraints which are syntactically present in the constraint graph. If these sets are defined using entailment, a more powerful definition of compatibility is obtained, which might be complete (no counter-examples are known to us). However, using entailment has two ill effects: first, the extended algorithm is still not complete, because no complete entailment algorithm is known; and second, it is slower, because our incomplete entailment algorithm is rather costly.

## 8 Implementation

How do these pieces fit together? First, the type inference engine analyzes the program using the rules of figure 2. As new constraints appear, their closure is computed incrementally, which guarantees that they have a solution. At any point, canonization, garbage collection and minimization can be applied; our theoretical development guarantees that their combination preserves closure. Thus, the simplification process fits smoothly into the regular inference process.

When to perform simplification? At least at each `let` node, because the environment would otherwise contain unsimplified type schemes, and the simplification work would be needlessly duplicated. At other nodes, we have a choice; in practice, only garbage collection is performed, because it is cheap and effective enough.

Finally, when a type scheme must be presented to the user, we apply our "external" simplification method to it; that is, we replace each type variable with its unique bound, whenever allowed by the occur check.

| | Lines | Time | Lines/s | Closure | Can. | G.C. | Min. | w/o Min. | Caml-Light |
|---|---|---|---|---|---|---|---|---|---|
| Graphs library | 900 | 2s | 450 | 45% | 10% | 30% | 15% | 2s | 1s |
| CL 0.74 standard library | 4300 | 5s | 860 | 35% | 20% | 25% | 15% | 13s | 6s |
| Format library | 1100 | 7s | 160 | 25% | 10% | 25% | 40% | 8s | 1s |
| MLgraph library | 9900 | 27s | 370 | 35% | 10% | 25% | 30% | 93s | 13s |

Figure 8: Implementation's performance

Although the type language described in this paper is reduced to a bare minimum, our implementation has a very rich type language, featuring record and variant types with row variables [16], and an interesting type-based exception analysis. In particular, it can handle arbitrary Caml-Light programs, provided they are translated into our language. This translation removes all type declarations[3] and interface specifications. This yields much more precise typings, but makes the typechecker's task much heavier.

Figure 8 presents a few performance measurements on existing Caml-Light libraries. The prototype, compiled into machine language using Objective Caml 1.07, was tested on a 150MHz Pentium Pro processor; timings are in user time. The last column shows Caml-Light's compilation time, and roughly represents a user's expectation; the previous one shows our typechecker's performance in the absence of minimization.

For Graphs and MLgraph, performance seems close to Caml-Light's. The standard library contains smaller functions, and is dealt with more easily; the Format library, on the other hand, contains functions with large concrete types—recall that we use almost no abstract types—and is more difficult to handle.

So, performance is reasonable when dealing with small or medium types, and worsens when handling large ones; the prototype does not behave linearly in this respect. However, it is nice to note that simplification is not a bottleneck, since closure itself is non-linear. This is not entirely surprising, since closure has, in theory, cubic time complexity. Also, note that the benefits of minimization outweigh its cost; that was not the case in [9].

## 9 Directions

To improve performance, one possibility is to investigate incremental simplification algorithms—all of our three algorithms work on a whole type scheme, rather than only on the most recently added constraints.

On the other hand, our performance figures suggest that our simplification techniques are reasonably fast, i.e. of the same order as closure itself. Thus, another possibility is to concentrate not on the simplification algorithms, but on the constraint creation engine itself—that is, on the formulation of the type inference rules. Indeed, our use of $\lambda$-lifting incurs a loss of sharing between the various branches of a type inference derivation. Sharing is explicitly restored by the context intersection operation, but this entails extra closure and simplification work.

Formulating the typing rules without making use of $\lambda$-lifting would thus produce a more intuitive, and possibly more efficient, system. It would also help deal with imperative constructs. Since our current system does not support unquantified type variables, it deals with expansive `let` definitions by rewriting them into $\beta$-redexes [19]. However, top-level `let` definitions cannot be rewritten in such a way, so they are accepted only if they define a monomorphic value—hence, a small loss of flexibility.

Such a system seems to require a more complex subtyping rule—which is why we adopted $\lambda$-lifting in the first place. So, it is a challenging research subject.

Finally, let us mention that better practical performance would of course be obtained if the code being tested used appropriate abstract type definitions and module interfaces.

## 10 Related work

The typing rules used in this paper are the same as in [18]. Older systems [7, 14] have the same valid *programs*, but fewer valid *typings*, because they lack a polymorphic subsumption rule. Here, as in [18], scheme subsumption is the one and only theoretical basis for all simplification methods, which is simple and elegant.

The type system proposed by Sulzmann *et al.* [17] is fairly close to ours, but does not use $\lambda$-lifting. Still, it is not the ideal system we discussed in the previous section, because its subtyping rule is too weak: in particular, it does not allow garbage collection in the global constraint graph.

As far as simplification is concerned, our current algorithms are strictly more powerful than those given in [6, 1, 18]. Minimization is slightly less powerful, in theory, than the heuristics of [14], because the latter are based on entailment; however, in practice, it is much more effective and efficient.

A performance comparison with [1] or [9] is difficult, because the analysis performed, as well as the machine and compiler used, vary widely. It seems safe to say that our results are at least as good.

Palsberg [13] and Henglein [10] study efficient type inference for an object calculus. It differs largely from our system (in particular, Henglein takes advantage of the fact that object types are *invariant* to improve efficiency). However, some of the techniques presented here (garbage collection and minimization) might carry over to it.

## Conclusion

This paper describes our work towards an efficient, streamlined type inference engine in the presence of subtyping. Our starting point was an existing type system with several known simplification strategies (canonization, garbage collection). Still, implementing it in an efficient way was not possible, mainly by lack of a systematic substitution algorithm, but also because the issue of data representation was not settled; some transformations had conflicting effects.

[3] except datatype declarations, which are turned into parameterized abstract type declarations. Because our language has extensible variant types, some pattern matchings would otherwise receive unexpected typings.

We give new algorithms: a very efficient minimization algorithm, which eliminates the need for any substitution heuristics, and a combined canonization/garbage collection algorithm. In addition, we deal with the problem of data representation by identifying invariants which should be preserved during type inference: the small terms invariant and the mono-polarity invariant. They have beneficial effects in several areas. Some transformations which improve readability, but break the invariants, are postponed until the type must be displayed to the user.

The result is a full framework for type inference with subtyping, with a clean and simple theory, leading to efficient algorithms. At present, the most promising research topic appears to be the elimination of $\lambda$-lifting, yielding a system closer to ML, but with greater theoretical complexity.

### References


[1] Alexander Aiken and Manuel Fähndrich. Making set-constraint based program analyses scale. Technical Report CSD-96-917, University of California, Berkeley, September 1996. URL: `http://http.cs.berkeley.edu/~manuel/papers/scw96.ps.gz`.

[2] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In Andre Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, CA, June 1992. IEEE Computer Society Press. URL: `http://http.cs.berkeley.edu/~aiken/ftp/lics92.ps`.

[3] Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping. Technical Report CSD-96-909, University of California, Berkeley, July 1996. URL: `http://http.cs.berkeley.edu/~aiken/ftp/quant.ps`.

[4] Alexander S. Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming & Computer Architecture*, pages 31–41. ACM Press, June 1993. URL: `http://http.cs.berkeley.edu/~aiken/ftp/fpca93.ps`.

[5] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, January 1994. URL: `http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps`.

[6] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 169–184, 1995. URL: `http://www.cs.jhu.edu/~trifonov/papers/sptio.ps.gz`.

[7] Jonathan Eifrig, Scott Smith, and Valery 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. URL: `http://www.elsevier.nl/locate/entcs/volume1.html`.

[8] Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR96-266, Rice University, November 1996. URL: `http://www.cs.rice.edu/CS/PLT/Publications/tr96-266.ps.gz`.

[9] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248, Las Vegas, Nevada, June 1997. URL: `http://www.cs.rice.edu/CS/PLT/Publications/pldi97-ff.ps.gz`.

[10] Fritz Henglein. Breaking through the $n^3$ barrier: Faster object type inference. In Benjamin Pierce, editor, *Proc. 4th Int'l Workshop on Foundations of Object-Oriented Languages (FOOL), Paris, France*, January 1997. URL: `http://www.cs.indiana.edu/hyplan/pierce/fool/henglein.ps.gz`.

[11] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, NY, 1971.

[12] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[13] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. URL: `http://www.cs.purdue.edu/homes/palsberg/paper/ic95-p.ps.gz`.

[14] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, January 1996. URL: `http://pauillac.inria.fr/~fpottier/publis/ICFP96.ps.gz`.

[15] François Pottier. *Type inference in the presence of subtyping: from theory to practice.* PhD thesis, Université Paris VII, July 1998. URL: `http://pauillac.inria.fr/~fpottier/publis/thesis-fpottier.ps.gz`.

[16] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz`.

[17] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In Benjamin Pierce, editor, *Proc. 4th Int'l Workshop on Foundations of Object-Oriented Languages (FOOL), Paris, France*, January 1997. URL: `http://www.cs.indiana.edu/hyplan/pierce/fool/sulzmann.ps.gz`.

[18] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. SV, September 1996. URL: `http://www.cs.jhu.edu/~trifonov/papers/subcon.ps.gz`.

[19] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Rice University, February 1993.

$$
\begin{array}{lllll}
\texttt{[ Nil ]} \le v_1 & v_3 \le v_4 & v_5 \le v_6 & v_7 \le v_8 & v_4 \le v_6 \to v_7 \\
v_9 \le v_{10} & v_{11} \le v_{12} & v_{13} \le v_{14} & v_{10} \le v_4 \to v_{11} & v_{12} \le v_{14} \to v_{15} \\
v_{15} \le v_{16} & v_8 * v_{16} \le v_{17} & v_{19} \to v_{20} \le v_{21} & v_2 \le v_5 * v_{13} & \texttt{[ Cons of } v_{17} \texttt{ ]} \le v_{18} \\
v_1 \le v_{20} & v_{18} \le v_{20} & v_{22} \le v_9 & v_3 \to v_{21} \le v_{22} & v_{19} \le \texttt{[ Nil | Cons of } v_2 \texttt{ ]}
\end{array}
$$

Figure 9: Before closure

$$
\begin{array}{rcl}
\texttt{[ Nil ]} \le & v_1 & \le v_{15},\, v_{16},\, v_{20} \\
 & v_2 & \le v_5 * v_{13} \\
v_4 \le & v_3 & \le v_4,\, v_6 \to v_7 \\
v_3 \le & v_4 & \le v_3,\, v_6 \to v_7 \\
 & v_5 & \le v_6 \\
v_5 \le & v_6 & \\
 & v_7 & \le v_8 \\
v_7 \le & v_8 & \\
v_3 \to v_{21},\, v_{22} \le & v_9 & \le v_{10},\, v_4 \to v_{11} \\
v_3 \to v_{21},\, v_9,\, v_{22} \le & v_{10} & \le v_4 \to v_{11} \\
v_{19} \to v_{20},\, v_{21} \le & v_{11} & \le v_{12},\, v_{14} \to v_{15} \\
v_{19} \to v_{20},\, v_{11},\, v_{21} \le & v_{12} & \le v_{14} \to v_{15} \\
 & v_{13} & \le v_{14},\, v_{19},\, \texttt{[ Nil | Cons of } v_2 \texttt{ ]} \\
v_{13} \le & v_{14} & \le v_{19},\, \texttt{[ Nil | Cons of } v_2 \texttt{ ]} \\
\texttt{[ Nil | Cons of } v_{17} \texttt{ ]},\, v_1,\, v_{18},\, v_{20} \le & v_{15} & \le v_{16} \\
\texttt{[ Nil | Cons of } v_{17} \texttt{ ]},\, v_1,\, v_{18},\, v_{15},\, v_{20} \le & v_{16} & \\
v_8 * v_{16} \le & v_{17} & \\
\texttt{[ Cons of } v_{17} \texttt{ ]} \le & v_{18} & \le v_{15},\, v_{16},\, v_{20} \\
v_{13},\, v_{14} \le & v_{19} & \le \texttt{[ Nil | Cons of } v_2 \texttt{ ]} \\
\texttt{[ Nil | Cons of } v_{17} \texttt{ ]},\, v_1,\, v_{18} \le & v_{20} & \le v_{15},\, v_{16} \\
v_{19} \to v_{20} \le & v_{21} & \le v_{11},\, v_{12},\, v_{14} \to v_{15} \\
v_3 \to v_{21} \le & v_{22} & \le v_9,\, v_{10},\, v_4 \to v_{11}
\end{array}
$$

Figure 10: After closure

## A A full example

This appendix shows the type inference engine at work on a small, but rather typical example, namely the classic `map` operation. The type language used here has products and extensible variant types (without row variables, for the sake of simplicity). The expression language has corresponding constructs (pairs, data constructors and pattern matching), as well as a fix-point operator `rec`.

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

The type inference rules indicate that the expression has type $v_{22}$, together with the constraints given by figure 9. Let us compute their closure; it is given by figure 10, in a way which looks more like a constraint graph: there is one line per type variable, mentioning its lower and upper bounds.

It turns out that the closure computation did not introduce any symbolic $\sqcup$ or $\sqcap$ constructs. So, in this case, the canonization algorithm has nothing to do. We can now compute polarities. The fix-point computation starts by marking the entry point, $v_{22}$, as positive, and then propagates marks through the above constraint graph. We find that $v_6, v_8, v_{16}, v_{17}, v_{20}, v_{21}, v_{22}$ are positive, while $v_2, v_3, v_5, v_7, v_{13}, v_{19}$ are negative.

Given this information, we can perform garbage collection. Positive (resp. negative) variables lose all of their upper (resp. lower) bounds; furthermore, constraints involving two variables are kept only if the left-hand one is negative and the right-hand one is positive. This yields

$$
\begin{array}{rcl}
 & v_2 & \le v_5 * v_{13} \\
 & v_3 & \le v_6 \to v_7 \\
 & v_5 & \le v_6 \\
v_5 \le & v_6 & \\
 & v_7 & \le v_8 \\
v_7 \le & v_8 & \\
 & v_{13} & \le \texttt{[ Nil | Cons of } v_2 \texttt{ ]} \\
\texttt{[ Nil | Cons of } v_{17} \texttt{ ]} \le & v_{16} & \\
v_8 * v_{16} \le & v_{17} & \\
 & v_{19} & \le \texttt{[ Nil | Cons of } v_2 \texttt{ ]} \\
\texttt{[ Nil | Cons of } v_{17} \texttt{ ]} \le & v_{20} & \\
v_{19} \to v_{20} \le & v_{21} & \\
v_3 \to v_{21} \le & v_{22} &
\end{array}
$$

We can now run the minimization algorithm. We compute the largest equivalence relation such that two equivalent variables have the same polarity, the same successors and predecessors, and equivalent constructed bounds. In this case, it is easy to see that the only non-trivial equivalence classes are $\{v_{13}, v_{19}\}$ and $\{v_{16}, v_{20}\}$.

Note that replacing $v_{19}$ with $v_{13}$ can be viewed as recognizing a partially unrolled fix-point. Indeed, it essentially consists in replacing $F(\mu t.F(t))$ with $\mu t.F(t)$, where $F$ is the type operator

$$t \mapsto \texttt{[ Nil | Cons of } v_5 * t \texttt{ ]}$$

By collapsing the equivalence classes, we obtain

$$
\begin{aligned}
v_2 &\leq v_5 * v_{13} \\
v_3 &\leq v_6 \to v_7 \\
v_5 &\leq v_6 \\
v_5 \leq v_6 & \\
v_7 &\leq v_8 \\
v_7 \leq v_8 & \\
v_{13} &\leq \texttt{[ Nil | Cons of } v_2 \texttt{ ]} \\
\texttt{[ Nil | Cons of } v_{17} \texttt{ ]} \leq v_{16} & \\
v_8 * v_{16} \leq v_{17} & \\
v_{13} \to v_{16} \leq v_{21} & \\
v_3 \to v_{21} \leq v_{22} &
\end{aligned}
$$

Type simplification is over; as far as the internal engine is concerned, this is the result. At this point, "external" simplification strategies may be applied to make the type scheme more readable. We replace each variable with its unique bound (except where disallowed by the occur check). Finally, we obtain that `map` has type $(v_6 \to v_8) \to v_{13} \to v_{16}$ where

$$
\begin{aligned}
v_{13} &\leq \texttt{[ Nil | Cons of } v_6 * v_{13} \texttt{ ]} \\
\texttt{[ Nil | Cons of } v_8 * v_{16} \texttt{ ]} &\leq v_{16}
\end{aligned}
$$

which is optimal, given our type language.

If we want to go a little further, we can notice that the above inequalities can be replaced by equalities. (The proof of correctness is trivial. Garbage collection would replace these equalities with the original inequalities; since garbage collection is correct, the two type schemes are equivalent.) Thus, `map`'s type could be printed as

$$
\begin{aligned}
(v_6 \to v_8) \to \mu t.\texttt{[ Nil | Cons of } v_6 * t \texttt{ ]} \\
\to \mu t.\texttt{[ Nil | Cons of } v_8 * t \texttt{ ]}
\end{aligned}
$$