Optimal Representations of Polymorphic Types with Subtyping (Extended Abstract)

Alexander Aiken[1]* and Edward L. Wimmers[2] and Jens Palsberg[3]

[1] EECS Department, University of California at Berkeley, Berkeley, CA 94720-1776.
[2] IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120-6099.
[3] Department of Computer Science, Purdue University, West Lafayette, IN 47907.

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.

1 Introduction

Contemporary type systems include many 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". Subtyping is used in, e.g., object-oriented type systems and in program analyses 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 usage context.

A number of type systems 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], type systems for object-oriented languages [EST95], and program analyses aimed at program optimization [AWL94, HM94]. In short, the combination of subtyping and polymorphism is useful, with many applications.

When taken together, subtyping and polymorphism induce equivalences on types that can be exploited to simplify the representation of types. Our main result is that, in a simple type language with a least type $\bot$ 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

* This work was supported by an NSF NYI award CCR-9457812.

complexity that is sensitive 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 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 by example. A few definitions are needed first. Consider the following simple type language:

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

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

$$\begin{aligned} \tau &\preceq \tau \\ \bot &\preceq \tau \\ \tau &\preceq \top \\ \tau_1 \preceq \tau_1' \wedge \tau_2 \preceq \tau_2' &\Leftrightarrow \tau_1' \rightarrow \tau_2 \preceq \tau_1 \rightarrow \tau_2' \end{aligned}$$

Quantified types are defined by:

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

For the moment, we rely on the reader's intuition about the meaning of quantified types. Quantified types are formalized in Section 2.

Consider the type $\forall\alpha.\forall\beta.\alpha \rightarrow \beta$. 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 $\top \rightarrow \bot$. The only function with this type is the one that diverges for all possible inputs.

It turns out that, in fact, $\forall\alpha.\forall\beta.\alpha \rightarrow \beta \equiv \top \rightarrow \bot$ in the standard *ideal* model of types [MPS84]. As argued above, the type with fewer variables is better for human readability and the speed of type inference (since it reduces the number of variables in instances of the polymorphic type, in this case to zero). The reasoning required to discover that $\forall\alpha.\forall\beta.\alpha \rightarrow \beta$ 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]. Note that this 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.

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 unneeded in a program to testing whether certain type variables can be replaced by $\perp$ 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 (eliminates as many variables as possible) for the simple type language defined above, as well as for a type language with recursive types. The various algorithms are practical and efficient, running in time $\mathcal{O}(VN)$, where V is the number of variables and N is the size of the type considered as a string. These algorithms have been implemented and in use since 1993, but with the exception of code documentation little has been written previously on the subject. The algorithm has been implemented and used in Illyria[4], the systems reported in [AW93], a large scale program analysis system for the functional language FL [AWL94], and a general-purpose constraint-based program analysis system [FA96].

Other recent systems based on constrained types have also pointed out the importance of variable elimination. In [EST95], Eifrig, Smith, and Trifonov and separately Pottier [Pot96] describe methods for eliminating unnecessary variables from constraint sets. Both of these methods are heuristic; i.e., they are sound but not complete. Constraint simplification is also a component of the systems described in [Kae92, Smi94].

Our focus in this paper is quite different. The question of variable elimination arises in any type system with polymorphism and subtyping, not just in systems with constrained types. Our purpose is to explore the basic structure of this problem in the simplest settings. To the best of our knowledge, we present the first sound and complete algorithms for variable elimination in polymorphic types. As the variable elimination algorithm we study is a commonly used sound (but incomplete) heuristic in more complex systems, our results also help characterize the fragment of such systems for which the algorithm is complete.

There is another application of our results. As stated above, variable elimination is important in constraint-based type systems, and ongoing research has sought to settle the question of whether there is a sound and complete algorithm for variable elimination in such systems. Our results show that completeness is quite sensitive to the structure of the underlying domain, even in the simplest settings. Innocuous properties of a domain (e.g., whether $\perp = \top \to \perp$) make completeness more difficult to obtain. 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.

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

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 position in a type. The main lemma shows that solely monotonic quantified variables can be replaced by $\bot$; solely anti-monotonic quantified variables can be replaced by $\top$.

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.

For lack of space, this extended abstract omits additional extensions to sound but incomplete algorithms to systems with union and intersection types and constrained types. The interested reader is referred to the full version of the paper [**?**].

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

1. $\mathcal{D}_0 \subseteq \mathcal{D}_1$ or, more generally, there is monomorphism from $\mathcal{D}_0$ to $\mathcal{D}_1$.
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 the following:

standard function types
If $x_1 \rightarrow x_2 \preceq y_1 \rightarrow y_2$, then $y_1 \preceq x_1$ and $x_2 \preceq y_2$.
strong 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 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 $t_1 \rightarrow t_2$ 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 is more general than requiring they be the same. In particular, since the $\forall$ quantifier introduces a glb operation (hence produces a value in $\mathcal{D}_1$) and the $\rightarrow$ operation is performed only on elements of

$\mathcal{D}_0$, the $\forall$ quantifier cannot appear inside of a $\rightarrow$ operation. If it happens that $\mathcal{D}_0 = \mathcal{D}_1$, then the semantic domain supports $\forall$ quantifiers inside of the $\rightarrow$ 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 1 Minimal Semantic Model. Let $\mathcal{D}_0 = \mathcal{D}_1$ be the three element set $\{\bot, \top \rightarrow \top, \top\}$ and let $\preceq$ be the partial order $\bot \preceq \top \rightarrow \top \preceq \top$. In this domain, all function types are the same—the domain does little more than detect that something is a function. For all $x, y \in \mathcal{D}$, $x \rightarrow y = \top \rightarrow \top$. It is easy to check that $\mathcal{D}$ satisfies all properties required of a semantic domain as well as strong glb types. The only property missing is standard function types (e.g., because $\bot \rightarrow \bot \preceq \top \rightarrow \top$, but $\top \not\preceq \bot$).

Example 2 Standard Model. Let $\mathcal{D}_0$ be the set consisting of $\bot$ and $\top$ and closed under the pairing operation (denoted using the $\rightarrow$ symbol). An obvious partial order is induced on $\mathcal{D}_0$ (and no two elements of $\mathcal{D}_0$ are equal). 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 show that $\mathcal{D}_1$ has standard function types and strong glb types.

3 Syntax

Our first type language has only type variables and function types. In this language and the extensions we consider quantification is only at the outermost level.

Definition 2. *Unquantified simple type expressions* are generated by the grammar:

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

where α ranges over a family of type variables. *Quantified simple type expressions* have the form

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

where each α_i is a type variable and τ is an unquantified simple type expression. The type τ is the *body* of the type.

Since $n = 0$ is a possibility in Definition 2, unquantified simple type expressions are also quantified simple type expressions. 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* $\theta : \text{Vars} \to \mathcal{D}_0$ 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.

1. $\theta(\top) = \top$
2. $\theta(\bot) = \bot$
3. $\theta(\tau_1 \to \tau_2) = \theta(\tau_1) \to \theta(\tau_2)$
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 $\mathcal{D}_0$. The following proposition is straightforward.

Proposition 4. $\theta(\forall\alpha_1 \ldots \forall\alpha_n.\tau) = \sqcap\{\theta[\alpha_1 \leftarrow x_1 \ldots \alpha_n \leftarrow x_n](\tau) \mid x_1, \ldots, x_n \in \mathcal{D}_0\}$

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 $\theta(\sigma_1) = \theta(\sigma_2)$.

We conclude this section by noting that the requirement in strong glb types that every subset of $\mathcal{D}_0$ satisfy the stated property is unnecessarily restrictive. For practical purposes it is only necessary that subsets of $\mathcal{D}_0$ actually denotable by type expressions have the property. *Standard glb types* capture this weaker notion.

Definition 5. Let S_0 be a subset of $\mathcal{D}_0$. Then S_0 is called *expr-definable* iff $S_0 = \sqcap\{\theta[\alpha_1 \leftarrow x_1 \ldots \alpha_n \leftarrow x_n](\tau) \mid x_1, \ldots, x_n \in \mathcal{D}_0\}$ for some type expression τ and some assignment θ.

standard glb types

If S_0 is an expr-definable subset of $\mathcal{D}_0$ and $x_1 \in \mathcal{D}_0$, then $\sqcap S_0 \preceq x_1$ iff $\exists x_0 \in S_0$ s.t. $x_0 \preceq x_1$.

Strong glb types clearly imply standard glb types. Our results only require the domain have standard glb types. Notice that, unlike the notion of strong glb types, which is purely semantic, standard glb types also has a syntactic component.

Example 3. Consider the domain equation

$$V \equiv N + (V \to V)$$

where N is the integers. This is essentially the equation of the ideal model in [MPS84]. The ideal model does not have strong glb types. For example, consider the set $S_0 = \{3, 4\}$. The glb of this set is $\{\bot\}$, but there is neither 3 nor 4 is less than $\bot$.

Let $\mathcal{D}_1$ be downward-closed, limit-closed subsets of V and let $\mathcal{D}_0$ consist of $\bot$, $\top$, and the elements of N closed under $\rightarrow$. Define a mapping T as follows:

$$\begin{array}{l} T(\top) = V \\ T(\bot) = \{\bot\} \\ T(i) = \{i, \bot\} \\ T(X \rightarrow Y) = \{f \mid \forall x \in T(X)(f(x) \in T(Y))\} \end{array}$$

It is easy to see that T embeds $\mathcal{D}_0$ in $\mathcal{D}_1$.

We now sketch by example why $\mathcal{D}_1$ has standard glbs. Consider the type expression $\bot \rightarrow \bot$. Let τ be the type expression that makes the set S_0 definable. That is, $S_0 = \{\theta[\alpha_1 \leftarrow x_1, \ldots, \alpha_n \leftarrow x_n](\tau) \mid x_1, \ldots, x_n \in \mathcal{D}_0\}$. The key is to show for each possible τ that either (1) $\sqcap S_0$ is not less than or equal to $\bot \rightarrow \bot$, or (2) S_0 contains an element less than or equal to $\bot \rightarrow \bot$. We may assume without loss of generality that τ contains no variables other than $\alpha_1, \ldots, \alpha_n$ since every possible θ can only assign values to the other variables that were already realized by some explicit type expression involving $\bot, \top, \rightarrow$, or an element of N.

The proof strategy is to argue by induction on the structure of τ. Consider the possibilities for τ. If $\tau = \bot$ or α, then $\bot \in S_0$ and the result follows since (2) is true. If $\tau = \top$ or an element of N, the result follows since (1) is true. Now assume that $\tau = \tau_1 \rightarrow \tau_2$. For simplicity, we make the additional assumption that V is lifted (i.e., that $\top \rightarrow \bot \neq \bot$); the argument if V is not lifted is similar but has more cases. If τ_2 contains a $\rightarrow$, $\top$, or an element of N, then (1) holds. If τ_2 is $\bot$ or a variable, then (2) holds for all possible τ_1.

We remark that the standard ideal model does not satisfy all of the requirements in Definition 1. In particular, $\top \rightarrow \bot = \bot$ and the domain does not have standard function types since $x \rightarrow \top = y \rightarrow \top$ for all $x, y \in \mathcal{D}_0$. The former problem is easily fixed by working with a lifted domain. We will not consider extensions to deal with the latter here; our algorithm is thus sound but not complete for such a model.

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 6. 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 does not occur 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)? In one direction we have

$$\theta(\forall\alpha.\tau) \preceq \theta[\alpha \leftarrow \tau'](\tau) = \theta(\tau[\alpha \leftarrow \tau'])$$

Then, using Definition 3, 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 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 7. *Pos* and *Neg* are defined as follows:

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

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

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

$$\begin{aligned} \mathit{Pos}(\alpha \to \beta) &= \{\beta\} \\ \mathit{Neg}(\alpha \to \beta) &= \{\alpha\} \end{aligned}$$

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

Lemma 8. Let $d', d \in \mathcal{D}_0$ where $d' \preceq d$. Let θ be any assignment.

1. If $\alpha \notin \mathit{Pos}(\tau)$, then $\theta[\alpha \leftarrow d](\tau) \preceq \theta[\alpha \leftarrow d'](\tau)$.
2. If $\alpha \notin \mathit{Neg}(\tau)$, 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 $\alpha \in \mathit{Pos}(\alpha)$, 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 $\alpha \notin \mathit{Pos}(\tau)$. By the definition of *Pos*, we know

$$\alpha \notin \mathit{Neg}(\tau_1) \cup \mathit{Pos}(\tau_2)$$

  Applying the lemma inductively to τ_1 and τ_2, we have

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

  Combining these two lines using axiom 5 of a semantic domain (Definition 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 9.

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

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

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

Lemma 10. If σ is a quantified simple type expression, 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. Assume first that $\forall\alpha.\sigma = \forall\alpha.\tau$ where τ is an unquantified simple type expression and that $\alpha \notin \mathit{Neg}(\tau)$. Note that

$$\begin{array}{ll} \quad \theta(\forall\alpha.\tau) & \\ = \sqcap\{\theta[\alpha \leftarrow x](\tau) | x \in \mathcal{D}_0\} & \\ \preceq \theta[\alpha \leftarrow \bot](\tau) & \text{since } \bot \text{ is a possible choice for } x \\ = \theta(\tau[\alpha \leftarrow \bot]) & \\ = \sqcap\{\theta[\alpha \leftarrow x](\tau[\alpha \leftarrow \bot]) | x \in \mathcal{D}_0\} & \text{since } \alpha \text{ does not occur in } \tau[\alpha \leftarrow \bot] \\ \preceq \sqcap\{\theta[\alpha \leftarrow x](\tau) | x \in \mathcal{D}_0\} & \text{by part 2 of Corollary 9} \\ = \theta(\forall\alpha.\tau) & \end{array}$$

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.

We are interested in quantified types for which as many variables have been eliminated using the conditions of Lemma 10 as possible. Returning to our canonical example,

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

Definition 11. 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 12 (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 13. Let σ be any quantified simple type expression. Then $\sigma \equiv VEP(\sigma)$ and $VEP(\sigma)$ is reduced.

Proof. Equivalence follows easily from Lemma 10. 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.

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 14. 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.

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 1) then every reduced simple type expression is irredundant (Theorem 26).

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.

To avoid a proliferation of subscripts, we from here on use s and t as well as τ for simple type expressions.

Definition 15. 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 16 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 16. 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 17. 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 $\to$ 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 $\{s_1 \to s_2 \preceq t_1 \to t_2\} \cup S$. 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 $\{s \preceq t\} \cup B(S)$.

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 (and if t_1 and t_2 are the bodies of reduced equivalent types) 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 18. 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 18 is that if two reduced types $\forall V_1.\tau_1$ and $\forall V_2.\tau_2$ are α-convertible, then $B(\{\tau_1 \preceq \tau_2\})$ 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 19. 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 $\to$ 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 20. Any (V_1, V_2)-miniscule system of constraints is (V_1, V_2)-convertible.

Proof. Let θ_0 be the assignment that assigns $\bot$ 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 $s_i \preceq t'_i \to t''_i \in B(S)$. 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 19.
2. $\bot \preceq t'_i \to t''_i$ is ruled out by Property 7 of Definition 19, since no assignment satisfies $t'_i \to t''_i \preceq \bot$.
3. $\top \preceq t'_i \to t''_i$ is ruled out by Property 6 of Definition 19, since no assignment satisfies $\top \preceq t'_i \to t''_i$.
4. If v is a variable not in V_1, let $\theta = \theta_1$. Then Property 6 of Definition 19 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 $\theta = \theta_0$. Note that $v \notin V_2$ since V_1 and V_2 are disjoint. Then Property 7 of Definition 19 is violated because for all θ' that agree with θ off of V_2, we have $\theta'(t'_i \to t''_i) \not\preceq \bot = \theta_0(v) = \theta(v) = \theta'(v)$.

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

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

1. $\bot \preceq \bot$ is ruled out by Property 3 in Definition 19.
2. $\top \preceq \bot$ is ruled out by Property 6 in Definition 19 since no assignment satisfies $\top \preceq \bot$.
3. If v is a variable not in V_1, let $\theta = \theta_1$. Then Property 6 in Definition 19 is violated since for all θ' that agree with θ off of V_1, we have that $\theta'(v) = \theta(v) = \theta_1(v) = \top \not\preceq \bot = \theta'(\bot)$.
   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 19, there is a constraint $s' \preceq v$ in S. There are four possible cases for s':
   (a) $s' = \bot$. In this case, $\bot \preceq v \preceq \bot$ is in S and hence Property 7 is violated by taking $\theta = \theta_1$.
   (b) $s' = \top$. In this case, $\top \preceq v \preceq \bot$ 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 $\theta'(v) = \theta(v) = \top \not\preceq \bot = \theta(v') = \theta'(v')$.
   (d) $s' = v' \notin V_1$. In this case, $v' \preceq v \preceq \bot$ violates Property 6 by taking $\theta = \theta_1$ since $\theta(v') = \top$.

This proves that $\bot$ cannot occur as an upper bound in S. By symmetry, $\bot$ can not occur as a lower bound in S, and hence $\bot$ 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 $\theta'(v'') = \theta(v'') = \top \not\preceq \bot = \theta(v') = \theta'(v')$. 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 21. 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 22. 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 19 all hold:

1. By compatibility V_1 and V_2 are disjoint sets of variables.
2. By Part 3 of Definition 16 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 2 of Definition 16.
4. Let $v \in V_1 \cup V_2$. We claim there is an $i \leq n$ such that $v \in \mathit{Pos}(s_i) \cup \mathit{Neg}(t_i)$. 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 17, 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.

One final technical lemma is required before we can show that the variable elimination procedure is complete. The intuition behind Lemma 23 is that if $B(\{\tau_1 \preceq \tau_2\})$ is a (V_1, V_2)-convertible system of constraints with bijection f (recall Definition 18), then $B(\{\tau_1 \preceq f(\tau_2)\}) = \emptyset$. This intuition is not quite correct, because there may be variables in τ_1 or τ_2 that are not in $V_1 \cup V_2$. In the following lemma, $\mathit{vars}(t)$ is the set of variables appearing in t.

Lemma 23. Assume that

1. $B(S)$ is a subset of a (V_1, V_2)-convertible system of constraints with bijection f from V_1 to V_2.
2. For each constraint $s \preceq t \in S$, we have $\mathit{vars}(s) \cap \mathit{vars}(t) \cap (V_1 \cup V_2) = \emptyset$. In other words, any variables common to s and t are not in $V_1 \cup V_2$.

Define

$$F(x) = \begin{cases} f(x) \text{ if } x \in V_1 \\ \quad x \text{ otherwise} \end{cases}$$

We extend F from variables to terms in the usual way. Define

$$S' = \{F(t) \preceq F(t') | t \preceq t' \in S\}$$

The claim is that $B(S') = \emptyset$.

Proof. The proof is by induction on the complexity of S.

– $S = \emptyset$. Then $S' = \emptyset$ and $B(\emptyset) = \emptyset$.

- $S = \{t \preceq t\} \cup S_1$. By assumption (2), $\mathit{vars}(t) \cap (V_1 \cup V_2) = \emptyset$. By the definition of F it follows that $F(t) = t$. Using the definition of B, it is easy to see that because S satisfies assumptions (1) and (2) with bijection f that S_1 also satisfies assumptions (1) and (2) with the same bijection f. Now we have

$$\begin{array}{ll} \emptyset & \\ = B(S_1') & \text{by induction} \\ = B(\{t \preceq t\} \cup S_1') & \text{definition of } B \\ = B(\{F(t) \preceq F(t)\} \cup S_1') & F(t) = t \\ = B(S') & \end{array}$$

- $S = \{t_1 \to t_2 \preceq s_1 \to s_2\} \cup S_1$. Let $T = \{s_1 \preceq t_1, t_2 \preceq s_2\} \cup S_1$. Using the definition of B, it is easy to check that T satisfies conditions (1) and (2) using the bijection f. By induction $B(T') = \emptyset$. Then

$$\begin{array}{ll} \emptyset & \\ = B(\{F(s_1) \preceq F(t_1), F(t_2) \preceq F(s_2)\} \cup S_1') & \text{by induction} \\ = B(\{F(t_1) \to F(t_2) \preceq F(s_1) \to F(s_2)\} \cup S_1') & \text{definition of } B \\ = B(\{F(t_1 \to t_2) \preceq F(s_1 \to s_2)\} \cup S_1') & \text{definition of } F \\ = B(S') & \end{array}$$

- $S = \{s \preceq t\} \cup S_1$ and no previous case applies. Then $B(S) = \{s \preceq t\} \cup B(S_1)$. Since $B(S)$ is (V_1, V_2)-convertible, it follows that $s = \alpha$ and $t = \beta$ for some distinct variables α and β and that either $F(\alpha) = \beta$ and $F(\beta) = \beta$ or $F(\alpha) = \alpha$ and $F(\beta) = \alpha$. The rest is similar to the case for $t \preceq t$ above.

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

Theorem 24. 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 compatible. It suffices to show that σ_1 and σ_2 are α-equivalent.

By Lemma 22, $B(\{\tau_1 \preceq \tau_2\})$ is a (V_1, V_2)-miniscule system of constraints. By Lemma 20, $B(\{\tau_1 \preceq \tau_2\})$ is a (V_1, V_2)-convertible system of constraints; let f be corresponding bijection mapping variables in V_1 to V_2. Define

$$F(x) = \begin{cases} f(x) \text{ if } x \in V_1 \\ \quad x \text{ otherwise} \end{cases}$$

Because σ_1 and σ_2 are compatible, $\mathit{vars}(\tau_1) \cap \mathit{vars}(\tau_2) \cap (V_1 \cup V_2) = \emptyset$. Then, by Lemma 23, we have

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

Since F is the identity on τ_1 it follows that

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

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

Corollary 25. 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. 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 24. But since they have no quantifiers, α-equivalence implies that $\tau = \tau'$.

Finally, the following theorem states our main result.

Theorem 26. 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 14. 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 14, σ' is reduced. By Theorem 24, 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 26 shows that a syntactic test (reduced) is equivalent to a semantic test (irredundant). Theorem 26 requires that the semantic domain has standard function types. The following examples show that this assumption is necessary.

Example 4. Consider the minimal semantic domain (Example 1). 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 5. 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 27 shows that the Variable Elimination Procedure (Procedure 12) is complete provided that the semantic domain has standard function types.

Theorem 27. 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 13 and Theorem 26.

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 27 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 6. 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.

Let the set of type expressions be $\top$ and $\bot$ closed under $\to$. For this domain, we define the $\to$ operator as follows. The four possibilities for combining $\top$ and $\bot$ using $\to$ map to the corresponding elements of $\mathcal{D}_0$. For all other types $y_0 \to y_1$, either y_1 or y_2 (or both) is a function type. If either y_0 or y_1 is a function type, define $y_0 \to y_1 = x$.

In this domain $\theta(\forall\alpha.(\alpha \to \alpha)) = x$ for all assignments θ. This follows because $x \preceq \top \to \top$ and $x \preceq \bot \to \bot$ and any other $y \to y = x$ (e.g., $(\bot \to \bot) \to (\bot \to \bot) = x$). Now we have that $\forall\alpha.(\alpha \to \alpha) \equiv (\bot \to \bot) \to \bot$ and, in fact, the quantified type is equivalent to exactly those unquantified types with a function type in one or both of the domain or range. 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 28. 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 29. 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 30. 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:

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 $i = 1, \ldots, n$.

Note that Definition 30 is well-formed because assignments are applied only to unquantified simple type expressions, an operation that already has meaning (see Definition 3).

Lemma 31. Let E and E' be contractive systems of equations and assume the semantic domain has contractive solutions.

1. If E defines the variables $\delta_1, \ldots, \delta_n$ and E' contains exactly the definitions for any other defined variables of E, then $(\theta[\delta_1 \leftarrow \theta^E(\delta_1), \ldots, \delta_n \leftarrow \theta^E(\delta_n)])^{E'} = \theta^E$.
2. If E does not mention variables $\alpha_1, \ldots, \alpha_m$, then $(\theta[\alpha_1 \leftarrow d_1, \ldots, \alpha_m \leftarrow d_m])^E = \theta^E[\alpha_1 \leftarrow d_1, \ldots, \alpha_m \leftarrow d_m]$

Proof. 1. Let $\theta_1 = \theta[\delta_1 \leftarrow \theta^E(\delta_1), \ldots, \delta_n \leftarrow \theta^E(\delta_n)]$. If α is not defined by E', then $\theta^E(\alpha) = \theta_1(\alpha)$. If $\delta = \tau$ is a definition in E', then $\theta^E(\delta) = \theta^E(\tau)$. By uniqueness of $\theta_1^{E'}$, it follows that $\theta_1^{E'} = \theta^E$ as desired.

2. By repeated applications, it suffices to consider the case $m = 1$. If β is not defined by E, it is easy to see that $\theta^E[\alpha_1 \leftarrow d_1](\beta) = \theta[\alpha_1 \leftarrow d_1](\beta)$. If $\delta = \tau$ is a definition in E, then $\theta^E[\alpha_1 \leftarrow d_1](\delta) = \theta^E(\delta) = \theta^E(\tau) = \theta^E[\alpha \leftarrow d_1](\tau)$. By uniqueness of $(\theta[\alpha_1 \leftarrow d_1])^E$, it follows that $(\theta[\alpha_1 \leftarrow d_1])^E = \theta^E[\alpha_1 \leftarrow d_1]$.

An assignment is extended to (quantified) recursive type expressions as follows:

Definition 32.

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 33 shows that if a domain has contractive solutions, then definitions of "unused" variables can be dropped.

Lemma 33. Assume the domain has contractive solutions. Let E by a set of equations a let $E' \subseteq E$. Assume that whenever a defined variable δ of E occurs in τ_0/E', then δ is a defined variable of E'. Then $\tau_0/E \equiv \tau_0/E'$.

Proof. Let $\delta_1, \ldots, \delta_m$ be the variables defined by E but not E'. Then we have:

$$
\begin{array}{ll}
\theta(\tau_0/E) & \\
= \theta^E(\tau_0) & \\
= \theta[\delta_1 \leftarrow \theta^E(\delta_1), \ldots, \delta_m \leftarrow \theta^E(\delta_m)]^{E'}(\tau_0) & \text{by part 1 of Lemma 31} \\
= \theta^{E'}[\delta_1 \leftarrow \theta^E(\delta_1), \ldots, \delta_m \leftarrow \theta^E(\delta_m)](\tau_0) & \text{by part 2 of Lemma 31} \\
= \theta^{E'}(\tau_0) & \text{since } \delta_1, \ldots, \delta_m \text{ do not appear in } \tau_0
\end{array}
$$

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 8). 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 35 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 34. 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, $\tau_0 \neq \alpha$, 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 35. 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 1) has contractive solutions, type continuity, and standard glb types, but it lacks standard function types. The standard semantic model (Example 2) 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 36. 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 $\bot$ using the $\to$ operator. Thus, $\top$ and $\bot$ are elements of $\mathcal{D}_0$ and every other element x of $\mathcal{D}_0$ is equal to $x' \to x''$ 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 [Cou79].

Let $x \preceq_0 y$ hold for all $x, y \in \mathcal{D}_0$. Let $x \preceq_{i+1} y$ hold iff $x = \bot$ or $y = \top$ or $x = x' \to x''$ and $y = y' \to y''$ 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 $i \geq 0$ [AC93].

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

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

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 monotonic 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 definable by a system of equations contractive in F's argument. To see this, note that in the base case $F^0(x) =_0 F^0(y)$, since every value is $=_0$ to every other value. Recall that $=_i$ means equal to depth i (where depth is the number of nested constructors) and that $F(z)$ is equivalent to a system of equations with occurrences of z embedded inside at least one constructor (contractiveness). Therefore, for the inductive step, it suffices to note that if $F^i(x) =_i F^i(y)$ (i.e., equal to a depth of i constructors) then $F(F^i(x)) =_{i+1} F(F^i(y))$ (i.e., equal to a depth of $i+1$ constructors).

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.

5.2 Soundness

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

Definition 37. Pos' and Neg' are sets of variables such that

1. If α is not defined in E, then $\mathit{Pos}'(\alpha/E) = \{\alpha\}$ and $\mathit{Neg}'(\alpha/E) = \emptyset$.
2. If $\delta = \tau$ is in E, then $\mathit{Pos}'(\delta/E) = \mathit{Pos}'(\tau/E) \cup \{\delta\}$ and $\mathit{Neg}'(\delta/E) = \mathit{Neg}'(\tau/E)$.
3. $\mathit{Pos}'(\bot/E) = \mathit{Neg}'(\bot/E) = \emptyset$
4. $\mathit{Pos}'(\top/E) = \mathit{Neg}'(\top/E) = \emptyset$
5. $\mathit{Pos}'(\tau_1 \to \tau_2/E) = \mathit{Pos}'(\tau_2/E) \cup \mathit{Neg}'(\tau_1/E)$ and $\mathit{Neg}'(\tau_1 \to \tau_2/E) = \mathit{Neg}'(\tau_2/E) \cup \mathit{Pos}'(\tau_1/E)$

Let D be the set of E's defined variables. Then $\mathit{Pos}(\tau/E) = \mathit{Pos}'(\tau/E) - D$ and $\mathit{Neg}(\tau/E) = \mathit{Neg}'(\tau/E) - D$.

Note that *Pos* and *Neg* exclude defined variables while Pos' and Neg' include defined variables. Many functions satisfy these equations. For example, choosing

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

satisfies the equations, but the least solution is

$$\mathit{Pos}(\delta/\delta = \delta \to \delta) = \mathit{Neg}(\delta/\delta = \delta \to \delta) = \emptyset$$

Our results apply to the least solutions of the equations. It is easy to construct the least sets for *Pos* and *Neg* by adding variables only as necessary to satisfy the clauses of Definition 37.

The following relationship between defined and regular variables is easy to show using Definition 37. The intuition is that if α is positive (resp. negative) in the definition of δ, then α is positive (resp. negative) in any position where δ appears positively and negative (resp. positive) in any position where δ appears negatively.

Lemma 38. If $\alpha \in \mathit{Pos}'(\delta/E)$ then

$$\begin{aligned}\delta \in \mathit{Neg}'(\tau/E) \Rightarrow \alpha \in \mathit{Neg}'(\tau/E)\\ \delta \in \mathit{Pos}'(\tau/E) \Rightarrow \alpha \in \mathit{Pos}'(\tau/E)\end{aligned}$$

If $\alpha \in \mathit{Neg}'(\delta/E)$ then

$$\begin{aligned}\delta \in \mathit{Neg}'(\tau/E) \Rightarrow \alpha \in \mathit{Pos}'(\tau/E)\\ \delta \in \mathit{Pos}'(\tau/E) \Rightarrow \alpha \in \mathit{Neg}'(\tau/E)\end{aligned}$$

Lemma 39. Let τ be a recursive type expression and let $d_1, d_2 \in \mathcal{D}_0$ where $d_1 \preceq d_2$. Let θ be any assignment. If the semantic domain has contractive solutions and type continuity, then

1. if $\alpha \notin \mathit{Pos}(\tau)$, then $\theta[\alpha \leftarrow d_2](\tau) \preceq \theta[\alpha \leftarrow d_1](\tau)$.
2. if $\alpha \notin \mathit{Neg}(\tau)$, then $\theta[\alpha \leftarrow d_1](\tau) \preceq \theta[\alpha \leftarrow d_2](\tau)$.

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 8. 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 $\alpha \in \mathit{Pos}(\delta_1/E)$ and $\alpha \in \mathit{Neg}(\delta_1/E)$, then the result is vacuously true. If $\alpha \notin \mathit{Pos}(\delta_1/E)$ and $\alpha \notin \mathit{Neg}(\delta_1/E)$, 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 33, 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.

Assume $\alpha \notin \mathit{Pos}(\delta_1/E)$ and $\alpha \in \mathit{Neg}(\delta_1/E)$. We claim $\delta_1 \notin \mathit{Neg}'(\tau_1/E)$. To see this, note that

$$\begin{aligned}&\delta_1 \in \mathit{Neg}'(\tau_1/E) \wedge \alpha \in \mathit{Neg}'(\delta_1/E) &\\ &\Rightarrow \alpha \in \mathit{Pos}'(\tau_1/E) &\text{by Lemma 38}\\ &\Rightarrow \alpha \in \mathit{Pos}'(\delta_1/E) &\text{by Definition 37}\\ &\Rightarrow \alpha \in \mathit{Pos}(\delta_1/E) &\text{by Definition 37}\end{aligned}$$

which violates the assumption $\alpha \notin \mathit{Pos}(\delta_1/E)$. Therefore $\delta_1 \notin \mathit{Neg}'(\tau_1/E)$. Let E' be E with the equation $\delta_1 = \tau_1$ deleted. Now

$$\begin{array}{l} \quad \delta_1 \notin \mathit{Neg}'(\tau_1/E) \\ \Rightarrow \delta_1 \notin \mathit{Neg}'(\tau_1/E') \text{ see below} \\ \Rightarrow \delta_1 \notin \mathit{Neg}(\tau_1/E') \ \text{ since } \mathit{Neg}(\tau_1/E') \subseteq \mathit{Neg}'(\tau_1/E') \end{array}$$

The second line follows because deleting equations from E can only decrease the least solutions of the equations for Pos' and Neg'.

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 $\alpha \notin \mathit{Pos}(\tau_1/E')$, 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

$$\begin{array}{lr} \quad F_{d_0}(h(d_0)) & \\ = \theta[\alpha \leftarrow d_0][\delta_1 \leftarrow h(d_0)](\tau_1/E') & \\ = (\theta[\alpha \leftarrow d_0][\delta_1 \leftarrow h(d_0)])^{E'}(\tau_1) & \\ = ((\theta[\alpha \leftarrow d_0]^{E})^{E'}(\tau_1) & \text{by part 3 of Definition 30} \\ = (\theta[\alpha \leftarrow d_0])^{E}(\tau_1) & \text{by part 1 of Lemma 31} \\ = (\theta[\alpha \leftarrow d_0])^{E}(\delta_1) & \\ = \theta[\alpha \leftarrow d_0](\delta_1/E) & \\ = h(d_0) & \end{array}$$

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.

Corollary 40. 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*:

$$\mathit{Pos}(\forall\alpha.\sigma) = \mathit{Pos}(\sigma) - \{\alpha\}$$
$$\mathit{Neg}(\forall\alpha.\sigma) = \mathit{Neg}(\sigma) - \{\alpha\}$$

Lemma 41. If σ is a quantified recursive type expression and the semantic domain has type continuity and contractive solutions, then

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

Proof. Same as the proof for Lemma 10.

Example 7. 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 41 allows us to conclude that

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

The next example shows that the assumption of type continuity is needed in the proof of Lemma 41.

Example 8. Consider the type expression $\forall\alpha.(\delta/\delta = \alpha \to \delta)$. If Lemma 41 holds, then we have

$$\begin{array}{ll} \ \ \delta/\delta = \top \to \delta & \\ \equiv \forall\alpha.(\delta/\delta = \alpha \to \delta) & \text{by Lemma 41} \\ \preceq (\delta/\delta = \bot \to \delta) & \text{since } \bot \text{ is an instance of } \alpha \end{array}$$

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 41.

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 41 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:

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

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 8 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 36 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 42. 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 41.

Theorem 43. 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 14.

5.3 Completeness

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

Definition 44. 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 45. 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 17 and so is omitted. The most important new case is Part 6 of Definition 44. In this clause, note that s and t must be regular variables of E and E' respectively. Thus $B(S)$ does not mention any defined variables. This observation is needed to show that if $\theta^E(t_1) \preceq \theta^{E'}(t_2)$ then θ is a solution of $B(\{t_1/E \preceq t_2/E'\})$.

Lemma 46. Let $\forall V_1.\tau_1/E_1$ and $\forall V_2.\tau_2/E_2$ be compatible recursive type expressions. If the semantic domain has contractive solutions, standard function types, and standard glb types, then $B(\{\tau_1/E_1 \preceq \tau_2/E_2\})$ is a (V_1, V_2)-miniscule system of constraints.

Proof. Let $B(\{\tau_1/E_1 \preceq \tau_2/E_2\}) = \{s_1 \preceq t_1, \ldots, s_n \preceq t_n\}$. We show that $B(\{\tau_1/E_1 \preceq \tau_2/E_2\})$ satisfies the conditions of Definition 19.

1. By compatibility V_1 and V_2 are disjoint sets of variables.
2. By Part 3 of Definition 44 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 44, 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 $B(\{\tau_1/E_1 \preceq \tau_2/E_2\})$.
4. Same as Part 4 of the proof of Lemma 22 (but using Definition 44).
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 45, 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 47. 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 $\sigma_1 = \forall V_1.\tau_1/E$ and α-convert σ'' to $\sigma_2 = \forall V_2.\tau_2/E'$ in such a way that σ_1 and σ_2 are (V_1, V_2) compatible. By Lemma 46, $B(\tau_1/E \preceq \tau_2/E')$ is a (V_1, V_2)-miniscule system of constraints. By Lemma 20, $B(\tau_1/E \preceq \tau_2/E')$ 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 48. 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 43. 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 43, σ' is reduced. By Theorem 47, σ and σ' have the same number of quantified variables. Hence, σ is irredundant as desired.

Theorem 49. 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 42 and Theorem 48.

6 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.

7 Acknowledgements

The authors would like to thank Martin Abadi for useful discussions about connections to the ideal model as well as the anonymous referees for their criticisms.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proc. POPL'91.

[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.

[Cou79] Bruno Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.

[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.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California. LISP Pointers V, 1*, pages 193–204, June 1992.

[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.

[PO95] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

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

[Smi94] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.