Principal Type Schemes for Functional Programs with Overloading and Subtyping

Geoffrey S. Smith*
Cornell University†

December 1994

Abstract

We show how the Hindley/Milner polymorphic type system can be extended to incorporate overloading and subtyping. Our approach is to attach *constraints* to quantified types in order to restrict the allowed instantiations of type variables. We present an algorithm for inferring principal types and prove its soundness and completeness. We find that it is necessary in practice to simplify the inferred types, and we describe techniques for type simplification that involve shape unification, strongly connected components, transitive reduction, and the monotonicities of type formulas.

1 Introduction

Many algorithms have the property that they work correctly on many different types of input; such algorithms are called *polymorphic*. A polymorphic type system supports polymorphism by allowing some programs to have multiple types, thereby allowing them to be used with greater generality.

The popular polymorphic type system due to Hindley and Milner [7, 10, 3] uses universally quantified type formulas to describe the types of polymorphic programs. Each program has a best type, called the *principal type*, that captures all possible types for the program. For example, the program $\lambda f.\lambda x.f(f\,x)$ has principal type $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$; any other type for this program can be obtained by instantiating the universally quantified type variable α appropriately. Another pleasant feature of the Hindley/Milner type system is the possibility of performing *type inference*—principal types can be inferred automatically, without the aid of type declarations.

However, there are two useful kinds of polymorphism that cannot be handled by the Hindley/Milner type system: *overloading* and *subtyping*. In the

*This work was supported jointly by the NSF and DARPA under grant ASC-88-00465.

†Author's current address: School of Computer Science, Florida International University, Miami, FL 33199; smithg@fiu.edu

Hindley/Milner type system, an assumption set may contain at most *one* typing assumption for any identifier; this makes it impossible to express the types of an overloaded operation like multiplication. For $*$ has types $int \to int \to int$[1] and $real \to real \to real$ (and perhaps others), but it does not have type $\forall\alpha.\alpha \to \alpha \to \alpha$. So any single typing $* : \sigma$ is either too narrow or too broad. As for subtyping, the Hindley/Milner system does not provide for subtype inclusions such as $int \subseteq real$.

This paper extends the Hindley/Milner system to incorporate overloading and subtyping, while preserving the existence of principal types and the ability to do type inference. In order to preserve principal types, we need a richer set of type formulas. The key device needed is *constrained (universal) quantification*, in which quantified variables are allowed only those instantiations that satisfy a set of *constraints*.

To deal with overloading, we require *typing* constraints of the form $x : \tau$, where x is an overloaded identifier. To see the need for such constraints, consider a function $expon(x, n)$ that calculates x^n, and that is written in terms of $*$ and 1, which are overloaded. Then the types of *expon* should be all types of the form $\alpha \to int \to \alpha$, provided that $* : \alpha \to \alpha \to \alpha$ and $1 : \alpha$; these types are described by the formula $\forall\alpha$ **with** $* : \alpha \to \alpha \to \alpha,\ 1 : \alpha \,.\, \alpha \to int \to \alpha$.

To deal with subtyping, we require *inclusion* constraints of the form $\tau \subseteq \tau'$. Consider, for example, the function $\lambda f.\lambda x.f(f\ x)$. In the Hindley/Milner system, this function has principal type $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$. But in the presence of subtyping, this type is no longer principal—if $int \subseteq real$, then $\lambda f.\lambda x.f(f\ x)$ has type $(real \to int) \to (real \to int)$, but this type is not deducible from $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$. The principal type turns out to be $\forall\alpha, \beta$ **with** $\beta \subseteq \alpha \,.\, (\alpha \to \beta) \to (\alpha \to \beta)$.

A subtle issue that arises with the use of constrained quantification is the *satisfiability* of constraint sets. A type with an unsatisfiable constraint set is *vacuous*; it has no instances. We must take care, therefore, not to call a program well typed unless we can give it a type with a satisfiable constraint set.

1.1 Related Work

Overloading (without subtyping) has also been investigated by Kaes [8] and by Wadler and Blott [19]. Kaes' work restricts overloading quite severely; for example he does not permit constants to be overloaded. Both Kaes' and Wadler/Blott's systems ignore the question of whether a constraint set is satisfiable, with the consequence that certain nonsensical expressions are regarded as well typed. For example, in Wadler/Blott's system the expression $true + true$ is well typed, even though $+$ does not work on booleans. Kaes' system has similar difficulties.

Subtyping (without overloading) has been investigated by (among many others) Mitchell [11], Stansifer [15], Fuh and Mishra [4, 5], and Curtis [2]. Mitchell, Stansifer, and Fuh and Mishra consider type inference with subtyping, but their

[1] Throughout this paper, we write functions in curried form.

languages do not include a **let** expression; we will see that the presence of **let** makes it much harder to prove the completeness of our type inference algorithm. Curtis studies a very rich type system that is not restricted to *shallow* types. The richness of his system makes it hard to characterize much of his work; for example he does not address the completeness of his inference algorithm. Fuh and Mishra and Curtis also explore type simplification.

1.2 Outline of the Rest of the Paper

In Section 2, we give the rules of the type system. In Section 3, we present algorithm W_{os} for inferring principal types. Section 4 contains the proofs that W_{os} is sound and complete. Section 5 describes techniques for simplifying the types produced by W_{os}. Section 6 briefly discusses the problem of testing the satisfiability of a constraint set. Finally, Section 7 concludes with a number of examples of type inference.

2 The Type System

The language that we study is the simple *core-ML* of Damas and Milner [3]. Given a set of *identifiers* (x, y, a, $\leq$, 1, ...), the set of *expressions* is given by

$$e ::= x \mid \lambda x.e \mid e\,e' \mid \textbf{let } x = e \textbf{ in } e'.$$

Given a set of *type variables* ($\alpha, \beta, \gamma, \ldots$) and a set of *type constructors* (*int*, *bool*, *char*, *set*, *seq*, ...) of various arities, we define the set of (unquantified) *types* by

$$\tau ::= \alpha \mid \tau \to \tau' \mid \chi(\tau_1, \ldots, \tau_n)$$

where χ is an n-ary type constructor. If χ is 0-ary, then the parentheses are omitted. As usual, $\to$ associates to the right. Types will be denoted by τ, π, ρ, ϕ, or ψ. We say that a type is *atomic* if it is a type constant (that is, a 0-ary type constructor) or a type variable.

Next we define the set of quantified types, or *type schemes*, by

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n \textbf{ with } \mathcal{C}_1, \ldots, \mathcal{C}_m \,.\, \tau,$$

where each $\mathcal{C}_i$ is a *constraint*, which is either a typing $x : \pi$ or an inclusion $\pi \subseteq \pi'$. We use overbars to abbreviate sequences; for example $\alpha_1, \alpha_2, \ldots, \alpha_n$ is abbreviated as $\bar{\alpha}$.

A *substitution* is a set of simultaneous replacements for type variables:

$$[\alpha_1, \ldots, \alpha_n := \tau_1, \ldots, \tau_n]$$

where the α_i's are distinct. We write the application of substitution S to type σ as σS, and we write the composition of substitutions S and T as ST. A substitution S can be applied to a typing $x : \sigma$ or an inclusion $\tau \subseteq \tau'$, yielding $x : (\sigma S)$ and $(\tau S) \subseteq (\tau' S)$, respectively.

When a substitution is applied to a quantified type, the usual difficulties with bound variables and capture must be handled. We define

$$(\forall\bar{\alpha}\ \textbf{with}\ C\,.\,\tau)S \quad = \quad \forall\bar{\beta}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}]S\,.\,\tau[\bar{\alpha} := \bar{\beta}]S,$$

where $\bar{\beta}$ are fresh type variables occurring neither in $\forall\bar{\alpha}$ **with** $C\,.\,\tau$ nor in S.

We occasionally need *updated* substitutions. The substitution $S \oplus [\bar{\alpha} := \bar{\tau}]$ is the same as S, except that each α_i is mapped to τ_i.

We are now ready to give the rules of our type system. There are two kinds of assertions that we are interested in proving: typings $e : \sigma$ and inclusions $\tau \subseteq \tau'$. These assertions will in general depend on a set of *assumptions* A, which contains the typings of built-in identifiers (e.g. $1 : \mathit{int}$) and basic inclusions (e.g. $\mathit{int} \subseteq \mathit{real}$). So the basic judgements of our type system are $A \vdash e : \sigma$ ("from assumptions A it follows that expression e has type σ") and $A \vdash \tau \subseteq \tau'$ ("from assumptions A it follows that type τ is a subtype of type τ'").

More precisely, an *assumption set* A is a finite set of assumptions, each of which is either an identifier typing $x : \sigma$ or an inclusion $\tau \subseteq \tau'$. An assumption set A may contain more than one typing for an identifier x; in this case we say that x is *overloaded* in A. If there is an assumption about x in A, or if some assumption in A has a constraint $x : \tau$, then we say that x *occurs* in A.

The rules for proving typings are given in Figure 1 and the rules for proving inclusions are given in Figure 2. If C is a set of typings or inclusions, then the notation $A \vdash C$ represents

$$A \vdash \mathcal{C}, \quad \text{for all } \mathcal{C} \text{ in } C.$$

(This notation is used in rules ($\forall$-intro) and ($\forall$-elim).) If $A \vdash e : \sigma$ for some σ, then we say that e is *well typed* with respect to A.

Our typing rules (hypoth), ($\rightarrow$-intro), ($\rightarrow$-elim), and (let) are the same as in Damas and Milner [3], except for the restrictions on ($\rightarrow$-intro) and (let), which are necessary to avoid certain anomalies. Because of the restrictions, we need a rule, ($\equiv_\alpha$), to allow the renaming of bound program identifiers; this allows the usual block structure in programs. Also ($\equiv_\alpha$) allows the renaming of bound type variables.

It should be noted that rule (let) cannot be used to create an overloading for an identifier; as a result, the only overloadings in the language are those given by the initial assumption set.[2]

Rules ($\forall$-intro) and ($\forall$-elim) are unusual, since they must deal with constraint sets. These rules are equivalent to rules in [19], with one important exception: the second hypothesis of the ($\forall$-intro) rule allows a constraint set to be moved into a type scheme only if the constraint set is satisfiable. This restriction, which is not present in the system of [19], is crucial in preventing many nonsensical expressions from being well typed. For example, from the assumptions $+$:

[2] This is not to say that our system disallows user-defined overloadings; it would be simple to provide a mechanism allowing users to add overloadings to the initial assumption set. The only restriction is that such overloadings must have *global* scope; as observed in [19], *local* overloadings complicate the existence of principal typings.

(hypoth) $A \vdash x : \sigma$, if $x : \sigma \in A$

($\rightarrow$-intro)
$$\frac{A \cup \{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \rightarrow \tau'} \quad (x \text{ does not occur in } A)$$

($\rightarrow$-elim)
$$\frac{\begin{array}{l} A \vdash e : \tau \rightarrow \tau' \\ A \vdash e' : \tau \end{array}}{A \vdash e\, e' : \tau'}$$

(let)
$$\frac{\begin{array}{l} A \vdash e : \sigma \\ A \cup \{x : \sigma\} \vdash e' : \tau \end{array}}{A \vdash \textbf{let } x = e \textbf{ in } e' : \tau} \quad (x \text{ does not occur in } A)$$

($\forall$-intro)
$$\frac{\begin{array}{l} A \cup C \vdash e : \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \forall \bar{\alpha} \textbf{ with } C \,.\, \tau} \quad (\bar{\alpha} \text{ not free in } A)$$

($\forall$-elim)
$$\frac{\begin{array}{l} A \vdash e : \forall \bar{\alpha} \textbf{ with } C \,.\, \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \tau[\bar{\alpha} := \bar{\pi}]}$$

($\equiv_\alpha$)
$$\frac{\begin{array}{l} A \vdash e : \sigma \\ e \equiv_\alpha e' \\ \sigma \equiv_\alpha \sigma' \end{array}}{A \vdash e' : \sigma'}$$

($\subseteq$)
$$\frac{\begin{array}{l} A \vdash e : \tau \\ A \vdash \tau \subseteq \tau' \end{array}}{A \vdash e : \tau'}$$

Figure 1: Typing Rules

$$
\begin{array}{ll}
\text{(hypoth)} & A \vdash \tau \subseteq \tau', \text{ if } (\tau \subseteq \tau') \in A \\[2ex]
\text{(reflex)} & A \vdash \tau \subseteq \tau \\[2ex]
\text{(trans)} & \dfrac{A \vdash \tau \subseteq \tau' \quad A \vdash \tau' \subseteq \tau''}{A \vdash \tau \subseteq \tau''} \\[2ex]
((-) \to (+)) & \dfrac{A \vdash \tau' \subseteq \tau \quad A \vdash \rho \subseteq \rho'}{A \vdash (\tau \to \rho) \subseteq (\tau' \to \rho')} \\[2ex]
(seq(+)) & \dfrac{A \vdash \tau \subseteq \tau'}{A \vdash seq(\tau) \subseteq seq(\tau')}
\end{array}
$$

Figure 2: Subtyping Rules

$int \to int \to int$, $+ : real \to real \to real$, and $true : bool$, then without the satisfiability condition it would follow that $true + true$ has type

$$\forall \ \textbf{with} + : bool \to bool \to bool \,.\, bool$$

even though + doesn't work on *bool*!

Inclusion rule (hypoth) allows inclusion assumptions to be used, and rules (reflex) and (trans) assert that $\subseteq$ is reflexive and transitive. The remaining inclusion rules express the well-known monotonicities of the various type constructors [13]. For example, $\to$ is antimonotonic in its first argument and monotonic in its second argument. The name $((-) \to (+))$ compactly represents this information. Finally, rule $(\subseteq)$ links the inclusion sublogic to the typing sublogic—it says that an expression of type τ has any supertype of τ as well.

As an example, here is a derivation of the typing

$$\{\} \vdash \lambda f.\lambda x.f(f\ x) : \forall \alpha, \beta \ \textbf{with}\ \beta \subseteq \alpha \,.\, (\alpha \to \beta) \to (\alpha \to \beta).$$

We have

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash f : \alpha \to \beta \tag{1}$$

by (hypoth),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash x : \alpha \tag{2}$$

by (hypoth),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash (f\ x) : \beta \tag{3}$$

by ($\to$-elim) on (1) and (2),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash \beta \subseteq \alpha \tag{4}$$

by (hypoth),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash (f\ x) : \alpha \tag{5}$$

by ($\subseteq$) on (3) and (4),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash f(f\ x) : \beta \tag{6}$$

by ($\to$-elim) on (1) and (5),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta\} \vdash \lambda x.f(f\ x) : \alpha \to \beta \tag{7}$$

by ($\to$-intro) on (6),

$$\{\beta \subseteq \alpha\} \vdash \lambda f.\lambda x.f(f\ x) : (\alpha \to \beta) \to (\alpha \to \beta) \tag{8}$$

by ($\to$-intro) on (7),

$$\{\} \vdash (\beta \subseteq \alpha)[\beta := \alpha] \tag{9}$$

by (reflex), and finally

$$\{\} \vdash \lambda f.\lambda x.f(f\ x) : \forall \alpha, \beta\ \textbf{with}\ \beta \subseteq \alpha\,.\,(\alpha \to \beta) \to (\alpha \to \beta) \tag{10}$$

by ($\forall$-intro) on (8) and (9).

Given a typing $A \vdash e : \sigma$, other types for e may be obtained by extending the derivation with the ($\forall$-elim) and ($\subseteq$) rules. The set of types thus derivable is captured by the *instance relation*, $\geq_A$.

Definition 1 *($\forall \bar{\alpha}$* **with** *$C\,.\,\tau$) $\geq_A \tau'$ if there is a substitution $[\bar{\alpha} := \bar{\pi}]$ such that*

- *$A \vdash C[\bar{\alpha} := \bar{\pi}]$ and*
- *$A \vdash \tau[\bar{\alpha} := \bar{\pi}] \subseteq \tau'$.*

Furthermore we say that $\sigma \geq_A \sigma'$ if for all τ, $\sigma' \geq_A \tau$ implies $\sigma \geq_A \tau$. In this case we say that σ' is an instance *of σ with respect to A.*

Now we can define the important notion of a *principal typing*.

Definition 2 *The typing $A \vdash e : \sigma$ is said to be* principal *if for all typings $A \vdash e : \sigma'$, $\sigma \geq_A \sigma'$. In this case σ is said to be a principal type for e with respect to A.*

An expression may have many principal types; for example, in Section 5 we show how a complex principal type can be systematically transformed into a much simpler (and more useful) principal type.

We now turn to the problem of inferring principal types.

3 Type Inference

For type inference, we make some assumptions about the initial assumption set. In particular, we disallow inclusion assumptions like $int \subseteq (int \to int)$, in which the two sides of the inclusion do not have the same 'shape'. Furthermore, we disallow 'cyclic' sets of inclusions such as $bool \subseteq int$ together with $int \subseteq bool$. More precisely, we say that assumption set A has *acceptable inclusions* if

$W_{os}(A, e)$ is defined by cases:

1. e is x

   if x is overloaded in A with *lcg* $\forall\bar{\alpha}.\tau$,
   return $([\,], \{x : \tau[\bar{\alpha} := \bar{\beta}]\}, \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
   else if $(x : \forall\bar{\alpha}\ \mathbf{with}\ C\,.\,\tau) \in A$,
   return $([\,], C[\bar{\alpha} := \bar{\beta}], \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
   else *fail*.

2. e is $\lambda x.e'$

   if x occurs in A, then rename x to a new identifier;
   let $(S_1, B_1, \tau_1) = W_{os}(A \cup \{x : \alpha\}, e')$ where α is new;
   return $(S_1, B_1, \alpha S_1 \to \tau_1)$.

3. e is $e'e''$

   let $(S_1, B_1, \tau_1) = W_{os}(A, e')$;
   let $(S_2, B_2, \tau_2) = W_{os}(AS_1, e'')$;
   let $S_3 = \mathit{unify}(\tau_1 S_2, \alpha \to \beta)$ where α and β are new;
   return $(S_1S_2S_3, B_1S_2S_3 \cup B_2S_3 \cup \{\tau_2 S_3 \subseteq \alpha S_3\}, \beta S_3)$.

4. e is $\mathbf{let}\ x = e'\ \mathbf{in}\ e''$

   if x occurs in A, then rename x to a new identifier;
   let $(S_1, B_1, \tau_1) = W_{os}(A, e')$;
   let $(S_2, B_1', \sigma_1) = \mathit{close}(AS_1, B_1, \tau_1)$;
   let $(S_3, B_2, \tau_2) = W_{os}(AS_1S_2 \cup \{x : \sigma_1\}, e'')$;
   return $(S_1S_2S_3, B_1'S_3 \cup B_2, \tau_2)$.

Figure 3: Algorithm W_{os}

- A contains only constant inclusions (i.e. inclusions of the form $c \subseteq d$, where c and d are type constants), and
- the reflexive transitive closure of the inclusions in A is a partial order.

Less significantly, we do not allow assumption sets to contain any typings $x : \sigma$ where σ has an unsatisfiable constraint set; we say that an assumption set has *satisfiable constraints* if it contains no such typings.

Henceforth, we assume that the initial assumption set has acceptable inclusions and satisfiable constraints.

Principal types for our language can be inferred using algorithm W_{os}, given in Figure 3. W_{os} is a generalization of Milner's algorithm W [10, 3]. Given initial assumption set A and expression e, $W_{os}(A, e)$ returns a triple (S, B, τ), such that

$$AS \cup B \vdash e : \tau.$$

$close(A, B, \tau)$:

let $\bar{\alpha}$ be the type variables free in B or τ but not in A;
let C be the set of constraints in B in which some α_i occurs;
if A has no free type variables,
  then if B is satisfiable with respect to A, then $B' = \{\}$ else *fail*
  else $B' = B$;
return $([], B', \forall\bar{\alpha}$ **with** $C \,.\, \tau)$.

Figure 4: A simple function *close*

Informally, τ is the type of e, B is a set of constraints describing all the uses made of overloaded identifiers in e as well as all the subtyping assumptions made, and S is a substitution that contains refinements to the typing assumptions in A.

Case 1 of W_{os} makes use of the *least common generalization* (*lcg*) [12] of an overloaded identifier x, as a means of capturing any common structure among the overloadings of x. For example, the *lcg* of $*$ is $\forall\alpha.\alpha \to \alpha \to \alpha$.

Case 3 of W_{os} is the greatest departure from algorithm W. Informally, we type an application $e'e''$ by first finding types for e' and e'', then ensuring that e' is indeed a function, and finally ensuring that the type of e'' is a *subtype* of the domain of e'.

Case 4 of W_{os} uses a function *close*, a simple version of which is given in Figure 4. The idea behind *close* is to take a typing $A \cup B \vdash e : \tau$ and, roughly speaking, to apply ($\forall$-intro) to it as much as possible. Because of the satisfiability condition in our ($\forall$-intro) rule, *close* needs to check whether constraint set B is satisfiable with respect to A; we defer discussion of how this might be implemented until Section 6.

Actually, there is a considerable amount of freedom in defining *close*; one can give fancier versions that do more type simplification. We will explore this possibility in Section 5.

4 Correctness of W_{os}

In this section, we prove the correctness of W_{os}. To begin with, we state a number of lemmas that give useful and fairly obvious properties of the type system. The proofs, which typically use induction on the length of the derivation, are mostly straightforward and are omitted.[3]

First, derivations are preserved under substitution:

Lemma 3 *If $A \vdash e : \sigma$ then $AS \vdash e : \sigma S$. If $A \vdash \tau \subseteq \tau'$, then $AS \vdash \tau S \subseteq \tau' S$.*

Next we give conditions under which an assumption is not needed in a derivation:

Lemma 4 *If $A \cup \{x : \sigma\} \vdash y : \tau$, x does not occur in A, and x and y are distinct identifiers, then $A \vdash y : \tau$. If $A \cup \{x : \sigma\} \vdash \tau \subseteq \tau'$, then $A \vdash \tau \subseteq \tau'$.*

[3] Proofs can be found in [14].

Extra assumptions never cause problems:

Lemma 5 *If $A \vdash e : \sigma$ then $A \cup B \vdash e : \sigma$. If $A \vdash \tau \subseteq \tau'$ then $A \cup B \vdash \tau \subseteq \tau'$.*

More substantially, there is a *normal form* theorem for derivations. Let ($\forall$-elim$'$) be the following weakened ($\forall$-elim) rule:

$$(\forall\text{-elim}') \qquad \frac{(x : \forall\bar{\alpha}\ \textbf{with}\ C\,.\,\tau)\ \in\ A \qquad A \vdash C[\bar{\alpha} := \bar{\pi}]}{A \vdash x : \tau[\bar{\alpha} := \bar{\pi}].}$$

Write $A \vdash' e : \sigma$ if this typing is derivable in the system obtained by deleting the ($\forall$-elim) rule and replacing it with the ($\forall$-elim$'$) rule. In view of the following theorem, $\vdash'$ derivations may be viewed as a *normal form* for $\vdash$ derivations.

Theorem 6 *$A \vdash e : \sigma$ if and only if $A \vdash' e : \sigma$.*

Now we turn to properties of assumption sets with acceptable inclusions.

Definition 7 *Types τ and τ' have the* same shape *if either*

- *τ and τ' are atomic or*
- *$\tau = \chi(\tau_1, \ldots, \tau_n)$, $\tau' = \chi(\tau'_1, \ldots, \tau'_n)$, where χ is an n-ary type constructor, $n \geq 1$, and for all i, τ_i and τ'_i have the same shape.*

Lemma 8 *If A contains only atomic inclusions (i.e. inclusions among atomic types) and $A \vdash \tau \subseteq \tau'$, then τ and τ' have the same shape.*

Lemma 9 *If A contains only atomic inclusions and $A \vdash \tau \to \rho \subseteq \tau' \to \rho'$, then $A \vdash \tau' \subseteq \tau$ and $A \vdash \rho \subseteq \rho'$.*

Similar lemmas hold for the other type constructors.

Finally, we show the correctness of W_{os}. The properties of *close* needed to prove the soundness and completeness of W_{os} are extracted into the following two lemmas:

Lemma 10 *If $(S, B', \sigma) = close(A, B, \tau)$ succeeds, then for any e, if $A \cup B \vdash e : \tau$ then $AS \cup B' \vdash e : \sigma$. Also, every identifier occurring in B' or in σ occurs in B.*

Lemma 11 *Suppose that A has acceptable inclusions and $AR \vdash BR$. Then $(S, B', \sigma) = close(A, B, \tau)$ succeeds and*

- *$B' = \{\}$, if A has no free type variables;*
- *free-vars$(\sigma) \subseteq$ free-vars(AS); and*
- *there exists T such that*
  1. *$R = ST$,*

2. *$AR \vdash B'T$, and*

3. *$\sigma T \geq_{AR} \tau R$.*

The advantage of this approach is that *close* may be given *any* definition satisfying the above lemmas, and W_{os} will remain correct. We exploit this possibility in Section 5.

The soundness of W_{os} is given by the following theorem:

Theorem 12 *If $(S, B, \tau) = W_{os}(A, e)$ succeeds, then $AS \cup B \vdash e : \tau$. Also, every identifier in B is overloaded in A or occurs in a constraint of some assumption in A.*

The proof is straightforward by induction on the structure of e.

We now establish the completeness of W_{os}. If our language did not contain **let**, then we could directly prove the following theorem by induction.

Theorem *If $AS \vdash e : \tau$, AS has satisfiable constraints, and A has acceptable inclusions, then $(S_0, B_0, \tau_0) = W_{os}(A, e)$ succeeds and there exists a substitution T such that*

1. *$S = S_0T$, except on new type variables of $W_{os}(A, e)$,*

2. *$AS \vdash B_0T$, and*

3. *$AS \vdash \tau_0 T \subseteq \tau$.*

Unfortunately, the presence of **let** forces us to a less direct proof.

Definition 13 *Let A and A' be assumption sets. We say that A is* stronger than *A', written $A \succeq A'$, if A and A' contain the same inclusions and $A' \vdash x : \tau$ implies $A \vdash x : \tau$.*

Roughly speaking, $A \succeq A'$ means that A can do anything that A' can. One would expect, then, that we could prove the following lemma:

Lemma *If $A' \vdash e : \tau$, A' has satisfiable constraints, and $A \succeq A'$, then $A \vdash e : \tau$.*

This lemma is needed to prove the completeness theorem above, but it appears to defy a straightforward inductive proof.[4] This forces us to combine the completeness theorem and the lemma into a single theorem that yields both as corollaries and that allows both to be proved simultaneously. We now do this.

Theorem 14 *Suppose that $A' \vdash e : \tau$, A' has satisfiable constraints, $AS \succeq A'$, and A has acceptable inclusions. Then $(S_0, B_0, \tau_0) = W_{os}(A, e)$ succeeds and there exists a substitution T such that*

1. *$S = S_0T$, except on new type variables of $W_{os}(A, e)$,*

2. *$AS \vdash B_0T$, and*

[4] The key difficulty is that it is possible that $A \succeq A'$ and yet $A \cup C \not\succeq A' \cup C$.

3. $AS \vdash \tau_0 T \subseteq \tau$.

Proof: By induction on the structure of e. For simplicity, assume that the bound identifiers of e have been renamed so that they are all distinct and so that they do not occur in A. By Theorem 6, $A' \vdash' e : \tau$. Now consider the four possible forms of e:

- <u>e is x</u>

  By the definition of $AS \succeq A'$, we have $AS \vdash' x : \tau$. Without loss of generality, we may assume that the derivation of $AS \vdash' x : \tau$ ends with a (possibly trivial) use of ($\forall$-elim$'$) followed by a (possibly trivial) use of ($\subseteq$). If $(x : \forall\bar{\alpha}\ \textbf{with}\ C\,.\,\rho) \in A$, then $(x : \forall\bar{\beta}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}]S\,.\,\rho[\bar{\alpha} := \bar{\beta}]S) \in AS$, where $\bar{\beta}$ are the first distinct type variables not free in $\forall\bar{\alpha}\ \textbf{with}\ C\,.\,\rho$ or in S. Hence the derivation $AS \vdash' x : \tau$ ends with

$$\begin{array}{l} (x : \forall\bar{\beta}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}]S\,.\,\rho[\bar{\alpha} := \bar{\beta}]S) \in AS \\ AS \vdash' C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \\ \hline AS \vdash' x : \rho[\bar{\alpha} := \beta]S[\beta := \bar{\pi}] \\ AS \vdash' \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \subseteq \tau \\ \hline AS \vdash' x : \tau \end{array}$$

  We need to show that $(S_0, B_0, \tau_0) = W_{os}(A, x)$ succeeds and that there exists T such that

  1. $S = S_0 T$, except on new type variables of $W_{os}(A, x)$,
  2. $AS \vdash B_0 T$, and
  3. $AS \vdash \tau_0 T \subseteq \tau$.

  Now, $W_{os}(A, x)$ is defined by

  if x is overloaded in A with *lcg* $\forall\bar{\alpha}.\tau$,
   return $([\,], \{x : \tau[\bar{\alpha} := \bar{\beta}]\}, \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
  else if $(x : \forall\bar{\alpha}\ \textbf{with}\ C\,.\,\tau) \in A$,
   return $([\,], C[\bar{\alpha} := \bar{\beta}], \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
  else *fail*.

  If x is overloaded in A with *lcg* $\forall\bar{\gamma}.\rho_0$, then $(S_0, B_0, \tau_0) = W_{os}(A, x)$ succeeds with $S_0 = [\,]$, $B_0 = \{x : \rho_0[\bar{\gamma} := \bar{\delta}]\}$, and $\tau_0 = \rho_0[\bar{\gamma} := \bar{\delta}]$, where $\bar{\delta}$ are new. Since $\forall\bar{\gamma}.\rho_0$ is the *lcg* of x, $\bar{\gamma}$ are the only variables in ρ_0 and there exist $\bar{\phi}$ such that $\rho_0[\bar{\gamma} := \bar{\phi}] = \rho$. Let

$$T = S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]].$$

Then

$$
\begin{array}{ll}
 & \tau_0 T \\
= & \quad \ll \text{definition} \gg \\
 & \rho_0[\bar{\gamma} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]]) \\
= & \quad \ll \text{only } \bar{\delta} \text{ occur in } \rho_0[\bar{\gamma} := \bar{\delta}] \gg \\
 & \rho_0[\bar{\gamma} := \bar{\delta}][\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\gamma} \text{ occur in } \rho_0 \gg \\
 & \rho_0[\bar{\gamma} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\gamma} \text{ occur in } \rho_0 \gg \\
 & \rho_0[\bar{\gamma} := \bar{\phi}][\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \\
= & \quad \ll \text{by above} \gg \\
 & \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}].
\end{array}
$$

So

1. $S_0 T = S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] = S$, except on $\bar{\delta}$. That is, $S_0 T = S$, except on the new type variables of $W_{os}(A, x)$.
2. Since $B_0 T = \{x : \tau_0 T\} = \{x : \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]\}$, it follows that $AS \vdash B_0 T$.
3. We have $\tau_0 T = \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]$ and $AS \vdash \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \subseteq \tau$, so $AS \vdash \tau_0 T \subseteq \tau$.

If x is not overloaded in A, then $(S_0, B_0, \tau_0) = W_{os}(A, x)$ succeeds with $S_0 = []$, $B_0 = C[\bar{\alpha} := \bar{\delta}]$, and $\tau_0 = \rho[\bar{\alpha} := \bar{\delta}]$, where $\bar{\delta}$ are new. Observe that $[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}])$ and $[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]$ agree on C and on ρ.

> (The only variables in C or ρ are the $\bar{\alpha}$ and variables ϵ not among $\bar{\beta}$ or $\bar{\delta}$. Both substitutions map $\alpha_i \mapsto \pi_i$ and $\epsilon \mapsto \epsilon S$.)

Let T be $S \oplus [\bar{\delta} := \bar{\pi}]$. Then

1. $S_0 T = S \oplus [\bar{\delta} := \bar{\pi}] = S$, except on $\bar{\delta}$. That is, $S_0 T = S$, except on the new type variables of $W_{os}(A, x)$.
2. Also, $B_0 T = C[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}]) = C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]$ (by the above observation), so $AS \vdash B_0 T$.
3. Finally, $\tau_0 T = \rho[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}]) = \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]$ (by the above observation). Since $AS \vdash \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \subseteq \tau$, we have $AS \vdash \tau_0 T \subseteq \tau$.

- <u>e is $\lambda x.e'$</u>

  Without loss of generality, we may assume that the derivation of $A' \vdash' e : \tau$ ends with a use of ($\rightarrow$-intro) followed by a (possibly trivial) use of ($\subseteq$):

$$
\dfrac{\dfrac{A' \cup \{x : \tau'\} \vdash' e' : \tau''}{A' \vdash' \lambda x.e' : \tau' \rightarrow \tau''} \quad A' \vdash' (\tau' \rightarrow \tau'') \subseteq \tau}{A' \vdash' \lambda x.e' : \tau}
$$

where x does not occur in A'.

We must show that $(S_0, B_0, \tau_0) = W_{os}(A, \lambda x.e')$ succeeds and that there exists T such that

1. $S = S_0 T$, except on new type variables of $W_{os}(A, \lambda x.e')$,
2. $AS \vdash B_0 T$, and
3. $AS \vdash \tau_0 T \subseteq \tau$.

Now, $W_{os}(A, \lambda x.e')$ is defined by

> if x occurs in A, then rename x to a new identifier;
> let $(S_1, B_1, \tau_1) = W_{os}(A \cup \{x : \alpha\}, e')$ where α is new;
> return $(S_1, B_1, \alpha S_1 \to \tau_1)$.

By our renaming assumption, we can assume that x does not occur in A. Now we wish to use induction to show that the recursive call succeeds. The new type variable α is not free in A, so

$$AS \cup \{x : \tau'\} = (A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']).$$

Note next that $A' \cup \{x : \tau'\}$ has satisfiable constraints. Now we need $AS \cup \{x : \tau'\} \succeq A' \cup \{x : \tau'\}$. Both have the same inclusions. Suppose that $A' \cup \{x : \tau'\} \vdash y : \rho$. If $y \neq x$, then by Lemma 4, $A' \vdash y : \rho$. Since $AS \succeq A'$, we have $AS \vdash y : \rho$ and then by Lemma 5, $AS \cup \{x : \tau'\} \vdash y : \rho$. On the other hand, if $y = x$, then the derivation $A' \cup \{x : \tau'\} \vdash' y : \rho$ must be by (hypoth) followed by a (possibly trivial) use of $(\subseteq)$:

$$\frac{A' \cup \{x : \tau'\} \vdash' x : \tau' \qquad A' \cup \{x : \tau'\} \vdash' \tau' \subseteq \rho}{A' \cup \{x : \tau'\} \vdash' x : \rho}$$

Since $AS \succeq A'$, AS and A' contain the same inclusions. Therefore, $AS \cup \{x : \tau'\} \vdash' \tau' \subseteq \rho$, so by (hypoth) followed by $(\subseteq)$, $AS \cup \{x : \tau'\} \vdash' x : \rho$. Finally, $A \cup \{x : \alpha\}$ has acceptable inclusions. In summary,

- $A' \cup \{x : \tau'\} \vdash e' : \tau''$,
- $A' \cup \{x : \tau'\}$ has satisfiable constraints,
- $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \succeq A' \cup \{x : \tau'\}$, and
- $A \cup \{x : \alpha\}$ has acceptable inclusions.

So by induction, $(S_1, B_1, \tau_1) = W_{os}(A \cup \{x : \alpha\}, e')$ succeeds and there exists T_1 such that

1. $S \oplus [\alpha := \tau'] = S_1 T_1$, except on new variables of $W_{os}(A \cup \{x : \alpha\}, e')$,
2. $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \vdash B_1 T_1$, and
3. $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \vdash \tau_1 T_1 \subseteq \tau''$.

So $(S_0, B_0, \tau_0) = W_{os}(A, \lambda x.e')$ succeeds with $S_0 = S_1$, $B_0 = B_1$, and $\tau_0 = \alpha S_1 \to \tau_1$.

Let T be T_1. Then

1. Observe that

$$\begin{array}{ll} & S_0 T \\ = & \quad \ll \text{definition} \gg \\ & S_1 T_1 \\ = & \quad \ll \text{by part 1 of the use of induction above} \gg \\ & S \oplus [\alpha := \tau'], \text{ except on new variables of } W_{os}(A \cup \{x : \alpha\}, e') \\ = & \quad \ll \text{definition of } \oplus \gg \\ & S, \text{ except on } \alpha. \end{array}$$

   Hence $S_0 T = S$, except on the new type variables of $W_{os}(A \cup \{x : \alpha\}, e')$ and on α. That is, $S_0 T = S$, except on the new type variables of $W_{os}(A, \lambda x.e')$.

2. $B_0 T = B_1 T_1$ and, by part 2 of the use of induction above, we have $AS \cup \{x : \tau'\} \vdash B_1 T_1$. Since x does not occur in A, it follows from Theorem 12 that x does not occur in B_1. Hence Lemma 4 may be applied to each member of $B_1 T_1$, yielding $AS \vdash B_1 T_1$.

3. Finally,

$$\begin{array}{ll} & \tau_0 T \\ = & \quad \ll \text{definition} \gg \\ & \alpha S_1 T_1 \to \tau_1 T_1 \\ = & \quad \ll \alpha \text{ is not a new type variable of } W_{os}(A \cup \{x : \alpha\}, e') \gg \\ & \alpha(S \oplus [\alpha := \tau']) \to \tau_1 T_1 \\ = & \quad \ll \text{definition of } \oplus \gg \\ & \tau' \to \tau_1 T_1. \end{array}$$

   Now by part 3 of the use of induction above, $AS \cup \{x : \tau'\} \vdash \tau_1 T_1 \subseteq \tau''$, so by Lemma 4, $AS \vdash \tau_1 T_1 \subseteq \tau''$. By (reflex) and $((-) \to (+))$, it follows that $AS \vdash (\tau' \to \tau_1 T_1) \subseteq (\tau' \to \tau'')$. In other words, $AS \vdash \tau_0 T \subseteq (\tau' \to \tau'')$. Next, because $A' \vdash (\tau' \to \tau'') \subseteq \tau$ and $AS \succeq A'$, we have $AS \vdash (\tau' \to \tau'') \subseteq \tau$. So by (trans) we have $AS \vdash \tau_0 T \subseteq \tau$.

- <u>e is $e'e''$</u>

  Without loss of generality, we may assume that the derivation of $A' \vdash' e : \tau$ ends with a use of ($\to$-elim) followed by a use of ($\subseteq$):

$$\begin{array}{c} A' \vdash' e' : \tau' \to \tau'' \\ A' \vdash' e'' : \tau' \\ \hline A' \vdash' e'\, e'' : \tau'' \\ A' \vdash' \tau'' \subseteq \tau \\ \hline A' \vdash' e'\, e'' : \tau \end{array}$$

We need to show that $(S_0, B_0, \tau_0) = W_{os}(A, e'e'')$ succeeds and that there exists T such that

1. $S = S_0T$, except on new type variables of $W_{os}(A, e'e'')$,
2. $AS \vdash B_0T$, and
3. $AS \vdash \tau_0T \subseteq \tau$.

Now, $W_{os}(A, e'e'')$ is defined by

$$\begin{array}{l}\text{let } (S_1, B_1, \tau_1) = W_{os}(A, e');\\ \text{let } (S_2, B_2, \tau_2) = W_{os}(AS_1, e'');\\ \text{let } S_3 = \mathit{unify}(\tau_1S_2, \alpha \to \beta) \text{ where } \alpha \text{ and } \beta \text{ are new;}\\ \text{return } (S_1S_2S_3, B_1S_2S_3 \cup B_2S_3 \cup \{\tau_2S_3 \subseteq \alpha S_3\}, \beta S_3).\end{array}$$

By induction, $(S_1, B_1, \tau_1) = W_{os}(A, e')$ succeeds and there exists T_1 such that

1. $S = S_1T_1$, except on new type variables of $W_{os}(A, e')$,
2. $AS \vdash B_1T_1$, and
3. $AS \vdash \tau_1T_1 \subseteq (\tau' \to \tau'')$.

Since AS has acceptable inclusions, by Lemma 8 τ_1T_1 is of the form $\rho \to \rho'$, and by Lemma 9 we have $AS \vdash \tau' \subseteq \rho$ and $AS \vdash \rho' \subseteq \tau''$.

Now $AS = A(S_1T_1) = (AS_1)T_1$, as the new type variables of $W_{os}(A, e')$ do not occur free in A. So by induction, $(S_2, B_2, \tau_2) = W_{os}(AS_1, e'')$ succeeds and there exists T_2 such that

1. $T_1 = S_2T_2$, except on new type variables of $W_{os}(AS_1, e'')$,
2. $(AS_1)T_1 \vdash B_2T_2$, and
3. $(AS_1)T_1 \vdash \tau_2T_2 \subseteq \tau'$.

The new type variables α and β do not occur in A, S_1, B_1, τ_1, S_2, B_2, or τ_2. So consider $T_2 \oplus [\alpha, \beta := \rho, \rho']$:

$$\begin{array}{ll} & (\tau_1S_2)(T_2 \oplus [\alpha, \beta := \rho, \rho'])\\ = & \qquad \ll \alpha \text{ and } \beta \text{ do not occur in } \tau_1S_2 \gg\\ & \tau_1S_2T_2\\ = & \qquad \ll \text{no new type variable of } W_{os}(AS_1, e'') \text{ occurs in } \tau_1 \gg\\ & \tau_1T_1\\ = & \qquad \ll \text{by above} \gg\\ & \rho \to \rho'\end{array}$$

In addition, $(\alpha \to \beta)(T_2 \oplus [\alpha, \beta := \rho, \rho']) = \rho \to \rho'$ by definition, so $S_3 = \mathit{unify}(\tau_1S_2, \alpha \to \beta)$ succeeds and there exists T_3 such that

$$T_2 \oplus [\alpha, \beta := \rho, \rho'] = S_3T_3.$$

So $(S_0, B_0, \tau_0) = W_{os}(A, e'e'')$ succeeds with $S_0 = S_1S_2S_3$, $B_0 = B_1S_2S_3 \cup B_2S_3 \cup \{\tau_2S_3 \subseteq \alpha S_3\}$, and $\tau_0 = \beta S_3$.

Let T be T_3. Then

1. We have

$$
\begin{array}{ll}
 & S_0T \\
= & \quad \ll \text{definition} \gg \\
 & S_1S_2S_3T_3 \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
 & S_1S_2(T_2 \oplus [\alpha, \beta := \rho, \rho']) \\
= & \quad \ll \alpha \text{ and } \beta \text{ do not occur in } S_1S_2 \gg \\
 & S_1S_2T_2, \text{ except on } \alpha \text{ and } \beta \\
= & \quad \ll \text{by part 1 of second use of induction and since the} \gg \\
 & \quad \ll \text{new variables of } W_{os}(AS_1, e'') \text{ don't occur in } S_1 \gg \\
 & S_1T_1, \text{ except on the new type variables of } W_{os}(AS_1, e'') \\
= & \quad \ll \text{by part 1 of the first use of induction above} \gg \\
 & S, \text{ except on the new type variables of } W_{os}(A, e').
\end{array}
$$

Hence $S_0T = S$ except on the new type variables of $W_{os}(A, e'e'')$.

2. Next,

$$
\begin{array}{ll}
 & B_0T \\
= & \quad \ll \text{definition} \gg \\
 & B_1S_2S_3T_3 \cup B_2S_3T_3 \cup \{\tau_2S_3T_3 \subseteq \alpha S_3T_3\} \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
 & B_1S_2(T_2 \oplus [\alpha, \beta := \rho, \rho']) \cup B_2(T_2 \oplus [\alpha, \beta := \rho, \rho']) \\
 & \qquad \cup\{\tau_2(T_2 \oplus [\alpha, \beta := \rho, \rho']) \subseteq \alpha(T_2 \oplus [\alpha, \beta := \rho, \rho'])\} \\
= & \quad \ll \alpha \text{ and } \beta \text{ do not occur in } B_1S_2,\ B_2, \text{ or } \tau_2 \gg \\
 & B_1S_2T_2 \cup B_2T_2 \cup \{\tau_2T_2 \subseteq \rho\} \\
= & \quad \ll \text{by part 1 of second use of induction and since the} \gg \\
 & \quad \ll \text{new variables of } W_{os}(AS_1, e'') \text{ don't occur in } B_1 \gg \\
 & B_1T_1 \cup B_2T_2 \cup \{\tau_2T_2 \subseteq \rho\}
\end{array}
$$

By part 2 of the first and second uses of induction above, $AS \vdash B_1T_1$ and $AS \vdash B_2T_2$. By part 3 of the second use of induction above, $AS \vdash \tau_2T_2 \subseteq \tau'$. Also, we found above that $AS \vdash \tau' \subseteq \rho$. So by (trans), $AS \vdash \tau_2T_2 \subseteq \rho$. Therefore, $AS \vdash B_0T$.

3. Finally, $\tau_0T = \beta S_3T_3 = \beta(T_2 \oplus [\alpha, \beta := \rho, \rho']) = \rho'$. Now, $AS \vdash \rho' \subseteq \tau''$ and, since $A' \vdash \tau'' \subseteq \tau$ and $AS \succeq A'$, also $AS \vdash \tau'' \subseteq \tau$. So it follows from (trans) that $AS \vdash \tau_0T \subseteq \tau$.

- <u>e is **let** $x = e'$ **in** e''</u>

  Without loss of generality we may assume that the derivation of $A' \vdash' e : \tau$ ends with a use of (let) followed by a (possibly trivial) use of ($\subseteq$):

$$
\begin{array}{c}
A' \vdash' e' : \sigma \\
A' \cup \{x : \sigma\} \vdash' e'' : \tau' \\
\hline
A' \vdash' \textbf{let } x = e' \textbf{ in } e'' : \tau' \\
A' \vdash' \tau' \subseteq \tau \\
\hline
A' \vdash' \textbf{let } x = e' \textbf{ in } e'' : \tau
\end{array}
$$

where x does not occur in A'.

We need to show that $(S_0, B_0, \tau_0) = W_{os}(A, \textbf{let } x = e' \textbf{ in } e'')$ succeeds and that there exists T such that

1. $S = S_0 T$, except on new type variables of $W_{os}(A, \textbf{let } x = e' \textbf{ in } e'')$,
2. $AS \vdash B_0 T$, and
3. $AS \vdash \tau_0 T \subseteq \tau$.

Now, $W_{os}(A, \textbf{let } x = e' \textbf{ in } e'')$ is defined by

if x occurs in A, then rename x to a new identifier;
let $(S_1, B_1, \tau_1) = W_{os}(A, e')$;
let $(S_2, B_1', \sigma_1) = \mathit{close}(AS_1, B_1, \tau_1)$;
let $(S_3, B_2, \tau_2) = W_{os}(AS_1 S_2 \cup \{x : \sigma_1\}, e'')$;
return $(S_1 S_2 S_3, B_1' S_3 \cup B_2, \tau_2)$.

Since A' has satisfiable constraints and $A' \vdash e' : \sigma$, it can be shown that there exists an *unquantified* type τ'' such that $A' \vdash e' : \tau''$. Hence by induction, $(S_1, B_1, \tau_1) = W_{os}(A, e')$ succeeds and there exists T_1 such that

1. $S = S_1 T_1$, except on new type variables of $W_{os}(A, e')$,
2. $AS \vdash B_1 T_1$, and
3. $AS \vdash \tau_1 T_1 \subseteq \tau''$.

By 1 and 2 above, we have $(AS_1)T_1 \vdash B_1 T_1$. Since AS_1 has acceptable inclusions, by Lemma 11 it follows that $(S_2, B_1', \sigma_1) = \mathit{close}(AS_1, B_1, \tau_1)$ succeeds, $\mathit{free\text{-}vars}(\sigma_1) \subseteq \mathit{free\text{-}vars}(AS_1 S_2)$, and there exists a substitution T_2 such that $T_1 = S_2 T_2$ and $(AS_1)T_1 \vdash B_1' T_2$.

Now, in order to apply the induction hypothesis to the second recursive call we need to show that $AS \cup \{x : \sigma_1 T_2\} \succeq A' \cup \{x : \sigma\}$. First, note that $AS \cup \{x : \sigma_1 T_2\}$ and $A' \cup \{x : \sigma\}$ contain the same inclusions. Next we must show that $A' \cup \{x : \sigma\} \vdash y : \rho$ implies $AS \cup \{x : \sigma_1 T_2\} \vdash y : \rho$, Suppose that $A' \cup \{x : \sigma\} \vdash y : \rho$. If $y \neq x$, then by Lemma 4, $A' \vdash y : \rho$. Since $AS \succeq A'$, we have $AS \vdash y : \rho$ and then by Lemma 5, $AS \cup \{x : \sigma_1 T_2\} \vdash y : \rho$.

If, on the other hand, $y = x$, then our argument will begin by establishing that $A' \vdash e' : \rho$. We may assume that the derivation of $A' \cup \{x : \sigma\} \vdash' x : \rho$ ends with a use of ($\forall$-elim$'$) followed by a use of ($\subseteq$): if σ is of the form $\forall \bar{\beta}$ **with** $C \,.\, \rho'$ then we have

$$\frac{\dfrac{x : \forall \bar{\beta} \textbf{ with } C \,.\, \rho' \in A' \cup \{x : \sigma\} \qquad A' \cup \{x : \sigma\} \vdash' C[\bar{\beta} := \bar{\pi}]}{A' \cup \{x : \sigma\} \vdash' x : \rho'[\beta := \bar{\pi}]} \qquad A' \cup \{x : \sigma\} \vdash' \rho'[\bar{\beta} := \bar{\pi}] \subseteq \rho}{A' \cup \{x : \sigma\} \vdash' x : \rho}$$

It is evident that x does not occur in C (if x occurs in C, then the derivation of $A' \cup \{x : \sigma\} \vdash' C[\bar{\beta} := \bar{\pi}]$ will be infinitely high), so by Lemma 4

applied to each member of $C[\bar{\beta} := \bar{\pi}]$, it follows that $A' \vdash C[\bar{\beta} := \bar{\pi}]$. Also, by Lemma 4, $A' \vdash \rho'[\bar{\beta} := \bar{\pi}] \subseteq \rho$. Therefore the derivation $A' \vdash e' : \forall\bar{\beta}$ **with** $C \,.\, \rho'$ may be extended using ($\forall$-elim) and ($\subseteq$):

$$\begin{array}{l} A' \vdash e' : \forall\bar{\beta} \textbf{ with } C \,.\, \rho' \\ A' \vdash C[\bar{\beta} := \bar{\pi}] \\ \hline A' \vdash e' : \rho'[\beta := \bar{\pi}] \\ A' \vdash \rho'[\bar{\beta} := \bar{\pi}] \subseteq \rho \\ \hline A' \vdash e' : \rho \end{array}$$

So by induction there exists a substitution T_3 such that

1. $S = S_1T_3$, except on new type variables of $W_{os}(A, e')$,
2. $AS \vdash B_1T_3$, and
3. $AS \vdash \tau_1T_3 \subseteq \rho$.

By 1 and 2 above, we have $(AS_1)T_3 \vdash B_1T_3$, so by Lemma 11 it follows that there exists a substitution T_4 such that

1. $T_3 = S_2T_4$,
2. $AS_1T_3 \vdash B_1'T_4$, and
3. $\sigma_1T_4 \geq_{AS_1T_3} \tau_1T_3$.

Now,

$$\begin{array}{ll} & AS_1S_2T_2 \\ = & \quad \ll \text{by first use of Lemma 11 above} \gg \\ & AS_1T_1 \\ = & \quad \ll \text{by part 1 of the first use of induction above} \gg \\ & AS \\ = & \quad \ll \text{by part 1 of the second use of induction above} \gg \\ & AS_1T_3 \\ = & \quad \ll \text{by second use of Lemma 11 above} \gg \\ & AS_1S_2T_4 \end{array}$$

Hence T_2 and T_4 are equal when restricted to the free variables of AS_1S_2. Since, by Lemma 11, *free-vars*$(\sigma_1) \subseteq$ *free-vars*(AS_1S_2), it follows that $\sigma_1T_2 = \sigma_1T_4$.

So, by part 3 of the second use of Lemma 11 above,

$$\sigma_1T_2 \geq_{AS} \tau_1T_3.$$

Write σ_1T_2 in the form $\forall\bar{\alpha}$ **with** $D \,.\, \phi$. Then there exists a substitution $[\bar{\alpha} := \bar{\psi}]$ such that

$$AS \vdash D[\bar{\alpha} := \bar{\psi}]$$

and

$$AS \vdash \phi[\bar{\alpha} := \bar{\psi}] \subseteq \tau_1T_3.$$

Using Lemma 5, we have the following derivation:

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \forall \bar{\alpha}\ \textbf{with}\ D\,.\,\phi$$

by (hypoth),

$$AS \cup \{x : \sigma_1 T_2\} \vdash D[\bar{\alpha} := \bar{\psi}]$$

by above,

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \phi[\bar{\alpha} := \bar{\psi}]$$

by ($\forall$-elim),

$$AS \cup \{x : \sigma_1 T_2\} \vdash \phi[\bar{\alpha} := \bar{\psi}] \subseteq \tau_1 T_3$$

by above,

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \tau_1 T_3$$

by ($\subseteq$),

$$AS \cup \{x : \sigma_1 T_2\} \vdash \tau_1 T_3 \subseteq \rho$$

by part 3 of the second use of induction above, and finally

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \rho$$

by ($\subseteq$). This completes that proof that $AS \cup \{x : \sigma_1 T_2\} \succeq A' \cup \{x : \sigma\}$, which is the same as

$$(AS_1 S_2 \cup \{x : \sigma_1\})T_2 \succeq A' \cup \{x : \sigma\}.$$

Because $A' \vdash e' : \sigma$ and A' has satisfiable constraints, it is easy to see that $A' \cup \{x : \sigma\}$ has satisfiable constraints.

Finally, $AS_1 S_2 \cup \{x : \sigma_1\}$ has acceptable inclusions.

This allows us to apply the induction hypothesis a third time, showing that $(S_3, B_2, \tau_2) = W_{os}(AS_1 S_2 \cup \{x : \sigma_1\}, e'')$ succeeds and that there exists T_5 such that

1. $T_2 = S_3 T_5$, except on new variables of $W_{os}(AS_1 S_2 \cup \{x : \sigma_1\}, e'')$,
2. $AS \cup \{x : \sigma_1 T_2\} \vdash B_2 T_5$, and
3. $AS \cup \{x : \sigma_1 T_2\} \vdash \tau_2 T_5 \subseteq \tau'$.

So $(S_0, B_0, \tau_0) = W_{os}(A, \textbf{let}\ x = e'\ \textbf{in}\ e'')$ succeeds with $S_0 = S_1 S_2 S_3$, $B_0 = B_1' S_3 \cup B_2$, and $\tau_0 = \tau_2$.

Let T be T_5. Then

1. Observe that

$$
\begin{array}{ll}
 & S_0T \\
= & \quad \ll \text{definition} \gg \\
 & S_1S_2S_3T_5 \\
= & \quad \ll \text{part 1 of third use of induction; the new variables} \gg \\
 & \quad \ll \text{of } W_{os}(AS_1S_2 \cup \{x : \sigma_1\}, e'') \text{ don't occur in } S_1S_2 \gg \\
 & S_1S_2T_2, \text{ except on new variables of } W_{os}(AS_1S_2 \cup \{x : \sigma_1\}, e'') \\
= & \quad \ll \text{by the first use of Lemma 11 above} \gg \\
 & S_1T_1 \\
= & \quad \ll \text{by part 1 of first use of induction} \gg \\
 & S, \text{ except on new variables of } W_{os}(A, e').
\end{array}
$$

   So $S_0T = S$, except on new variables of $W_{os}(A, \textbf{let } x = e' \textbf{ in } e'')$.

2. Next, $B_0T = B_1'S_3T_5 \cup B_2T_5 = B_1'T_2 \cup B_2T_5$. By the first use of Lemma 11 above, $AS \vdash B_1'T_2$. By part 2 of the third use of induction above, $AS \cup \{x : \sigma_1T_2\} \vdash B_2T_5$. By Theorem 12, every identifier occurring in B_1 occurs in A. So, since x does not occur in A, x does not occur in B_1. By Lemma 10, every identifier occurring in the constraints of σ_1 occurs in B_1. Hence x does not occur in the constraints of σ_1. By Theorem 12, every identifier occurring in B_2 is overloaded in $AS_1S_2 \cup \{x : \sigma_1\}$ or occurs in some constraint of some assumption in $AS_1S_2 \cup \{x : \sigma_1\}$. Since x does not occur in AS_1S_2 or in the constraints of σ_1, it follows that x does not occur in B_2. Hence Lemma 4 and may be applied to each member of B_2T_5, yielding $AS \vdash B_2T_5$. So $AS \vdash B_0T$.

3. Now, $\tau_0T = \tau_2T_5$ and by part 3 of the third use of induction above and by Lemma 4, we have $AS \vdash \tau_0T \subseteq \tau'$. Also since $A' \vdash \tau' \subseteq \tau$ and since $AS \succeq A'$, it follows that $AS \vdash \tau' \subseteq \tau$. So by (trans), $AS \vdash \tau_0T \subseteq \tau$.

QED

Finally, we get our principal typing result:

Corollary 15 *Let A be an assumption set with satisfiable constraints, acceptable inclusions, and no free type variables. If e is well typed with respect to A, then $(S, B, \tau) = W_{os}(A, e)$ succeeds, $(S', B', \sigma) = close(A, B, \tau)$ succeeds, and the typing $A \vdash e : \sigma$ is principal.*

5 Type Simplification

A typical initial assumption set A_0 is given in Figure 5. Note that A_0 provides a least fixed-point operator *fix*, allowing us to write recursive programs.

Now let *lexicographic* be the following program:

$$A_0 = \left\{ \begin{array}{l} int \subseteq real, \\ if : \forall\alpha.bool \to \alpha \to \alpha \to \alpha, \\ = : \forall\alpha.\alpha \to \alpha \to bool, \\ * : int \to int \to int, \quad * : real \to real \to real, \\ true : bool, \quad false : bool, \\ cons : \forall\alpha.\alpha \to seq(\alpha) \to seq(\alpha), \\ car : \forall\alpha.seq(\alpha) \to \alpha, \quad cdr : \forall\alpha.seq(\alpha) \to seq(\alpha), \\ nil : \forall\alpha.seq(\alpha), \quad null? : \forall\alpha.seq(\alpha) \to bool, \\ \leq : real \to real \to bool, \quad \leq : char \to char \to bool, \\ fix : \forall\alpha.(\alpha \to \alpha) \to \alpha \end{array} \right\}$$

Figure 5: An Example Assumption Set

```
fix λleq.λx.λy.
      if (null? x)
        true
        if (null? y)
          false
          if (=  (car x) (car y))
            (leq (cdr x) (cdr y))
            (≤  (car x) (car y))
```

Function *lexicographic* takes two sequences x and y and tests whether x lexicographically precedes y, using $\leq$ to compare the elements of the sequences.

The computation

$$\begin{array}{l} (S, B, \tau) = W_{os}(A_0, lexicographic); \\ (S', B', \sigma) = close(A_0, B, \tau). \end{array}$$

produces a principal type σ for *lexicographic*. But if we use the simple *close* of Figure 4 we discover, to our horror, that we obtain the principal type

$$\forall\alpha,\gamma,\zeta,\epsilon,\delta,\theta,\eta,\lambda,\kappa,\mu,\nu,\xi,\pi,\rho,\sigma,\iota,\tau,\upsilon,\phi \ \ \textbf{with}$$
$$\left\{ \begin{array}{l} \gamma \subseteq seq(\zeta),\ bool \subseteq \epsilon,\ \delta \subseteq seq(\theta),\ bool \subseteq \eta,\ \gamma \subseteq seq(\lambda),\ \lambda \subseteq \kappa, \\ \delta \subseteq seq(\mu),\ \mu \subseteq \kappa,\ \gamma \subseteq seq(\nu),\ seq(\nu) \subseteq \xi,\ \delta \subseteq seq(\pi), \\ seq(\pi) \subseteq \rho,\ \sigma \subseteq \iota,\ \leq : \tau \to \tau \to bool,\ \gamma \subseteq seq(\upsilon),\ \upsilon \subseteq \tau, \\ \delta \subseteq seq(\phi),\ \phi \subseteq \tau,\ bool \subseteq \iota,\ \iota \subseteq \eta,\ \eta \subseteq \epsilon,\ (\xi \to \rho \to \sigma) \to \\ (\gamma \to \delta \to \epsilon) \subseteq (\alpha \to \alpha) \end{array} \right\} . \alpha$$

Such a type is clearly useless to a programmer, so, as a practical matter, it is essential for *close* to simplify the types that it produces.

We describe the simplification process by showing how it works on *lexicographic*. The call $W_{os}(A_0, lexicographic)$ returns

$$([\beta, o := \xi \to \rho \to \sigma, \rho \to \sigma],\ B,\ \alpha),$$

where

$$B = \left\{ \begin{array}{l} \gamma \subseteq seq(\zeta),\ bool \subseteq bool,\ bool \subseteq \epsilon,\ \delta \subseteq seq(\theta),\ bool \subseteq \eta, \\ \gamma \subseteq seq(\lambda),\ \lambda \subseteq \kappa,\ \delta \subseteq seq(\mu),\ \mu \subseteq \kappa,\ \gamma \subseteq seq(\nu),\ seq(\nu) \subseteq \\ \xi,\ \delta \subseteq seq(\pi),\ seq(\pi) \subseteq \rho,\ \sigma \subseteq \iota,\ \leq : \tau \to \tau \to bool, \\ \gamma \subseteq seq(\upsilon),\ \upsilon \subseteq \tau,\ \delta \subseteq seq(\phi),\ \phi \subseteq \tau,\ bool \subseteq \iota,\ \iota \subseteq \eta, \\ \eta \subseteq \epsilon,\ (\xi \to \rho \to \sigma) \to (\gamma \to \delta \to \epsilon) \subseteq (\alpha \to \alpha) \end{array} \right\}$$

This means that for any instantiation S of the variables in B such that $A_0 \vdash BS$, *lexicographic* has type αS. The problem is that B is so complicated that it is not at all clear what the possible satisfying instantiations are. It turns out, however, that we can make (generally partial) instantiations for some of the variables in B that are optimal, in that they yield a simpler, yet equivalent, type. This is the basic idea behind type simplification.

There are two ways for an instantiation to be optimal. First, an instantiation of some of the variables in B is clearly optimal if it is 'forced', in the sense that those variables can be instantiated in only one way if B is to be satisfied. The second way for an instantiation to be optimal is more subtle. Suppose that there is an instantiation T that makes B no harder to satisfy and that makes the body (in this example, α) no larger. More precisely, suppose that $A_0 \cup B \vdash BT$ and $A_0 \cup B \vdash \alpha T \subseteq \alpha$. Then by using rule ($\subseteq$), BT and αT can produce the same types as can B and α, so the instantiation T is optimal. We now look at how these two kinds of optimal instantiation apply in the case of *lexicographic*.

We begin by discovering a number of forced instantiations. Consider the constraint $\gamma \subseteq seq(\zeta)$ in B. By Lemma 8, this constraint can be satisfied only if γ is instantiated to some type of the form $seq(\chi)$; the partial instantiation $[\gamma := seq(\chi)]$ is forced. There is a procedure, *shape-unifier*, that finds the most general substitution U such that all the inclusions in BU are between types of the same shape.[5] In this case, U is

$$\left[\begin{array}{l} \gamma := seq(\chi), \\ \delta := seq(\psi), \\ \xi := seq(\omega), \\ \rho := seq(\alpha_1), \\ \alpha := seq(\delta_1) \to seq(\gamma_1) \to \beta_1 \end{array} \right]$$

The instantiations in U are all forced by shape considerations; making these forced instantiations produces the constraint set

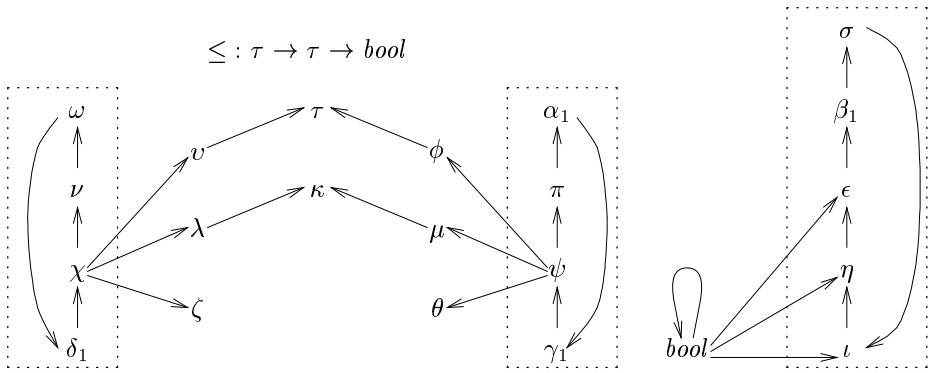
$$\{seq(\chi) \subseteq seq(\zeta), bool \subseteq bool, bool \subseteq \epsilon, seq(\psi) \subseteq seq(\theta), \ldots\}$$

and the body

$$seq(\delta_1) \to seq(\gamma_1) \to \beta_1.$$

We have made progress; we can now see that *lexicographic* is a function that takes two sequences as input and returns some output.

[5] Algorithms for shape unification are given in [14] and in [5].



body: $seq(\delta_1) \to seq(\gamma_1) \to \beta_1$

Figure 6: Atomic Inclusions for *lexicographic*

The new constraint set contains the inclusion $seq(\chi) \subseteq seq(\zeta)$. By our restrictions on subtyping, this constraint is equivalent to the simpler constraint $\chi \subseteq \zeta$. Similarly, any constraint of the form $\tau \to \rho \subseteq \tau' \to \rho'$ is equivalent to the pair of constraints $\tau' \subseteq \tau$ and $\rho \subseteq \rho'$. In this way, we can transform the constraint set into an equivalent set containing only *atomic inclusions*. The result of this transformation is shown graphically in Figure 6, where an inclusion $\tau_1 \subseteq \tau_2$ is denoted by drawing an arrow from τ_1 to τ_2. Below the representation of the constraint set we give the body.

Now notice that the constraint set in Figure 6 contains cycles; for example ω and ν lie on a common cycle. This means that if S is any instantiation that satisfies the constraints, we will have both $A_0 \vdash \omega S \subseteq \nu S$ and $A_0 \vdash \nu S \subseteq \omega S$. But since the inclusion relation is a partial order, it follows that $\omega S = \nu S$. In general, any two types within the same strongly connected component must be instantiated in the same way. If a component contains more than one type constant, then, it is unsatisfiable; if it contains exactly one type constant, then all the variables must be instantiated to that type constant; and if it contains only variables, then we may instantiate all the variables in the component to any chosen variable. We have surrounded the strongly connected components of the constraint set with dotted rectangles in Figure 6; Figure 7 shows the result of collapsing those components and removing any trivial inclusions of the form $\rho \subseteq \rho$ thereby created.

At this point, we are finished making forced instantiations; we turn next to instantiations that are optimal in the second sense described above. These are the monotonicity-based instantiations.

Consider the type $bool \to \alpha$. By rule $((-) \to (+))$, this type is *monotonic*

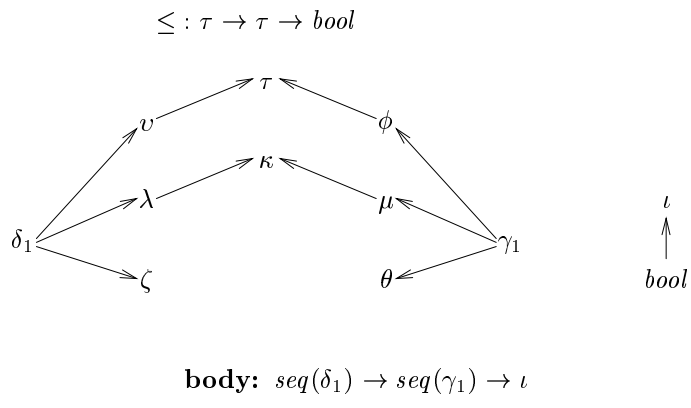


Figure 7: Collapsed Components of *lexicographic*

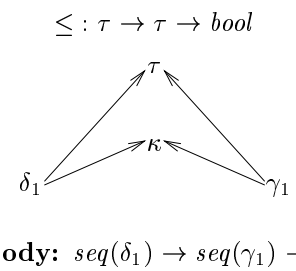


Figure 8: Result of Shrinking ι, υ, λ, ζ, ϕ, μ, and θ

in α: as α grows, a larger type is produced. In contrast, the type $\alpha \to bool$ is *antimonotonic* in α: as α grows, a smaller type is produced. Furthermore, the type $\beta \to \beta$ is both monotonic and antimonotonic in α: changing α has no effect on it. Finally, the type $\alpha \to \alpha$ is neither monotonic nor antimonotonic in α: as α grows, incomparable types are produced.

Refer again to Figure 7. The body $seq(\delta_1) \to (seq(\gamma_1) \to \iota)$ is antimonotonic in δ_1 and γ_1 and monotonic in ι. This means that to make the body smaller, we must boost δ_1 and γ_1 and shrink ι. Notice that ι has just one type smaller than it, namely $bool$. This means that if we instantiate ι to $bool$, all the inclusions involving ι will be satisfied, and ι will be made smaller. Hence the instantiation $[\iota := bool]$ is optimal. The cases of δ_1 and γ_1 are trickier—they both have more than one successor, so it does not appear that they can be boosted. If we boost δ_1 to υ, for example, then the inclusions $\delta_1 \subseteq \lambda$ and $\delta_1 \subseteq \zeta$ may be violated.

The variables υ, λ, ζ, ϕ, μ and θ, however, do have unique predecessors. Since the body is monotonic (as well as antimonotonic) in all of these variables, we may safely shrink them all to their unique predecessors. The result of these instantiations is shown in Figure 8.

Now we are left with a constraint graph in which no node has a unique predecessor or successor. We are still not done, however. Because the body $seq(\delta_1) \to seq(\gamma_1) \to bool$ is both monotonic and antimonotonic in κ, we can instantiate κ arbitrarily, even to an incomparable type, without making the body grow. It happens that the instantiation $[\kappa := \tau]$ satisfies the two inclusions $\delta_1 \subseteq \kappa$ and $\gamma_1 \subseteq \kappa$. Hence we may safely instantiate κ to τ.

Observe that we could have tried instead to instantiate τ to κ, but this would have violated the overloading constraint $\leq : \tau \to \tau \to bool$. This brings up a point not yet mentioned: before performing a monotonicity-based instantiation of a variable, we must check that all overloading constraints involving that variable are satisfied.

At this point, δ_1 and γ_1 have a unique successor, τ, so they may now be boosted. This leaves us with the constraint set $\{\leq : \tau \to \tau \to bool\}$ and the body $seq(\tau) \to seq(\tau) \to bool$. At last the simplification process is finished; we can now apply ($\forall$-intro) to produce the principal type

$$\forall \tau \ \textbf{with} \leq : \tau \to \tau \to bool \,.\, seq(\tau) \to seq(\tau) \to bool,$$

which is the type that one would expect for *lexicographic*.

The complete function *close* is given in Figure 9. Because this definition of *close* satisfies Lemmas 10 and 11, W_{os} remains correct if this new *close* is used.

One important aspect of *close* that has not been mentioned is its use of *transitive reductions* [1]. As we perform monotonicity-based instantiations, we maintain the set of inclusion constraints E_i in reduced form. This provides an efficient implementation of the guard of the **while** loop in the case where a variable α must be shrunk: in this case, the only possible instantiation for α is its unique predecessor in E_i, if it has one. Similarly, if α must be boosted, then its only possible instantiation is its unique successor, if it has one.

6 Satisfiability Checking

We say that a constraint set B is *satisfiable* with respect to an assumption set A if there is a substitution S such that $A \vdash BS$. Unfortunately, this turns out to be an undecidable problem, even in the absence of subtyping [14, 18]. This forces us to impose restrictions on overloading and/or subtyping.

In practice, overloadings come in fairly restricted forms. For example, the overloadings of $\leq$ would typically be

$$\begin{array}{l} \leq : char \to char \to bool, \\ \leq : real \to real \to bool, \\ \leq : \forall \alpha \ \textbf{with} \leq : \alpha \to \alpha \to bool \,. \\ \qquad seq(\alpha) \to seq(\alpha) \to bool \end{array}$$

Overloadings of this form are captured by the following definition.

Definition 16 *We say that x is* overloaded by constructors *in A if the lcg of x in A is of the form $\forall \alpha.\tau$ and if for every assumption $x : \forall \bar{\beta}$ **with** $C \,.\, \rho$ in A,*

$close(A, B, \tau)$:

let A_{ci} be the constant inclusions in A,
  B_i be the inclusions in B,
  B_t be the typings in B;
let $U = \textit{shape-unifier}(\{(\phi, \phi') \mid (\phi \subseteq \phi') \in B_i\})$;
let $C_i = \textit{atomic-inclusions}(B_i U) \cup A_{ci}$,
  $C_t = B_t U$;
let $V = \textit{component-collapser}(C_i)$;
let $S = UV$,
  $D_i = \textit{transitive-reduction}(\textit{nontrivial-inclusions}(C_i V))$,
  $D_t = C_t V$;
$E_i := D_i$; $E_t := D_t$; $\rho := \tau S$;
$\bar{\alpha} :=$ variables free in D_i or D_t or τS but not AS;
while there exist α in $\bar{\alpha}$ and π different from α such that
  $AS \cup (E_i \cup E_t) \vdash (E_i \cup E_t)[\alpha := \pi] \cup \{\rho[\alpha := \pi] \subseteq \rho\}$
do $E_i := \textit{transitive-reduction}(\textit{nontrivial-inclusions}(E_i[\alpha := \pi]))$;
  $E_t := E_t[\alpha := \pi]$;
  $\rho := \rho[\alpha := \pi]$;
  $\bar{\alpha} := \bar{\alpha} - \alpha$
od
let $E = (E_i \cup E_t) - \{\mathcal{C} \mid AS \vdash \mathcal{C}\}$;
let E'' be the set of constraints in E in which some α in $\bar{\alpha}$ occurs;
if AS has no free type variables,
  then if $\textit{satisfiable}(E, AS)$ **then** $E' := \{\}$ **else** *fail*
  else $E' := E$;
return $(S, E', \forall\bar{\alpha}$ **with** E'' . $\rho)$.

Figure 9: Function *close*

- $\rho = \tau[\alpha := \chi(\bar{\beta})]$, *for some type constructor* χ, *and*
- $C = \{x : \tau[\alpha := \beta_i] \mid \beta_i \in \bar{\beta}\}$.

In a type system with overloading but no subtyping, the restriction to overloading by constructors allows the satisfiability problem to be solved efficiently.

On the other hand, for a system with subtyping but no overloading, it is shown in [20] and [9] that testing the satisfiability of a set of *atomic* inclusions is NP-complete. Testing the satisfiability of a set of arbitrary inclusions is shown in [16] to be PSPACE-hard, and [17] gives a DEXPTIME algorithm.[6]

In our system, which has both overloading and subtyping, the restriction to overloading by constructors is enough to make the satisfiability problem decidable [14]. But to get an efficient algorithm, it will be necessary to restrict the subtype relation. This remains an area for future study.

7 Conclusion

This paper gives a clean extension of the Hindley/Milner type system that incorporates overloading and subtyping. We have shown how principal types can be inferred using algorithms W_{os} and *close*. These algorithms have been implemented, and in this section we show the principal types inferred (with respect to the initial assumption set A_0 from Figure 5) for a number of example programs:

- We begin with function *reduce* (sometimes called *foldright*), with definition

$$\begin{array}{l} \mathit{fix}\ \lambda \mathit{reduce}. \\ \quad \lambda f.\lambda a.\lambda l.\ \mathit{if}\ (\mathit{null?}\ l\ ) \\ \qquad\qquad\qquad\quad a \\ \qquad\qquad\qquad\quad (f\ (\mathit{car}\ l)\ (\mathit{reduce}\ f\ a\ (\mathit{cdr}\ l))) \end{array}$$

  The type inferred for *reduce* is

$$\forall \beta_1, \zeta.\ (\beta_1 \to \zeta \to \zeta) \to \zeta \to \mathit{seq}(\beta_1) \to \zeta.$$

  This is the same type that ML would have inferred.

- Next we consider a variant of *reduce*.

$$\begin{array}{l} \mathit{fix}\ \lambda \mathit{reduce}. \\ \quad \lambda f.\lambda a.\lambda l.\ \mathit{if}\ (\mathit{null?}\ l\ ) \\ \qquad\qquad\qquad\quad a \\ \qquad\qquad\qquad\quad \mathit{if}\ (\mathit{null?}\ (\mathit{cdr}\ l)) \\ \qquad\qquad\qquad\qquad (\mathit{car}\ l) \\ \qquad\qquad\qquad\qquad (f\ (\mathit{car}\ l)\ (\mathit{reduce}\ f\ a\ (\mathit{cdr}\ l))) \end{array}$$

[6] Since our function *close* simplifies constraint sets before testing whether they are satisfiable, we actually need to deal only with the case of atomic inclusions.

Now the inferred type is

$$\forall \beta_1, \zeta \ \textbf{with} \ \beta_1 \subseteq \zeta \,.\, (\beta_1 \to \zeta \to \zeta) \to \zeta \to seq(\beta_1) \to \zeta.$$

Here ML would have unified β_1 and ζ.

- Function *max* is

$$\lambda x.\lambda y.\mathit{if}\ (\leq\ y\ x)\ x\ y$$

Its type is

$$\forall \alpha, \beta, \gamma, \delta \ \textbf{with} \ \alpha \subseteq \gamma,\ \alpha \subseteq \delta,\ \beta \subseteq \gamma,\ \beta \subseteq \delta,\ \leq : \delta \to \delta \to \mathit{bool}\,. \\ \alpha \to \beta \to \gamma\,.$$

This surprisingly complicated type cannot, it turns out, be further simplified without assuming more about the subtype relation.

- Finally, function *mergesort* is given in Figure 10. The type inferred for *mergesort* is

$$\forall \sigma_4 \ \textbf{with} \ \leq : \sigma_4 \to \sigma_4 \to \mathit{bool}\,.\, seq(\sigma_4) \to seq(\sigma_4).$$

Also, the type inferred for *split* is

$$\forall \delta_2.seq(\delta_2) \to seq(seq(\delta_2))$$

and the type inferred for *merge* is

$$\forall \iota_1, \theta_1, \eta_1, \kappa \ \textbf{with} \ \theta_1 \subseteq \kappa,\ \theta_1 \subseteq \eta_1,\ \iota_1 \subseteq \kappa,\ \iota_1 \subseteq \eta_1,\ \leq : \kappa \to \kappa \to \mathit{bool}\,. \\ seq(\iota_1) \to seq(\theta_1) \to seq(\eta_1),$$

which is very much like the type of function *max* above.

The fact that the types inferred in these examples are not too complicated suggests that this approach has the potential to be useful in practice.

We conclude by mentioning a few ways in which this work could be extended.

- Efficient methods for testing the satisfiability of constraint sets need to be developed.
- Because our type system can derive a typing in more than one way, the semantic issue of *coherence* [6] should be addressed.
- It would be nice to extend the language to include *record* types, which obey interesting subtyping rules, but which would appear to complicate type simplification.

7.1 Acknowledgements

I am grateful to David Gries and to Dennis Volpano for many helpful discussions of this work.

```
let split=
    fix λsplit.
        λlst. if (null? lst)
                  (cons nil (cons nil nil))
                  if (null? (cdr lst))
                    (cons lst (cons nil nil))
                    let pair=split (cdr (cdr lst)) in
                        (cons (cons (car lst) (car pair))
                              (cons (cons (car (cdr lst)) (car (cdr pair)))
                                    nil)) in
    let merge=
        fix λmerge.
            λlst1.λlst2.
                if (null? lst1)
                  lst2
                  if (null? lst2)
                    lst1
                    if (≤ (car lst1) (car lst2))
                      (cons (car lst1) (merge (cdr lst1) lst2))
                      (cons (car lst2) (merge lst1 (cdr lst2))) in
        fix λmergesort.
            λlst. if (null? lst)
                    nil
                    if (null? (cdr lst))
                      lst
                      let lst1lst2=split lst in
                          merge (mergesort (car lst1lst2))
                                (mergesort (car (cdr lst1lst2)))
```

Figure 10: Example *mergesort*

References

[1] Alfred V. Aho, Michael R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, June 1972.

[2] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis.* PhD thesis, Cornell University, January 1990.

[3] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1989.

[5] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[6] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[7] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[8] Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988.

[9] Patrick Lincoln and John C. Mitchell. Algorithmic aspects of type inference with subtypes. In *19th ACM Symposium on Principles of Programming Languages*, pages 293–304, 1992.

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

[11] John C. Mitchell. Coercion and type inference (summary). In *Eleventh ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[12] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[13] John C. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, 1985.

[14] Geoffrey S. Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, August 1991. Also available as TR 91–1230.

[15] Ryan Stansifer. Type inference with subtypes. In *Fifteenth ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[16] Jerzy Tiuryn. Subtype inequalities. In *IEEE Symposium on Logic in Computer Science*, pages 308–315, 1992.

[17] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 686–701. Springer-Verlag, April 1993.

[18] Dennis M. Volpano and Geoffrey S. Smith. On the complexity of ML typability with overloading. In *Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 15–28. Springer-Verlag, August 1991.

[19] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[20] Mitchell Wand and Patrick O'Keefe. On the complexity of type inference with coercion. In *Conference on Functional Programming Languages and Computer Architecture*, 1989.