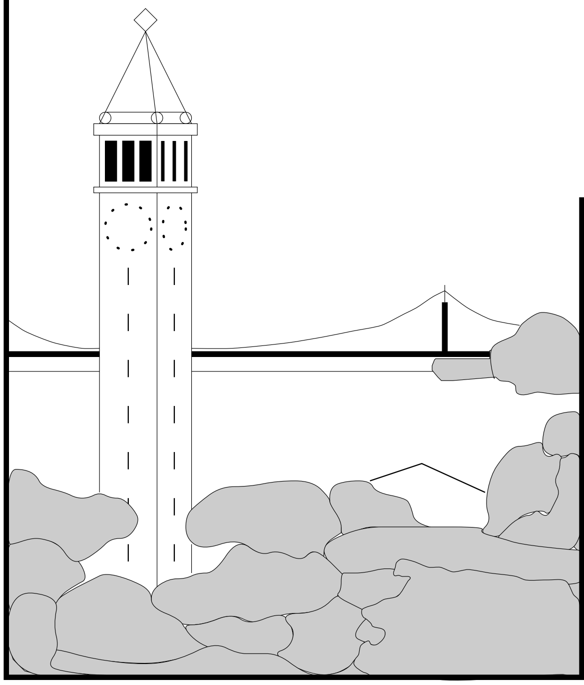
Optimal Representations of Polymorphic Types with Subtyping

Alexander Aiken *Edward L. Wimmers* *Jens Palsberg*



Report No. UCB/CSD-96-909

July 1996

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Optimal Representations of Polymorphic Types with Subtyping

Alexander Aiken[*] Edward L. Wimmers[†] Jens Palsberg[‡]

Abstract

Many type inference and program analysis systems include notions of subtyping and parametric polymorphism. When used together, these two features induce equivalences that allow types to be simplified by eliminating quantified variables. Eliminating variables both improves the readability of types and the performance of algorithms whose complexity depends on the number of type variables. We present an algorithm for simplifying quantified types in the presence of subtyping and prove it is sound and complete for non-recursive and recursive types. We also show that an extension of the algorithm is sound but not complete for a type language with intersection and union types, as well as for a language of constrained types.

1 Introduction

Contemporary type systems include a wide array of features, of which two of the most important are *subtyping* and *parametric polymorphism*. These two features are independently useful. Subtyping expresses relationships between types of the form "type τ_1 is less than type τ_2". Such relationships are useful, for example, in object-oriented type systems and in program analysis algorithms where a greatest (or least) element is required. Parametric polymorphism allows a parameterized type inferred for a program fragment to take on a different instance in every context where it is used. This feature has the advantage that the same program can be used at many different types.

A number of type systems have been proposed that combine subtyping and polymorphism, among other features. The intended purposes of these systems varies. A few examples are: studies of type systems themselves [CW85, Cur90, AW93], proposals for type systems for object-oriented languages [EST95], and program analysis systems used in program optimization [AWL94, HM94]. In short, the combination of subtyping and polymorphism is useful, with a wide range of applications.

When taken together, subtyping and polymorphism induce equivalences on types that can be exploited to simplify the representation of types. Our main technical result is that, in a simple type language with a least type $\perp$ and greatest type $\top$, for any type σ there is another type σ' that is equivalent to σ and σ' has the minimum number of quantified type variables. Thus, type simplification eliminates quantified variables wherever possible. Eliminating variables is desirable for three reasons. First, many type inference algorithms have computational complexity that is sensitive (both theoretically and practically) to the number of type variables. Second, eliminating variables makes types more readable. Third, simplification makes properties of types manifest that are otherwise implicit; in at least one case that we know

[*]Author's address: EECS Department, University of California at Berkeley, Berkeley, CA 94720-1776. Email: aiken@cs.berkeley.edu. This work was supported by an NSF NYI award, CCR-9457812

[†]Author's address: IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120-6099. Email: wimmers@almaden.ibm.com

[‡]Author's address: Laboratory for Computer Science, MIT, NE43-340 545 Technology Square, Cambridge MA, 02139. Email: palsberg@theory.lcs.mit.edu

of, these "hidden" properties are exactly the information needed to justify compiler optimizations based on type information [AWL94].

The basic idea behind variable elimination is best illustrated with an example. A few definitions are needed first. Consider a simple type language defined by the following grammar:

$$\tau ::= \alpha \mid \top \mid \bot \mid \tau_1 \to \tau_2$$

In this grammar, α is a type variable. Following standard practice, we use $\alpha, \beta, \ldots$ for type variables and $\tau, \tau', \tau_1, \tau_2, \ldots$ for types. The subtyping relation is a partial order $\preceq$ on types, which is the least relation satisfying

$$\begin{array}{rcl} \tau & \preceq & \tau \\ \bot & \preceq & \tau \\ \tau & \preceq & \top \\ \tau_1 \preceq \tau_1' \wedge \tau_2 \preceq \tau_2' & \Leftrightarrow & \tau_1' \to \tau_2 \preceq \tau_1 \to \tau_2' \end{array}$$

Quantified types are given by the following grammar:

$$\sigma ::= \tau \mid \forall \alpha . \sigma$$

For the moment, we rely on the reader's intuition about the meaning of quantified types. A formal semantics of quantified types is presented in Section 2.

Consider the type $\forall \alpha . \forall \beta . \alpha \to \beta$. Any function with this type takes an input of an arbitrary type α and produces an output of any (possibly distinct) arbitrary type β. What functions have this type? The output β must be included in all possible types; there is only one such type $\bot$. The input α, however, must include all possible types; there is only one such type $\top$. Thus, one might suspect that this type is equivalent to $\top \to \bot$. The only function with this type is the one that diverges for all possible inputs.

It turns out that, in fact, $\forall \alpha . \forall \beta . \alpha \to \beta \equiv \top \to \bot$ in the standard *ideal* model of types [MPS84]. As argued above, the type with fewer variables is better for human readability, the speed of type inference, and for the automatic exploitation of type information by a compiler. We briefly illustrate these three claims.

The reasoning required to discover that $\forall \alpha . \forall \beta . \alpha \to \beta$ represents an everywhere-divergent function is non-trivial. There is a published account illustrating how types inferred from ML programs (which have polymorphism but no subtyping) can be used to detect non-terminating functions exactly as above [Koe94]. The previous example is the simplest one possible; the problem of understanding types only increases with the size of the type and expressiveness of the type language. The following example is taken from the system of [AW93], a subtype inference system with polymorphism. In typing a term, the inference algorithm in this system generates a system of subtyping constraints that must be satisfied. The solution of the constraints gives the desired type. Constraints are generated as follows: If f has type $\alpha \to \beta$ and x has type γ, then for an application $f\,x$ to be well-typed it must be the case that $\gamma \preceq \alpha$. Figure 1 shows the type generated for the divergent lambda term $(\lambda x . x\, x)(\lambda x . x\, x)$. The type has the form

$$\forall \alpha_1, \ldots, \alpha_8 . (\alpha_6 / S)$$

Informally, the meaning of this type is α_6 for any assignment to the variables $\alpha_1, \ldots, \alpha_8$ that simultaneously satisfies all the constraints in S.

This type is equal to $\bot$, a fact proven by our algorithm extended to handle constraints. The type $\bot$ is sound, since the term is divergent. This example illustrates both improved readability and the

$$
\forall\alpha_1 \ldots \forall\alpha_8 .\, \alpha_6 \ \Big/ \left\{ \begin{array}{rcccl}
\alpha_4 \rightarrow \alpha_6 & \preceq & \alpha_1 & \preceq & \alpha_5 \rightarrow \alpha_6 \\
 & & \alpha_1 & \preceq & \alpha_2 \rightarrow \alpha_3 \\
\alpha_1 & \preceq & \alpha_2 & \preceq & \alpha_5 \rightarrow \alpha_6 \\
\bot & \preceq & \alpha_3 & \preceq & \top \\
\alpha_2 & \preceq & \alpha_4 & \preceq & \alpha_5 \rightarrow \alpha_6 \\
\alpha_4 & \preceq & \alpha_5 & \preceq & \alpha_4 \\
\bot & \preceq & \alpha_6 & \preceq & \alpha_3 \\
\alpha_4 \rightarrow \alpha_6 & \preceq & \alpha_7 & \preceq & \alpha_1 \\
\alpha_3 & \preceq & \alpha_8 & \preceq & \top
\end{array} \right.
$$

Figure 1: A quantified type of eight variables qualified by constraints.

possibility of more efficient inference. To use the polymorphic type $\forall\alpha_1, \ldots, \alpha_n.(\tau/S)$ in different contexts, the variables must be instantiated and the constraints duplicated for each usage context. Eliminating variables simplifies the representation, making this very expensive aspect of type inference less costly.

Finally, simplifying types can improve not only the speed but the quality of program analyses. For example, the *soft typing* system of [AWL94] reduces the problem of identifying where runtime type checks are not required in a program to testing whether certain type variables can be replaced by $\bot$ in a quantified type. This is exactly the task performed by elimination of variables in quantified types.

Our main contribution is a variable elimination algorithm that is sound and complete (i.e., eliminates as many variables as possible) for the simple type language defined above, as well as for a type language with recursive types. We extend the algorithm to type languages with intersection and union types and to type languages with subsidiary constraints. For these latter two cases, the techniques we present are sound but not complete. Combining the completeness results for the simpler languages with examples illustrating the incompleteness of the algorithm in the more expressive settings, we shed some light on the sources of incompleteness.

The various algorithms are practical and efficient, running in linear or quadratic time in the size of the type. The algorithm for simplifying quantified types with subsidiary constraints has been in use for several years, but with the exception of code documentation little has been written previously on the subject. The algorithm has been implemented and used in Illyria[1], the systems reported in [AW93], and a large scale program analysis system for the functional language FL [AWL94]. This last application was by far the largest and best engineered. The quality of this system depended critically on eliminating variables wherever possible.

Other recent systems based on constrained types have also pointed out the importance of variable elimination. In [EST95], Eifrig, Smith, and Trifonov describe a variable elimination method similar, but not identical too, the one in Section 7. Pottier gives a method that can eliminate redundant variables from constraint sets [Pot96]. Both of these methods are heuristic; i.e., they are sound but not complete. Our focus in this paper is quite different. The problem of eliminating variables from polymorphic types appears quite difficult in the more expressive type languages (consider the incomplete algorithms just discussed). Our purpose is to explore the structure of the problem in restricted cases, and to understand what makes the problem harder in the case of constrained types. To the best of our knowledge, we present the first sound and complete algorithms for variable elimination in any type system.

[1]The source code for the Illyria system and an interactive demo are available at URL `http://www.cs.berkeley.edu/ aiken/Illyria-demo.html`.

Rather than work in a specific semantic domain, we state axioms that a semantic domain must satisfy for our techniques to apply (Section 2). Section 3 gives the syntax for type expressions as well as their interpretation in the semantic domain.

Section 4 proves the results for the case of *simple type expressions*, which are non-recursive types. For quantified simple types, variable elimination produces an equivalent type with the minimum number of quantified variables. Furthermore, all equivalent types with the minimum number of quantified variables are *α-equivalent*—they are identical up to the names and order of quantified variables.

The intuition behind the variable elimination procedure is easy to convey. Type variables may be classed as *monotonic* (*positive*) or *anti-monotonic* (*negative*) based on their syntactic position in a type. Intuitively, the main lemma shows that quantified variables that are solely monotonic can be eliminated in favor of $\bot$; quantified variables that are solely anti-monotonic can be eliminated in favor of $\top$. Section 4.2 proves that the strategy of eliminating either monotonic or anti-monotonic variables is complete for the simple type language. Variables that are both monotonic and anti-monotonic cannot be eliminated.

Section 5 extends the basic variable elimination algorithm to a type language with recursive types. The extended algorithm is again both sound and complete, but it is no longer the case that all equivalent types with the minimum number of quantified variables are α-equivalent.

Section 6 extends the algorithm to intersection and union types. This language is the first extension for which the techniques are sound but not complete. Examples are given showing sources of incompleteness. Finally, Section 7 extends the algorithm to a type language with subsidiary constraints, as in Figure 1. This is the most general type language we consider. Section 8 concludes with a few remarks on related work.

2 Semantic Domains

Rather than work with a particular semantic domain, we axiomatize the properties needed to prove the corresponding theorems about eliminating quantified variables.

Definition 2.1 A semantic domain $\mathcal{D} = (\mathcal{D}_0, \mathcal{D}_1, \preceq, \sqcap)$ satisfies the following properties:

1. $\mathcal{D}_0 \subseteq \mathcal{D}_1$
2. a partial order on $\mathcal{D}_1$ denoted by $\preceq$.
3. a minimal element $\bot \in \mathcal{D}_0$ such that $\bot \preceq x$ for all $x \in \mathcal{D}_1$.
4. a maximal element $\top \in \mathcal{D}_0$ such that $x \preceq \top$ for all $x \in \mathcal{D}_1$.
5. a binary operation $\rightarrow$ on $\mathcal{D}_0$ such that if $y_1 \preceq x_1$ and $x_2 \preceq y_2$, then $x_1 \rightarrow x_2 \preceq y_1 \rightarrow y_2$. Furthermore, $\bot \rightarrow \top \neq \top$ and $\top \rightarrow \bot \neq \bot$.
6. a greatest lower bound operation $\sqcap$ on $\mathcal{D}_1$ such that if $D \subseteq \mathcal{D}_1$, then $\sqcap D$ is the greatest lower bound (or *glb*) of D.

In addition, the semantic domain $\mathcal{D}$ may satisfy some (or all) of the following properties:

standard function types
If $x_1 \rightarrow x_2 \preceq y_1 \rightarrow y_2$, then $y_1 \preceq x_1$ and $x_2 \preceq y_2$.

standard glb types
If $S_0 \subseteq \mathcal{D}_0$ and $x_1 \in \mathcal{D}_0$, then $\sqcap S_0 \preceq x_1$ iff $\exists x_0 \in S_0$ s.t. $x_0 \preceq x_1$.

An unusual aspect of Definition 2.1 is that a domain is built from two sets $\mathcal{D}_0$ and $\mathcal{D}_1$. This structure parallels the two distinct operations provided in the type language: function space $t_1 \to t_2$ and universal quantification $\forall \ldots$ (see Section 3). These operations impose different requirements on the semantic domain, so allowing $\mathcal{D}_0$ and $\mathcal{D}_1$ to be different provides more generality than requiring that they be the same. In particular, since the $\forall$ quantifier introduces a glb operation (and hence produces a value in $\mathcal{D}_1$) and the $\to$ operation can be performed only on elements of $\mathcal{D}_0$, the $\forall$ quantifier cannot appear inside of a $\to$ operation. If the semantic domain has the property that $\mathcal{D}_0 = \mathcal{D}_1$, then it supports $\forall$ quantifiers inside of the $\to$ operation. It is worth noting that separating $\mathcal{D}_0$ and $\mathcal{D}_1$ not only generalizes but simplifies some of our results.

The following two examples illustrate the most important features of semantic domains and are used throughout the paper.

Example 2.2 (Minimal Semantic Model) Let $\mathcal{D}_0 = \mathcal{D}_1$ be the three element set $\{\bot, \top \to \top, \top\}$ where $\bot \preceq \top \to \top \preceq \top$. In this domain, all function types are the same and this type domain does little more than detect that something is a function. For all $x, y \in \mathcal{D}$, $x \to y = \top \to \top$. It is easy to check that $\mathcal{D}$ satisfies all properties required of a semantic domain as well as standard glb types. The only property missing is standard function types (e.g., because $\bot \to \bot \preceq \top \to \top$, but $\top \not\preceq \bot$).

Example 2.3 (Standard Model) Let $\mathcal{D}_0$ be the set consisting of $\bot$ and $\top$ and closed under the pairing operation (denoted using the $\to$ symbol). An obvious partial order is induced on $\mathcal{D}_0$. Let $\mathcal{D}_1$ consist of all the non-empty, upward-closed subsets of $\mathcal{D}_0$. Intuitively, each element of $\mathcal{D}_1$ represents the glb of its members. Define $d_0 \preceq d_1$ iff $d_0 \supseteq d_1$. Note that there is an obvious inclusion mapping from $\mathcal{D}_0$ to $\mathcal{D}_1$ by mapping each element of $\mathcal{D}_0$ to the upward-closure of the singleton set consisting of that element. It is easy to see that $\mathcal{D}_1$ has standard glb types.

The construction of $\mathcal{D}_1$ from $\mathcal{D}_0$ used in Example 2.3 is a general procedure for building a $\mathcal{D}_1$. Given a domain $\mathcal{D}_0$, the domain $\mathcal{D}_1$ can be defined to be the non-empty, upward-closed subsets of $\mathcal{D}_0$. Each element of $\mathcal{D}_1$ represents the glb of its members.

3 Syntax

The first type language we consider has only type variables and function types. In this language, as in all extensions we consider, quantification is shallow (occurs only at the outermost level).

Definition 3.1 *Unquantified simple type expressions* are generated by the grammar:

$$\tau ::= \alpha \mid \top \mid \bot \mid \tau_1 \to \tau_2$$

where α is a family of type variables.

A *quantified simple type expression* has the form

$$\forall \alpha_1 \ldots \forall \alpha_n . \tau$$

where α_i is a type variable for $i = 1, \ldots, n$ and τ is an unquantified simple type expression. The type τ is called the *body* of the type.

Since $n = 0$ is a possibility in Definition 3.1, every unquantified simple type expression is also a quantified simple type expression. In the sequel, we use σ for a quantified type expression (perhaps with no quantifiers), and τ for a type expression without quantifiers.

A type variable is *free* in a quantified type expression if it appears in the body but not in the list of quantified variables. To give meaning to a quantified type, it is necessary to specify the meaning of its free variables. An *assignment* $\theta : \mathrm{Vars} \rightarrow \mathcal{D}_0$ is a map from variables to the semantic domain. The assignment $\theta[\alpha \leftarrow \tau]$ is the assignment θ modified at point α to return τ.

An assignment is extended from variables to (quantified) simple type expressions as follows:

Definition 3.2

1. $\theta(\top) = \top$
2. $\theta(\bot) = \bot$
3. $\theta(\tau_1 \rightarrow \tau_2) = \theta(\tau_1) \rightarrow \theta(\tau_2)$
4. $\theta(\forall\alpha.\tau) = \sqcap\{\theta[\alpha \leftarrow x](\tau) | x \in \mathcal{D}_0\}$

Note that unquantified simple type expressions are assigned meanings in $\mathcal{D}_0$ whereas quantified simple type expressions typically have meanings in $\mathcal{D}_1$ but not in $\mathcal{D}_0$.

Proposition 3.3 $\theta(\forall\alpha_1 \ldots \forall\alpha_n.\tau) = \sqcap\{\theta[\alpha_1 \leftarrow x_1 \ldots \alpha_n \leftarrow x_n](\tau) \mid x_1, \ldots, x_n \in \mathcal{D}_0\}$

Proof: Follows immediately from Definition 3.2. □

Our results for eliminating variables in quantified types hinge on knowledge about when two type expressions have the same meaning in the semantic domain. However, because type expressions may have free variables, the notion of equality must also take into account possible assignments to those free variables. We say that two quantified type expressions σ_1 and σ_2 are *equivalent*, written $\sigma_1 \equiv \sigma_2$, if for all assignments θ, we have $\theta(\sigma_1) = \theta(\sigma_2)$.

4 Simple Type Expressions

This section presents an algorithm for eliminating quantified type variables in simple type expressions and proves that the algorithm is sound. The following definition formalizes what it means to correctly eliminate as many variables from a type as possible:

Definition 4.1 A type expression σ is *irredundant* if for all σ' such that $\sigma' \equiv \sigma$, it is the case that σ has no more quantified variables than σ'.

In general, irredundant types are not unique. It is easy to show that renaming quantified variables does not change the meaning of a type, provided we observe the usual rules of capture. Thus, $\forall\alpha.\sigma \equiv \forall\beta.\sigma[\alpha \leftarrow \beta]$ provided that β is not free in σ. It is also true that types distinguished only by the order of quantified variables are equivalent. That is, $\forall\alpha.\forall\beta.\sigma \equiv \forall\beta.\forall\alpha.\sigma$. Our main result is that for every type there is a unique (up to renaming and reordering of bound variables) irredundant type that is equivalent.

Since equivalence ($\equiv$) is a semantic notion, irredundancy is also semantic in nature and cannot be determined by a trivial examination of syntax. The key question is: Under what circumstances can a type $\forall\alpha.\tau$ be replaced by some type $\tau[\alpha \leftarrow \tau']$ (for some type expression τ' not containing α)? From Definition 3.2, it follows that

$$\forall\alpha.\tau \equiv \tau[\alpha \leftarrow \tau']$$

if and only if for all assignments θ

$$\forall d \in \mathcal{D}_0.\ \theta(\tau[\alpha \leftarrow \tau']) \preceq \theta[\alpha \leftarrow d](\tau)$$

In other words, a type $\sigma = \forall\alpha.\tau$ is equivalent to $\tau[\alpha \leftarrow \tau']$ whenever for all assignments θ, we have $\theta(\tau[\alpha \leftarrow \tau'])$ is the minimal element of of the set $\{\theta[\alpha \leftarrow x](\tau) | x \in \mathcal{D}_0\}$ to which the glb operation is applied in computing σ's meaning under θ.

The difficulty in computing irredundant types is that the function-space constructor $\rightarrow$ is *anti-monotonic* in its first position. That is, $\tau_1 \preceq \tau_2$ implies that $\tau_1 \rightarrow \tau \succeq \tau_2 \rightarrow \tau$. Thus, determining the minimal element of a greatest lower bound computation may require maximizing or minimizing a variable, depending on whether the type is monotonic or anti-monotonic in that variable. Intuitively, to eliminate as many variables as possible, variables in anti-monotonic positions should be set to $\top$, while others in monotonic positions should be set to $\bot$. We define functions *Pos* and *Neg* that compute a type's set of monotonic and anti-monotonic variables, respectively.

Definition 4.2 *Pos* and *Neg* are defined as follows:

$$\begin{array}{rcl} Pos(\alpha) & = & \{\alpha\} \\ Pos(\tau_1 \rightarrow \tau_2) & = & Neg(\tau_1) \cup Pos(\tau_2) \\ Pos(\top) & = & \emptyset \\ Pos(\bot) & = & \emptyset \\ \\ Neg(\alpha) & = & \emptyset \\ Neg(\tau_1 \rightarrow \tau_2) & = & Pos(\tau_1) \cup Neg(\tau_2) \\ Neg(\top) & = & \emptyset \\ Neg(\bot) & = & \emptyset \end{array}$$

As an example, for the type $\alpha \rightarrow \beta$ we have

$$\begin{array}{rcl} Pos(\alpha \rightarrow \beta) & = & \{\beta\} \\ Neg(\alpha \rightarrow \beta) & = & \{\alpha\} \end{array}$$

The following lemma shows that *Pos* and *Neg* correctly characterize variables in monotonic and anti-monotonic positions respectively.

Lemma 4.3 Let $d', d \in \mathcal{D}_0$ where $d' \preceq d$. Let θ be any assignment.

1. If $\alpha \notin Pos(\tau)$, then $\theta[\alpha \leftarrow d](\tau) \preceq \theta[\alpha \leftarrow d'](\tau)$.
2. If $\alpha \notin Neg(\tau)$, then $\theta[\alpha \leftarrow d'](\tau) \preceq \theta[\alpha \leftarrow d](\tau)$.

Proof: This proof is an easy induction on the structure of τ.

- If $\tau = \bot$ or $\tau = \top$, then $\theta[\alpha \leftarrow d'](\tau) = \theta(\tau) = \theta[\alpha \leftarrow d](\tau)$, so both (1) and (2) hold.

- If $\tau = \alpha$, then $\alpha \in Pos(\alpha)$, so (1) holds vacuously. For (2), we have

$$\theta[\alpha \leftarrow d'](\alpha) = d' \preceq d = \theta[\alpha \leftarrow d](\alpha)$$

- Let $\tau = \tau_1 \to \tau_2$. We prove only (1), as the proof for (2) is symmetric. So assume that $\alpha \notin Pos(\tau)$. By the definition of Pos, we know

$$\alpha \notin Neg(\tau_1) \cup Pos(\tau_2)$$

Applying the lemma inductively to τ_1 and τ_2, we have

$$\begin{array}{rcl} \theta[\alpha \leftarrow d'](\tau_1) & \preceq & \theta[\alpha \leftarrow d](\tau_1) \\ \theta[\alpha \leftarrow d](\tau_2) & \preceq & \theta[\alpha \leftarrow d'](\tau_2) \end{array}$$

Combining these two lines using axiom 5 of a semantic domain (Definition 2.1) it follows that

$$\theta[\alpha \leftarrow d](\tau_1 \to \tau_2) \preceq \theta[\alpha \leftarrow d'](\tau_1 \to \tau_2)$$

which proves the result.

□

Corollary 4.4

1. If $\alpha \notin Pos(\tau)$, then $\theta(\tau[\alpha \leftarrow \top]) \preceq \theta(\tau) \preceq \theta(\tau[\alpha \leftarrow \bot])$ holds for all assignments θ.
2. If $\alpha \notin Neg(\tau)$, then $\theta(\tau[\alpha \leftarrow \bot]) \preceq \theta(\tau) \preceq \theta(\tau[\alpha \leftarrow \top])$ holds for all assignments θ.

4.1 Variable Elimination

Our algorithm for eliminating variables from quantified types is based on the computation of Pos and Neg. Before presenting the variable elimination procedure, we extend Pos and Neg to quantified types:

$$\begin{array}{rcl} Pos(\forall\alpha.\sigma) & = & Pos(\sigma) - \{\alpha\} \\ Neg(\forall\alpha.\sigma) & = & Neg(\sigma) - \{\alpha\} \end{array}$$

The following lemma gives sufficient conditions for a variable to be eliminated.

Lemma 4.5 If σ is a quantified simple type expression, then

$$\begin{array}{rcl} \alpha \notin Neg(\sigma) & \Rightarrow & \forall\alpha.\sigma \equiv \sigma[\alpha \leftarrow \bot] \\ \alpha \notin Pos(\sigma) & \Rightarrow & \forall\alpha.\sigma \equiv \sigma[\alpha \leftarrow \top] \end{array}$$

Proof: Assume first that $\forall\alpha.\sigma = \forall\alpha.\tau$ where τ is an unquantified simple type expression and that $\alpha \notin Neg(\tau)$. Note that

$$\begin{array}{lll} & \theta(\forall\alpha.\tau) & \\ = & \sqcap\{\theta[\alpha \leftarrow x](\tau) | x \in \mathcal{D}_0\} & \\ \preceq & \theta[\alpha \leftarrow \bot](\tau) & \text{since } \bot \text{ is a possible choice for } x \\ = & \theta(\tau[\alpha \leftarrow \bot]) & \\ = & \sqcap\{\theta[\alpha \leftarrow x](\tau[\alpha \leftarrow \bot]) | x \in \mathcal{D}_0\} & \text{since } \alpha \text{ does not occur in } \tau[\alpha \leftarrow \bot] \\ \preceq & \sqcap\{\theta[\alpha \leftarrow x](\tau) | x \in \mathcal{D}_0\} & \text{by part 2 of Corollary 4.4} \\ = & \theta(\forall\alpha.\tau) & \end{array}$$

Therefore, $\theta(\forall\alpha.\tau) = \theta(\tau[\alpha \leftarrow \bot])$ for all assignments θ. For the general (quantified) case $\forall\alpha_1, \ldots, \alpha_n.\tau$, observe that any variable α_i for $1 \leq i \leq n$ can be moved to the innermost position of the type by a sequence of bound variable interchanges and renamings, at which point the reasoning for the base case above can be applied. The proof for the second statement ($\alpha \notin \mathit{Pos}(\sigma)$) is symmetric. $\square$

We are interested in quantified types for which as many variables have been eliminated using the conditions of Lemma 4.5 as possible. Returning to our canonical example,

$$\begin{array}{lll} & \forall\alpha.\forall\beta.\alpha \to \beta & \\ \equiv & \forall\beta.\top \to \beta & \text{since } \alpha \notin \mathit{Pos}(\forall\beta.\alpha \to \beta) \\ \equiv & \top \to \bot & \text{since } \alpha \notin \mathit{Neg}(\top \to \beta) \end{array}$$

Definition 4.6 A quantified simple type expression σ is *reduced* if

- σ is unquantified; or
- $\sigma = \forall\alpha.\sigma'$ and furthermore $\alpha \in \mathit{Pos}(\sigma') \wedge \alpha \in \mathit{Neg}(\sigma')$ and σ' is reduced.

Note that the property of being reduced is distinct from the property of being irredundant. "Reduced" is a syntactic notion and does not depend on the semantic domain. Irredundancy is a semantic notion, because it involves testing the expression's meaning against the meaning of other type expressions.

Procedure 4.7 (Variable Elimination Procedure (VEP)) Given a quantified type expression $\forall\alpha_1 \ldots \forall\alpha_n.\tau$, compute the sets $\mathit{Pos}(\tau)$ and $\mathit{Neg}(\tau)$. Let $VEP(\forall\alpha_1 \ldots \forall\alpha_n.\tau)$ be the type obtained by:

1. dropping any quantified variable not used in τ,
2. setting any quantified variable α where $\alpha \notin \mathit{Pos}(\forall\alpha_1 \ldots \forall\alpha_n.\tau)$ to $\bot$,
3. setting any quantified variable α where $\alpha \notin \mathit{Neg}(\forall\alpha_1 \ldots \forall\alpha_n.\tau)$ to $\top$,
4. and retaining any other quantified variable.

Theorem 4.8 Let σ be any quantified simple type expression. Then $\sigma \equiv VEP(\sigma)$ and $VEP(\sigma)$ is reduced.

Proof: Equivalence follows easily from Lemma 4.5. To see that $VEP(\sigma)$ is reduced, observe that any quantified variable not satisfying conditions (1)–(3) of the Variable Elimination Procedure must occur both positively and negatively in the body of σ. $\square$

A few remarks on the Variable Elimination Procedure are in order. The algorithm can be implemented very efficiently. Only two linear passes over the structure of the type are needed: one to compute the *Pos* and *Neg* sets (which can be done using a using a hash-table or bit-vector implementation of sets) and another to perform any substitutions. In addition, the algorithm is idempotent, so $VEP(VEP(\sigma)) = VEP(\sigma)$.

Theorem 4.9 Every irredundant simple type expression is reduced.

Proof: Let σ be an irredundant simple type expression. Since σ is irredundant, $VEP(\sigma)$ has at least as many quantified variables as σ. Therefore $VEP(\sigma) = \sigma$; i.e., the Variable Elimination Procedure does not remove any variables from σ. Since $VEP(\sigma)$ is reduced, σ is a reduced simple type expression. $\square$

4.2 Completeness

If σ is a quantified simple type expression, then $VEP(\sigma)$ is an equivalent reduced simple type expression, possibly with fewer quantified variables. In this section, we address whether additional quantified variables can be eliminated from a reduced type. In other words, is a reduced simple type expression irredundant? We show that if the semantic domain $\mathcal{D}$ has standard function types (Definition 2.1) then every reduced simple type expression is irredundant (Theorem 4.20).

For semantic domains with standard function types, the Variable Elimination Procedure is complete in the sense that no other algorithm can eliminate more quantified variables and preserve equivalence. The completeness proof shows that whenever two reduced types are equivalent, then they are syntactically identical, up to renamings and reorderings of quantified variables.

To simplify the presentation that follows, we introduce some new notation and terminology. By analogy with the α-reduction of the lambda calculus, two quantified simple type expressions are *α-equivalent* iff either can be obtained from the other by a series of reorderings or capture-avoiding renamings of quantified variables. We sometimes use the notation $\forall\{\alpha_1, \ldots, \alpha_n\}.\tau$ to denote $\forall\alpha_1 \ldots \forall\alpha_n.\tau$. Using a set instead of an ordered list involves no loss of generality since duplicates never occur in reduced expressions and variable order can be permuted freely.

4.3 Constraint Systems

Proving completeness requires a detailed comparison of the syntactic structure of equivalent reduced types. This comparison is more intricate than might be expected; in addition, in the sequel we perform a similar analysis to prove that variable elimination is complete for recursive types. This section develops the technical machinery at the heart of both completeness proofs.

Definition 4.10 A *system of constraints* is a set of inclusion relations between unquantified simple type expressions $\{\ldots s \preceq t \ldots\}$. A *solution* of the constraints is any assignment θ such that $\theta(s) \preceq \theta(t)$ holds for all constraints $s \preceq t$ in the set.

Definition 4.11 gives an algorithm B that compares two unquantified simple type expressions t_1 and t_2. The comparison is expressed in terms of constraints; the function B transforms a constraint $t_1 \preceq t_2$ into a system of constraints on the variables of t_1 and t_2. Intuitively, $B(\{t_1 \preceq t_2\})$ summarizes what must be true about the variables of the two types whenever the relationship $t_1 \preceq t_2$ holds.

Definition 4.11 Let S be a set of unquantified constraints. $B(S)$ is a set of constraints defined by the following rules. These clauses are to be applied in order with the earliest one that applies taking precedence.

1. $B(\emptyset) = \emptyset$
2. $B(\{t \preceq t\} \cup S) = B(S)$.
3. $B(\{s_1 \to s_2 \preceq t_1 \to t_2\} \cup S) = B(\{t_1 \preceq s_1, s_2 \preceq t_2\} \cup S)$.
4. Otherwise, $B(\{s \preceq t\} \cup S) = \{s \preceq t\} \cup B(S)$.

Lemma 4.12 Let S be a system of constraints. If $\mathcal{D}$ is a semantic domain with standard function types, then every solution of S is a solution of $B(S)$.

Proof: Let the *complexity* of S be the pair *(number of $\rightarrow$ symbols in S, number of constraints in S)*. Complexity is ordered lexicographically, so $(i, j) < (i', j')$ if $i < i'$ or $i = i'$ and $j < j'$. The result is proven by induction on the complexity of S, with one case for each clause in the definition of B:

1. $B(\emptyset) = \emptyset$. The result clearly holds.

2. Since any assignment is a solution of $t \preceq t$, any solution θ of $\{t \preceq t\} \cup S$ is also a solution of S. By induction, θ is a solution of $B(S)$.

3. Let θ be a solution of $\{s_1 \rightarrow s_2 \preceq t_1 \rightarrow t_2\} \cup S$. Since the domain has standard function types, it follows that θ is also a solution of $\{t_1 \preceq s_1, s_2 \preceq t_2\} \cup S$. By induction, θ is a solution of $B(\{t_1 \preceq s_1, s_2 \preceq t_2\} \cup S)$.

4. In the final case, by induction every solution of S is a solution of $B(S)$. Therefore all solutions of $\{s \preceq t\} \cup S$ are solutions of $\{s \preceq t\} \cup B(S)$.

$\square$

The completeness proof uses an analysis of the constraints $B(t_1 \preceq t_2)$ where t_1 and t_2 are the bodies of reduced equivalent types. Observe that if t_1 and t_2 differ only in the names of variables, then $B(t_1 \preceq t_2)$ is a system of constraints between variables. Furthermore, it turns out that if t_1 and t_2 are actually renamings of each other, then the constraints $B(t_1 \preceq t_2)$ define this renaming in both directions. Proving this claim is a key step in the proof. This discussion motivates the following definition:

Definition 4.13 A system S of constraints is (V_1, V_2)-*convertible* iff V_1, V_2 are disjoint sets and there is a bijection f from V_1 to V_2 such that $S = \{\alpha \preceq f(\alpha) | \alpha \in V_1\} \cup \{f(\alpha) \preceq \alpha | \alpha \in V_1\}$

The idea behind Definition 4.13 is that if two reduced types $\forall V_1.\tau_1$ and $\forall V_2.\tau_2$ are α-convertible, then $B(\tau_1, \tau_2)$ is a (V_1, V_2)-convertible system of constraints (provided V_1 and V_2 are disjoint). It is easiest to prove this fact by first introducing an alternative characterization of convertible constraint systems, which is given in the following technical definition and lemma.

Definition 4.14 A system of constraints $\{s_1 \preceq t_1, \ldots, s_n \preceq t_n\}$ is (V_1, V_2)-*miniscule* iff the following all hold:

1. V_1 and V_2 are disjoint sets of variables.

2. for all $i \leq n$, at most one of s_i and t_i is a $\rightarrow$ expression.

3. for all $i \leq n$, s_i and t_i are different expressions.

4. for each $v \in V_1 \cup V_2$, there exists $i \leq n$ such that $v \in \mathit{Pos}(s_i) \cup \mathit{Neg}(t_i)$

5. for each $v \in V_1 \cup V_2$, there exists $i \leq n$ such that $v \in \mathit{Neg}(s_i) \cup \mathit{Pos}(t_i)$

6. for every assignment θ there is a assignment θ' such that $\theta(v) = \theta'(v)$ for all $v \notin V_1$ and $\theta'(s_i) \preceq \theta'(t_i)$ holds for all $i \leq n$.

7. for every assignment θ there is a assignment θ' such that $\theta(v) = \theta'(v)$ for all $v \notin V_2$ and $\theta'(t_i) \preceq \theta'(s_i)$ holds for all $i \leq n$. (Note the reverse order of t_i and s_i.)

Lemma 4.15 Any (V_1, V_2)-miniscule system of constraints is (V_1, V_2)-convertible.

Proof: Let θ_0 be the assignment that assigns $\perp$ to every variable, let θ_1 be the assignment that assigns $\top$ to every variable, and let S be a (V_1, V_2)-miniscule system of constraints. The first step is to show that no $\to$ expressions can occur in S. It is easy to check that if we reverse all inequalities we get a (V_2, V_1)-miniscule system of constraints. Thus, by symmetry, to show that $\to$ cannot occur in S it suffices to show that $\to$ cannot occur in any upper bound in S.

For the sake of obtaining a contradiction, assume that $s_i \preceq t'_i \to t''_i \in B(\tau_1 \preceq \tau_2)$. We show that each of the four possible forms for s_i is impossible.

1. $s'_i \to s''_i \preceq t'_i \to t''_i$ is ruled out by Property 2 of Definition 4.14.

2. $\perp \preceq t'_i \to t''_i$ is ruled out by Property 7 of Definition 4.14, since no assignment satisfies $t'_i \to t''_i \preceq \perp$.

3. $\top \preceq t'_i \to t''_i$ is ruled out by Property 6 of Definition 4.14, since no assignment satisfies $\top \preceq t'_i \to t''_i$.

4. If v is a variable not in V_1, let θ be the assignment mapping all variables to $\top$. Then Property 6 of Definition 4.14 is violated because for all θ' that agree with θ off of V_1, we have $\theta'(v) = \theta(v) = \theta_1(v) = \top \not\preceq \theta'(t'_i \to t''_i)$.

   If $v \in V_1$, let θ be the assignment mapping all variables to $\perp$. Note that $v \notin V_2$ since V_1 and V_2 are disjoint. Then Property 7 of Definition 4.14 is violated because for all θ' that agree with θ off of V_2, we have $\theta'(t'_i \to t''_i) \not\preceq \perp = \theta_0(v) = \theta(v) = \theta'(v)$.

This completes the proof that $\to$ cannot occur in S.

The next step is to show that $\perp$ cannot occur in S. By symmetry it suffices to show that $\perp$ cannot occur as an upper bound in S. There are three cases to consider.

1. $\perp \preceq \perp$ is ruled out by Property 3 in Definition 4.14.

2. $\top \preceq \perp$ is ruled out by Property 6 in Definition 4.14 since no assignment satisfies $\top \preceq \perp$.

3. If v is a variable not in V_1, let $\theta = \theta_1$. Then Property 6 in Definition 4.14 is violated since for all θ' that agree with θ off of V_1, we have that $\theta'(v) = \theta(v) = \theta_1(v) = \top \not\preceq \perp = \theta'(\perp)$.

   If $v \in V_1$, a complex case argument is needed because Property 6 is not directly violated. By Properties 5 and 2 of Definition 4.14, there is a constraint $s' \preceq v$ in S. There are four possible cases for s':

   (a) $s' = \perp$. In this case, $\perp \preceq v \preceq \perp$ is in S and hence Property 7 is violated by taking $\theta = \theta_1$.

   (b) $s' = \top$. In this case, $\top \preceq v \preceq \perp$ violates Property 6 since it is never satisfied by any assignment.

   (c) $s' = v' \in V_1$. In this case, $v' = v$ is ruled out by Property 3. So we may assume that v' and v are different variables. Property 7 is violated by taking $\theta = \theta_0[v \leftarrow \top]$ since if θ' agrees with θ off of V_2 the constraint $v \preceq v'$ is violated since $\theta'(v) = \theta(v) = \top \not\preceq \perp = \theta(v') = \theta'(v')$.

   (d) $s' = v' \notin V_1$. In this case, $v' \preceq v \preceq \perp$ violates Property 6 by taking $\theta = \theta_1$ since $\theta(v') = \top$.

This proves that $\perp$ cannot occur as an upper bound in S. By symmetry, $\perp$ can not occur as a lower bound in S, and hence $\perp$ cannot occur anywhere in S. An analogous argument shows that $\top$ can not occur anywhere in S either.

Thus, every element of S is of the form $v' \preceq v''$ for variables v', v''. We now show that $v', v'' \in V_1 \cup V_2$. Suppose that $v' \notin V_1 \cup V_2$. If $v'' \in V_1$, then Property 7 is violated by taking $\theta = \theta_0[v'' \leftarrow \top]$ since for all θ' that agree with θ off of V_2, we have that $\theta'(v'') = \theta(v'') = \top \not\preceq \perp = \theta(v') = \theta'(v')$. If $v'' \notin V_1$, then

Property 6 is violated by taking $\theta = \theta_0[v' \leftarrow \top]$ since for all θ' that agree with θ off of V_1, we have that $\theta'(v') = \theta(v') = \top \not\preceq \bot = \theta(v'') = \theta'(v'')$. Therefore, the supposition that $v' \notin V_1 \cup V_2$ is false and it follows that $v' \in V_1 \cup V_2$. A similar argument shows that $v'' \in V_1 \cup V_2$.

If both v' and v'' are in V_1, then Property 7 is violated by taking $\theta = \theta_0[v'' \leftarrow \top]$ since for all θ' that agree with θ off of V_2, we have that $\theta'(v'') = \theta(v'') = \top \not\preceq \bot = \theta(v') = \theta'(v')$. This shows that not both v' and v'' are in V_1. A symmetric argument shows that not both v' and v'' are in V_2. Thus, it follows that for every constraint $s_i \preceq t_i$ in S, either $s_i \in V_1$ and $t_i \in V_2$ or $s_i \in V_2$ and $t_i \in V_1$.

Next we show that if $v_0 \preceq v_1 \preceq v_2$, then $v_0 = v_2$. First assume that $v_1 \in V_1$. If v_0 and v_2 are different variables, then Property 6 is violated by taking $\theta = \theta_0[v_0 \leftarrow \top]$ since for all θ' that agree with θ off of V_1, we have that $\theta'(v_0) = \theta(v_0) = \top \not\preceq \bot = \theta(v_2) = \theta'(v_2)$. Hence, in the case that $v_1 \in V_1$, it follows that $v_0 = v_2$. A similar argument shows that if $v_1 \in V_2$, then $v_0 = v_2$.

The next goal is to show that for every $v_1 \in V_1$, there exists a unique $v_2 \in V_2$ such that $v_1 \preceq v_2$ is in S. By Property 4, there is at least one such v_2. Let v_2' be any variable such that $v_1 \preceq v_2'$ is in S. By Property 5, there is a v_0 such that $v_0 \preceq v_1$ is in S. It follows that $v_2 = v_0 = v_2'$ which proves that v_2 is unique.

Define a function f mapping V_1 to V_2 so that $v_1 \preceq f(v_1)$ is in S. By Property 5, for any $v_1 \in V_1$, there is a v_0 such that $v_0 \preceq v_1$ is in S. It follows that $v_0 = f(v_1)$. This proves that $S \subseteq \{\alpha \preceq f(\alpha) | \alpha \in V_1\} \cup \{f(\alpha) \preceq \alpha | \alpha \in V_1\}$. Since every constraint in S has the form $v' \preceq v''$ where either v' or v'' is in V_1 and since the upper and lower bounds are unique (because $v_0 \preceq v_1 \preceq v_2 \in S$ implies that $v_0 = v_2$), it follows that there are no extra elements of S. Therefore, $S = \{\alpha \preceq f(\alpha) | \alpha \in V_1\} \cup \{f(\alpha) \preceq \alpha | \alpha \in V_1\}$. Thus S is (V_1, V_2)-convertible as desired. □

4.4 From Constraints to Completeness

The definitions and lemmas of Section 4.3 are the building blocks of the completeness proof. Before finally presenting the proof, we need one last definition:

Definition 4.16 Two simple type expressions $\forall V_1.\tau_1$ and $\forall V_2.\tau_2$ are *compatible* iff $\forall V_1.\tau_1$ and $\forall V_2.\tau_2$ are equivalent reduced simple type expressions such that V_1 and V_2 are disjoint and no variable in V_1 occurs in τ_2 and no variable in V_2 occurs in τ_1.

The important part of the definition of compatibility is that the type expressions are reduced and equivalent. The conditions regarding quantified variables are there merely to simplify proofs. There is no loss of generality because α-conversion can be applied to convert any two equivalent reduced type expressions into compatible expressions.

Lemma 4.17 Let $\forall V_1.\tau_1$ and $\forall V_2.\tau_2$ be compatible type expressions. If the semantic domain has standard function types and standard glb types, then $B(\tau_1 \preceq \tau_2)$ is a (V_1, V_2)-miniscule system of constraints.

Proof: Let $B(\tau_1 \preceq \tau_2) = \{s_1 \preceq t_1, \ldots, s_n \preceq t_n\}$. We prove that the conditions in Definition 4.14 all hold:

1. By compatibility V_1 and V_2 are disjoint sets of variables.

2. By Part 2 of Definition 4.11 at most one of s_i and t_i is a $\rightarrow$ expression.

3. For all $i \leq n$, s_i and t_i are different expressions by Part 3 of Definition 4.11.

4. Let $v \in V_1 \cup V_2$. We claim there is an $i \leq n$ such that $v \in Pos(s_i) \cup Neg(t_i)$. This fact is proven by induction on the number of steps needed to compute $B(\{\tau_1 \preceq \tau_2\})$ using the fact the expressions are reduced and hence all variables in both V_1 and V_2 occur both positively and negatively.

5. Proof similar to the previous step.

6. Let θ be any assignment. We must show that there is an assignment θ' such that $\theta(v) = \theta'(v)$ for all $v \notin V_1$ and $\theta'(s_i) \preceq \theta'(t_i)$ holds for all $i \leq n$. Since $\theta(\forall V_1.\tau_1) \preceq \theta(\forall V_2.\tau_2)$, it follows that $\theta(\forall V_1.\tau_1) \preceq \theta(\tau_2)$. Since the semantic domain has standard glb types, it follows that $\theta'(\tau_1) \preceq \theta(\tau_2)$ holds for some θ' that agrees with θ except possibly on V_1. Since no variable in V_1 occurs in τ_2, we know $\theta'(\tau_1) \preceq \theta'(\tau_2)$. By Lemma 4.12, it follows that θ' is a solution to $B(\tau_1 \preceq \tau_2)$.

7. To show that for every assignment θ there is an assignment θ' such that $\theta(v) = \theta'(v)$ for all $v \notin V_2$ and $\theta'(t_i) \preceq \theta'(s_i)$ holds for all $i \leq n$, reverse the roles of τ_1 and τ_2. This argument relies on the fact that $B(\tau_2 \preceq \tau_1)$ can be obtained from $B(\tau_1 \preceq \tau_2)$ by reversing the direction of the $\preceq$ symbol.

□

We are now ready to state and prove the first of the major theorems concerning completeness.

Theorem 4.18 If the semantic domain has standard function types and standard glb types, then any two reduced simple type expressions are equivalent iff they are α-equivalent.

Proof: The if-direction is clear and does not even require that the semantic domain have standard function types. To prove the only-if direction, let σ' and σ'' be two reduced simple type expressions. If necessary, α-convert σ' to $\sigma_1 = \forall V_1.\tau_1$ and α-convert σ'' to $\sigma_2 = \forall V_2.\tau_2$ so that σ_1 and σ_2 are (V_1, V_2) compatible. It suffices to show that σ_1 and σ_2 are α-equivalent.

By Lemma 4.17, $B(\tau_1 \preceq \tau_2)$ is a (V_1, V_2)-miniscule system of constraints. By Lemma 4.15, $B(\tau_1 \preceq \tau_2)$ is a (V_1, V_2)-convertible system of constraints; let F be the corresponding bijection mapping variables in V_2 to V_1. A simple induction using the definition of B shows that

$$B(\tau_1 \preceq F(\tau_2)) = \emptyset$$

from which it follows (again using the definition of B) that $\tau_1 = F(\tau_2)$. This shows that σ_1 and σ_2 are α-equivalent as desired. □

Corollary 4.19 If the semantic domain has standard function types, then no two different unquantified simple type expressions are equivalent.

Proof: Given a semantic domain $\mathcal{D}$ construct another semantic domain $\mathcal{D}'$ such that $\mathcal{D}_0 = \mathcal{D}'_0$ and $\mathcal{D}'$ has standard glb types using the construction in Example 2.3. Using the semantic domain $\mathcal{D}'$ suffices because the meaning of an unquantified type expression is always an element of $\mathcal{D}_0$ and $\mathcal{D}'_0 = \mathcal{D}_0$. If τ and τ' are equivalent, unquantified simple type expressions, then they are reduced and hence α-equivalent by Theorem 4.18. But since they have no quantifiers, α-equivalence implies that $\tau = \tau'$. □

Finally, the following theorem states our main result.

Theorem 4.20 If the semantic domain has standard function types and standard glb types, then a simple type expression is reduced iff it is irredundant.

Proof: The if direction follows from Theorem 4.9. To prove the only-if direction, let σ be a reduced simple type expression with the goal of proving that σ is irredundant. Let σ' be an irredundant type that is equivalent to σ. (Such a σ' can always be found by picking it to be a type expression equivalent to σ with the smallest possible number of quantified variables.) By Theorem 4.9, σ' is reduced. By Theorem 4.18, σ is α-equivalent to σ'. Therefore, it follows that σ and σ' have the same number of quantified variables. Hence, σ is irredundant as desired. □

Theorem 4.20 shows that a syntactic test (reduced) is equivalent to a semantic test (irredundant). Theorem 4.20 requires that the semantic domain has standard function types. The following examples show that this assumption is necessary.

Example 4.21 Consider the minimal semantic domain (Example 2.2). It is clear that $\forall\alpha.(\alpha \to \alpha) \equiv (\top \to \top)$ in the minimal semantic domain. Therefore, $\forall\alpha.(\alpha \to \alpha)$ is reduced but not irredundant.

Example 4.22 In the semantic domain used in [AW93], $x \to \top = y \to \top$ regardless of the values of x and y, because if the answer can be anything (i.e., $\top$), it does not matter what the domain is. In this case, $\forall\alpha.((\alpha \to \alpha) \to \top) \equiv \top \to \top$. Thus, $\forall\alpha.((\alpha \to \alpha) \to \top)$ is not irredundant even though it is reduced.

Theorem 4.23 shows that the Variable Elimination Procedure (Procedure 4.7) is complete provided that the semantic domain has standard function types.

Theorem 4.23 Let σ be a quantified simple type expression. If the semantic domain has standard function types and standard glb types, then $VEP(\sigma)$ is an irredundant simple type expression equivalent to σ.

Proof: Follows easily from Theorem 4.8 and Theorem 4.20. □

To summarize, for simple type expressions the Variable Elimination Procedure that removes quantified variables occurring positively or negatively in a type produces an equivalent type with the minimum number of quantified variables. Furthermore, this type is unique up to the renaming and order of quantified variables.

A good feature of Theorem 4.23 is that the irredundant type expression produced by the Variable Elimination Procedure has no more arrows than the original type expression. This need not be the case if the semantic domain does not have standard function types.

Example 4.24 Let $\mathcal{D}_0 = \mathcal{D}_1 = \{\bot, \top \to \bot, x, \bot \to \bot, \top \to \top, \bot \to \top, \top\}$ where x is a function type, $\top \to \bot$ is less than x, and x is less than the other three function types. If either y_0 or y_1 is a function type, then $y_0 \to y_1 = x$. Therefore, $\theta(\forall\alpha.(\alpha \to \alpha)) = x$ for all assignments θ (since, for example, $(\bot \to \bot) \to (\bot \to \bot) = x$). This implies that $\forall\alpha.(\alpha \to \alpha) \equiv (\bot \to \bot) \to \bot$. Even though $\forall\alpha.(\alpha \to \alpha)$ has only one arrow, every irredundant type expression equivalent to $\forall\alpha.(\alpha \to \alpha)$ has at least two arrows.

5 Recursive Type Expressions

This section extends the basic variable elimination algorithm to a type language with recursive types. The proofs of soundness and completeness parallel the structure of the corresponding proofs for the non-recursive case.

New issues arise in two areas. First, there is new syntax for recursive type equations, which requires corresponding extensions to the syntax-based algorithms (*Pos*, *Neg*, and B). Second, two new conditions on the semantic domain are needed. Roughly speaking, the two conditions are (a) that recursive equations have solutions in the semantic domain (which is needed to give meaning to recursive type expressions) and (b) that the ordering $\preceq$ satisfies a continuity property (which is required to guarantee correctness of the *Pos* and *Neg* computations). It is surprising that condition (b) is needed not just for completeness, but even for soundness. Fortunately, standard models of recursive types (including the ideal model and regular trees) satisfy both conditions.

5.1 Preliminaries

We begin by defining a type language with recursive types. We first require the technical notion of a *contractive equation*.

Definition 5.1 Let $\delta_1, \ldots, \delta_n$ be distinct type variables and let $\tau_1, \ldots, \tau_n$ be unquantified simple type expressions. A variable α is *contractive in an equation* $\delta_1 = \tau_1$ if every occurrence of α in τ_1 is inside a constructor (such as $\rightarrow$). A system of equations

$$\delta_1 = \tau_1 \wedge \ldots \wedge \delta_n = \tau_n$$

is *contractive* iff each δ_i is contractive in every equation of the system.

Contractiveness is a standard technical condition in systems with recursive types [MPS84]. Contractiveness is necessary for equations to have unique solutions (e.g., an equation such as $\delta = \delta$ may have many solutions). The results of this section only apply to systems of contractive equations.

Definition 5.2 An *(unquantified) recursive type expression* is of the form: τ/E where E is a set of contractive equations and τ is an unquantified simple type expression.

Throughout this section, we use $\delta, \delta_1, \delta', \ldots$ for the *defined* variables that are given definitions in the set of equations E, and we use $\alpha, \alpha', \alpha_1, \ldots$ to indicate the *regular* variables, i.e., those that are not given definitions. To give meaning to recursive type expressions, the equations in a recursive type must have solutions in the semantic domain. The following definition formalizes this requirement.

Definition 5.3 A semantic domain has *contractive solutions* iff for every contractive system E of equations

$$\delta_1 = \tau_1 \wedge \ldots \wedge \delta_n = \tau_n$$

and for every assignment θ, there exists a unique assignment θ^E such that the following hold:

1. $\theta^E(\alpha) = \theta(\alpha)$ for all $\alpha \notin \{\delta_1, \ldots, \delta_n\}$
2. $\theta^E(\delta_i) = \theta^E(\tau_i)$ for all $i = 1, \ldots, n$.

Furthermore, if $\theta_1(\alpha) = \theta_2(\alpha)$ for every regular variable α of E, then $\theta_1^E(\delta) = \theta_2^E(\delta)$ for every defined variable δ of E.

Note that Definition 5.3 is well-formed because assignments are applied only to unquantified simple type expressions, an operation that already has meaning (see Definition 3.2). An assignment is extended to (quantified) recursive type expressions as follows:

Definition 5.4

1. $\theta(\tau/E) = \theta^E(\tau)$ for any unquantified simple type expression τ.

2. $\theta(\forall\alpha.\tau/E) = \sqcap\{\theta[\alpha \leftarrow x](\tau/E) | x \in \mathcal{D}_0\}$

Just as for simple type expressions, every unquantified simple type expression is assigned a meaning in $\mathcal{D}_0$ whereas quantified simple type expressions typically have meanings that are in $\mathcal{D}_1$ but not in $\mathcal{D}_0$. Lemma 5.5 shows that if a domain has contractive solutions, then definitions of "unused" variables can be dropped.

Lemma 5.5 Assume the domain has contractive solutions. Let E by a set of equations a let $E' \subseteq E$. Assume that whenever δ is a defined variable of E and δ occurs in τ_0/E', then δ is a defined variable of E'. Then $\tau_0/E \equiv \tau_0/E'$.

Proof: Let θ be any assignment. First note that $\theta^E(\delta_i) = \theta^E(\tau_i)$ for each equation $\delta_i = \tau_i$ in E'. By uniqueness of $(\theta^E)^{E'}$, it follows that $(\theta^E)^{E'} = \theta^E$. Since θ^E and θ agree on the regular variables of E', it follows that $(\theta^E)^{E'}$ and $\theta^{E'}$ agree on the defined variables of E'. If a variable α occurs in τ_0 and α is a regular variable of E', then α is also a regular variable of E and hence $(\theta^E)^{E'}(\alpha) = \theta^{E'}(\alpha)$. Hence $(\theta^E)^{E'}$ and $\theta^{E'}$ agree on all the variables that occur in τ_0. Therefore, $(\theta^E)^{E'}(\tau_0) = \theta^{E'}(\tau_0)$. Putting these facts together gives us that $\theta(\tau_0/E) = \theta^E(\tau_0) = (\theta^E)^{E'}(\tau_0) = \theta^{E'}(\tau_0) = \theta(\tau_0/E')$. Therefore, $\tau_0/E \equiv \tau_0/E'$ as desired. □

Surprisingly, even though contractive solutions guarantee that equations have unique solutions, this is not sufficient for soundness of the Variable Elimination Procedure. The crux of the problem is found in the reasoning that justifies using *Pos* and *Neg* as the basis for replacing variables by $\top$ or $\bot$ (Lemma 4.3). The *Pos* and *Neg* algorithms traverse a type expression to compute the set of positive and negative variables of the expression. In the case of recursive types, *Pos* and *Neg* can be regarded as using finite unfoldings of the recursive equations. We must ensure that these finite approximations correctly characterize the limit, which is the "infinite" unfolding of the equations. Readers familiar with denotational semantics will recognize this requirement as a kind of continuity property. Definition 5.7 defines *type continuity*, which formalizes the appropriate condition. Later in this section we give an example showing that type continuity is in fact necessary.

Definition 5.6 A *definable operator* is a function $F : \mathcal{D}_0 \to \mathcal{D}_0$ such that there is a recursive type expression $\tau_0/\bigwedge_{i=1}^m \delta_i = \tau_i$, a substitution θ, and a (regular) variable α such that α is contractive in all equations and

$$F(d) = \theta[\alpha \leftarrow d](\tau_0/\bigwedge_{i=1}^{m} \delta_i = \tau_i)$$

holds for all $d \in \mathcal{D}_0$.

Definition 5.7 A semantic domain $\mathcal{D}$ has *type-continuity* iff for every monotonic, definable operator F and every $d', d'' \in \mathcal{D}_0$,

$$(F(d'') = d'' \ \wedge \ F(d') \preceq d') \ \Rightarrow \ d'' \preceq d'$$

The minimal semantic model (Example 2.2) has contractive solutions, type continuity, and standard glb types, but it lacks standard function types. The standard semantic model (Example 2.3) has standard glb types and standard function types but lacks contractive solutions (e.g., because the equation $\delta = \delta \to \delta$ has no solution). The standard model does have type continuity, but without contractive solutions type continuity is not very interesting; for the standard model, the only monotonic definable operators with a fixed point are constant functions. The standard semantic model can be extended to the usual regular tree model to provide contractive solutions without sacrificing the other properties.

Lemma 5.8 The usual semantic domain of regular trees has contractive solutions, standard glb types, standard function types, and type continuity.

Proof: We briefly sketch the usual semantic domain $\mathcal{D}_0$ of regular trees. This discussion is not intended to give a detailed construction of the domain, but rather to highlight the important features. As usual, $\mathcal{D}_1$ consists of the non-empty upward closed subsets of $\mathcal{D}_0$. Therefore, the semantic domain has standard glb types.

A finite or infinite tree is *regular* if it has only a finite number of subtrees. The set $\mathcal{D}_0$ consists of the regular trees built from $\top$ and $\perp$ using the $\rightarrow$ operator. Thus, $\top$ and $\perp$ are elements of $\mathcal{D}_0$ and every other element x of $\mathcal{D}_0$ is equal to $x' \rightarrow x''$ for some $x', x'' \in \mathcal{D}_0$. Furthermore, x' and x'' are unique. It is well-known that such a domain has contractive solutions.

Let $x \preceq_0 y$ hold for all $x, y \in \mathcal{D}_0$. Let $x \preceq_{i+1} y$ hold iff $x = \perp$ or $y = \top$ or $x = x' \rightarrow x''$ and $y = y' \rightarrow y''$ and $x'' \preceq_i y''$ and $y' \preceq_i x'$. Notice that $\preceq_{i+1} \subseteq \preceq_i$. Then $x \preceq y$ holds iff $x \preceq_i y$ holds for all $i \geq 0$.

First we check that $\preceq$ has standard function types.

$$\begin{array}{ll} & x' \rightarrow x'' \preceq y' \rightarrow y'' \\ \Leftrightarrow & \forall i (x' \rightarrow x'' \preceq_{i+1} y' \rightarrow y'') \\ \Leftrightarrow & \forall i (x'' \preceq_i y'' \text{ and } y' \preceq_i x') \\ \Leftrightarrow & \forall i (x'' \preceq_i y'') \text{ and } \forall i (y' \preceq_i x') \\ \Leftrightarrow & x'' \preceq y'' \text{ and } y' \preceq x' \end{array}$$

Thus $\preceq$ has standard function types.

Next we check that $\mathcal{D}_0$ has type continuity. Let $x =_i y$ stand for $x \preceq_i y$ and $y \preceq_i x$. Let F be a definable operator. For any $x, y \in \mathcal{D}_0$,

$$F^i(x) =_i F^i(y)$$

This fact follows by induction on i, using the fact that F is contractive in its argument. Let d' and d'' be elements of $\mathcal{D}_0$ such that $F(d'') = d''$ and $F(d') \preceq d'$. It is easy to see by induction that $F^i(d'') = d''$ and, using monotonicity of F, that $F^i(d') \preceq d'$. Therefore, we have

$$d'' = F^i(d'') =_i F^i(d') \preceq d'$$

for all i. Hence, $d'' \preceq_i d'$ holds for all i. By definition of $\preceq$, it follows that $d'' \preceq d'$ and we conclude that the domain has type continuity. $\square$

5.2 Soundness

In this section, we extend variable elimination to recursive types. The first step is to extend the notion of *Pos* and *Neg* to include defined variables (recall that defined variables are denoted by δ):

Definition 5.9 *Pos* and *Neg* are sets of regular variables such that

1. If α is not defined in E, then $Pos(\alpha/E) = \{\alpha\}$ and $Neg(\alpha/E) = \emptyset$.
2. If $\delta = \tau$ is in E, then $Pos(\delta/E) = Pos(\tau/E)$ and $Neg(\delta/E) = Neg(\tau/E)$.
3. $Pos(\perp/E) = Neg(\perp/E) = \emptyset$
4. $Pos(\top/E) = Neg(\top/E) = \emptyset$

5. $Pos(\tau_1 \to \tau_2/E) = Pos(\tau_2/E) \cup Neg(\tau_1/E)$
   and $Neg(\tau_1 \to \tau_2/E) = Neg(\tau_2/E) \cup Pos(\tau_1/E)$

Many functions *Pos* and *Neg* satisfy these equations. For example, choosing

$$Pos(\delta/\delta = \delta \to \delta) = Neg(\delta/\delta = \delta \to \delta) = \{\alpha_4, \alpha_{29}\}$$

satisfies the equations, but the least solution is

$$Pos(\delta/\delta = \delta \to \delta) = Neg(\delta/\delta = \delta \to \delta) = \emptyset$$

Our results apply to any functions *Pos* and *Neg* satisfying Definition 5.9, but an implementation should compute the smallest possible sets of positive and negative variables. It is easy to construct the least sets for *Pos* and *Neg* by adding variables only as necessary to satisfy the clauses of Definition 5.9. Notice that a defined variable never occurs in the answer set of either *Pos* or *Neg*.

Lemma 5.10 Let τ be a recursive type expression. If the semantic domain has contractive solutions and type continuity, then

1. if $\alpha \notin Pos(\tau)$ and $d_1 \preceq d_2$, then $\theta[\alpha \leftarrow d_2](\tau) \preceq \theta[\alpha \leftarrow d_1](\tau)$ holds for all assignments θ.
2. if $\alpha \notin Neg(\tau)$ and $d_1 \preceq d_2$, then $\theta[\alpha \leftarrow d_1](\tau) \preceq \theta[\alpha \leftarrow d_2](\tau)$ holds for all assignments θ.

Proof: Let $\tau = \tau_0/E$. The result is proven by induction on the number of equations in E with a sub-induction on the structure of τ_0. The sub-induction on τ_0's structure proceeds as in Lemma 4.3. The interesting case is the new base case where τ_0 is a defined variable δ_1 with $\delta_1 = \tau_1$ in E.

Assume $\tau = \delta_1/E$ where $\delta_1 = \tau_1$ is an equation in E. If $\alpha \in Pos(\delta_1/E)$ and $\alpha \in Neg(\delta_1/E)$, then the result is vacuously true. If $\alpha \notin Pos(\delta_1/E)$ and $\alpha \notin Neg(\delta_1/E)$, then let E' be those equations in E that do not contain α and do not (recursively) refer to a defined variable that contains α in its definition. Using Lemma 5.5, it can be shown that $\delta_1/E' \equiv \delta_1/E$, so it suffices to prove the result for δ_1/E'. Notice that α does not occur in δ_1/E'. It is easy to check that $\theta[\alpha \leftarrow d](\delta_1/E') = \theta(\delta_1/E')$ holds for all $d \in \mathcal{D}_0$ and the result follows.

If $\alpha \notin Pos(\delta_1/E)$ and $\alpha \in Neg(\delta_1/E)$, then let E' be E with the equation $\delta_1 = \tau_1$ deleted. If δ_1 were in $Neg(\tau_1/E')$, then the fact that $\alpha \in Neg(\delta_1/E)$ would force α to be in $Pos(\delta_1/E)$ as well, which is contradiction. Hence, $\delta_1 \notin Neg(\tau_1/E')$. Fix an assignment θ. For each $d_0 \in \mathcal{D}_0$, define $F_{d_0}(d) = \theta[\alpha \leftarrow d_0][\delta_1 \leftarrow d](\tau_1/E')$. It is clear that F_{d_0} is a definable operator. By the induction hypothesis, F_{d_0} is a monotonic operator. It is easy to see that $\alpha \notin Pos(\tau_1/E')$, so it also follows from the induction hypothesis that F is anti-monotonic in its subscript. More formally, if $d_1 \preceq d_2$, then $F_{d_2}(d) \preceq F_{d_1}(d)$ holds for every $d \in \mathcal{D}_0$.

Define a function h on $\mathcal{D}_0$ by

$$h(d_0) = \theta[\alpha \leftarrow d_0](\delta_1/E)$$

Now we have

$$\begin{array}{lll}
& F_{d_0}(h(d_0)) & \\
= & \theta[\alpha \leftarrow d_0][\delta_1 \leftarrow h(d_0)](\tau_1/E') & \\
= & (\theta[\alpha \leftarrow d_0][\delta_1 \leftarrow h(d_0)])^{E'}(\tau_1) & \\
= & ((\theta[\alpha \leftarrow d_0]^{E})^{E'}(\tau_1) & \text{by furthermore clause of Definition 5.3} \\
= & (\theta[\alpha \leftarrow d_0])^{E}(\tau_1) & \text{by uniqueness of } (\theta^E)^{E'} \text{ relative to } \theta^E \text{ in Definition 5.3} \\
= & (\theta[\alpha \leftarrow d_0])^{E}(\delta_1) & \\
= & \theta[\alpha \leftarrow d_0](\delta_1/E) & \\
= & h(d_0) &
\end{array}$$

Thus, $F_{d_0}(h(d_0)) = h(d_0)$ holds for every $d_0 \in \mathcal{D}_0$. Let $d_1 \preceq d_2$. $F_{d_2}(h(d_2)) = h(d_2)$. $F_{d_2}(h(d_1)) \preceq$

$F_{d_1}(h(d_1)) = h(d_1)$. By type continuity, $h(d_2) \preceq h(d_1)$ which is the desired result.

If $\alpha \in \mathit{Pos}(\delta_1/E)$ and $\alpha \notin \mathit{Neg}(\delta_1/E)$, then the proof is omitted since it is similar to the case where $\alpha \notin \mathit{Pos}(\delta_1/E)$ and $\alpha \in \mathit{Neg}(\delta_1/E)$. Like the previous case, F is monotonic in its argument; unlike the previous case, F is monotonic in its subscript. $\Box$

Corollary 5.11 Assume the semantic domain has type continuity and contractive solutions.

1. If $\alpha \notin \mathit{Pos}(\tau/E)$, then $\theta((\tau/E)[\alpha \leftarrow \top]) \preceq \theta(\tau/E) \preceq \theta((\tau/E)[\alpha \leftarrow \bot])$ holds for all assignments θ.
2. If $\alpha \notin \mathit{Neg}(\tau/E)$, then $\theta((\tau/E)[\alpha \leftarrow \bot]) \preceq \theta(\tau/E) \preceq \theta((\tau/E)[\alpha \leftarrow \top])$ holds for all assignments θ.

The rest of the soundness results proceed as before. In particular, the Variable Elimination Procedure remains unaffected, except that it uses the new definitions of *Pos* and *Neg*. Just as in Section 4, we extend *Pos* and *Neg*:

$$\begin{aligned} \mathit{Pos}(\forall\alpha.\sigma) &= \mathit{Pos}(\sigma) - \{\alpha\} \\ \mathit{Neg}(\forall\alpha.\sigma) &= \mathit{Neg}(\sigma) - \{\alpha\} \end{aligned}$$

Lemma 5.12 If σ is a quantified recursive type expression and the semantic domain has type continuity and contractive solutions, then

$$\begin{aligned} \alpha \notin \mathit{Neg}(\sigma) &\Rightarrow \forall\alpha.\sigma \equiv \sigma[\alpha \leftarrow \bot] \\ \alpha \notin \mathit{Pos}(\sigma) &\Rightarrow \forall\alpha.\sigma \equiv \sigma[\alpha \leftarrow \top] \end{aligned}$$

Proof: Same as the proof for Lemma 4.5. $\Box$

Example 5.13 Let $\sigma = (\delta_3 \to \delta_3) \to (\delta_2 \to \delta_1)/\delta_1 = \alpha_1 \to \delta_1 \wedge \delta_2 = \alpha_2 \to \delta_2 \wedge \delta_3 = \alpha_3 \to \delta_3$. Note that $\mathit{Pos}(\sigma) = \{\alpha_2, \alpha_3\}$ and $\mathit{Neg}(\sigma) = \{\alpha_1, \alpha_3\}$. Assuming that the semantic domain has contractive solutions and type continuity, Lemma 5.12 allows us to conclude that

$$\begin{aligned} &\forall\alpha_1\forall\alpha_2\forall\alpha_3.((\delta_3 \to \delta_3) \to (\delta_2 \to \delta_1)/\ \delta_1 = \alpha_1 \to \delta_1 \wedge \delta_2 = \alpha_2 \to \delta_2 \wedge \delta_3 = \alpha_3 \to \delta_3) \\ \equiv\ &\forall\alpha_3.((\delta_3 \to \delta_3) \to (\delta_2 \to \delta_1)/\ \delta_1 = \top \to \delta_1 \wedge \delta_2 = \bot \to \delta_2 \wedge \delta_3 = \alpha_3 \to \delta_3) \end{aligned}$$

The next example shows that the assumption of type continuity is needed in the proof of Lemma 5.12.

Example 5.14 Consider the type expression $\forall\alpha.(\delta/\delta = \alpha \to \delta)$. If Lemma 5.12 holds, then we have

$$\begin{aligned} &\delta/\delta = \top \to \delta \\ \equiv\ &\forall\alpha.(\delta/\delta = \alpha \to \delta) && \text{by Lemma 5.12} \\ \preceq\ &(\delta/\delta = \bot \to \delta) && \text{since } \bot \text{ is an instance of } \alpha \end{aligned}$$

Let $\delta_0 = \top \to \delta_0$ and $\delta_1 = \bot \to \delta_1$ be elements of the semantic domain. Any semantic domain in which it is *not* the case that $\delta_0 \preceq \delta_1$ serves as a counterexample to the conclusion of Lemma 5.12.

Take the semantic domain to be the set of regular trees and define $x \preceq'_0 y$ to hold iff $x = y$. Let $x \preceq'_{i+1} y$ hold iff $x = \bot$, $y = \top$, or $\exists x_1, x_2, y_1, y_2 (x = x_1 \to x_2 \wedge y = y_1 \to y_2 \wedge y_1 \preceq'_i x_1 \wedge x_2 \preceq'_i y_2)$. Let $x \preceq' y$ hold iff $x \preceq'_i y$ for some i. Next, notice that $\delta_0 \preceq'_{i+1} \delta_1$ iff $\top \to \delta_0 \preceq'_{i+1} \bot \to \delta_1$ iff $\bot \preceq'_i \top \wedge \delta_0 \preceq'_i \delta_1$. It is easy to see by induction that $\delta_0 \not\preceq'_i \delta_1$ is true for all i. Hence, $\delta_0 \not\preceq' \delta_1$. Thus, the conclusion of Lemma 5.12 does not hold for this semantic domain.

This semantic domain has contractive solutions, standard function types, and standard glb types. What it lacks is type continuity, and it is instructive to see why. Consider the two definable operators:

$$\begin{aligned} F_\bot(d) &= [\alpha \leftarrow d](\bot \to \alpha) \\ F_\top(d) &= [\alpha \leftarrow d](\top \to \alpha) \end{aligned}$$

Let $\top \to \top \to \ldots$ be the infinite regular tree where $\top$ appears in the domain of every "$\to$". Observe that

$$F_\top(\top \to \top \to \ldots) = \top \to \top \to \ldots$$

Note that for all d we have $F_\top(d) \preceq' F_\bot(d)$. In particular,

$$F_\top(\bot \to \bot \to \ldots) \preceq' F_\bot(\bot \to \bot \to \ldots) = \bot \to \bot \to \ldots$$

If the domain had type continuity, it would follow that

$$\top \to \top \to \ldots \preceq' \bot \to \bot \to \ldots$$

As shown above, this relation does not hold, so therefore the domain does not have type continuity.

As discussed at the beginning of this section, type continuity is needed to guarantee that the finite computation performed by *Pos* and *Neg* is consistent with the orderings on all finite and infinite trees. Example 5.14 shows how the problem arises when $x \not\preceq' y$, but $x \preceq_i y$ for all i (where $\preceq_i$ is the relation used in Lemma 5.8 to define the usual ordering on regular trees). Thus, in contrast to the case of simple expressions where no additional assumptions on the semantic domain are needed for soundness, type continuity is needed to prove soundness for recursive type expressions.

Theorem 5.15 Let σ be any quantified recursive type expression. If the semantic domain has type continuity and contractive solutions, then $\sigma \equiv VEP(\sigma)$ and $VEP(\sigma)$ is a reduced recursive type expression.

Proof: Follows easily from Lemma 5.12. $\square$

Theorem 5.16 If the semantic domain has type continuity and contractive solutions, then every irredundant recursive type expression is reduced.

Proof: Same as the proof of Theorem 4.9. $\square$

5.3 Completeness

In this section, we face concerns similar to those found in Section 4.2.

Definition 5.17 Let S be a set of unquantified constraints. Define $B(S)$ to be the smallest set of constraints such that the following all hold. These clauses are to be applied in order, with the earliest one that applies taking precedence.

1. $B(\emptyset) = \emptyset$
2. If t is $\top$, $\bot$, or a regular variable, then $B(\{t/E \preceq t/E'\} \cup S) = B(S)$.
3. $B(\{s_1 \to s_2/E \preceq t_1 \to t_2/E'\} \cup S) = B(\{t_1/E' \preceq s_1/E, s_2/E \preceq t_2/E'\} \cup S)$.

4. If $\delta = \tau$ is in E, then $B(\{\delta/E \preceq t\} \cup S) = B(\{\tau/E \preceq t\} \cup S)$.

5. If $\delta = \tau$ is in E', then $B(\{s \preceq \delta/E'\} \cup S) = B(\{s \preceq \tau/E'\} \cup S)$.

6. Otherwise, $B(\{s/E \preceq t/E'\} \cup S) = \{s \preceq t\} \cup B(S)$.

Lemma 5.18 Assume that $\mathcal{D}$ is a semantic domain with contractive solutions, standard function types, and standard glb types. If θ is a solution of $\{t_1/E \preceq t_2/E'\}$, then it is a solution of $B(\{t_1/E \preceq t_2/E'\})$.

Proof: The proof is very similar to the proof of Lemma 4.12 and so is omitted. The most important new case is Part 6 of Definition 5.17. In this clause, note that dropping the associated constraints may increase the set of solutions, which is permitted by the statement of the lemma. □

Lemma 5.19 Let $\forall V_1.\tau_1$ and $\forall V_2.\tau_2$ be compatible recursive type expressions. If the semantic domain has contractive solutions, standard function types, and standard glb types, then $B(\{\tau_1 \preceq \tau_2\})$ is a (V_1, V_2)-miniscule system of constraints.

Proof: Let $B(\{\tau_1 \preceq \tau_2\}) = \{s_1 \preceq t_1, \ldots, s_n \preceq t_n\}$. We show that $B(\{\tau_1 \preceq \tau_2\})$ satisfies the conditions of Definition 4.14.

1. By compatibility V_1 and V_2 are disjoint sets of variables.

2. By Part 3 of Definition 5.17 at most one of s_i and t_i is a $\rightarrow$ expression.

3. Consider a constraint $t \preceq t$. Constraints of the form $\bot \preceq \bot$, $\top \preceq \top$, and $\alpha \preceq \alpha$ are eliminated by Part 2 of Definition 5.17, constraints $t \rightarrow t' \preceq t \rightarrow t'$ are eliminate by Part 3, and constraints $\delta \preceq \delta$ are eliminated by Parts 4 and 5. Therefore, for all t, we have $t \preceq t$ is not in $B(\{\tau_1 \preceq \tau_2\})$.

4. Same as Part 4 of the proof of Lemma 4.17 (but using Definition 5.17).

5. Proof similar to the previous step.

6. Let θ be any assignment. We must show that there is an assignment θ' such that $\theta(v) = \theta'(v)$ for all $v \notin V_1$ and $\theta'(s_i) \preceq \theta'(t_i)$ holds for all $i \leq n$. Since
$$\theta(\forall V_1.\tau_1/E_1) \preceq \theta(\forall V_2.\tau_2/E_2),$$
it follows that
$$\theta(\forall V_1.\tau_1/E_1) \preceq \theta(\tau_2/E_2).$$
Since the semantic domain has standard glb types, it follows that $\theta'(\tau_1/E_1) \preceq \theta(\tau_2/E_2)$ holds for some θ' that agrees with θ except possibly on V_1. Since no variable in V_1 occurs in τ_2/E_2, we know $\theta'(\tau_1/E_1) \preceq \theta'(\tau_2/E_2)$. By Lemma 5.18, it follows that θ' is a solution to $B(\tau_1/E_1 \preceq \tau_2/E_2)$.

7. Similar to the previous step with the roles of τ_1 and τ_2 reversed.

□

Theorem 5.20 If the semantic domain has contractive solutions, standard glb types, and standard function types, then any two reduced recursive type expressions have the same number of quantified variables.

Proof: Let σ' and σ'' be two reduced recursive type expressions. If necessary, α-convert σ' to $\sigma_1 = \forall V_1.\tau_1$ and α-convert σ'' to $\sigma_2 = \forall V_2.\tau_2$ in such a way that σ_1 and σ_2 are (V_1, V_2) compatible. By Lemma 5.19, $B(\tau_1/E \preceq \tau_2/E')$ is a (V_1, V_2)-miniscule system of constraints. By Lemma 4.15, $B(\tau_1/E \preceq \tau_2/E')$ is a (V_1, V_2)-convertible system of constraints, which implies that $|V_1| = |V_2|$. □

Unlike the case of simple expressions, two equivalent reduced types need not be α-equivalent. For example, consider a semantic domain that has contractive solutions. Let $\delta_0 = \delta_0 \to \delta_0$ and $\delta_1 = (\delta_1 \to \delta_1) \to (\delta_1 \to \delta_1)$. These two types exist since the domain has contractive solutions. By substituting $(\delta_0 \to \delta_0)$ in for δ_0, we obtain $\delta_0 = (\delta_0 \to \delta_0) \to (\delta_0 \to \delta_0)$. Since the domain has contractive solutions, it follows that $\delta_0 = \delta_1$. Clearly, the type expressions $\delta_0/\delta_0 = \delta_0 \to \delta_0$ and $\delta_1/\delta_1 = (\delta_1 \to \delta_1) \to (\delta_1 \to \delta_1)$ are not α-equivalent.

Theorem 5.21 If the semantic domain has contractive solutions, type continuity, standard function types, and standard glb types, then a recursive type expression is reduced iff it is irredundant.

Proof: The if-direction follows from Theorem 5.16. To prove the only-if direction, let σ be a reduced recursive type expression with the goal of proving that σ is irredundant. Let σ' be an irredundant recursive type expression that is equivalent to σ. (Such a σ' can always be found by picking it to be a type expression equivalent to σ with the smallest possible number of quantified variables.) By Theorem 5.16, σ' is reduced. By Theorem 5.20, σ and σ' have the same number of quantified variables. Hence, σ is irredundant as desired. □

Theorem 5.22 Let σ be quantified recursive type expression. If the semantic domain has contractive solutions, type continuity, standard glb types, and standard function types, then $VEP(\sigma)$ is an irredundant recursive type expression equivalent to σ.

Proof: Follows easily from Theorem 5.15 and Theorem 5.21. □

6 Intersection and Union Types

In this section we extend our results to type languages with union and intersection types. This is the first point at which the technique of eliminating variables that appear solely in monotonic or anti-monotonic positions is sound but not complete.

6.1 Preliminaries

As a first step union and intersection types are added to simple type expressions.

Definition 6.1 *Extended type expressions* are generated by the grammar

$$\tau ::= \alpha \mid \top \mid \bot \mid \tau_1 + \tau_2 \mid \tau_1 \cdot \tau_2 \mid \tau_1 \to \tau_2$$

Extended quantified types are adapted in the obvious way to use extended type expressions instead of simple type expressions. The operations $+$ and $\cdot$ are interpreted as least-upper bound and greatest-lower bound, respectively. To give meaning to extended type expressions an assumption is needed about the upper and lower bounds that exist in the domain.

Definition 6.2 A semantic domain $\mathcal{D} = (\mathcal{D}_0, \mathcal{D}_1, \preceq, \sqcap)$ has *standard upper and lower bounds* if every pair of elements $\tau_1, \tau_2 \in \mathcal{D}_0$ have a least upper bound $\tau_1 \sqcup \tau_2$ and a greatest lower bound $\tau_1 \sqcap \tau_2$ in $\mathcal{D}_0$.

Note that requiring $\tau_1 \sqcap \tau_2$ exist is different from having standard glb types, as standard glb types are glb's of (potentially) infinite sets in $\mathcal{D}_1$. The following lemma shows that natural domains have standard upper and lower bounds.

Lemma 6.3 Assume that for every $x \in \mathcal{D}_0$ it is the case that $x = \top$, $x = \bot$, or $x = x_1 \rightarrow x_2$ for some x_1 and x_2. If $\mathcal{D}_0$ has standard function types, then the domain has standard upper and lower bounds.

Proof: We must show that $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in \mathcal{D}_0$. It is easy to check that the following equations cover all possibilities:

$$\begin{array}{rclcrcl}
\top \sqcup x & = & \top & \quad & x \sqcup \top & = & \top \\
\bot \sqcup x & = & x & & x \sqcup \bot & = & x \\
\bot \sqcap x & = & \bot & & x \sqcap \bot & = & \bot \\
\top \sqcap x & = & x & & x \sqcap \top & = & x \\
x_1 \rightarrow y_1 \sqcup x_2 \rightarrow y_2 & = & x_1 \sqcap x_2 \rightarrow y_1 \sqcup y_2 & & x_1 \rightarrow y_1 \sqcap x_2 \rightarrow y_2 & = & x_1 \sqcup x_2 \rightarrow y_1 \sqcap y_2
\end{array}$$

The eight equations on the first four lines are easy to verify. To justify the equation $x_1 \rightarrow y_1 \sqcup x_2 \rightarrow y_2 = x_1 \sqcap x_2 \rightarrow y_1 \sqcup y_2$, note that

$$\begin{array}{rcl}
x_1 \sqcap x_2 & \preceq & x_1 \\
x_1 \sqcap x_2 & \preceq & x_2 \\
y_1 & \preceq & y_1 \sqcup y_2 \\
y_2 & \preceq & y_1 \sqcup y_2
\end{array}$$

from which it follows that $x_1 \sqcap x_2 \rightarrow y_1 \sqcup y_2$ is an upper bound of both $x_1 \rightarrow y_1$ and $x_2 \rightarrow y_2$. Let $a \rightarrow b$ be any other upper bound of $x_1 \rightarrow y_1$ and $x_2 \rightarrow y_2$. Since the domain has standard function types, we have $a \preceq x_1$ and $a \preceq x_2$, so $a \preceq x_1 \sqcap x_2$. Similarly $y_1 \sqcup y_2 \preceq b$. Therefore, $x_1 \sqcap x_2 \rightarrow y_1 \sqcup y_2$ is the least upper bound. The justification of the last equation is symmetric. □

It follows immediately from Lemma 6.3 that the Standard Model (Example 2.3) and Regular Tree Model (Lemma 5.8) both have standard upper and lower bounds.

Given an assignment θ, the meanings of the new type operations are:

$$\begin{array}{rcl}
\theta(\tau_1 + \tau_2) & = & \theta(\tau_1) \sqcup \theta(\tau_2) \\
\theta(\tau_1 \cdot \tau_2) & = & \theta(\tau_1) \sqcap \theta(\tau_2)
\end{array}$$

6.2 Soundness for Non-Recursive Types

We first extend *Pos* and *Neg* to include the new operations.

$$\begin{array}{rcl}
Pos(\tau_1 + \tau_2) & = & Pos(\tau_1) \cup Pos(\tau_2) \\
Neg(\tau_1 \cdot \tau_2) & = & Neg(\tau_1) \cup Neg(\tau_2)
\end{array}$$

We can now restate the basic lemma needed to prove soundness for the non-recursive case.

Lemma 6.4 Let τ be any extended simple type expression. Let $d, d' \in \mathcal{D}_0$ where $d \preceq d'$. If the domain has standard upper and lower bounds, then

1. If $\alpha \notin Pos(\tau)$, then $\theta(\tau[\alpha \leftarrow d']) \preceq \theta(\tau[\alpha \leftarrow d])$ holds for all assignments θ.

2. If $\alpha \notin Neg(\tau)$, then $\theta(\tau[\alpha \leftarrow d]) \preceq \theta(\tau[\alpha \leftarrow d'])$ holds for all assignments θ.

Proof: This proof is by induction on the structure of τ and is an easy extension of the proof of Lemma 4.3. Let $d', d \in \mathcal{D}_0$ where $d' \preceq d$, and let θ be any assignment. There are two new cases:

- Let $\tau = \tau_1 + \tau_2$. Assume $\alpha \notin Pos(\tau)$. By the definition of Pos, we know

$$\alpha \notin Pos(\tau_1) \cup Pos(\tau_2)$$

and therefore

$$\begin{array}{rcl} \theta[\alpha \leftarrow d'](\tau_1) & \preceq & \theta[\alpha \leftarrow d](\tau_1) \\ \theta[\alpha \leftarrow d'](\tau_2) & \preceq & \theta[\alpha \leftarrow d](\tau_2) \end{array}$$

follow by induction. The relationships still hold if the right-hand sides are made larger, so

$$\begin{array}{rcl} \theta[\alpha \leftarrow d'](\tau_1) & \preceq & \theta[\alpha \leftarrow d](\tau_1) \sqcup \theta[\alpha \leftarrow d](\tau_2) \\ \theta[\alpha \leftarrow d'](\tau_2) & \preceq & \theta[\alpha \leftarrow d](\tau_1) \sqcup \theta[\alpha \leftarrow d](\tau_2) \end{array}$$

Combining these two inequalities we get

$$\theta[\alpha \leftarrow d'](\tau_1) \sqcup \theta[\alpha \leftarrow d'](\tau_2) \ \preceq\ \theta[\alpha \leftarrow d](\tau_1) \sqcup \theta[\alpha \leftarrow d](\tau_2)$$

The proof for the subcase $\alpha \notin Neg(\tau)$ is similar.

- Let $\tau = \tau_1 \cdot \tau_2$. This case is very similar to the previous one, with $\sqcap$ substituted for $\sqcup$.

□

An inspection of the results from Section 4.1 shows that the proofs of Lemma 4.5 and Theorem 4.8 depend only on Lemma 4.3 and not on a particular language of type expressions. Therefore, by Lemma 6.4, it is immediate that Procedure 4.7 is a sound variable elimination procedure for extended simple types in domains with standard upper and lower bounds.

While variable elimination is sound for extended simple type expressions, it is not complete.

Example 6.5 In either the Standard Model or Regular Tree Model we have

$$\forall \alpha.(\alpha \to \alpha) + \top \equiv \top$$

Clearly, the first type is reduced and not irredundant.

Similarly, $\forall \alpha.(\alpha \to \alpha) \cdot \bot \equiv \bot$. In general, the Pos and Neg computations overestimate the set of positive and negative variables for expressions $\tau_1 + \tau_2$ where $\theta(\tau_1) \preceq \theta(\tau_2)$ for all θ (and similarly for $\cdot$).

A subtler source of incompleteness arises from interaction between universal quantification and unions and intersections.

Example 6.6

$$\begin{array}{lll} & \forall \alpha, \beta. \alpha \cdot \beta \to \alpha \cdot \beta & \\ = & \sqcap\{\theta[\alpha \leftarrow x_1, \beta \leftarrow x_2](\alpha \cdot \beta \to \alpha \cdot \beta) | x_1, x_2 \in \mathcal{D}_0\} & \text{by Proposition 3.3} \\ = & \sqcap\{x_1 \sqcap x_2 \to x_1 \sqcap x_2) | x_1, x_2 \in \mathcal{D}_0\} & \\ = & \sqcap\{x \to x | x \in \mathcal{D}_0\} & \text{since } \{x_1 \sqcap x_2 | x1, x_2 \in \mathcal{D}_0\} = \mathcal{D}_0 \\ = & \forall \alpha. \alpha \to \alpha & \end{array}$$

Note that there is no explicit relationship between α and β in the type. The relationship follows from the fact that the variables are always used together and the universal quantification.

6.3 Improvements

We do not know a complete version of the Variable Elimination Procedure in the presence of union and intersection types. In this section we briefly illustrate some heuristic improvements that have been useful in practice [AM91, FA96]. As illustrated in Section 6.2, redundant intersections and unions are a significant source of incompleteness. This suggests the following procedure:

Procedure 6.7 (Extended Variable Elimination Procedure (EVEP)) Let σ be an extended quantified type.

1. Let σ_1 be the result of replacing any subexpression $\tau_1 + \tau_2$ in σ by τ_2 if $\theta(\tau_1) \preceq \theta(\tau_2)$ for all assignments θ.
2. Let σ_2 be the result of replacing any subexpression $\tau_1 \cdot \tau_2$ in σ' by τ_2 if $\theta(\tau_2) \preceq \theta(\tau_1)$ for all assignments θ.
3. Let $\sigma_3 = VEP(\sigma_2)$.
4. Halt if no variables are eliminated in (3); the result is σ_3. Repeat (1)-(3) on σ_3 otherwise.

Note that deciding whether a type is equivalent to $\top$ or $\bot$ in all assignments is not necessarily easy, depending on the expressiveness of the type language under consideration.

The interesting aspect of Procedure 6.7 is that iterating the elimination of intersections, unions, and variables is necessary—the body of the Extended Variable Elimination Procedure is not idempotent. For example:

$$\begin{array}{lll} & \forall\alpha,\beta.\beta \rightarrow (\alpha \cdot \beta) & \\ \equiv & \forall\beta.\beta \rightarrow (\bot \cdot \beta) & \text{since } \alpha \text{ is not negative} \\ \equiv & \forall\beta.\beta \rightarrow \bot & \text{since } \bot \cdot \tau = \bot \\ \equiv & \top \rightarrow \bot & \text{since } \beta \text{ is not positive} \end{array}$$

Since each iteration but the last of the Extended Variable Elimination Procedure eliminates at least one variable, the complexity is at worst the product of the number of quantified variables and the size of the original type, which is at most quadratic in the size of the type.

6.4 Soundness and Incompleteness for Recursive Types

In this section we consider extended recursive types.

Definition 6.8 An *extended recursive type* has the form

$$\tau / \bigwedge_{1 \leq i \leq n} \delta_i = \tau_i$$

where the equations are contractive and $\tau, \tau_1, \ldots, \tau_n$ are extended type expressions.

It will come as no surprise that, in addition to standard upper and lower bounds, the domain must have contractive solutions and type continuity for variable elimination to be sound for extended recursive types. An inspection of the statement and proof of Lemma 5.10 shows that it does not depend on a particular definition of type, but only on type continuity and soundness of the non-recursive case. Thus, adding the hypothesis that the domain has standard upper and lower bounds to Lemma 5.10, and substituting Lemma 6.4 for Lemma 4.3 in the proof of the lemma, gives a proof of soundness for variable elimination on extended recursive types.

Because extended simple types are a subset of the extended recursive types and the VEP is incomplete for simple types, it follows that variable elimination is incomplete for extended recursive types.

7 Constrained Type Expressions

This section presents results for types with polymorphism and subtyping constraints, which is also called *bounded polymorphism* or *constrained types*. This language is the most general that we consider.

7.1 Preliminaries

We begin by defining a type language with subsidiary subtyping constraints.

Definition 7.1 An *(unquantified) constrained type expression* has the form τ_0/C where C is a set of constraints of the form

$$\begin{array}{ccc} \tau_1 & \preceq & \tau_1' \\ \ldots & & \ldots \\ \tau_n & \preceq & \tau_n' \end{array}$$

where τ_i and τ_i' are unquantified simple type expressions for all $1 \leq i \leq n$.

Unlike the case of recursive types, note that Definition 7.1 makes no distinction between "regular" and "defined" variables—all variables are regular.

Definition 7.2 Let θ be any assignment. Then

1. $\theta(\tau/C) = \theta(\tau)$ provided that θ is a solution of C.

2. $\theta(\forall\alpha_1, \ldots, \alpha_n.\tau/C) =$

   $\sqcap\{\theta[\alpha_1 \leftarrow x_1, \ldots, \alpha_n \leftarrow x_n](\tau/C) | x_1, \ldots, x_n \in \mathcal{D}_0$ and $\theta[\alpha_1 \leftarrow x_1, \ldots, \alpha_n \leftarrow x_n]$ is a solution of $C\}$

The meaning of an unquantified constrained type τ/C under assignment θ is undefined unless θ is a solution of C. Furthermore, the $\sqcap$ operation in the meaning of a quantified constrained type under assignment θ is restricted to those modifications of θ that satisfy the constraints. It is easy to see that constrained types are a generalization of recursive types, because any recursive type

$$\forall\alpha_1, \ldots, \alpha_m.\tau/\ \delta_1 = \tau_1 \rightarrow \tau_1' \wedge \ldots \wedge \delta_n = \tau_n \rightarrow \tau_n'$$

can be written as a constrained type

$$\forall\alpha_1, \ldots, \alpha_m.\tau/\ \delta_1 \preceq \tau_1 \rightarrow \tau_1' \wedge \delta_1 \succeq \tau_1 \rightarrow \tau_1' \wedge \ldots \wedge \delta_n \preceq \tau_n \rightarrow \tau_1' \wedge \delta_n \succeq \tau_n \rightarrow \tau_1'$$

It is also worth noting that it is well-defined for a quantified constrained type to have an inconsistent system of constraints. For example, if $C = \top \preceq \alpha \preceq \bot$, then

$$\begin{array}{rl} & \theta(\forall\alpha.\tau/C) \\ = & \sqcap\{\theta[\alpha \leftarrow x](\tau/C) | x \in \mathcal{D}_0 \text{ and } \theta[\alpha \leftarrow x] \text{ is a solution of } C\} \\ = & \sqcap\{\} \\ = & \top \end{array}$$

An important feature of constrained types is that the constraints may have multiple lower (or upper) bounds for a single variable, such as

$$\tau_1 \to \tau_2 \preceq \alpha \ \wedge\ \tau_3 \to \tau_4 \preceq \alpha \ \wedge\ \beta \preceq \gamma \ \wedge\ \beta \preceq \tau_5 \to \tau_6$$

In any solution of these constraints, α must be an upper bound of $\tau_1 \to \tau_2$ and $\tau_3 \to \tau_4$, and β must be a lower bound of γ and $\tau_5 \to \tau_6$.

To give algorithms for eliminating quantified variables from constrained types, it is necessary to characterize the solutions of constraints. To minimize the number of new concepts needed to explain the algorithms in the case of constrained types, we build on the results of Section 5 by characterizing solutions of constraints in terms of equations.

Definition 7.3 A system C of constraints is *closed* iff

$$\begin{array}{rcl} \tau_1 \preceq \alpha \in C \wedge \alpha \preceq \tau_2 \in C & \Rightarrow & \tau_1 \preceq \tau_2 \in C \\ \tau_1 \to \tau_2 \preceq \tau_3 \to \tau_4 \in C & \Rightarrow & \tau_3 \preceq \tau_1 \in C \wedge \tau_2 \preceq \tau_4 \in C \end{array}$$

A closed system C is *consistent* iff $\top \preceq \bot \notin C$.

Definition 7.3 is taken from [EST95]. Intuitively, closing a system of constraints C is equivalent to solving C, and if the closed system has no inconsistent constraints, then it has solutions. Instead of asserting that closed consistent systems have solutions directly, we characterize those solutions in terms of equations.

Definition 7.4 Let C be a closed consistent system of constraints. Let the variables of C be $\delta_1, \ldots, \delta_n$. For each variable δ_i appearing in C, define

$$\begin{array}{rcl} L^C_{\delta_i} & = & \bot + \Sigma\{\tau | \tau \preceq \delta_i \in C \text{ and if } \tau \text{ is a variable } \delta_j, \text{ then } j < i\} \\ U^C_{\delta_i} & = & \top \cdot \Pi\{\tau | \tau \succeq \delta_i \in C \text{ and if } \tau \text{ is a variable } \delta_j, \text{ then } j < i\} \end{array}$$

Let $\alpha_1, \ldots, \alpha_n$ be fresh variables. Define a system of equations E_C for C:

$$\bigwedge_{1 \le i \le n} \delta_i = L^C_{\delta_i} + (\alpha_i \cdot U^C_{\delta_i})$$

The intuition behind Definition 7.4 is that any solution for the equation for δ_i ranges between $L^C_{\delta_i}$ (when $\alpha_i = \bot$) and $U^C_{\delta_i}$ (when $\alpha_i = \top$). For example, consider the system C of constraints

$$\delta_1 \preceq \delta_2 \ \wedge\ \top \to \bot \preceq \delta_1 \ \wedge\ \delta_1 \preceq \bot \to \top$$

Closing this system gives

$$\delta_1 \preceq \delta_2 \ \wedge\ \top \to \bot \preceq \delta_1 \ \wedge\ \delta_1 \preceq \bot \to \top \ \wedge\ \top \to \bot \preceq \delta_2 \ \wedge\ \bot \preceq \top$$

which is consistent. The equations E_C are

$$\begin{array}{rcl} \delta_1 & = & (\top \to \bot) + (\alpha_1 \cdot (\bot \to \top)) \\ \delta_2 & = & ((\top \to \bot) + \delta_1) + (\alpha_2 \cdot \top) \end{array}$$

This example shows that the equations E_C are not necessarily contractive, since δ_1 appears outside of a constructor in the equation for δ_2. However, E_C is always equivalent to a contractive system of equations.

Lemma 7.5 Let E_C be a system of equations for a closed consistent system of constraints C. Then there is a system of constraints E'_C that is contractive such that E_C and E'_C have the same solutions.

Proof: Examination of L_{δ_i} and U_{δ_i} in Definition 7.4 shows that if δ_j occurs outside of a $\rightarrow$ expression in the equation for δ_i, then $j < i$. We show by induction on i how to construct a contractive equation for δ_i. The equation for δ_1 has no variable δ_k outside of a $\rightarrow$ expression, so the equation for δ_1 is already contractive. Assume that $\delta_1, \ldots, \delta_{i-1}$ have contractive equations. Any variable δ_j outside of a $\rightarrow$ expression in the equation for δ_i can be eliminated by substituting the right-hand side of an equation for δ_j. Because $j < i$, we can choose a contractive equation for δ_j, in which case the resulting equation for δ_i is also contractive. □

Applying Lemma 7.5 to the example system of equations above, the contractive system is

$$\begin{aligned} \delta_1 &= (\top \rightarrow \bot) + (\alpha_1 \cdot (\bot \rightarrow \top)) \\ \delta_2 &= ((\top \rightarrow \bot) + (\top \rightarrow \bot) + (\alpha_1 \cdot (\bot \rightarrow \top))) + (\alpha_2 \cdot \top) \end{aligned}$$

Definition 7.6 A domain is *adequate* if

1. the domain has contractive solutions,
2. the domain has standard upper and lower bounds,
3. and for any consistent closed system C of constraints over $\delta_1, \ldots, \delta_n$,

$$\theta \text{ is a solution of } C \Leftrightarrow \exists \theta'.\ \theta' \text{ is a solution of } E_C \text{ and } \forall 1 \leq i \leq n.\ \theta(\delta_i) = \theta'(\delta_i)$$

Lemma 7.7 In an adequate domain, closed consistent systems of constraints have solutions.

Proof: By Part 3 of Definition 7.6, a closed consistent system C has solutions iff E_C has solutions. By Lemma 7.5, E_C is equivalent to a contractive system of equations. By Part 1 of Definition 7.6, contractive equations always have solutions in an adequate domain. □

A construction of an adequate domain is given in [AW93].

7.2 Soundness

The equivalence between constraints and equations in an adequate domain suggests that variable elimination can be performed by first translating from constraints to equations, applying the results of Section 6.4 to eliminate variables, and then translating back (if desired) to constraints. We can improve on this procedure with a modified Extended Variable Elimination Procedure that takes advantage of the structure of constraint systems.

Procedure 7.8 Let $\sigma = \forall \delta_1, \ldots, \delta_n.\tau/C$ be a quantified constrained type. Let

$$\sigma' = \forall \alpha_1, \ldots, \alpha_n.\tau/E_C$$

be the corresponding extended quantified type. Perform the following steps on σ':

1. Let σ_1 be the result of replacing any subexpression $\tau_1 + \tau_2$ in σ' by τ_2 if $\theta(\tau_1) \preceq \theta(\tau_2)$ for all assignments θ. In particular, if any equation of E_C has the form

$$\delta_i = L^C_{\delta_i} + \top \cdot U^C_{\delta_i}$$

then replace the equation by

$$\delta_i = U^C_{\delta_i}$$

since $\theta(L^C_{\delta_i}) \preceq \theta(U^C_{\delta_i})$ for any solution θ of the constraints.

2. Let σ_2 be the result of replacing any subexpression $\tau_1 \cdot \tau_2$ in σ' by τ_2 if $\theta(\tau_2) \preceq \theta(\tau_1)$ for all assignments θ. In particular, if any equation of E_C has the form

$$\delta_i = L^C_{\delta_i} + \bot \cdot U^C_{\delta_i}$$

then replace the equation by

$$\delta_i = L^C_{\delta_i}$$

3. Let $\sigma_3 = VEP(\sigma_2)$.

4. Halt if no variables are eliminated in (3); the result is σ_3. Repeat (1)-(3) on σ_3 otherwise.

Example 7.9 Consider the type

$$\forall \delta_1, \delta_2.\ \delta_1 \to \delta_2 /\ \delta_1 \preceq \delta_2$$

The extended quantified type is

$$\forall \alpha_1, \alpha_2 . \delta_1 \to \delta_2 / \quad \delta_1 = \bot + \alpha_1 \cdot \delta_2 \quad \wedge \quad \delta_2 = \bot + \alpha_2 \cdot \top$$

Now $Pos(\delta_1 \to \delta_2) = \{\alpha_2\}$ and $Neg(\delta_1 \to \delta_2) = \{\alpha_1, \alpha_2\}$. Thus α_1 can be set to $\top$; performing this substitution and simplifying gives:

$$\forall \alpha_2 . \delta_1 \to \delta_2 /\ \delta_1 = \delta_2 \quad \wedge \quad \delta_2 = \alpha_2$$

Substituting α_2 for the other variables gives:

$$\forall \alpha_2 . \alpha_2 \to \alpha_2$$

Soundness is easy to prove for the procedure given above.

Lemma 7.10 Let $\sigma = \forall \delta_1, \ldots, \delta_n . \tau / C$ and assume that C is closed and consistent. If the domain is adequate and has type continuity, then Procedure 7.8 is sound for σ.

Proof: Follows from Definition 7.6 and Lemma 5.10. □

We note that it is easy to give an algorithm that implements the effect of Procedure 7.8 on the constraints directly, without requiring translations to and from equations.

8 Conclusions

Polymorphic types with subtyping have rich structure. In this paper, we have shown that for simple non-recursive types and recursive types, it is possible to compute an optimal representation of a polymorphic type in the sense that no other equivalent type has fewer quantified variables. Thus, the optimal representation can be interpreted as having the minimum polymorphism needed to express the type.

In more complex type languages, in particular in languages with union and intersection types, the same methods are sound but incomplete. The completeness results for the simpler type languages show that the source of incompleteness is in fact union and intersection types in these languages. The problem of whether there is a sound and complete variable elimination procedure for languages with intersection and union types remains open.

We have also given a sound variable elimination procedure for polymorphic constrained types. Variable elimination is critically important in implementations of type systems using constrained types [FA96], and in fact the desire to better understand variable elimination in this setting was the original motivation for this work. However, the problem of whether there is a sound and complete procedure for eliminating variables in polymorphic constrained types also remains open.

References

[AM91] A. Aiken and B. Murphy. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[Cur90] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Parc, February 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '96*, 1995.

[FA96] M. Fähndrich and A. Aiken. Making set-constraint program analyses scale. In *CP96 Workshop on Set Constraints*, August 1996.

[HM94] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.

[Koe94] A. Koenig. An anecdote about ML type inference. In *Proceedings of the USENIX 1994 Symposium on Very High Level Languages*, October 1994.

[MPS84] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymophic types. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

[Pot96] F. Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133, May 1996.