Simplifying subtyping constraints: a theory

François Pottier[*]
Francois.Pottier@inria.fr

August 25, 2000

Abstract

This paper offers a theoretical study of constraint simplification, a fundamental issue for the designer of a practical type inference system with subtyping.

In the simpler case where constraints are equations, a simple isomorphism between constrained type schemes and finite state automata yields a complete constraint simplification method. Using it as a guide for the intuition, we move on to the case of subtyping, and describe several simplification algorithms. Although no longer complete, they are conceptually simple, efficient, and very effective in practice.

Overall, this paper gives a concise theoretical account of the techniques found at the core of our type inference system. Our study is restricted to the case where constraints are interpreted in a non-structural lattice of regular terms. Nevertheless, we highlight a small number of general ideas, which explain our algorithms at a high level and may be applicable to a variety of other systems.

1 Introduction

1.1 Subtyping and type inference

In a typed programming language, a function application $(e_1\, e_2)$ is legal if and only if there exists a type τ_2 which is both a valid type for the argument e_2 and a valid domain type for the function e_1.

In the simply-typed λ-calculus, the set of all valid types of a given (un-annotated) expression e has a very regular structure: it is either empty, or exactly the set of all substitution instances of a *most general* type τ. Then, inferring the (most general) type of an expression reduces to solving a set of equations between types [Wan87]. The addition of `let`-polymorphism, as done in ML [Mil78], essentially preserves this fact.

These systems have type instantiation as their only notion of type compatibility. In particular, they view any two ground types as incompatible unless they are equal. For instance, assume machine integers and floating-point numbers are described by two base types, namely `int` and `real`. Then, the application $(\mathit{fact}\, x)$ is illegal if fact and x have (most general) types `int` $\rightarrow$ `int` and `real`, respectively. This is a good point, since it is certainly a programming error. On the other hand, if log and n have (most general) types `real` $\rightarrow$ `real` and `int`, respectively, then the application $(\mathit{log}\, n)$ is deemed illegal as well. Yet, because integers are mathematically a subset of reals, one may actually wish for this term to be accepted.

To overcome this limitation, Mitchell [Mit84] suggests enriching these type systems with *subtyping.* This involves introducing a partial order $\leq$ on types, together with a new typing

[*]Address: INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France. This paper is to appear in *Information & Computation.*

rule, stating that if $\tau \leq \tau'$ holds (read: if τ is a *subtype* of τ') then every expression of type τ has type τ' as well. For instance, choosing the strict ordering `int` $<$ `real` causes $(\mathit{fact}\, x)$ to remain ill-typed, while $(\mathit{log}\, n)$ becomes well-typed, because $n :$ `int` now implies $n :$ `real`. Subtyping is not, in general, limited to base types: Cardelli [Car88] equips record types with a natural subtyping relation, allowing information about any number of fields to be discarded. In addition to its intrinsic interest, such a system provides a possible basis for the study of object-oriented languages.

Systems equipped with subtyping have a combination of type instantiation and subtyping as their notion of type compatibility. As a result, the type inference problem no longer reduces to solving a set of equations. Instead, it requires solving a set of inequalities, usually called *constraints* [Mit84, Pal95]. This process is theoretically straightforward, but costly, because the efficient unification algorithms developed to solve equations [JK90] can no longer be used.

Why, then, should we wish to perform type inference? Would it not be sufficient to require the programmer to supply type annotations, and merely check their consistency? Let us give two reasons why type inference is useful. First, it frees the programmer from the burden of declaring the type of every program variable—a tedious task in many widespread languages—and allows him to naturally write polymorphic code. Second, type inference may be viewed as a simple way of describing program analyses [PO95], whose results may be used, for instance, to drive compiler optimizations.

1.2 Simplification

Our aim, then, is to study the type inference problem in the presence of subtyping, and to compare it with the original problem, where subtyping is reduced to equality.

The constraint system to be solved is the same in both cases; its size is linear in the program size. (Though `let`-polymorphism may, in fact, cause it to grow exponentially, it is an accepted fact that it "usually" does not.) However, while equations can be solved in quasi-linear time, solving inequalities between (non-atomic) terms typically requires (at least) cubic-time algorithms [AW93, Pal95, MR00]. Thus, an *efficiency* problem appears.

Every unification problem admits a most general solution. Thus, in the absence of subtyping, every program has a most general type. It is often compact and easily intelligible. On the other hand, many classes of subtyping problems do not have most general solutions. Then, describing the set of all valid types of a given program requires printing the constraint system itself, which often involves many auxiliary type variables. Thus, a *readability* issue also arises.

To address these problems, it seems necessary to *simplify* systems of subtyping constraints, i.e. to reduce them to smaller, equivalent systems. This topic has received continued attention in the past few years [Aik94, AF96, AWP96, FFSA98, AFFS98, Fäh99, EST95a, TS96, FF96, FF97, Fla97, Pot96, Pot98a, Pot98b, Pot98c]. Indeed, designing a reasonable simplification algorithm is not easy. It must be correct and efficient. Ideally, it should also be complete, i.e. produce optimal results. Unfortunately, achieving completeness involves solving the *constraint entailment* problem, which may be much more complex than constraint solving. In our framework, for instance, entailment has been shown PSPACE-hard, but its decidability is still unsettled [HR98, NP99]. For this reason, practical constraint simplification algorithms are often incomplete.

1.3 Choices

Defining a type system with subtyping involves two main choices. First, one must choose a constraint logic, i.e. define a constraint language and its interpretation within a model.

Second, one must define a set of typing rules. Because typing judgements involve constraints, the rules reduce the typing problem to a series of assertions expressed within the constraint logic. These two choices are mostly orthogonal, as pointed out by [OSW99].

As far as the first choice is concerned, the array of possibilities is extremely wide. The model may have covariant type constructors only, or it may have contravariant constructors as well. (In the former case, constraint systems may have smallest solutions.) It may or may not have recursive types. (If present, they may give smoother mathematical properties to the model, leading to simpler algorithms.) The model, equipped with the subtype ordering, may or may not form a lattice. (If it does, then more aggressive simplifications become valid.) Types may be interpreted as ideals [MPS86] or as terms. (The former interpretation assigns more precise meanings to union and intersection types. On the other hand, it may be more complex; axioms such as $\bot = c(\bot)$, where c is any unary strict type constructor, make constraint solving more difficult.) When types are interpreted as terms, subtyping may be atomic (i.e. only constant type constructors may be comparable), structural (i.e. only type constructors of identical arity may be comparable), or non-structural (even type constructors with different arities may be comparable). Constraints may be interpreted within a fixed model, or within a family thereof. (If the former, then deeper simplifications are usually possible. On the other hand, user-extensible subtype hierarchies require the latter.)

Changes in the constraint logic greatly affect the complexity of the resolution and entailment problems (as well as the formulation of the corresponding algorithms). For this reason, we will focus on a single case, while hoping that (some of) our methods may be applicable to (some) other logics. More specifically, we choose to interpret types in the fixed model of all regular terms generated by $\bot$, $\rightarrow$ and $\top$, with arities 0, 2 and 0, respectively. Subtyping is interpreted in the model by ordering these constructors as given and viewing $\rightarrow$ as a contra/co-variant type constructor. This yields a non-structural subtyping relationship, which forms a lattice. Although this case may seem very simple, generalizing it to more elaborate non-structural term lattices is straightforward (see e.g. [Pot00a]) and requires no fundamental changes to the theory or to the algorithms.

The second choice definitely has less impact on the system as a whole. Although many variants have appeared in the literature, most of them are very close in spirit. The idea is to extend the Hindley-Milner type discipline [Mil78] with constraints, while keeping `let`-polymorphism. Perhaps the most elegant formal exposition of this idea is the system HM(X) by Odersky, Sulzmann *et al.* [OSW99, SMZ99]. Here, however, we will use a set of typing rules inspired by Trifonov and Smith [TS96], with a few technical modifications to the type inference rules. This somewhat uncommon presentation allows us to deal with closed (i.e. fully universally quantified) type schemes only, making a formal description of constraint simplification—the central topic of the present paper—easier.

1.4 Overview

In this paper, we present a type inference system with subtyping, designed with constraint simplification in mind. Its inference rules are written so as to generate amenable constraint systems. We describe three simplification algorithms, designed to be used in combination with one another; they are simple, efficient and effective. We emphasize the parallel between the case of equality and that of subtyping, and show that these algorithms are based, in both cases, on the same broad ideas. In fact, in the case of equality, their combination yields a complete simplification strategy. Although it is no longer complete in the case of subtyping, we believe it produces good results in practice.

This paper is laid out as follows. Section 2 introduces the necessary theoretical background, namely a set of ground types ordered by subtyping, a core language, a set of typing rules, and an equivalent set of type inference rules. The type inference rules describe a deterministic algorithm, which maps an expression to a constrained type scheme. The constraints thus generated may be viewed, at will, as equations or as subtyping constraints. In the former case, we obtain a type inference system close to that of ML; in the latter, a system equipped with the full power of subtyping. Section 3 studies the simpler one, and suggests a complete simplification method by borrowing concepts from automata theory. This section should help the reader form general intuitions about the structure and behavior of constraints. We hope these ideas are applicable in other contexts; in particular, they may be transferred to our more complex system, which is the topic of section 4. In this section, which forms the theoretical body of the paper, we formally describe and prove several constraint simplification algorithms, based on the same ideas. Section 5 shows these algorithms at work on a simple example. Section 6 reviews related work.

This paper borrows ideas from several existing works. One of its novel aspects is their seamless integration: we describe a clean, simple theory, which leads directly to an efficient implementation [Pot00b]. Another contribution is in the area of presentation. First, thanks to a carefully thought-out mathematical layout, we are able to present our formal results with almost no auxiliary steps, and with substantially smaller proofs than in earlier works. Second, we highlight the similarity of our methods with those applicable in the case of equality constraints; by doing so, we hope to help the reader grasp the essential ideas behind our algorithms. Thus, this paper may constitute a good introduction to the theoretical issues behind constraint-based type inference.

Before beginning our technical exposition, let us recall that the focus of this paper is on constraint simplification. Because of this decision, several issues related to the design of a constraint-based type inference system have been left aside. Among them, one may mention certain fundamental theoretical results, such as type safety; various implementation concerns, including efficiency measurements; extensions of the core language necessary to obtain a full-blown programming language; etc. These issues are discussed at length in [Pot98c, Pot98b]. Lastly, we do not address the issue of entailment, i.e. we do not attempt to give an algorithm to decide whether two given type schemes are in the subsumption relation. Indeed, we do not have a need for such an algorithm, because all of the simplification algorithms presented in this paper provably preserve the meaning of their input. Nevertheless, the entailment problem is closely linked to the issue of constraint simplification; we refer the interested reader to [AC93, KPS93, Pot98c, HR98, NP99].

2 A constraint-based type inference system

2.1 Ground types

Ground types are the regular trees built with the elementary constructors $\bot$, $\top$ and $\rightarrow$. They are the simplest kind of types, since they are (possibly recursive) types without variables. They are monomorphic; polymorphism shall be introduced later by considering type *schemes* which denote sets of ground types.

Definition 1 *Let the* ground signature Σ_g *consist of* $\bot$ *and* $\top$ *with arity* 0 *and* $\rightarrow$ *with arity* 2. *A* path p *is a finite string of* 0*'s and* 1*'s, i.e. an element of* $\{0,1\}^*$. ϵ *denotes the empty path. The length of a path* p *is denoted by* $|p|$. *Its* parity $\pi(p)$ *is the number of* 0*'s it contains, taken modulo* 2. *A* ground tree τ *is a partial function from paths into* Σ_g, *whose domain is non-empty and prefix-closed, and such that* $\tau(p0)$ *and* $\tau(p1)$ *are defined iff* $\tau(p) = \rightarrow$. *Given* $p \in \mathrm{dom}(\tau)$, *the* subtree of τ rooted at p, *written* $\tau_{|p}$, *is the tree* $q \mapsto \tau(pq)$. *A tree is* finite *iff its domain is finite. A tree is* regular *iff it has a finite number of subtrees. A* ground type *is a regular ground tree. We denote the set of ground types by*

$\mathbb{T}$. $\bot$ *(resp.* $\top$*) stands for the tree* τ *such that* $\mathrm{dom}(\tau) = \{\epsilon\}$ *and* $\tau(\epsilon) = \bot$ *(resp.* $\top$*). If* τ_0 *and* τ_1 *are trees,* $\tau_0 \to \tau_1$ *stands for the tree* τ *defined by* $\tau(\epsilon) = \to$, $\tau(0p) = \tau_0(p)$ *and* $\tau(1p) = \tau_1(p)$.

The set of ground types is equipped with a partial order, called *subtyping*.

Definition 2 *A family of orderings over ground types is defined inductively as follows. First,* $\leq_0$ *is uniformly true. Second, for any* $k \in \mathbb{N}^+$, $\tau \leq_{k+1} \tau'$ *holds iff at least one of the following is true:*

- $\tau = \bot$;
- $\tau' = \top$;
- $\exists \tau_0 \tau_1 \tau_0' \tau_1' \quad \tau = \tau_0 \to \tau_1$, $\tau' = \tau_0' \to \tau_1'$, $\tau_0' \leq_k \tau_0$ *and* $\tau_1 \leq_k \tau_1'$.

Subtyping, denoted by $\leq$*, is the intersection of these orderings.*

$(\mathbb{T}, \leq)$ forms a lattice. Its operators $\sqcup$ and $\sqcap$ can be defined in several ways, e.g. using automata products, finite approximations or a fix-point theorem. But their definition is of little interest in itself, and we shall be content with the following characterization.

Theorem 1 *The set of ground types* $\mathbb{T}$*, equipped with the subtyping relation, is a lattice. We denote its least upper bound and greatest lower bound operators by* $\sqcup$ *and* $\sqcap$*, respectively. These operators are of course associative and commutative. In addition, they are characterized by the following identities:*

$$\begin{array}{cc}
\bot \sqcup \tau = \tau & \bot \sqcap \tau = \bot \\
\top \sqcup \tau = \top & \top \sqcap \tau = \tau
\end{array}$$
$$(\tau_1 \to \tau_2) \sqcup (\tau_1' \to \tau_2') = (\tau_1 \sqcap \tau_1') \to (\tau_2 \sqcup \tau_2')$$
$$(\tau_1 \to \tau_2) \sqcap (\tau_1' \to \tau_2') = (\tau_1 \sqcup \tau_1') \to (\tau_2 \sqcap \tau_2')$$

2.2 Types

We will soon describe our type system, which is a logic for deriving typing judgments about programs. We wish the system to enjoy most general typings: so, informally speaking, the set of a program's ground types should be expressible with a single typing judgment. That is, a possibly infinite set of possibly infinite ground types should be described by a single logical assertion—which must be finite. To allow this, we now introduce *types*, which may contain *type variables*. Using recursive constraints on variables, any given ground type can be finitely described; in addition, quantification over type variables allows giving a finite description of certain infinite sets of ground types. To sum up, type variables serve two different purposes: they encode recursive structure, and they allow polymorphism.

Definition 3 *Let* $\mathcal{V}$ *be a denumerable set of* type variables*, denoted by* α*,* β*, etc. The set of* types*, denoted by* $\mathcal{T}$*, is defined by*

$$\tau ::= \alpha \mid \bot \mid \top \mid \tau \to \tau$$

A type is said to be constructed *iff it is not a variable.*

Definition 4 *A* ground substitution *is a total mapping from type variables to ground types. A* renaming *is a bijection between two subsets of* $\mathcal{V}$*. Ground substitutions and renamings are straightforwardly extended to types.*

Definition 5 *The sets of* positive *and* negative free variables *of a type τ, respectively denoted by* $\mathrm{fv}^+(\tau)$ *and* $\mathrm{fv}^-(\tau)$, *are defined by*

$$\begin{array}{rlrl}
\mathrm{fv}^+(\alpha) & = \{\alpha\} & \mathrm{fv}^-(\alpha) & = \varnothing \\
\mathrm{fv}^+(\bot) & = \varnothing & \mathrm{fv}^-(\bot) & = \varnothing \\
\mathrm{fv}^+(\top) & = \varnothing & \mathrm{fv}^-(\top) & = \varnothing \\
\mathrm{fv}^+(\tau_0 \to \tau_1) & = \mathrm{fv}^-(\tau_0) \cup \mathrm{fv}^+(\tau_1) & \mathrm{fv}^-(\tau_0 \to \tau_1) & = \mathrm{fv}^+(\tau_0) \cup \mathrm{fv}^-(\tau_1)
\end{array}$$

The set of free variables *of τ, denoted by* $\mathrm{fv}(\tau)$, *is defined by*

$$\mathrm{fv}(\tau) = \mathrm{fv}^+(\tau) \cup \mathrm{fv}^-(\tau)$$

2.3 Constrained type schemes

Like that of ML, our type system offers `let`-polymorphism. Thus, typing judgments associate programs not merely with types, but with *type schemes.*

A constrained type scheme is essentially a type—its *body*—where variables are allowed to assume arbitrary values, within the limits of certain *constraints.* Hence, a type scheme represents a set of ground types, which is obtained—roughly speaking—by applying all solutions of the constraints to the body.

Constraint-based type systems have appeared in order to deal with subtyping assumptions in typing judgments. However, they can also describe classic equality-based systems, such as ML itself. For this reason, we will give two variants of our type system: one where constraints are to be interpreted as equations, and one where they truly denote subtyping relationships. The former is of course simpler, but still interesting, because it presents many common points with the latter, especially in the area of constraint simplification, where the same broad concepts apply. Studying it first will allow us to identify methods which generally apply to all constraint-based systems, as opposed to those specific to our interpretation of subtyping.

However, even in the simpler case, our system exhibits a significant departure from ML, because, following Trifonov and Smith [TS96], we choose a formulation where all type schemes are *closed*, i.e. with no free type variables.

This decision gives rise to a system where type schemes are stand-alone: their meaning does not depend on any external assumptions. (Defining the denotation of a type scheme *with* free type variables would require supplying an assignment of ground types to these free variables.) It also removes the need to maintain a global constraint set, constraining those variables which are free in the environment, since there are none. Furthermore, we will notice that two distinct branches of a type inference derivation now share no type variables. These properties lead to a simplification, and a better understanding, of the theory, as well as to a more straightforward implementation.

In ML, it is incorrect to generalize over a type variable if it appears free in the environment. So, how can we hope to be able to universally quantify over all variables? The solution is to move the environment into the type scheme itself. This presentation is known as *λ-lifting*, for it essentially amounts to pretending that we are dealing solely with closed program terms. Its functioning will be detailed by the typing rules (see section 2.4). More precisely, information concerning `let`-bound variables remains stored inside an external environment, while information about λ-bound variables appears in a *context* which is part of type schemes.

Definition 6 *Assume an ordering $\prec$ on ground types, which can be chosen to be $=$ or $\leq$.*

The forthcoming definitions depend on the choice on $\prec$, so we end up defining two variants of the type system, based either on equality or on subtyping.

Definition 7 *Assume a denumerable set of* λ-identifiers*, denoted by* x*,* y*, ...*

Definition 8 *A* ground context *is a finite map from* λ*-identifiers to ground types. The ordering* $\prec$ *is extended to ground contexts as follows:*

$$A \prec A' \quad \iff \quad \forall x \in \mathrm{dom}(A') \quad x \in \mathrm{dom}(A) \wedge A(x) \prec A'(x)$$

A ground ctype *is a pair of a ground context* A *and a ground type* τ*, written* $A \Rightarrow \tau$*. The ordering* $\prec$ *is extended to ground ctypes by setting*

$$(A \Rightarrow \tau) \prec (A' \Rightarrow \tau') \quad \iff \quad (A' \prec A) \wedge (\tau \prec \tau')$$

Definition 9 *A* context A *is a finite map from* λ*-identifiers to types. If* $x \notin \mathrm{dom}(A)$*, then* $A[x \mapsto \tau]$ *is the context which extends* A *by mapping* x *to* τ*. if* $x \in \mathrm{dom}(A)$*, then* $A \setminus x$ *is the context which is undefined at* x *and which coincides with* A *elsewhere. A* ctype *is a pair of a context* A *and a type* τ*, written* $A \Rightarrow \tau$*. Ground substitutions are extended straightforwardly to contexts and ctypes.*

Definition 10 *A* constraint *is a pair of types, written* $\tau \prec \tau'$*. A ground substitution* ρ *is a* solution *of it iff* $\rho(\tau) \prec \rho(\tau')$*; we then write* $\rho \vdash \tau \prec \tau'$*. When* $\prec$ *stands for* $\leq$*, we say* ρ *is a* k-solution *of* $\tau \leq \tau'$ *iff* $\rho(\tau) \leq_k \rho(\tau')$*; we then write* $\rho \vdash_k \tau \prec \tau'$*. We write* $\rho \vdash C$ *(resp.* $\rho \vdash_k C$*) when* $\rho \vdash c$ *(resp.* $\rho \vdash_k c$*) holds for all* $c \in C$*.*

Definition 11 Type schemes *are defined by*

$$\sigma ::= A \Rightarrow \tau \mid C$$

where A *denotes a context,* τ *a type and* C *a constraint set. (The symbol* $\mid$ *should be interpreted here as a literal, not as a choice.) Let* $\mathrm{fv}(\sigma)$ *stand for the set of all type variables which appear in* A*,* τ *or* C*. The* order *of* σ *is* $|\,\mathrm{fv}(\sigma)\,|$*.*

Intuitively speaking, all variables of a type scheme are to be considered as universally quantified. However, we shall not write any quantifiers explicitly. Formally speaking, no implicit α-conversion is allowed on type schemes; α-conversion shall be dealt with explicitly. This decision allows a rigorous description of the way fresh variables and renamings are handled.

We now define the denotation of a type scheme as a set of ground ctypes.

Definition 12 *The* denotation $[\![\sigma]\!]$ *of a type scheme* σ *is the union of the* $\prec$*-upper cones generated by its ground instances. That is,*

$$[\![A \Rightarrow \tau \mid C]\!] = \{A' \Rightarrow \tau' \,;\, \exists \rho \vdash C \quad \rho(A \Rightarrow \tau) \prec A' \Rightarrow \tau'\}$$

Informally speaking, a type scheme is simply a way of describing a set of ground ctypes. Thus, its denotation is precisely this set, i.e. the set of ground ctypes which the program would receive in a system without polymorphism. Since subtyping allows weakening a program's ground ctype, it is natural for a scheme's denotation to be upward closed, hence the use of upper cones in its definition. It is now clear that a type scheme is more general than another one iff it represents a larger set of ground ctypes; thus, subsumption between type schemes is defined as set-theoretic inclusion of their denotations, as follows.

Definition 13 *Given two type schemes* σ_1 *and* σ_2*, the former is said to be* more general *than the latter iff* $[\![\sigma_1]\!] \supseteq [\![\sigma_2]\!]$*; we shall then write* $\sigma_1 \preccurlyeq \sigma_2$*. In other words,* σ_1 *is more general than* σ_2 *iff for any ground instance of* σ_2*, there exists a ground instance of* σ_1 *which is smaller with respect to* $\prec$*. Formally,*

$$(A_1 \Rightarrow \tau_1 \mid C_1) \preccurlyeq (A_2 \Rightarrow \tau_2 \mid C_2)$$

is thus equivalent to

$$\forall \rho_2 \vdash C_2 \quad \exists \rho_1 \vdash C_1 \quad \rho_1(A_1 \Rightarrow \tau_1) \prec \rho_2(A_2 \Rightarrow \tau_2)$$

We write $\sigma_1 \approx \sigma_2$ *when* $\sigma_1 \preccurlyeq \sigma_2$ *and* $\sigma_2 \preccurlyeq \sigma_1$*.*

The relation $\preccurlyeq$ was introduced in [TS96], where it is written $\leq^{\forall}$.

2.4 Typing rules

The language we are interested in is core ML, that is, a λ-calculus equipped with a `let` construct. For the sake of simplicity, we separate λ-bound identifiers from `let`-bound ones, by placing them in two distinct syntactic classes.

Definition 14 *Assume given a denumerable set of* `let`-identifiers*, denoted by* X, Y, ... Expressions *are defined by*

$$e ::= x \mid \lambda x.e \mid e\,e \mid X \mid \texttt{let}\, X = e \,\texttt{in}\; e$$

Definition 15 Environments *are defined by*

$$\Gamma ::= \varnothing \mid \Gamma; X : \sigma$$

Environment access is defined, as usual, by

$$(\Gamma; X : \sigma)(X) = \sigma \qquad\qquad (\Gamma; Y : \sigma)(X) = \Gamma(X) \quad \textit{when } X \neq Y$$

Note that environments contain information about `let`-bound variables only. Associating types to λ-bound variables is done inside type schemes, as shown by the typing rules given in figure 1.

Definition 16 *An expression* e *is* well typed *in an environment* Γ *iff there exists a type scheme* σ*, whose denotation is non-empty, such that* $\Gamma \vdash e : \sigma$*.*

Recall that the denotation of a type scheme $A \Rightarrow \tau \mid C$ is non-empty if and only if C admits a solution. Thus, to determine whether a program is well-typed, one must not only build a typing derivation, but also make sure that it yields a solvable constraint set.

Also, recall that the relation $\preccurlyeq$, as well as the notion of denotation, depend on our choice of $\prec$. So, there are two variants of this type system, one based on equality, the other based on subtyping.

In this system, one rule is devoted to each syntactic construct; in addition, rule (Sub), called the *subsumption* rule, allows reformulating the type scheme at any point, with great flexibility. It allows arbitrary α-conversions, as well as simplifications of the constraint system.

These rules aim at simplicity. Still, we expect the unfamiliar reader to wonder why contexts are made part of type schemes. Let us explain. Contexts are part of the λ-lifting mechanism, which allows us to emulate the behavior of ML, while using universally quantified variables exclusively. But how can we express "monomorphic" types, since all variables must be universally quantified? Here is an example. Consider the expression

$$\lambda x.\texttt{let}\, Y = x \,\texttt{in}\; \lambda f.(f\,Y\,Y)$$

Let us type this expression in ML. Y's type is a monomorphic variable α. So, the two uses of Y do not involve any instantiation, and the expression's type is $\alpha \rightarrow (\alpha \rightarrow \alpha \rightarrow \epsilon) \rightarrow \epsilon$. In our system, on the contrary, Y's type is $(x : \alpha) \Rightarrow \alpha$, according to rule (Var). Here,

$$\frac{A(x) = \tau}{\Gamma \vdash x : A \Rightarrow \tau \mid C} \qquad (\textsc{Var})$$

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

$$\frac{\Gamma \vdash e_1 : A \Rightarrow \tau_2 \rightarrow \tau \mid C \qquad \Gamma \vdash e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash e_1\, e_2 : A \Rightarrow \tau \mid C} \qquad (\textsc{App})$$

$$\frac{\Gamma(X) = \sigma}{\Gamma \vdash X : \sigma} \qquad (\textsc{LetVar})$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let}\, X = e_1 \,\texttt{in}\, e_2 : \sigma_2} \qquad (\textsc{Let})$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \preccurlyeq \sigma'}{\Gamma \vdash e : \sigma'} \qquad (\textsc{Sub})$$

Figure 1: Typing rules

α is (implicitly) universally quantified. So, if one were free to use rule (Sub) to perform renamings, the two uses of Y could yield two distinct schemes $(x : \beta) \Rightarrow \beta$ and $(x : \gamma) \Rightarrow \gamma$. However, the typing rule for function application requires that its two branches share the same context. So, necessarily, β and γ must be the same variable, and the sub-expression $\lambda f.(f\,Y\,Y)$ has type $(x : \beta) \Rightarrow (\beta \rightarrow \beta \rightarrow \epsilon) \rightarrow \epsilon$. Once the λ-abstraction is performed, the whole expression receives type $\beta \rightarrow (\beta \rightarrow \beta \rightarrow \epsilon) \rightarrow \epsilon$, as expected. To sum up, all variables which appear in the context actually have monomorphic behavior; this is caused by a sharing constraint on contexts, which is enforced whenever two branches of the derivation are brought together. So, we are able to do away with the notion of unquantified type variable; nonetheless, the system is correct, as stated below.

Statement 1 *Let e be an expression satisfying the following two conditions:*

- *each λ-identifier is bound at most once within e;*
- *if* `let` $X = e_1$ `in` e_2 *is a sub-expression of e, then X appears free within e_2.*

Assume e to be well-typed in the empty environment. Then e is safe with respect to a call-by-value semantics of the language.

The above two conditions are technical. The first one is made necessary by the way we "lift" λ-binders through `let` binders; the second one is required to make rule (Let) safe with respect to a call-by-value semantics [TS96]. They are not restrictive, since any expression can be rewritten, without altering its semantics, so as to satisfy them. Indeed, to satisfy the first condition, an appropriate renaming of λ-bound variables shall do; to fulfill the second one, it suffices to replace the construct `let` $X = e_1$ `in` e_2 with `let` $X = e_1$ `in` $(\lambda\_.e_2)\,X$ whenever X does not appear free in e_2.

$$\frac{\alpha, \beta \notin F}{[F]\ \Gamma \vdash_{\mathrm{I}} x : [F \cup \{\alpha, \beta\}]\ (x \mapsto \alpha) \Rightarrow \beta \mid \{\alpha \prec \beta\}} \quad (\textsc{Var}_{\mathrm{I}})$$

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

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

$$\frac{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\ \Gamma \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2 \\ [F'']\ A_1 \wedge A_2 = [F''']\ A \mid C_m \\ \alpha, \beta \notin F''' \\ C = C_1 \cup C_2 \cup C_m \cup \{\alpha \prec \beta, \tau_1 \prec \tau_2 \to \alpha\}\end{array}}{[F]\ \Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F''' \cup \{\alpha, \beta\}]\ A \Rightarrow \beta \mid C} \quad (\textsc{App}_{\mathrm{I}})$$

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

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

Figure 2: Type inference rules

The reader may point out that these conditions are not preserved by reduction, which poses a problem when attempting to express a subject reduction property. However, we shall not attempt to prove statement 1 in this paper, because we choose to focus on the issue of constraint simplification. We remove these conditions and give a full subject reduction proof—for the case where $\prec$ stands for the subtyping relation—in [Pot98b, Pot98c]. Doing so requires a more complex formulation of the type system, which is why we choose simplicity here.

Lastly, one may notice that safety—with respect to any semantics—comes for free in the pure λ-calculus, since there are no possible execution errors. However, the safety proof given in [Pot98b, Pot98c] is not based on this remark, and can be extended to more complex calculi.

2.5 Type inference rules

The typing rules introduced above cannot be directly used to infer an expression's type. First, they are not syntax directed, because of rule (Sub). Second, rule (App) places sharing constraints on its premises: A, τ_2 and C appear in both premises. So, we now define a set of type inference rules, which specify a type reconstruction algorithm; they are given in figure 2. The main difference with the typing rules is the disappearance of the subtyping rule, which has been built into the application rule. (The "$[F]$" annotations, although noisy, are trivial; they allow an explicit treatment of fresh variables.)

Rule (App$_{\mathrm{I}}$) uses the following definition, which describes how contexts are brought together whenever two branches of the derivation meet.

Definition 17 *The assertion $[F]\ A_1 \wedge A_2 = [F']\ A \mid C$ stands, by definition, for the following conjunction:*

- $\mathrm{dom}(A) = \mathrm{dom}(A_1) \cup \mathrm{dom}(A_2)$*;*
- $\forall x \in \mathrm{dom}(A_1) \cap \mathrm{dom}(A_2) \quad A(x) \in \mathcal{V} \setminus F$*;*
- $\forall x \in \mathrm{dom}(A_1) \setminus \mathrm{dom}(A_2) \quad A(x) = A_1(x)$*;*
- $\forall x \in \mathrm{dom}(A_2) \setminus \mathrm{dom}(A_1) \quad A(x) = A_2(x)$*;*
- $F' = F \cup \{A(x)\,;\, x \in \mathrm{dom}(A_1) \cap \mathrm{dom}(A_2)\}$*;*
- $C = \{A(x) \prec A_i(x)\,;\, x \in \mathrm{dom}(A_1) \cap \mathrm{dom}(A_2), i \in \{1,2\}\}$.

Informally speaking, we say that A is the *meet* of the two contexts A_1 and A_2. It is essentially the least demanding context which guarantees that both A_1's and A_2's expectations about the expression's runtime environment are fulfilled.

The type inference rules are sound and complete with respect to the typing rules—that is, they infer a most general type scheme for the expression at hand.

Statement 2 *The type inference rules are correct with respect to the typing rules; that is, $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma$ implies $\Gamma \vdash e : \sigma$.*

Statement 3 *The type inference rules are complete with respect to the typing rules. That is, if $\Gamma \vdash e : \sigma$ then, for any finite $F \subseteq \mathcal{V}$, there exists a finite $F' \subseteq \mathcal{V}$ and a type scheme $\sigma' \preccurlyeq \sigma$ such that $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma'$. Furthermore, σ' is uniquely determined, up to a renaming, by Γ and e.*

These rules are very close, in spirit, to those of Trifonov and Smith [TS96]. However, we have brought a few subtle, but important modifications, so as to produce type schemes which satisfy a couple of interesting properties. First, any such scheme is made up of *small terms* only. Second, when $\prec$ stands for the subtyping relationship, the scheme contains no *bipolar* variables. Both properties shall be used throughout the paper to simplify statements and proofs. We prove the former here; the latter is introduced in section 4.2.

Definition 18 *A* small term *is a type term of the form $\bot$, $\top$ or $\alpha_0 \rightarrow \alpha_1$, i.e. a term whose strict sub-terms are type variables. A type scheme $A \Rightarrow \tau \mid C$ is* made up of small terms *iff it satisfies the following conditions:*

- *for all $x \in \mathrm{dom}(A)$, $A(x)$ is a type variable;*
- *τ is a type variable;*
- *for all $(\tau \prec \tau') \in C$, either τ and τ' are type variables, or one is a variable and the other is a small term.*

Theorem 2 *If $[\Gamma]\ F \vdash_{\mathrm{I}} e : [F']\ \sigma$, then σ is made up of small terms.*

Proof. Straightforward induction on the structure of the type inference derivation. $\square$

The small terms property allows reasoning about sharing between sub-terms, and is a key requirement in our formulation of minimization (see section 4.5). It is already to be found, for instance, in the theory of unification [Hue76]. Among works more closely related to ours, Aiken and Wimmers [AW92] and Palsberg [Pal95] use a similar convention.

3 Simplifying equality constraints

We are done introducing our type inference system, which specifies how to associate a constrained type scheme with a given program. We shall now focus our attention onto the main issue of interest here: how to simplify an inferred type scheme, without affecting its meaning. We begin, in this section, with the simpler case where $\prec$ is chosen to be $=$, i.e. where constraints are equations.

In common presentations of equality-based type systems, no equations appear; instead, their most general unifiers are computed directly. Here, however, we explicitly deal with equality constraints, so as to highlight the similarity with the more complex case of subtyping constraints.

In this section, and in this section only, we choose to deal with simplified type schemes, of the form $\tau \mid C$. We shall not concern ourselves with contexts, because their presence does not add any difficulty to the simplification issue.

3.1 Preliminaries

Let us begin with a few straightforward facts concerning term automata [KPS93].

Definition 19 *A* term automaton *is a tuple $\mathcal{A} = (Q, q_0, \delta, l)$ where:*

- *Q is a finite set of* states*,*
- *$q_0 \in Q$ is the* start state*,*
- *$\delta : Q \times \{0,1\} \to Q$ is a (partial)* transition function*,*
- *$l : Q \to \Sigma_g \cup \mathcal{V}$ is a* labeling function*,*

such that for any state $q \in Q$ and for any $i \in \{0,1\}$, $\delta(q,i)$ is defined iff $l(q) = \to$.

A state $q \in Q$ is said to be free *iff its label is a variable, i.e. $l(q) \in \mathcal{V}$. The* order *of $\mathcal{A}$ is the number of its states, i.e. $|\,Q\,|$.*

A term automaton is essentially a way of representing a type term, possibly recursive and possibly with free type variables. Such a representation is more compact than a classic tree representation, because of its ability to express sharing between nodes.

Definition 20 *Let $\mathcal{A} = (Q, q_0, \delta, l)$ be a term automaton. Extend δ to a partial function $\hat{\delta} : Q \times \{0,1\}^* \to Q$. Then, $\mathcal{A}$ describes a function $\tau_{\mathcal{A}}$ from paths into $\Sigma_g \cup \mathcal{V}$, defined by $p \mapsto l(\hat{\delta}(q_0, p))$.*

Rather than viewing an automaton as a type term, possibly containing type variables, we can also choose to view it as a set of ground types.

Definition 21 *Let $\mathcal{A}$ be a term automaton. The* ground instance *of $\mathcal{A}$ through a ground substitution ρ is the ground type τ defined as follows: for all paths p,*

- *if $\tau_{\mathcal{A}}(p) \in \Sigma_g$, then $\tau(p) = \tau_{\mathcal{A}}(p)$;*
- *if $\tau_{\mathcal{A}}(p)$ is a type variable $\alpha \in \mathcal{V}$, then $\tau_{|p} = \rho(\alpha)$.*

The denotation *of a term automaton $\mathcal{A}$ is the set of its ground instances.*

Statement 4 *A term automaton's denotation is non-empty.*

Statement 5 *Two term automata $\mathcal{A}$ and $\mathcal{B}$ have the same denotation iff $\tau_{\mathcal{A}}$ and $\tau_{\mathcal{B}}$ are equal up to a renaming of variables.*

$$\frac{\alpha = e \qquad \alpha = e'}{\alpha = e = e'} \qquad (\textsc{Fuse})$$

$$\frac{e = \top = \top}{e = \top} \qquad (\textsc{Decompose}_\top)$$

$$\frac{e = \bot = \bot}{e = \bot} \qquad (\textsc{Decompose}_\bot)$$

$$\frac{e = \alpha_0 \rightarrow \alpha_1 = \beta_0 \rightarrow \beta_1}{e = \alpha_0 \rightarrow \alpha_1 \qquad \alpha_0 = \beta_0 \qquad \alpha_1 = \beta_1} \qquad (\textsc{Decompose}_\rightarrow)$$

Figure 3: Solving multi-equations

3.2 Simplifying multi-equations

The type inference algorithm generates equations. However, it is best to introduce a more general notion of *multi-equation*, as is often done in works on unification [Hue76, JK90, Rém92].

Definition 22 *A* multi-equation *is a set of terms $\{\tau_1, \dots, \tau_n\}$, written $\tau_1 = \dots = \tau_n$. An equality constraint $\tau_1 = \tau_2$ can be viewed as a multi-equation. The notion of solution is extended straightforwardly to multi-equations and to sets thereof. A multi-equation is* made up of small terms *iff all of its members are variables or small terms.*

In order to determine that a program is well-typed, we need to make sure that its associated type scheme has a non-empty denotation, i.e. that its constraint set has a solution. This is done by applying a set of rewriting rules to the multi-equation set, as follows.

Theorem 3 *Consider a type scheme $\sigma = \alpha_0 \mid C$, where C is a multi-equation set, made up of small terms. Rewrite C according to the rules of figure 3, until none applies; let C' denote the result of this process. Then, C' is also made up of small terms, and has the same solutions as C. Furthermore,*

- *if C' contains at least one multi-equation of the form $e = \tau = \tau'$, where neither τ nor τ' are variables, then $[\![\sigma]\!]$ is empty;*
- *otherwise, C' is said to be in* canonical form. *It can easily be viewed as a term automaton, whose order equals that of σ, and whose denotation coincides with $[\![\sigma]\!]$. As a corollary, $[\![\sigma]\!]$ is non-empty.*

Proof. With an appropriate definition of weight (e.g. give weight 1 to variables, $\bot$ and $\top$, and weight 2 to the $\rightarrow$ symbol), it is easy to verify that each rewriting rule causes the total weight of the multi-equation set to decrease. Hence, the process must terminate. Each rewriting rule obviously preserves the solution space, as well as the small terms property.

Assume C' contains a multi-equation with two non-variable terms. Then, these terms must have incompatible head constructors, because none of the decomposition rules in figure 3 applies. So, C' has no solution. On the other hand, assume C' is in canonical form;

then, each multi-equation contains at most one non-variable term. Additionally, because rule (FUSE) no longer applies, each variable appears in at most one multi-equation. In each multi-equation, choose a unique representative, equal to its non-variable term when it has one, and to an arbitrary member otherwise. For each $\alpha \in \mathrm{fv}(\sigma)$, let $\mathrm{repr}(\alpha)$ denote the representative of α's multi-equation, if α appears in some multi-equation, and α itself otherwise. Define a term automaton $\mathcal{A} = (Q, q_0, \delta, l)$ as follows:

- $Q = \mathrm{fv}(\sigma)$;
- $q_0 = \alpha_0$;
- for $i \in \{0, 1\}$, $\delta(\alpha, i) = \alpha_i$ when $\mathrm{repr}(\alpha) = \alpha_0 \to \alpha_1$;
- $l(\alpha) = \mathrm{repr}(\alpha)(\epsilon)$.

It is straightforward to verify that the ground instances of this automaton are exactly those of σ; hence, its denotation coincides with $[\![\sigma]\!]$. The non-emptiness result stems from statement 4. □

Theorem 3 yields an algorithm to determine whether a type scheme has a non-empty denotation; this makes type inference decidable. However, it also shows that a canonical type scheme can be viewed as a term automaton; we now establish the converse, showing that the two notions are equivalent.

Theorem 4 *Let $\mathcal{A}$ be a term automaton. Then, there exists a canonical type scheme σ, of the same order, whose denotation coincides with $\mathcal{A}$'s.*

Proof. Assume $\mathcal{A} = (Q, q_0, \delta, l)$. Choose some injective map $q \in Q \mapsto \alpha_q \in \mathcal{V}$. Define a multi-equation set C by

- for each $\alpha \in \mathrm{rng}(l)$, $\{\alpha_q\,;\, l(q) = \alpha\} \in C$;
- for each $q \in Q$ such that $l(q) = \bot$, $\{\alpha_q, \bot\} \in C$;
- for each $q \in Q$ such that $l(q) = \top$, $\{\alpha_q, \top\} \in C$;
- for each $q \in Q$ such that $l(q) = \to$, $\{\alpha_q, \alpha_{\delta(q,0)} \to \alpha_{\delta(q,1)}\} \in C$.

Define $\sigma = \alpha_{q_0} \mid C$. It is straightforward to verify that $[\![\sigma]\!]$ coincides with $\mathcal{A}$'s denotation. □

The equivalence between canonical type schemes and term automata gives rise to an essential idea: the well-known minimization procedure for finite-state automata carries over to canonical type schemes.

Theorem 5 *Let σ be a canonical type scheme. Among the canonical type schemes equivalent to σ, there is one of minimal order, which can be computed in time $O(n \log n)$, where n is the order of σ.*

Proof. Thanks to theorems 3 and 4, we can state the problem in terms of automata. Given an automaton $\mathcal{A}$, of order n, we must compute an automaton $\mathcal{B}$, whose denotation equals that of $\mathcal{A}$, and which is minimal for this property. According to statement 5, we can equivalently require $\tau_{\mathcal{A}} = \tau_{\mathcal{B}}$. Hence, the problem simply consists in minimizing the labeled finite state automaton $\mathcal{A}$, which can be done in time $O(n \log n)$ [Hop71]. □

$$\alpha_0 \quad \text{where} \quad \left\{ \begin{array}{ll} \alpha_0 = \alpha_1 = \alpha_2 \rightarrow \alpha_3 & \alpha_5 = \alpha_6 = \top \\ \alpha_3 = \alpha_5 \rightarrow \alpha_6 & \alpha_2 = \alpha_4 \end{array} \right.$$

Figure 4: A sample type scheme, in canonical form

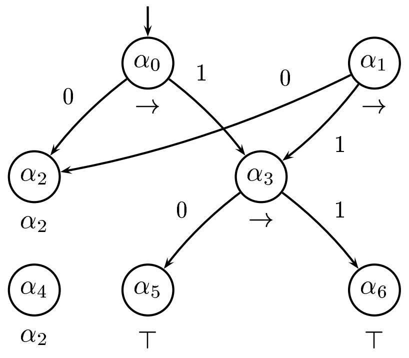

Figure 5: The same, viewed as an automaton

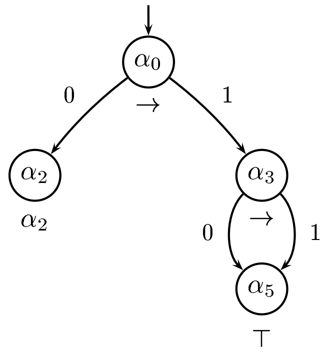

Figure 6: The minimized automaton

$$\alpha_0 \quad \text{where} \quad \left\{ \begin{array}{ll} \alpha_0 = \alpha_2 \rightarrow \alpha_3 & \alpha_5 = \top \\ \alpha_3 = \alpha_5 \rightarrow \alpha_5 & \end{array} \right.$$

Figure 7: The same, viewed again as a type scheme

Thus, it is possible to minimize the number of variables of a type scheme—which we adopt as a measure of its complexity—in quasi-linear time. Figures 4 to 7 illustrate this procedure. Our starting point is a type scheme whose multi-equation set has been put in canonical form after the rules of figure 3. Theorem 3 allows us to view it as an automaton (figure 5), which we then minimize. Minimization is a well-known, two-step process: first eliminate any states not reachable from the start state, then merge equivalent states. In broad terms, two states are equivalent if their labels are equal and if they carry transitions, with equal labels, whose end states are in turn equivalent. This process yields the automaton shown in figure 6. Finally, theorem 4 allows us to turn this automaton back into a type scheme. Of course, thinking in terms of automata allows a simple explanation of the process, but isn't mandatory; the minimization procedure can be described directly in terms of multi-equations, if one so wishes.

How does this procedure compare to the usual resolution process used in ML type inference? An ML type checker computes the most general solution of the equation set, using unification. This essentially amounts to putting the type scheme in canonical form, by applying the rules of figure 3, then merging all members of a single multi-equation. Our algorithm goes one step further, since variables belonging to different multi-equations can also be merged, provided they stand for equivalent states of the automaton. In fact, our simplification procedure is complete—it yields a type scheme with a minimal number of variables. Since our schemes are made up of small terms, this is a meaningful measure of their complexity.

Theoretically speaking, our decision of working with small terms allows us to easily highlight the isomorphism between type schemes and term automata. More intuitively, one might say that breaking a large type term down into a series of small terms, linked together by equations, essentially amounts to labelling each node of the original term with a type variable. Identifying variables is then tantamount to sharing nodes in the original type term, thus yielding a more compact representation. Of course, a user is likely to prefer a more readable representation, with fewer variables and larger terms; it is easy to revert to such a representation for display purposes. (For instance, the type scheme of figure 7 can be printed as $\alpha_2 \rightarrow \top \rightarrow \top$.) This is already the case in typical ML implementations, where types are internally represented by directed acyclic graphs, but printed as trees. It is important to carefully distinguish the two representations, since the latter is typically exponentially larger. In other words, an internal representation must favor efficiency; converting to an external representation, which offers better readability, must be delayed until the result is ready for the user to be seen.

To conclude, we have studied a complete simplification procedure for constrained type schemes, in the case where constraints are equations. It consists of three main steps: putting the constraints in canonical form, eliminating unreachable variables, and merging equivalent variables. We shall now move on to the case of subtyping, and discover that, although details become more complex, the same broad ideas apply.

4 Simplifying subtyping constraints

4.1 Solving constraints

As in section 3, our first task is to find an algorithm to decide whether a given constraint set has a solution. Indeed, doing so is required to determine whether a program is well-typed. Our goal, in this section, is to describe such an algorithm.

We begin with a fundamental technical result, which describes a weak, sufficient condition for a constraint set to have a solution. It will form the basis for the proof of the constraint solving algorithm. We prove a fairly powerful version of this result, allowing ground constants to appear in constraints. (Since ground types may be infinite, writing down these extended constraints would require some finite representation; however, we will not need to do so.) Thanks to this generalization, this result also forms the basis for the proof of the garbage collection algorithm (see section 4.3).

Definition 23 *A constraint set* with ground constants *is a set C of subtyping constraints of the form $\tau \leq \tau'$, where τ and τ' are either two variables, one variable and a small term, or one variable and a ground type. Define the assertion $C \Vdash^{+1} \tau \leq \tau'$ to mean*

$$\forall k \geq 0 \quad \forall \rho \vdash_k C \quad \rho \vdash_{k+1} \tau \leq \tau'$$

Define $C^{\downarrow}(\alpha) = \{\tau\,;\, \tau \notin \mathcal{V} \wedge \tau \leq \alpha \in C\}$ and $C^{\uparrow}(\alpha) = \{\tau\,;\, \tau \notin \mathcal{V} \wedge \alpha \leq \tau \in C\}$. C is said to be weakly closed *iff the following conditions are met:*

1. *$\alpha \leq \beta \in C$ and $\beta \leq \gamma \in C$ imply $\alpha \leq \gamma \in C$;*
2. *$\alpha \leq \beta \in C$ and $\tau \in C^{\downarrow}(\alpha)$ imply $\exists \tau' \in C^{\downarrow}(\beta) \quad C \Vdash^{+1} \tau \leq \tau'$;*
3. *$\alpha \leq \beta \in C$ and $\tau' \in C^{\uparrow}(\beta)$ imply $\exists \tau \in C^{\uparrow}(\alpha) \quad C \Vdash^{+1} \tau \leq \tau'$;*
4. *$\tau \in C^{\downarrow}(\alpha)$ and $\tau' \in C^{\uparrow}(\alpha)$ imply $C \Vdash^{+1} \tau \leq \tau'$.*

Theorem 6 *Let C be a constraint set with ground constants. If C is weakly closed, then C has a solution.*

Proof. Note that this proof only uses conditions 2 and 4 of definition 23. The other conditions shall be required by further theorems, such as theorem 11.

Let $V = \text{fv}(C)$. Consider the set $\mathbb{T}^V$ of ground substitutions of domain V. We define a map S from $\mathbb{T}^V$ into itself by

$$\rho \mapsto \left(\alpha \mapsto \bigsqcup_{\tau \in C^{\downarrow}(\alpha)} \rho(\tau)\right)$$

Assuming $\mathbb{T}^V$ is viewed as a metric space, equipped with the usual distance between (tuples of) infinite trees [Cou83], it is easy to verify that S is $\frac{1}{2}$-contractive. Thus, it has a unique fix-point ρ.

We shall now verify that ρ is a solution of C. This is done by proving that it is a k-solution of C, for all $k \geq 0$, by induction over k. The base case is immediate, since $\leq_0$ is uniformly true (see definition 2). It remains to prove, assuming $\rho \vdash_k C$, that $\rho \vdash_{k+1} C$.

Consider a constraint of the form $\alpha \leq \beta \in C$. Because C satisfies condition 2 of definition 23, we have $\forall \tau \in C^{\downarrow}(\alpha) \quad \exists \tau' \in C^{\downarrow}(\beta) \quad C \Vdash^{+1} \tau \leq \tau'$. Since $\rho \vdash_k C$, this implies $\forall \tau \in C^{\downarrow}(\alpha) \quad \exists \tau' \in C^{\downarrow}(\beta) \quad \rho(\tau) \leq_{k+1} \rho(\tau')$, which in turn entails $(\bigsqcup_{\tau \in C^{\downarrow}(\alpha)} \rho(\tau)) \leq_{k+1} (\bigsqcup_{\tau' \in C^{\downarrow}(\beta)} \rho(\tau'))$. This statement is none other than $\rho(\alpha) \leq_{k+1} \rho(\beta)$.

Next, consider a constraint of the form $\tau \leq \alpha \in C$, where $\tau \notin \mathcal{V}$. Then, $\tau \in C^{\downarrow}(\alpha)$. So, by definition of ρ, $\rho(\tau) \leq \rho(\alpha)$. In particular, $\rho(\tau) \leq_{k+1} \rho(\alpha)$.

Finally, consider a constraint of the form $\alpha \leq \tau' \in C$, where $\tau' \notin \mathcal{V}$. Then, $\tau' \in C^{\uparrow}(\alpha)$. Pick some $\tau \in C^{\downarrow}(\alpha)$. Then, condition 4 of definition 23, together with our induction hypothesis, yield $\rho \vdash_{k+1} \tau \leq \tau'$, i.e. $\rho(\tau) \leq_{k+1} \rho(\tau')$. Since this holds for all $\tau \in C^{\downarrow}(\alpha)$, we also have $(\bigsqcup_{\tau \in C^{\downarrow}(\alpha)} \rho(\tau)) \leq_{k+1} \rho(\tau')$, i.e. $\rho(\alpha) \leq_{k+1} \rho(\tau')$. This concludes the proof. □

Theorem 6 is a nice tool to exhibit solutions of a constraint set. However, it is not clear, given an arbitrary constraint set, how it can be put in weakly closed form. So, we shall now define a stronger, but simpler, notion of closure, which can be computed more easily. This is the notion originally proposed by Eifrig, Smith and Trifonov [EST95b].

Definition 24 *The partial function subc, defined as follows, breaks a constraint whose members are variables or small terms down into a set of equivalent constraints:*

$$\begin{array}{ll} \mathrm{subc}(\alpha \leq \tau) = \{\alpha \leq \tau\} & \mathrm{subc}(\tau \leq \alpha) = \{\tau \leq \alpha\} \\ \mathrm{subc}(\bot \leq \tau) = \varnothing & \mathrm{subc}(\tau \leq \top) = \varnothing \end{array}$$
$$\mathrm{subc}(\alpha_0 \rightarrow \alpha_1 \leq \alpha'_0 \rightarrow \alpha'_1) = \{\alpha'_0 \leq \alpha_0, \alpha_1 \leq \alpha'_1\}$$

Definition 25 *Let C be a constraint set, made up of small terms. C is said to be* closed *iff whenever $\{\tau \leq \alpha, \alpha \leq \tau'\} \subseteq C$,* subc$(\tau \leq \tau')$ *is defined and included in C. From now on, a type scheme $A \Rightarrow \tau \mid C$ is said to be closed iff C is closed.*

In plain words, the above definition means that a constraint set is closed iff it is stable through a combination of transitivity and structural decomposition. Let us now verify, as announced, that closure entails weak closure; which means, considering theorem 6, that any closed constraint set admits a solution.

Theorem 7 *Any closed constraint set C is weakly closed.*

Proof. It is clear that C satisfies condition 1 of definition 23.

Assume $\alpha \leq \beta \in C$. Let $\tau \in C^{\downarrow}(\alpha)$. Because C is closed, $\mathrm{subc}(\tau \leq \beta) = \{\tau \leq \beta\} \subseteq C$. So, $\tau \in C^{\downarrow}(\beta)$. This is sufficient to establish condition 2 of definition 23; just pick $\tau' = \tau$. Symmetrically, condition 3 is satisfied.

Now, assume $\tau \in C^{\downarrow}(\alpha)$ and $\tau' \in C^{\uparrow}(\alpha)$. Because C is closed, $\mathrm{subc}(\tau \leq \tau')$ is defined and part of C. Thus, any k-solution of C is, in particular, a k-solution of $\mathrm{subc}(\tau \leq \tau')$. Moreover, considering the definition of subc, it is easy to verify that any k-solution of $\mathrm{subc}(\tau \leq \tau')$ is a $(k+1)$-solution of $\tau \leq \tau'$. Condition 4 of definition 23 ensues. □

To conclude this section, we present an algorithm which puts a given constraint set in closed form, if it has a solution, and fails otherwise. This algorithm is used to determine whether a given program is well-typed. Its bad complexity: $O(n^3)$, as well as the size of its output: $O(n^2)$, are among the main reasons why constraint simplification is required.

Theorem 8 *Let C be a constraint set, made up of small terms. Let C^2 denote*

$$C \cup \left(\bigcup_{\{\tau \leq \alpha, \alpha \leq \tau'\} \subseteq C} \mathrm{subc}(\tau \leq \tau') \right)$$

If the sequence $C, C^2, C^4, \ldots$ is infinite, then it reaches a fix-point C^∞, which is the smallest closed constraint set containing C; its solution space is equal to C's and non-empty. (C^∞ is called the closure *of C.) Otherwise, C has no solution.*

Proof. For an arbitrary C, it is clear that C^2 is equivalent to C if it is defined, and that C has no solution otherwise (i.e. if subc is applied outside of its domain). Thus, if some element of the sequence is undefined, then C has no solution. Otherwise, the sequence must reach a fix-point C^∞, because any newly created constraint involves existing terms, and there is only a finite number of such constraints. It is clear that C^∞ is the smallest closed set containing C. According to theorem 7, C^∞ is also weakly closed; by theorem 6, it admits a solution. □

While building a type inference derivation, we wish to make sure, at every step, that the expression at hand is well-typed, so as to detect errors as soon as possible. So, we must maintain our constraint sets in closed form. This may be done incrementally, taking advantage of the fact that each type inference rule adds a few fresh constraints to a closed constraint set; an incremental algorithm is described in [Pot98c, Pot98b]. Of course, if we use such an algorithm, then our simplification algorithms must preserve the closure property; this ensures that we may perform simplifications transparently at any point.

4.2 Polarities

If σ is the type scheme associated to an expression e, it would be interesting to distinguish the type variables of σ which represent an input (i.e. some data expected by the expression e) from those which represent an output (i.e. some result supplied by e). We shall annotate each type variable with a $-$ sign in the former case, and with a $+$ sign in the latter case. Of course, it is possible for a variable to carry both signs at once; we call such a variable *bipolar*. Some variables, on the other hand, carry no sign at all; we call those *neutral*. Thus, we shall associate a pair of Boolean flags, which we call *polarity*, to each variable. This information will serve to guide all of our simplification algorithms.

Definition 26 *Consider a weakly closed type scheme $\sigma = (A \Rightarrow \epsilon \mid C)$, made up of small terms. The set of* positive *variables of σ, and the set of* negative *variables of σ, respectively denoted by $\mathrm{fv}^+(\sigma)$ and $\mathrm{fv}^-(\sigma)$, are the smallest subsets P and N of $\mathrm{fv}(\sigma)$ such that*

- $\epsilon \in P$
- $\mathrm{rng}(A) \subseteq N$
- $\forall \alpha \in P \quad \mathrm{fv}^+(C^{\downarrow}(\alpha)) \subseteq P \wedge \mathrm{fv}^-(C^{\downarrow}(\alpha)) \subseteq N$
- $\forall \alpha \in N \quad \mathrm{fv}^+(C^{\uparrow}(\alpha)) \subseteq N \wedge \mathrm{fv}^-(C^{\uparrow}(\alpha)) \subseteq P$

Polarities may be easily computed as a smallest fix-point. The time required is linear in the size of the constraint set. Indeed, visiting a variable's constructed lower (resp. upper) bounds has to be done at most once, namely when the variable first becomes positive (resp. negative). Thus, each constraint is traversed at most once; whence the result.

Trifonov and Smith [TS96] introduced polarities as a refinement of our notion of reachability [Pot96], which would only detect neutral variables, and used them to drive garbage collection (see section 4.3). However, they did not mention certain useful properties of polarities, which we shall now describe.

Intuitively speaking, each positive variable of σ represents a piece of data computed by e and accessible as a part of its result. Assume e is placed inside a context $\mathcal{C}$, yielding an expression $\mathcal{C}[e]$ whose associated scheme is σ'. $\mathcal{C}[e]$'s result might still contain some parts of e's result, meaning that the corresponding variables are still positive in σ'; others may have been dropped, meaning that the corresponding variables are no longer positive in σ'. However, any value computed by e, but inaccessible through its result, obviously remains inaccessible through $\mathcal{C}[e]$'s result; which means that any variables *not* positive in σ cannot become positive in σ'. An analogous property holds for negative variables. In other words, polarities *decrease* as one walks down a type inference derivation. This property is formalized by the following theorem.

Theorem 9 *Consider an instance of one of the type inference rules of figure 2, whose output is a type scheme σ. Pick some $\alpha \in \mathrm{fv}(\sigma)$, and assume α also appears in σ', where σ' is one of the rule's premises. Then, $\alpha \in \mathrm{fv}^+(\sigma)$ (resp. $\mathrm{fv}^-(\sigma)$) implies $\alpha \in \mathrm{fv}^+(\sigma')$ (resp. $\mathrm{fv}^-(\sigma')$).*

Proof. The only non-trivial case is that of rule (App$_\mathrm{I}$). We use the notations of figure 2. For $i \in \{1, 2\}$, let $\sigma_i = (A_i \Rightarrow \tau_i \mid C_i)$; assume C_i is closed. Define

$$P = \mathrm{fv}^+(\sigma_1) \cup \mathrm{fv}^+(\sigma_2) \cup \{\beta\}$$
$$N = \mathrm{fv}^-(\sigma_1) \cup \mathrm{fv}^-(\sigma_2) \cup \{\alpha\} \cup \mathrm{fv}(A)$$

We wish to show that P and N are conservative approximations of the polarities in σ, i.e. that they satisfy the recursive equations of definition 26. However, recall that computing polarities requires the constraint set to be closed. Thus, these equations must be applied to C^∞, not to C itself; we need some information about C^∞ in order to prove that the equations hold.

Let the assertion τ^+ stand for the conjunction $\mathrm{fv}^+(\tau) \subseteq P \wedge \mathrm{fv}^-(\tau) \subseteq N$. (The assertion τ^- is defined symmetrically.) Notice that $C_1 \cup C_2$ is closed, because these sets have disjoint domains. Let us call "new" the constraints in $C_m \cup \{\alpha \leq \beta, \tau_1 \leq \tau_2 \rightarrow \alpha\}$, as well as any constraints arising from the subsequent closure computation. It is straightforward to verify that whenever a small term τ appears on the left-hand (resp. right-hand) side of a new constraint, then τ^+ (resp. τ^-) holds.

This guarantees that the equations of definition 26, applied to $A \Rightarrow \beta \mid C^\infty$, are satisfied by P and N. Because $\mathrm{fv}^+(\sigma)$ and $\mathrm{fv}^-(\sigma)$ are the smallest solutions of these equations, we have $\mathrm{fv}^+(\sigma) \subseteq P$ and $\mathrm{fv}^-(\sigma) \subseteq N$. In particular, $\mathrm{fv}^+(\sigma) \cap \mathrm{fv}(\sigma_i) \subseteq \mathrm{fv}^+(\sigma_i)$ and $\mathrm{fv}^-(\sigma) \cap \mathrm{fv}(\sigma_i) \subseteq \mathrm{fv}^-(\sigma_i)$; which is the desired result. □

Theorem 9 guarantees that a variable's polarity decreases during its lifetime. As a corollary, if the type inference rules are written so as to never cause a fresh variable to be bipolar—and so they are—then no bipolar variables can ever appear in a type inference derivation.

Theorem 10 *Assume* $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma$. *If none of the* $\Gamma(X)$, *for* $X \in \mathrm{dom}(\Gamma)$, *contains a bipolar variable, then neither does* σ.

Proof. First, we check that whenever a fresh variable is created by one of the type inference rules, it is not bipolar. Consider, for instance, rule (Var$_{\mathrm{I}}$). It creates two variables α and β. The former appears in the context of the type scheme, while the latter appears in its body. Hence, α is negative, and β is positive. According to definition 26, polarities can only travel from a variable to a small term, so the constraint $\alpha \leq \beta$ does not cause α (resp. β) to become positive (resp. negative). Note, on the other hand, that in the type scheme $(x \mapsto \gamma) \Rightarrow \gamma$, γ *is* bipolar; splitting γ into two variables α and β, linked by a constraint, is the technical trick which allows us not to create any bipolar variables. Rule (App$_{\mathrm{I}}$) contains a similar trick.

Second, theorem 9 tells us that if a variable is bipolar at a certain point, then it must have been so since the moment it was created. According to the previous paragraph, this is impossible; whence the result. □

This result is used to simplify various definitions and proofs, in particular concerning garbage collection and canonization. Of course, we will need to prove that our simplification algorithms also cause polarities to decrease, so we can perform simplifications at any point without breaking this property.

4.3 Garbage collection

Computing the closure of a constraint set typically yields a large number of constraints. Many of them are useful as intermediate steps of the closure computation, but are no longer essential once it is over. More precisely, we shall now show that the only meaningful constraints in a closed scheme $A \Rightarrow \epsilon \mid C$ are the following:

- those which link a positive (resp. negative) variable α to an element of $C^\downarrow(\alpha)$ (resp. $C^\uparrow(\alpha)$)—they give information about the structure of a piece of data supplied (resp. expected) by the expression;
- those which link a negative variable to a positive one—they represent a possible flow of data from one of the expression's inputs to one of its outputs.

Any other constraints are superfluous, i.e. do not affect the scheme's denotation. Thus, we can simply forget about them; this process, proposed by Trifonov and Smith [TS96], is called *garbage collection*. Note that all neutral variables are discarded; in our analogy with section 3, garbage collection corresponds to the removal of unreachable nodes in a finite automaton. It does more than that, however, since it also removes certain edges between reachable nodes.

Definition 27 *Consider σ as in definition 26. The image of σ through* garbage collection, *denoted by* $\mathrm{GC}(\sigma)$*, is the type scheme $A \Rightarrow \epsilon \mid D$, where D is a subset of C defined as follows:*

- $\alpha \leq \beta \in D$ *iff* $\alpha \leq \beta \in C$, $\alpha \in \mathrm{fv}^-(\sigma)$ *and* $\beta \in \mathrm{fv}^+(\sigma)$*;*
- $D^{\downarrow}(\alpha)$ *equals* $C^{\downarrow}(\alpha)$ *if* $\alpha \in \mathrm{fv}^+(\sigma)$*, and* $\varnothing$ *otherwise;*
- $D^{\uparrow}(\alpha)$ *equals* $C^{\uparrow}(\alpha)$ *if* $\alpha \in \mathrm{fv}^-(\sigma)$*, and* $\varnothing$ *otherwise.*

Theorem 11 *Consider σ as in definition 27. Then $\sigma \approx \mathrm{GC}(\sigma)$.*

Proof. Write $\sigma' = \mathrm{GC}(\sigma)$. Since σ' has fewer constraints, it is clear that $\sigma' \preccurlyeq \sigma$. So, we need to prove $\sigma \preccurlyeq \sigma'$. According to definition 13, this is equivalent to

$$\forall \rho' \vdash D \quad \exists \rho \vdash C \quad \rho(A \Rightarrow \epsilon) \leq \rho'(A \Rightarrow \epsilon)$$

Pick some $\rho' \vdash D$. We now wish to prove that the following constraint set *with ground constants* (see definition 23) admits a solution:

$$C \cup \{\epsilon \leq \rho'(\epsilon)\} \cup \{\rho'(A(x)) \leq A(x)\,;\, x \in \mathrm{dom}(A)\}$$

We shall do so by proving that the following constraint set—which contains the previous one, according to definition 26—is weakly closed:

$$\begin{aligned} C &\cup \{\rho'(\beta) \leq \alpha\,;\, \beta \in \mathrm{fv}^-(\sigma) \wedge \beta \leq \alpha \in C^r\} \\ &\cup \{\alpha \leq \rho'(\beta)\,;\, \beta \in \mathrm{fv}^+(\sigma) \wedge \alpha \leq \beta \in C^r\} \end{aligned}$$

(where C^r denotes the reflexive closure of C, i.e. $\alpha \leq \beta \in C^r$ iff $\alpha = \beta$ or $\alpha \leq \beta \in C$). Let E denote this set.

Because C satisfies condition 1 of definition 23, so does E. Using the same property, it is easy to check that E satisfies conditions 2 and 3. There remains to check condition 4. Assume $\tau \in E^{\downarrow}(\alpha)$ and $\tau' \in E^{\uparrow}(\alpha)$. Four cases arise, depending on whether τ and τ' are small terms or ground terms:

- Both τ and τ' are small terms. Then, $\tau \in C^{\downarrow}(\alpha)$ and $\tau' \in C^{\uparrow}(\alpha)$. The result is immediate, considering C meets condition 4.
- Both τ and τ' are ground terms. Then, according to the definition of E, τ is equal to $\rho'(\beta)$, for some $\beta \in \mathrm{fv}^-(\sigma)$ such that $\beta \leq \alpha \in C^r$. Symmetrically, τ' is of the form $\rho'(\beta')$, for some $\beta' \in \mathrm{fv}^+(\sigma)$ such that $\alpha \leq \beta' \in C^r$. Because C satisfies condition 1 of definition 23, $\beta \leq \beta' \in C^r$. If $\beta = \beta'$, then $\tau = \tau'$ and the result is immediate. So, we can assume $\beta \leq \beta' \in C$. Since $\beta \in \mathrm{fv}^-(\sigma)$ and $\beta' \in \mathrm{fv}^+(\sigma)$, definition 27 specifies that $\beta \leq \beta' \in D$. Since $\rho' \vdash D$, $\rho'(\beta) \leq \rho'(\beta')$; that is, $\tau \leq \tau'$ holds.
- τ is a small term and τ' is a ground term. As before, τ' is of the form $\rho'(\beta')$, for some $\beta' \in \mathrm{fv}^+(\sigma)$ such that $\alpha \leq \beta' \in C^r$. On the other hand, we must have $\tau \in C^{\downarrow}(\alpha)$. If $\alpha \leq \beta' \in C$, considering that C satisfies condition 2 of definition 23, there exists a small

term $\tau'' \in C^{\downarrow}(\beta')$ such that $C \Vdash^{+1} \tau \leq \tau''$. If, on the other hand, $\alpha = \beta'$, then the same holds (simply pick $\tau'' = \tau$). Pick some $\rho \vdash_k E$. We then have $\rho(\tau) \leq_{k+1} \rho(\tau'')$. Furthermore, because $\beta' \in \mathrm{fv}^+(\sigma)$, definition 27 specifies that $\tau'' \in D^{\downarrow}(\beta')$. Since $\rho' \vdash D$, this entails $\rho'(\tau'') \leq \rho'(\beta')$. Now, we need to reason by cases on the structure of τ'':

- Assume τ'' is of the form $\delta_0 \to \delta_1$. Since $\beta' \in \mathrm{fv}^+(\sigma)$, definition 26 specifies that $\delta_1 \in \mathrm{fv}^+(\sigma)$ and $\delta_0 \in \mathrm{fv}^-(\sigma)$. According to the definition of E, $\delta_1 \leq \rho'(\delta_1) \in E$. Since $\rho \vdash_k E$, this implies $\rho(\delta_1) \leq_k \rho'(\delta_1)$. Symmetrically, $\rho'(\delta_0) \leq_k \rho(\delta_0)$. As a consequence, $\rho(\delta_0 \to \delta_1) \leq_{k+1} \rho'(\delta_0 \to \delta_1)$. In other words, $\rho(\tau'') \leq_{k+1} \rho'(\tau'')$.
- Assume τ'' is equal to $\bot$ or $\top$. Then, the same holds, i.e. $\rho(\tau'') \leq_{k+1} \rho'(\tau'')$.

We can now combine, by transitivity, the three results obtained above:

$$\rho(\tau) \leq_{k+1} \rho(\tau'') \leq_{k+1} \rho'(\tau'') \leq \rho'(\beta')$$

This implies $\rho(\tau) \leq_{k+1} \rho'(\beta')$. That is, $\rho \vdash_{k+1} \tau \leq \tau'$, which is the desired result.

- The last case is symmetrical to the previous one. □

It is easy to check that garbage collection preserves polarities. Furthermore, provided bipolar variables are disallowed, its output is closed, as stated below. This important remark was missing from [TS96].

Theorem 12 *Consider σ as in definition 27. If $\mathrm{fv}^+(\sigma) \cap \mathrm{fv}^-(\sigma) = \varnothing$, then $\mathrm{GC}(\sigma)$ is closed.*

Proof. Write $\mathrm{GC}(\sigma) = A \Rightarrow \epsilon \mid D$, as in definition 27. As per definition 25, assume $\{\tau \leq \alpha, \alpha \leq \tau'\} \subseteq D$. Then, $\alpha \in \mathrm{fv}^+(\sigma)$, because it appears on the right-hand side of a constraint in D. Symmetrically, $\alpha \in \mathrm{fv}^-(\sigma)$. This is impossible, by hypothesis, so D is (vacuously) closed. □

4.4 Canonization

In section 3, in order to view a multi-equation system as a finite state automaton, we required it to be in canonical form, i.e. to equate each variable with at most one non-variable term. Similarly, in the case of subtyping, we say that a constraint set is in *canonical form* iff each variable has exactly one non-variable lower (resp. upper) bound. We shall require this property before we attempt to minimize constraint sets. In this section, we give an algorithm, called *canonization*, which computes a canonical form of an arbitrary constraint set.

Definition 28 *Let $\sigma = A \Rightarrow \epsilon \mid C$ be a type scheme, made up of small terms, containing no bipolar variables, such that $\sigma = \mathrm{GC}(\sigma)$.*

Let V (resp. W) range over non-empty subsets of $\mathrm{fv}^-(\sigma)$ (resp. $\mathrm{fv}^+(\sigma)$). For each such V (resp. W) of cardinality greater than 1, pick a fresh variable γ_V (resp. λ_W). (By fresh variables, we mean that these variables are pairwise distinct, and distinct from σ's variables.) Define the rewriting function *r^- (resp. r^+) according to figure 8. The first two lines define r^- (resp. r^+) on non-empty sets of negative (resp. positive) variables; they are then extended to sets of negative (resp. positive) small terms.*

The image of σ through canonization, *denoted by $\mathrm{Can}(\sigma)$, is $A \Rightarrow \epsilon \mid D$, where the constraint set D is given by figure 9. It is clear that $\mathrm{Can}(\sigma)$ is in canonical form.*

$$\begin{aligned} r^+(\{\alpha\}) &= \alpha & r^-(\{\alpha\}) &= \alpha \\ r^+(W) &= \lambda_W \text{ when } |\,W\,| > 1 & r^-(V) &= \gamma_V \text{ when } |\,V\,| > 1 \end{aligned}$$

$$\begin{aligned} r^+(\{\bot\} \cup S) &= r^+(S) & r^-(\{\top\} \cup S) &= r^-(S) \\ r^+(\{\top\} \cup S) &= \top & r^-(\{\bot\} \cup S) &= \bot \\ r^+(\varnothing) &= \bot & r^-(\varnothing) &= \top \end{aligned}$$

$$r^+(\{\alpha_1 \to \beta_1, \ldots, \alpha_n \to \beta_n\}) = r^-(\{\alpha_1, \ldots, \alpha_n\}) \to r^+(\{\beta_1, \ldots, \beta_n\})$$
$$r^-(\{\alpha_1 \to \beta_1, \ldots, \alpha_n \to \beta_n\}) = r^+(\{\alpha_1, \ldots, \alpha_n\}) \to r^-(\{\beta_1, \ldots, \beta_n\})$$

Figure 8: Definition of the rewriting functions

$$r^-(V) \leq r^+(W) \in D \text{ iff } \exists \alpha \in V \quad \exists \beta \in W \quad \alpha \leq \beta \in C$$

$$\begin{aligned} D^{\downarrow}(\alpha) &= \{r^+(C^{\downarrow}(\alpha))\} & D^{\uparrow}(\alpha) &= \{r^-(C^{\uparrow}(\alpha))\} \\ D^{\downarrow}(\gamma_V) &= \{\bot\} & D^{\uparrow}(\lambda_W) &= \{\top\} \\ D^{\downarrow}(\lambda_W) &= \{r^+(\bigcup_{\alpha \in W} C^{\downarrow}(\alpha))\} & D^{\uparrow}(\gamma_V) &= \{r^-(\bigcup_{\alpha \in V} C^{\uparrow}(\alpha))\} \end{aligned}$$

Figure 9: Canonization

The basic idea behind canonization is simple: introduce fresh variables to stand for least upper bounds and greatest lower bounds of existing variables. For instance, "$\alpha \sqcup \beta$" may be represented by a fresh variable $\lambda_{\{\alpha,\beta\}}$, together with the constraints $\alpha \leq \lambda_{\{\alpha,\beta\}}$ and $\beta \leq \lambda_{\{\alpha,\beta\}}$. A straightforward definition of canonization, based on this principle, is given by Trifonov and Smith [TS96]. However, it involves intermediate closure computations, which generate many superfluous constraints. For instance, α and β above must be positive, because the least upper bound expressions which arise during canonization always involve positive variables. Since there are no bipolar variables, α and β cannot be negative. So, the fresh constraints $\alpha \leq \lambda_{\{\alpha,\beta\}}$ and $\beta \leq \lambda_{\{\alpha,\beta\}}$ shall be removed by the next garbage collection pass. In between, though, these constraints will take part in a closure computation, and their transitive consequences may survive garbage collection. Rather than going through the process of adding superfluous constraints as part of canonization, performing a closure computation, and then eliminating them, we give a more detailed description of canonization, whose output is provably closed, and which does not generate these unnecessary constraints, thus saving time.

For the sake of simplicity, our definition creates an exponential number of fresh variables. Of course, an implementation shall create a fresh γ_V or λ_W only on demand, i.e. when it appears in the constructed bound of an existing variable—which may be an original variable α, or may itself be a γ or a λ.

Considering our strong hypotheses on σ, it is easy to prove that $\mathrm{Can}(\sigma)$ is closed. Furthermore, we may prove that existing variables see their polarity decrease during canonization. These results mean that we may apply canonization transparently at any point of the type inference process, while still performing incremental closure computations, and relying on the assumption that no bipolar variables exist. They are proved below.

Theorem 13 *Consider σ as in definition 28. Then, $\mathrm{Can}(\sigma)$ is closed. Furthermore,*

$$\mathrm{fv}^+(\mathrm{Can}(\sigma)) \subseteq \{\lambda_W\} \cup \mathrm{fv}^+(\sigma)$$
$$\mathrm{fv}^-(\mathrm{Can}(\sigma)) \subseteq \{\gamma_V\} \cup \mathrm{fv}^-(\sigma)$$

As a corollary, there are no bipolar variables in $\mathrm{Can}(\sigma)$.

Proof. We first verify that two constraints of D involving variables can never be combined by transitivity. It suffices to notice that $r^-(V)$ can never be equal to $r^+(W)$, because the former is of the form γ_V or $\alpha \in \mathrm{fv}^-(\sigma)$, while the latter is of the form λ_W or $\alpha \in \mathrm{fv}^+(\sigma)$. Since σ has no bipolar variables, $\mathrm{fv}^+(\sigma) \cap \mathrm{fv}^-(\sigma) = \varnothing$.

To fully verify the requirement of definition 25, it essentially suffices to further notice that $\sigma = \mathrm{GC}(\sigma)$. This implies that for all $\alpha \in \mathrm{fv}^+(\sigma)$ (resp. $\alpha \in \mathrm{fv}^-(\sigma)$), $C^\uparrow(\alpha)$ (resp. $C^\downarrow(\alpha)$) is empty; which implies $D^\uparrow(\alpha) = \{\top\}$ (resp. $D^\downarrow(\alpha) = \{\bot\}$). The desired property follows easily; thus, $\mathrm{Can}(\sigma)$ is closed.

This result allows us to compute polarities. We verify that $\{\lambda_W\} \cup \mathrm{fv}^+(\sigma)$ and $\{\gamma_V\} \cup \mathrm{fv}^-(\sigma)$ satisfy the fix-point equations of definition 26, applied to $\mathrm{Can}(\sigma)$. To do so, it suffices to notice that a λ_W never appears in negative (resp. positive) position in a non-variable lower (resp. upper) bound—a symmetric result holds of γ_V—and that any $\alpha \in \mathrm{fv}(\sigma)$ appears in fewer positions than in C. □

We are now ready to prove the correctness of the canonization algorithm.

Theorem 14 *Consider σ as in definition 28. Then $\sigma \approx \mathrm{Can}(\sigma)$.*

Proof. Let us use the notations of definition 28. We first show that $\mathrm{Can}(\sigma) \preccurlyeq \sigma$, i.e.

$$\forall \rho \vdash C \quad \exists \rho' \vdash D \quad \rho'(A \Rightarrow \epsilon) \leq \rho(A \Rightarrow \epsilon)$$

Pick some $\rho \vdash C$. Define ρ' by

$$\rho'(\alpha) = \rho(\alpha) \qquad \rho'(\gamma_V) = \prod_{\alpha \in V} \rho(\alpha) \qquad \rho'(\lambda_W) = \coprod_{\alpha \in W} \rho(\alpha)$$

One easily checks that, for any W, $\rho'(r^+(W)) = \bigsqcup_{\alpha \in W} \rho(\alpha)$. Similarly, $\rho'(r^-(V)) = \bigsqcap_{\alpha \in V} \rho(\alpha)$. It is then straightforward to extend these results to sets of small terms, rather than sets of variables. Finally, using these results, it is a matter of routine to ascertain that ρ' satisfies D.

The other direction of the proof is slightly more difficult, because, as we explained before, our definition of canonization contains a built-in garbage collection step. We introduce an intermediate type scheme $\sigma' = A \Rightarrow \epsilon \mid E$, where E is defined by

$$E = D \cup \{\alpha \leq \lambda_W \,;\, \alpha \in W\} \cup \{\gamma_V \leq \alpha \,;\, \alpha \in V\}$$

First, let us show that $\sigma \preccurlyeq \sigma'$, i.e.

$$\forall \rho \vdash E \quad \exists \rho' \vdash C \quad \rho'(A \Rightarrow \epsilon) \leq \rho(A \Rightarrow \epsilon)$$

It is sufficient to prove that E entails C, i.e. $\forall \rho \vdash E \quad \rho \vdash C$. Pick some $\rho \vdash E$. It is clear that for any W, $(\bigsqcup_{\alpha \in W} \rho(\alpha)) \leq \rho(r^+(W))$. A symmetric result holds of any set of negative variables V. As above, these results can be transferred to sets of small terms. Using them, it is easy to check that any solution of E also satisfies C.

There remains to prove that $\sigma' \preccurlyeq \text{Can}(\sigma)$. We shall do so by noticing that the constraints in $E \setminus D$ are superfluous, according to garbage collection. The result shall then follow from theorem 11. Our first objective is to prove that E is weakly closed, which entitles us to apply garbage collection to σ'.

First, we check that E satisfies condition 1 of definition 23. Consider two constraints $\{\varphi \leq \psi, \psi \leq \xi\} \subseteq E$. If both appear in D, then so does $\varphi \leq \xi$, because D is closed. Besides, at least one of them must appear in D, because otherwise they would be of the form $\gamma_V \leq \alpha$ and $\alpha \leq \lambda_W$, which would require α to be bipolar. So, let us assume $\varphi \leq \psi \in D$ and $\psi \leq \xi \notin D$. (The other case is symmetric.) Then, the latter is of the form $\alpha \leq \lambda_W$, where $\alpha \in W$. Thus, the former must be of the form $r^-(V) \leq \alpha$, where $\beta \leq \alpha \in C$ for some $\beta \in V$. These properties are enough to guarantee that $r^-(V) \leq \lambda_W \in D$. Hence, E satisfies condition 1 of definition 23.

Then, we check that E satisfies condition 2 of definition 23. Because D is closed, it suffices to verify that whenever $\alpha \in W$ and $\tau \in E^{\downarrow}(\alpha)$, there exists some $\tau' \in E^{\downarrow}(\lambda_W)$ such that $E \Vdash^{+1} \tau \leq \tau'$. In other words, any k-solution ρ of E must satisfy $\rho(r^+(C^{\downarrow}(\alpha))) \leq_{k+1} \rho(r^+(\bigcup_{\beta \in W} C^{\downarrow}(\beta)))$. Because $\alpha \in W$, $C^{\downarrow}(\alpha)$ is a subset of $\bigcup_{\beta \in W} C^{\downarrow}(\beta)$. Thus, what we need to prove is a monotonicity property of r^+; it is easy to prove it in the case of variables first, and to transfer it to the case of small terms.

By symmetry, E also satisfies condition 3 of definition 23. Finally, because D is closed, it satisfies condition 4 of definition 23, and so does E. We have verified that E is weakly closed. Thus, according to theorem 11, we may throw away some of σ''s constraints, as allowed by its polarities, and obtain an equivalent type scheme.

Consider a constraint of the form $\alpha \leq \lambda_W$, where $\alpha \in W$. $\alpha \in W$ implies $\alpha \in \text{fv}^+(\sigma)$; since polarities decrease during canonization, $\alpha \notin \text{fv}^-(\text{Can}(\sigma))$. Furthermore, constraints between variables do not affect the polarity computation, so σ' and $\text{Can}(\sigma)$ have the same polarities. This implies $\alpha \notin \text{fv}^-(\sigma')$. Since α is not negative in σ', the constraint $\alpha \leq \lambda_W$ may be thrown away without affecting σ''s denotation. The same is true of constraints of the form $\gamma_V \leq \alpha$, where $\alpha \in V$. It follows that all constraints in $E \setminus D$ are actually superfluous, and $\sigma' \approx \text{Can}(\sigma)$. This concludes the proof. $\square$

4.5 Minimization

The simplification method developed in section 3 is based on the minimization of finite automata, which consists of two steps: eliminate any unreachable states, then identify all states which recognize the same language. In the case of subtyping, the first step is performed by garbage collection, which discards superfluous variables and constraints. It is also possible to design an algorithm in charge of performing the second step, as suggested by Flanagan and Felleisen [FF96, FF97, Fla97]. We now present this algorithm, adapted to our system, and name it *minimization.* It detects equivalent variables, using a method reminiscent of the way equivalent states of a finite automaton are found, and then merges them. We begin with the definition of the criterion which allows considering certain variables as equivalent.

Definition 29 *Let V be a set of type variables. Any equivalence relation $\equiv$ on V is extended to the set of small terms whose variables are in V:*

$$\bot \equiv \bot \qquad \top \equiv \top$$
$$\alpha_0 \to \alpha_1 \equiv \beta_0 \to \beta_1 \iff (\alpha_0 \equiv \beta_0) \wedge (\alpha_1 \equiv \beta_1)$$

Definition 30 *Let C be a constraint set. For $\alpha \in \mathrm{fv}(C)$, define*

$$\mathrm{pred}_C(\alpha) = \{\beta\,;\, \beta \leq \alpha \in C\}$$
$$\mathrm{succ}_C(\alpha) = \{\beta\,;\, \alpha \leq \beta \in C\}$$

Definition 31 *Let $\sigma = A \Rightarrow \epsilon \mid C$ be a type scheme in canonical form, made up of small terms, containing no bipolar variables, such that $\sigma = \mathrm{GC}(\sigma)$. For any $\alpha \in \mathrm{fv}(\sigma)$, $C^{\downarrow}(\alpha)$ (resp. $C^{\uparrow}(\alpha)$) is a singleton; so, by abuse of language, we shall use the same notation to refer to its unique element.*

An equivalence relation $\equiv$, of domain $\mathrm{fv}(\sigma)$, *is* compatible with σ *iff $\alpha \equiv \beta$ implies all of the following:*

1. $\{\alpha, \beta\} \subseteq \mathrm{fv}^{+}(\sigma)$ *or* $\{\alpha, \beta\} \subseteq \mathrm{fv}^{-}(\sigma)$;
2. $\mathrm{pred}_C(\alpha) = \mathrm{pred}_C(\beta)$ *and* $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$;
3. $C^{\downarrow}(\alpha) \equiv C^{\downarrow}(\beta)$ *and* $C^{\uparrow}(\alpha) \equiv C^{\uparrow}(\beta)$.

We now prove that the above conditions are indeed sufficient to ensure correctness, i.e. if we identify the variables of a type scheme according to a compatible equivalence relation, then we obtain an equivalent type scheme.

Definition 32 *Consider σ as in definition 31; let $\equiv$ be a partition compatible with σ. The quotient $\sigma/_{\equiv}$ is defined—up to a renaming—as $\pi(\sigma)$, where π is any mapping of* $\mathrm{fv}(\sigma)$ *into $\mathcal{V}$ such that*

$$\forall \alpha, \beta \in \mathrm{fv}(\sigma) \quad \alpha \equiv \beta \iff \pi(\alpha) = \pi(\beta)$$

Theorem 15 *Consider σ and $\equiv$ as in definition 32. Then, $\sigma/_{\equiv} \approx \sigma$.*

Proof. The assertion $\sigma \preccurlyeq \sigma/_{\equiv}$ clearly holds, because the latter is the image of the former through the substitution π. Conversely, let us show that $\sigma/_{\equiv} \preccurlyeq \sigma$. Let ρ be a solution of C. We need to exhibit a solution ρ' of $\pi(C)$ such that $\rho'(\pi(A \Rightarrow \epsilon)) \leq \rho(A \Rightarrow \epsilon)$.

Consider an equivalence class of $\equiv$. Because of condition 1 of definition 31, it must be either a subset of $\mathrm{fv}^{+}(\sigma)$, or a subset of $\mathrm{fv}^{-}(\sigma)$. We denote it by V (resp. W) in the former (resp. latter) case. We denote the image of its elements through π by φ_V (resp. φ_W). Define ρ' by

$$\rho'(\varphi_V) = \bigsqcup_{\alpha \in V} \rho(\alpha) \qquad \rho'(\varphi_W) = \bigsqcap_{\alpha \in W} \rho(\alpha)$$

We remark that for any $\alpha \in \mathrm{fv}^+(\sigma)$, $\rho'(\pi(\alpha)) \leq \rho(\alpha)$ holds; symmetrically, for any $\alpha \in \mathrm{fv}^-(\sigma)$, we have $\rho(\alpha) \leq \rho'(\pi(\alpha))$.

Let us now verify that ρ' is a solution of $\pi(C)$. We begin by checking that any constraint between variables is satisfied. Such a constraint is necessarily of the form $\varphi_V \leq \varphi_W$; furthermore, because of condition 2 of definition 31, we have

$$\forall \alpha \in V \quad \forall \beta \in W \quad \alpha \leq \beta \in C$$

Because ρ satisfies C, this implies

$$\forall \alpha \in V \quad \forall \beta \in W \quad \rho(\alpha) \leq \rho(\beta)$$

which, considering the definition of ρ', is exactly $\rho'(\varphi_V) \leq \rho'(\varphi_W)$.

We then check that any constraint between a variable and a small term is satisfied. Such a constraint may be written $\pi(C^{\downarrow}(\alpha)) \leq \pi(\alpha)$—the other case is symmetric. If $\alpha \in \mathrm{fv}^-(\sigma)$, this is immediate, because $C^{\downarrow}(\alpha) = \bot$. Assume $\alpha \in \mathrm{fv}^+(\sigma)$. According to the definition of ρ', our goal can then be written

$$\forall \alpha' \equiv \alpha \quad \rho'(\pi(C^{\downarrow}(\alpha))) \leq \rho(\alpha')$$

Assume $\alpha' \equiv \alpha$. Thanks to condition 3 of definition 31, we have $C^{\downarrow}(\alpha') \equiv C^{\downarrow}(\alpha)$, so these terms have the same image through π. Additionally, because ρ satisfies C, $\rho(C^{\downarrow}(\alpha')) \leq \rho(\alpha')$ holds. So, it suffices to prove

$$\rho'(\pi(C^{\downarrow}(\alpha'))) \leq \rho(C^{\downarrow}(\alpha'))$$

which is a straightforward consequence of our above remarks concerning ρ'.

There only remains to verify that $\rho'(\pi(A \Rightarrow \epsilon)) \leq \rho(A \Rightarrow \epsilon)$, which is again a direct consequence of said remarks. □

To obtain an algorithm, there remains to show, given a type scheme σ, how to compute an equivalence relation compatible with σ. Of course, we wish to identify as many variables as possible, so we wish to compute the coarsest such relation.

Theorem 16 *Consider σ as in definition 31. Then, there exists a coarsest equivalence relation compatible with σ. It can be computed in time $O(dn \log n)$, where $n = |\,\mathrm{fv}(\sigma)\,|$, and d is the degree of the graph $\{(\alpha, \beta)\,;\, \alpha \leq \beta \in C\}$.*

Proof. If τ is a small term, let $\mathrm{head}(\tau) \in \Sigma_g$ denote its head constructor. To each $\alpha \in \mathrm{fv}(\sigma)$, associate a key, as follows:

$$\mathrm{key}(\alpha) = (1, \mathrm{pred}_C(\alpha), \mathrm{head}(C^{\downarrow}(\alpha))) \text{ if } \alpha \in \mathrm{fv}^+(\sigma)$$
$$\mathrm{key}(\alpha) = (0, \mathrm{succ}_C(\alpha), \mathrm{head}(C^{\uparrow}(\alpha))) \text{ if } \alpha \in \mathrm{fv}^-(\sigma)$$

Define $\alpha \equiv_{\mathrm{key}} \beta$ to mean $\mathrm{key}(\alpha) = \mathrm{key}(\beta)$. Furthermore, for $i \in \{0, 1\}$, define a partial function δ_i from $\mathrm{fv}(\sigma)$ into itself by

$$\delta_i(\alpha) = \alpha_i \text{ if } \alpha \in \mathrm{fv}^+(\sigma) \text{ and } C^{\downarrow}(\alpha) = \alpha_0 \rightarrow \alpha_1$$
$$\delta_i(\alpha) = \alpha_i \text{ if } \alpha \in \mathrm{fv}^-(\sigma) \text{ and } C^{\uparrow}(\alpha) = \alpha_0 \rightarrow \alpha_1$$

Then, it is not difficult to see that an equivalence relation $\equiv$ is compatible with σ iff it is finer than $\equiv_{\mathrm{key}}$ and stable with respect to δ_0 and δ_1. (An equivalence relation $\equiv$ is *stable* with respect to a function f iff for every class B of $\equiv$, either f is undefined on all of B, or f is defined on all of B and $f(B)$ lies entirely within some class B'.)

So, the problem is now to find the coarsest refinement of a given partition which is stable with respect to a finite number of given functions. Indeed, such a refinement exists; Hopcroft [Hop71] gives an $O(n \log n)$ algorithm to compute it.

There remains to check how much time is necessary to compute $\equiv_{\mathrm{key}}$, δ_0 and δ_1. $\equiv_{\mathrm{key}}$ can be obtained by building a list of all variables in $\mathrm{fv}(\sigma)$, sorting it according to their keys, and then walking the list, taking advantage of the fact that variables related by $\equiv_{\mathrm{key}}$ must be adjacent in the sorted list. Comparing two keys takes time $O(d)$, because predecessor or successor sets of cardinality up to d have to be compared; so, the whole operation takes time $O(dn \log n)$. Building δ_0 and δ_1 can be done in time $O(n)$. $\square$

It is straightforward to check that minimization preserves polarities, as well as the closure property.

In the case of equality constraints, minimization was an optimal simplification method, as shown by theorem 5. Here, though, completeness is lost, because the criterion we use to detect equivalent variables is too coarse, as shown by the following example. Let F be a covariant type operator, distinct from the identity. (For instance, take $F(\alpha) = \top \rightarrow \alpha$.) Consider the type scheme

$$\alpha^- \rightarrow \beta^- \rightarrow \gamma^+ \mid \{\alpha^- \leq F\,\alpha^-,\, \beta^- \leq F\,\beta^-,\, F\,\gamma^+ \leq \gamma^+,\, \alpha^- \leq \gamma^+\}$$

Here, α and β cannot be in the same class. If they were, then the presence of the constraint $\alpha \leq \gamma$ would require $\beta \leq \gamma$ to be also present, which is not the case. However, the constraint $\alpha \leq \gamma$ is superfluous, because it is implied by the other constraints. (Indeed, $\alpha \leq F\,\alpha$ and $F\,\gamma \leq \gamma$ entail $\alpha \leq \gamma$.) If a complete axiomatization of entailment were known, it might be possible to use it to determine that α and β are equivalent. However, in its absence, we are left with an incomplete minimization algorithm, which relies on a syntactic criterion, namely the presence of the constraint $\beta \leq \gamma$, rather than on a semantic one, namely the fact that this relationship is implied by the constraint set.

Although situations similar to the above one do sometimes arise in practice, experience shows that minimization often produces an optimal result. So, this theoretical problem is not a practical issue; on the contrary, the criterion's simplicity is the key to the algorithm's efficiency.

5 Example

Our theoretical description is over; we now wish to show our algorithms at work on a simple example. Consider the expression $\lambda(x, y)$.`choose` (x, y) `or` (y, x). (We assume the language is extended with pairs, pair patterns, and a non-deterministic choice construct `choose`.) We will first compute a type scheme for this expression, by building a type inference derivation, then simplify it. (In a real implementation, simplifications may be applied at any point of the derivation; it is desirable to do so at least at every `let` node, to avoid moving an unsimplified type scheme into the environment.)

According to rule (VAR$_{\mathrm{I}}$), the first occurrence of x receives type $(x \mapsto v_1) \Rightarrow v_2$, together with the constraint $v_1 \leq v_2$. Similarly, the first occurrence of y receives type $(y \mapsto v_3) \Rightarrow v_4$, where $v_3 \leq v_4$. The pair construction rule, like the application rule, computes a meet of the two contexts, so (x, y) is assigned type $(x \mapsto v_1, y \mapsto v_3) \Rightarrow v_5$, where $v_2 \times v_4 \leq v_5$ is added to the above constraints.

$v_1 \leq v_2$	$v_3 \leq v_4$	$v_2 \times v_4 \leq v_5$
$v_6 \leq v_7$	$v_8 \leq v_9$	$v_9 \times v_7 \leq v_{10}$
$v_{11} \leq v_1$	$v_{11} \leq v_6$	$v_{12} \leq v_3$
$v_{12} \leq v_8$	$v_5 \leq v_{13}$	$v_{10} \leq v_{13}$
$v_{14} \leq v_{11} \times v_{12}$	$v_{14} \rightarrow v_{13} \leq v_{15}$	

Figure 10: The initial constraints

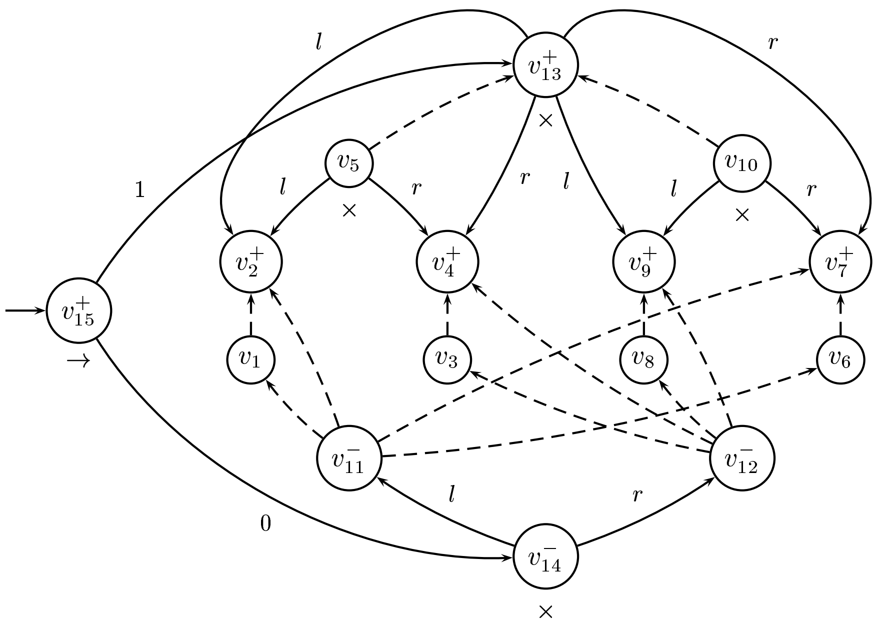

Figure 11: After closure

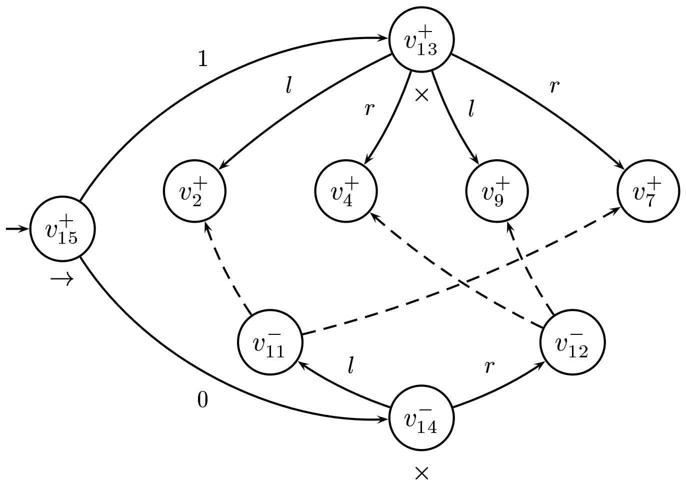

Figure 12: After garbage collection

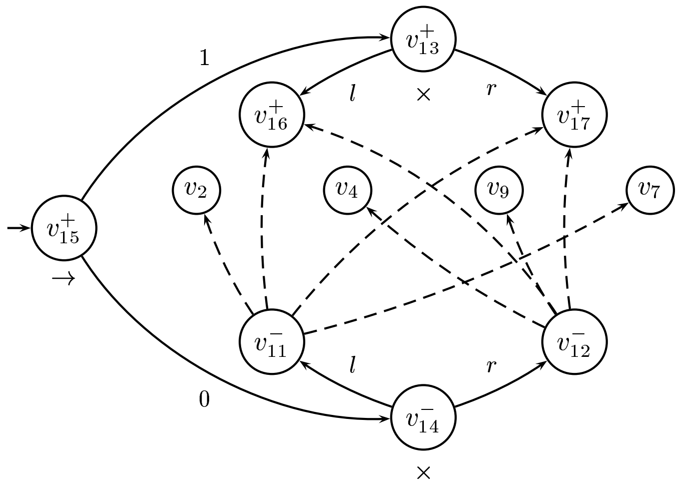

Figure 13: After canonization

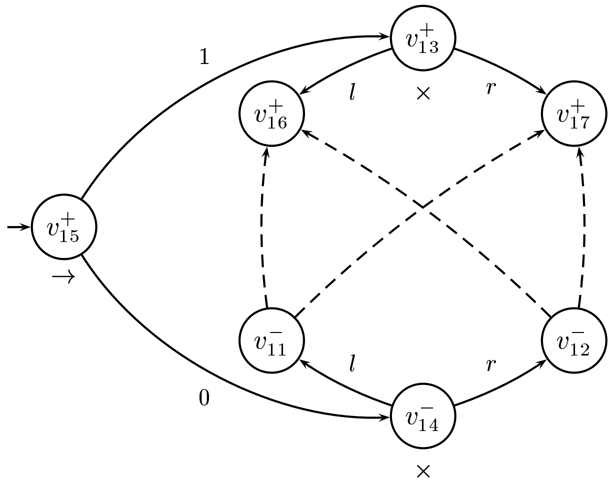

Figure 14: After a second pass of garbage collection

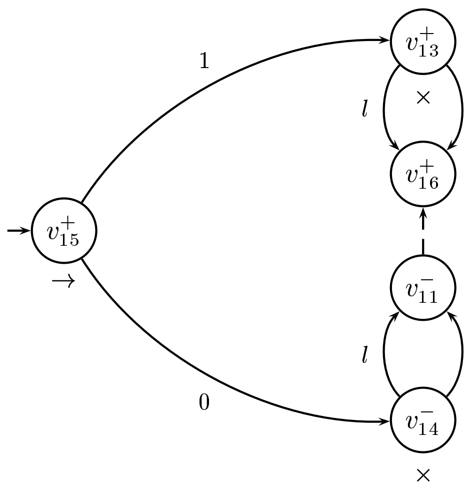

Figure 15: After minimization

$$v_{16}^{\pm} \times v_{16}^{\pm} \rightarrow v_{16}^{\pm} \times v_{16}^{\pm}$$

Figure 16: After pretty-printing

Similarly, the pair (y, x) receives type $(x \mapsto v_6, y \mapsto v_8) \Rightarrow v_{10}$, where $v_6 \leq v_7, v_8 \leq v_9, v_9 \times v_7 \leq v_{10}$.

The inference rule for the `choose` construct again computes a meet of the contexts, and merges the two result types. We obtain $(x \mapsto v_{11}, y \mapsto v_{12}) \Rightarrow v_{13}$, with the new constraints $v_{11} \leq v_1, v_{11} \leq v_6, v_{12} \leq v_3, v_{12} \leq v_8, v_5 \leq v_{13}, v_{10} \leq v_{13}$.

Finally, rule ($\textsc{Abs}_\textsc{I}$), extended to deal with pair patterns, removes the context entries for x and y and uses them to build a function type. We finally obtain type v_{15}, with fresh constraints $v_{14} \leq v_{11} \times v_{12}, v_{14} \rightarrow v_{13} \leq v_{15}$. The constraints obtained so far are grouped in figure 10.

We must now compute the closure of this constraint set, to ensure that the expression is well-typed. This adds the constraints $v_{11} \leq v_2, v_{11} \leq v_7, v_{12} \leq v_4, v_{12} \leq v_9, v_2 \times v_4 \leq v_{13}, v_9 \times v_7 \leq v_{13}$. No inconsistency is found, so the expression is type-correct; however, we now wish to simplify this type scheme.

Since the constraint set is closed, we may compute the polarity of each variable. The result is shown graphically in figure 11. Dashed edges represent subtyping relationships between variables. Solid edges link each variable v to the variables of its relevant constructed bounds, i.e. its constructed lower (resp. upper) bounds, when v is positive (resp. negative). Solid edges are labeled by 0, 1, l or r, to indicate domain, range, left component, and right component, respectively. Nodes are labeled with the head constructor(s) of their relevant constructed bounds. Thus, by using polarities to identify relevant bounds—which, in general, simplifies the figure—we obtain a graphical presentation similar to that of section 3. There are two main differences: first, the presence of subtyping edges; second, the fact that a variable may, at this point, have several relevant constructed bounds.

Since polarities are known, we may now apply garbage collection, to get rid of all superfluous constraints. All neutral variables, namely v_5, v_{10}, v_1, v_3, v_8 and v_6, disappear. This corresponds to the intuition—which is quite apparent on figure 11—that they are intermediate variables, which become useless after they have played a part in the closure computation. The result of garbage collection is shown by figure 12.

Things are now clearer. However, v_{13} has two constructed lower bounds, namely $v_2 \times v_4$ and $v_9 \times v_7$, and our minimization algorithm can only act on canonical sets, where each variable has exactly one constructed bound. (This corresponds, informally speaking, to the fact that only deterministic automata may be directly minimized.) So, we first apply our canonization algorithm, whose output is shown in figure 13. It creates two fresh variables, v_{16} and v_{17}. The former intuitively stands for $v_2 \sqcup v_9$, while the latter stands for $v_4 \sqcup v_7$.

Note that v_2, v_4, v_9 and v_7 have become neutral as a result of canonization. Since minimization expects its input to be stable by garbage collection, we must now run garbage collection again. Its output appears in figure 14. (The reader may be surprised to see that this algorithm has to be applied twice during the simplification process. In practice, this is not a problem at all, since it is very cheap. In theory, one may prove that canonization does not require its input to be stable through garbage collection, which allows each algorithm to be run exactly once. Doing so requires a heavier proof [Pot98b, Pot98c].)

A characteristic configuration, called a 2-*crown* in the literature, is now clearly apparent. The minimization algorithm will eliminate it. Indeed, v_{16} and v_{17} can be identified, because they have identical polarities, predecessor sets, and constructed lower bounds (namely $\bot$). Symmetrically, it is valid to merge v_{11} and v_{12}. The output of minimization is given by figure 15.

At this point, the result is clearly optimal, considering our two invariants: we chose to use small terms only, and to prohibit bipolar variables. This allowed an easier formulation of our algorithms and proofs—in particular, expressing minimization requires the first invariant, since there is otherwise no way to reason about sharing between type terms. Thus, we put the emphasis on *efficiency*. However, the computation is now over, and we wish to display its result. It is then perfectly acceptable to abandon these restrictions, in order to enhance *readability*. We apply a well-known simplification tactic [EST95a, AF96, AWP96, Pot96], which consists in replacing positive (resp. negative) variables with their lower (resp. upper) bound, if it is unique. This yields the type scheme displayed in figure 16, which is exactly what a programmer familiar with ML would have expected.

It is important to notice that the above invariants favor efficiency, at the expense of readability. We choose an efficient data representation during the whole type inference process, and switch to a more readable form for display. Trying to achieve efficiency and readability at the same time is a design mistake, since these goals put opposite requirements on the data representation: efficiency requires small terms, which allow improving sharing, while readability favors large terms, which help reduce the number of variables. This fact has already been pointed out while discussing our previous example, in section 3.

6 Related work

Closest to our work are the papers by Eifrig, Smith and Trifonov [EST95b, EST95a]. Their constraint logic is the same as ours; they perform constraint solving using the closure algorithm described in the present paper. Our definition of the scheme subsumption operator $\preccurlyeq$ comes from a later paper by Trifonov and Smith [TS96], where it is written $\leq^{\forall}$. We also adopt its formulation of the type inference rules, with a few enhancements, as explained in section 2.5. Moreover, this paper introduces garbage collection, a refinement of a technique for detecting unreachable variables proposed by the present author in [Pot96], as well as canonization. (Its description of canonization, however, is less precise, and may involve closure computations, whereas our definition is more detailed and allows proving that the closure property is preserved.)

Aiken and Wimmers [AW92, AW93] also study the problem of constraint-based type inference, but with a different interpretation of constraints. In our system, ground types are regular terms, and subtyping is defined explicitly on terms. Rather, Aiken *et al.* use the ideal model [MPS86]. Ground types are subsets of the model, and subtyping coincides with set-theoretic inclusion. In both cases, type inference involves constraint solving; however, in the former case, constraints are written in a dedicated formalism, whereas in the latter, the general theory of set constraints is used. As a result, their system is more expressive, as shown e.g. by its elaborate treatment of pattern matching [AWL94], but more complex. Its initial implementation [Aik94] contained unpublished simplification algorithms. More recent works by Aiken, Fähndrich *et al.* [AF96, FFSA98, AFFS98, Fäh99] describe various simplification techniques, many of which share common ideas with ours.

Flanagan and Felleisen [FF96, FF97, Fla97] also manipulate set constraints, in order to perform set-based analysis. Their system offers several common aspects with ours; in particular, it provided the inspiration for our *minimization* algorithm. The main difference probably lies in the treatment of functions. Indeed, in our system, a function's domain is the type of its *formal* argument, that is, the type of the objects it is able to handle; so, the $\rightarrow$ constructor must be contravariant with respect to its first argument. In Flanagan and Felleisen's system, on the contrary, a function's domain represents its *actual* argument, that is, the values passed to this function during the program's execution; so, the "dom" destructor is covariant. Furthermore, the constraint logic allows applying this destructor to objects other than functions. These decisions have advantages: every solvable constraint set has a smallest solution; entailment is decidable. On the other hand, solving the constraints no longer guarantees that the program is correct; an additional check becomes necessary. Hence, the theory is significantly modified.

Sulzmann *et al.* [OSW99, SMZ99] propose an abstract constraint-based type system, called HM(X). Whereas our paper offers a choice between equality constraints and a specific kind of subtyping constraints, they go one step further and parameterize their system by an arbitrary constraint logic, together with its constraint solving algorithm. Because it does not use our λ-lifting technique, their system is closer to the original Hindley-Milner presentation. As a drawback, the simplification issue is made slightly more complex. First, simplification algorithms (and their proofs) must distinguish between the variables which appear free in the environment and those which do not. Second, the presence of free variables makes implementing generalization and instantiation algorithms quite a subtle task, while it is trivial in our presentation. Sulzmann *et al.* do not address simplification or implementation issues.

Bourdoncle and Merz [BM96, BM97] propose a type system based on constrained type schemes, and apply it to an object-oriented language with multi-methods. After defining a subtyping relation between ground types, they lift it to the level of polymorphic type schemes, using a technique identical to ours. However, their constraint logic differs vastly. On the one hand, subtyping is structural and recursive types are absent, which allows decomposing any constraint system into one involving atoms (constants and variables) only. On the other hand, their subtyping relation is arbitrary and user-extensible, by contrast with our fixed lattice. As a result, different constraint resolution techniques are required; they are studied by Frey [Fre97].

Palsberg [Pal95] studies the problem of type inference for the core object calculus of Abadi and Cardelli [AC94a, AC94b]. He proposes an algorithm based on the same principle as that of Eifrig, Smith and Trifonov. However, the two systems exhibit a fundamental difference: whereas Eifrig *et al.*'s $\rightarrow$ constructor is contravariant with respect to its first argument and covariant with respect to the second one, Abadi and Cardelli's object types are *invariant*; that is, a subtyping relationship between two object types entails the *equality* of their common components. As shown by Henglein [Hen97], this peculiarity allows enhancing the inference algorithm's efficiency. However, to simulate function types in a satisfactory way, Abadi and Cardelli must introduce universally and existentially quantified types; in doing so, they lose type inference.

Müller, Niehren and Podelski [NMP97] take interest in the static analysis of the language Oz. The set of each program variable's possible values is approximated by a set of infinite terms. Once again, these sets are related by inclusion constraints. Moreover, for the program to be well-typed, the constraints must not merely admit a solution, but one that associates a non-empty set to each variable. For this reason, Müller *et al.* interpret constraints in the model of *non-empty* sets of terms. This system presents, in principle, common points with those mentioned above, but the details of constraint resolution, entailment and—if it were attempted—simplification differ widely. Also, note that this system only supports covariant type constructors.

Finally, let us mention Fuh and Mishra [FM88, FM89], who were precursors in the area of constraint simplification. Their work, however, deals with atomic constraints, as proposed by Mitchell [Mit84], and is of diminished interest today.

7 Conclusion

We have given a clean, comprehensive theoretical account of a constraint simplification system. This work brings together elements from various sources, and introduces several original ideas, so as to build a streamlined framework. We propose a combination of three simplification algorithms, which are simple and well-understood, as evidenced by the simplicity of their proofs. Practical experiments [Pot00b] show that this combination is efficient and effective, although the problem of designing a complete simplification method currently remains open.

The type system studied in this paper is reduced to an almost trivial core—in appearance. In fact, it is easy to extend it with advanced features, such as open record and variant types, reference types, etc. Furthermore, the essential ideas behind these algorithms are very general and should be applicable to a wide variety of systems—our study of the case of equality constraints supports this claim. In conclusion, we hope for this paper to constitute a sound theoretical basis for the development of constraint-based type inference systems.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993. URL: `http://research.microsoft.com/Users/luca/Papers/SRT.A4.ps`.

[AC94a] Martín Abadi and Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994. URL: `http://research.microsoft.com/Users/luca/Papers/PrimObj1stOrder.A4.ps`.

[AC94b] Martín Abadi and Luca Cardelli. A theory of primitive objects — second-order systems. In D. Sannella, editor, *Proc. of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 1–25, New York, N.Y., 1994. Springer Verlag. URL: `http://research.microsoft.com/Users/luca/Papers/PrimObj2ndOrder.A4.ps`.

[AF96] Alexander S. 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`.

[AFFS98] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A toolkit for constructing type- and constraint-based program analyses. *Lecture Notes in Computer Science*, 1473:76–96, 1998. URL: `http://www.cs.berkeley.edu/~aiken/papers/tic98.ps`.

[Aik94] Alexander S. Aiken. The Illyria system, 1994. URL: `http://http.cs.berkeley.edu:80/~aiken/Illyria-demo.html`.

[AW92] Alexander S. 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`.

[AW93] 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`.

[AWL94] 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`.

[AWP96] Alexander S. 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`.

[BM96] François Bourdoncle and Stephan Merz. On the integration of functional programming, class-based object-oriented programming, and multi-methods. Research Report 26, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Paris, March 1996. URL: `http://www.cma.ensmp.fr/Francois.Bourdoncle/mlsub.ps.Z`.

[BM97] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 302–315, Paris, January 1997. ACM. URL: `http://www.cma.ensmp.fr/Francois.Bourdoncle/popl97.ps.Z`.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66. URL: `http://research.microsoft.com/Users/luca/Papers/Inheritance.A4.ps`.

[Cou83] Bruno Courcelle. Fundamental properties of infinite trees. *Theoret. Comput. Sci.*, 25(2):95–169, March 1983.

[EST95a] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. *ACM SIGPLAN Notices*, 30(10):169–184, 1995. URL: `http://www.cs.jhu.edu/~trifonov/papers/sptio.ps.gz`.

[EST95b] 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`.

[FF96] 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`.

[FF97] 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`.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander S. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 Conference on Programming Languages Design and Implementation*, pages 85–96, Montréal, June 1998. URL: `http://www.cs.berkeley.edu/~manuel/papers/pldi98.ps`.

[Fäh99] Manuel Fähndrich. BANE*: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, 1999. URL: `http://research.microsoft.com/~maf/diss.ps`.

[Fla97] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, May 1997. URL: `http://www.cs.rice.edu/CS/PLT/Publications/thesis-flanagan.ps.gz`.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 2*, volume 352 of *LNCS*, pages 167–183, Berlin, March 1989. Springer.

[Fre97] Alexandre Frey. Satisfying subtype inequalities in polynomial space. In Pascal Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, number 1302 in Lecture Notes in Computer Science, pages 265–277, Paris, France, September 1997. Springer Verlag. URL: `http://www.cma.ensmp.fr/Alexandre.Frey/Publications/SAS97.ps.gz`.

[Hen97] 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.cis.upenn.edu/~bcpierce/fool/henglein.ps.gz`.

[Hop71] 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.

[HR98] Fritz Henglein and Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment. In *25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, July 1998. URL: `http://research.microsoft.com/~rehof/icalp98.ps`.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre* 1, 2, ..., ω. PhD thesis, Université Paris 7, September 1976.

[JK90] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. Technical Report 561, Université Paris-Sud, 91405 Orsay, France, April 1990.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proceedings POPL '93*, pages 419–428, 1993. URL: `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl93.ps.Z`.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mit84] John C. Mitchell. Coercion and type inference. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[MPS86] David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1–2):95–130, October–November 1986.

[MR00] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248(1–2), November 2000. URL: `http://www.cs.wisc.edu/wpis/papers/tcs_submission98r2.ps`.

[NMP97] Joachim Niehren, Martin Müller, and Andreas Podelski. Inclusion constraints over non-empty sets of trees. In Max Dauchet, editor, *Theory and Practice of Software Development, International Joint Conference CAAP/FASE/TOOLS*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1997. URL: `ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/ines97.ps.Z`.

[NP99] Joachim Niehren and Tim Priesnitz. Characterizing subtype entailment in automata theory. Technical report, Universität des Saarlandes, Programming Systems Lab, 1999. Submitted. URL: `http://www.ps.uni-sb.de/Papers/abstracts/pauto.html`.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps`.

[Pal95] 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`.

[PO95] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* ACM, January 1995. URL: `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl95.ps.Z`.

[Pot96] 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/fpottier-icfp96.ps.gz`.

[Pot98a] François Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 228–238, September 1998. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-icfp98.ps.gz`.

[Pot98b] François Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. PhD thesis, Université Paris 7, July 1998. URL: `http://pauillac.inria.fr/~fpottier/publis/these-fpottier.ps.gz`.

[Pot98c] François Pottier. Type inference in the presence of subtyping: from theory to practice. Technical Report 3483, INRIA, September 1998. URL: `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3483.ps.gz`.

[Pot00a] François Pottier. A 3-part type inference engine. Submitted for journal publication, May 2000. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz`.

[Pot00b] François Pottier. `Wallace`: an efficient implementation of type inference with subtyping, February 2000. URL: `http://pauillac.inria.fr/~fpottier/wallace/`.

[Rém92] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, 1992. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz`.

[SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999. URL: `http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz`.

[TS96] 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`.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987. URL: `ftp://ftp.ccs.neu.edu/pub/people/wand/papers/fundamenta-87.dvi`.