Polymorphic Type Inference with Overloading and Subtyping

Geoffrey S. Smith*
Cornell University†

Abstract. We show how the Hindley/Milner polymorphic type system can be extended to incorporate overloading and subtyping, by using constrained quantification. We describe an algorithm for inferring principal types and outline a proof of 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 [5, 7, 2] 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 $\lambda f.\lambda x.f(fx)$ has principal type $\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$; 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 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 $int \rightarrow int \rightarrow int$[1] and $real \rightarrow real \rightarrow real$ (and perhaps others), but it does not have type $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$. So any single typing $* : \sigma$ is either too narrow or too broad. Subtyping cannot be handled in the Hindley/Milner system, because there is no way to express 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*,

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

†Author's current address: Department of Computer Science and Information Systems, The American University, Washington, DC 20016–8116; geoffrey@gabriel.cas.american.edu

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

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 $\alpha \rightarrow int \rightarrow \alpha$, provided that $* : \alpha \rightarrow \alpha \rightarrow \alpha$ and $1 : \alpha$; these types are described by the formula $\forall\alpha$ **with** $* : \alpha \rightarrow \alpha \rightarrow \alpha,\ 1 : \alpha\ .\ \alpha \rightarrow int \rightarrow \alpha$.

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

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 [6] and by Wadler and Blott [14]. 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 [8], Stansifer [12], Fuh and Mishra [3, 4], and Curtis [1]. Mitchell, Stansifer, and Fuh and Mishra consider type inference with subtyping, but their 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.

2 The Type System

The language that we study is the simple *core-ML* of Damas and Milner [2]. 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

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

where χ is an n-ary type constructor. If χ is 0-ary, then the parentheses are omitted. As usual, $\rightarrow$ is taken to be right associative. Types will be denoted by τ, π, or ρ. 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

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n \textbf{ with } C_1, \ldots, C_m \,.\, \tau,$$

where each 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.

Now we 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. $1 : int$) and basic inclusions (e.g. $int \subseteq real$). 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

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

(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 [2], 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]

[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 [14], *local* overloadings complicate the existence of principal typings.

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

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

(→-elim) $$\frac{A \vdash e : \tau \rightarrow \tau' \quad A \vdash e' : \tau}{A \vdash e\,e' : \tau'}$$

(let) $$\frac{A \vdash e : \sigma \quad A \cup \{x : \sigma\} \vdash e' : \tau}{A \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau}$$ (x does not occur in A)

(∀-intro) $$\frac{A \cup C \vdash e : \tau \quad A \vdash C[\bar{\alpha} := \bar{\pi}]}{A \vdash e : \forall \bar{\alpha}\ \mathbf{with}\ C\,.\,\tau}$$ ($\bar{\alpha}$ not free in A)

(∀-elim) $$\frac{A \vdash e : \forall \bar{\alpha}\ \mathbf{with}\ C\,.\,\tau \quad A \vdash C[\bar{\alpha} := \bar{\pi}]}{A \vdash e : \tau[\bar{\alpha} := \bar{\pi}]}$$

($\equiv_\alpha$) $$\frac{A \vdash e : \sigma \quad e \equiv_\alpha e' \quad \sigma \equiv_\alpha \sigma'}{A \vdash e' : \sigma'}$$

($\subseteq$) $$\frac{A \vdash e : \tau \quad A \vdash \tau \subseteq \tau'}{A \vdash e : \tau'}$$

Figure 1: Typing Rules

(hypoth) $A \vdash \tau \subseteq \tau'$, if $(\tau \subseteq \tau') \in A$

(reflex) $A \vdash \tau \subseteq \tau$

(trans) $$\frac{A \vdash \tau \subseteq \tau' \quad A \vdash \tau' \subseteq \tau''}{A \vdash \tau \subseteq \tau''}$$

($(-) \rightarrow (+)$) $$\frac{A \vdash \tau' \subseteq \tau \quad A \vdash \rho \subseteq \rho'}{A \vdash (\tau \rightarrow \rho) \subseteq (\tau' \rightarrow \rho')}$$

($seq(+)$) $$\frac{A \vdash \tau \subseteq \tau'}{A \vdash seq(\tau) \subseteq seq(\tau')}$$

Figure 2: Subtyping Rules

Rules ($\forall$-intro) and ($\forall$-elim) are unusual, since they must deal with constraint sets. These rules are equivalent to rules in [14], 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 [14], is crucial in preventing many nonsensical expressions from being well typed. For example, from the assumptions $+ : int \rightarrow int \rightarrow int$, $+ : real \rightarrow real \rightarrow real$, and $true : bool$, then without the satisfiability condition it would follow that $true + true$ has type

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

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 [10]. For example, $\rightarrow$ is antimonotonic in its first argument and monotonic in its second argument. The name $((-) \rightarrow (+))$ 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.

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 $(\forall \bar{\alpha} \textbf{ with } C \,.\, \tau) \geq_A \tau'$ *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.

We now turn to the problem of inferring principal types.

3 Type Inference

For type inference, it is useful to assume that the initial assumption set has certain properties. In particular, we disallow inclusion assumptions like $int \subseteq (int \rightarrow int)$, 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

- A contains only constant inclusions (i.e. inclusions of the form $c \subseteq d$, where c and d are type constants), and

$W_{os}(A, e)$ 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 $(x : \forall\bar{\alpha}$ 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 $\alpha$ 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 $S_3 = unify(\tau_1 S_2, \alpha \to \beta)$ where $\alpha$ and $\beta$ are new;
   return $(S_1 S_2 S_3, B_1 S_2 S_3 \cup B_2 S_3 \cup \{\tau_2 S_3 \subseteq \alpha S_3\}, \beta S_3)$.

4. $e$ is let $x = e'$ 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 $(S_2, B_1', \sigma_1) = close(AS_1, B_1, \tau_1)$;
   let $(S_3, B_2, \tau_2) = W_{os}(AS_1 S_2 \cup \{x : \sigma_1\}, e'')$;
   return $(S_1 S_2 S_3, B_1' S_3 \cup B_2, \tau_2)$.
```

Figure 3: Algorithm W_{os}

- the transitive closure of the inclusions in A is antisymmetric.

These restrictions imply that only types of the same 'shape' can be related by inclusion, and that the inclusion relation 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 [7, 2]. Given initial assumption set A and expression e, $W_{os}(A, e)$ returns a triple (S, B, τ), such that

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

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.

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

let $\bar{\alpha}$ be the type variables free in $B$ or $\tau$ but not in $A$;
let $C$ be the set of constraints in $B$ in which some $\alpha_i$ 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*

Case 1 of W_{os} makes use of the *least common generalization* (*lcg*) [9] of an overloaded identifier x, as a means of capturing any common structure among the overloadings of x. For example, the *lcg* of $*$ is $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$.

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 outline the proof of correctness of W_{os}; complete proofs can be found in [11]. We begin with some useful properties of our type system.

Lemma 1 *If* $A \vdash e : \sigma$ *then* $AS \vdash e : \sigma S$. *If* $A \vdash \tau \subseteq \tau'$, *then* $AS \vdash \tau S \subseteq \tau' S$.

Let ($\forall$-elim$'$) be the following weakened ($\forall$-elim) rule:

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

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 2 $A \vdash e : \sigma$ *iff* $A \vdash' e : \sigma$.

Now we turn to 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 3 *If $(S, B', \sigma) = close(A, B, \tau)$ succeeds, then for any e, if $A \cup B \vdash e : \tau$ then $AS \cup B' \vdash e : \sigma$.*

Lemma 4 *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.

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

Theorem 5 *If $(S, B, \tau) = W_{os}(A, e)$ succeeds, then $AS \cup B \vdash e : \tau$.*

Now we turn to 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. *$S = S_0 T$, except on new type variables of $W_{os}(A, e)$,*
2. *$AS \vdash B_0 T$, and*
3. *$AS \vdash \tau_0 T \subseteq \tau$.*

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

Definition 3 *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 the following lemma would be true.

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

But the lemma appears to defy a straightforward inductive proof.[3] 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.

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

Theorem 6 *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. *$S = S_0 T$, except on new type variables of $W_{os}(A, e)$,*
2. *$AS \vdash B_0 T$, and*
3. *$AS \vdash \tau_0 T \subseteq \tau$.*

Finally, we get the following principal typing result:

Corollary 7 *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 will contain assumptions such as $int \subseteq real$, $if : \forall\alpha.bool \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$, $\leq\, : real \rightarrow real \rightarrow bool$, $\leq\, : char \rightarrow char \rightarrow bool$, and $car : \forall\alpha.seq(\alpha) \rightarrow \alpha$. Also, we need a typing for a least fixed-point operator *fix*, allowing us to express recursion: $fix : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha$.

Now let *lexicographic* be the following program:

```
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

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

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

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

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 $W_{os}(A_0, \mathit{lexicographic})$ returns

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

where

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

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 $\gamma \subseteq \mathit{seq}(\zeta)$ in B. By our restrictions on subtyping, this constraint can be satisfied only if γ is instantiated to some type of the form $\mathit{seq}(\chi)$; the partial instantiation $[\gamma := \mathit{seq}(\chi)]$ 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.[4] In this case, U is

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

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

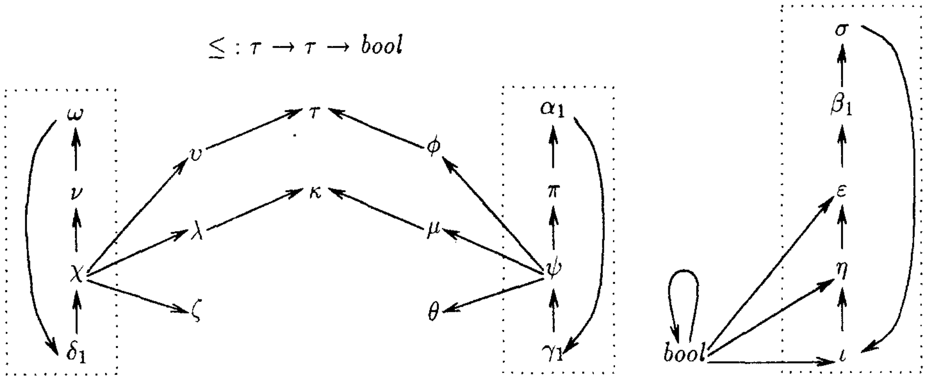
$$\{\mathit{seq}(\chi) \subseteq \mathit{seq}(\zeta), \mathit{bool} \subseteq \mathit{bool}, \mathit{bool} \subseteq \varepsilon, \mathit{seq}(\psi) \subseteq \mathit{seq}(\theta), \ldots\}$$

and the body

$$\mathit{seq}(\delta_1) \rightarrow \mathit{seq}(\gamma_1) \rightarrow \beta_1.$$

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

[4] Such a procedure is given in [4] with the name MATCH.



$\leq : \tau \rightarrow \tau \rightarrow bool$

$$\begin{tikzcd}
\omega \arrow[dddddd, bend right] & & \tau & & \alpha_1 \arrow[dddddd, bend left] & & \sigma \arrow[dddddd, bend left] \\
& \upsilon \arrow[ur] & & \phi \arrow[ul] & & & \beta_1 \arrow[u] \\
\nu \arrow[uu] & & \kappa & & \pi \arrow[uu] & & \varepsilon \arrow[u] \\
& \lambda \arrow[ur] & & \mu \arrow[ul] & & & \\
\chi \arrow[uu] \arrow[uuur] \arrow[ur] \arrow[dr] & & & & \psi \arrow[uu] \arrow[uuul] \arrow[ul] \arrow[dl] & & \eta \arrow[uu] \\
& \zeta & & \theta & & & \\
\delta_1 \arrow[uu] & & & & \gamma_1 \arrow[uu] & bool \arrow[loop above] \arrow[r] \arrow[uur] \arrow[uuuur] & \iota \arrow[uu]
\end{tikzcd}$$

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

Figure 5: 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 $\chi \subseteq \zeta$. 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 5, 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 5 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 5; Figure 6 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 $bool \rightarrow \alpha$. By rule $((-) \rightarrow (+))$, this type is *monotonic* in α: as α grows, a larger type is produced. In contrast, the type $\alpha \rightarrow bool$ is *antimonotonic* in α: as α grows, a smaller type is produced. Furthermore, the type $\beta \rightarrow \beta$ is both monotonic and antimonotonic in α: changing α has no effect on it. Finally, the type $\alpha \rightarrow \alpha$ is neither monotonic nor antimonotonic in α: as α grows, incomparable types are produced.

$\leq : \tau \rightarrow \tau \rightarrow \mathit{bool}$

$$\begin{tikzcd}
 & & & \tau & & & & \\
 & \upsilon \arrow[urr] & & & & \phi \arrow[ull] & & \\
 & & & \kappa & & & & \\
 & \lambda \arrow[urr] & & & & \mu \arrow[ull] & & \iota \\
\delta_1 \arrow[uuur] \arrow[ur] \arrow[dr] & & & & & & \gamma_1 \arrow[uuul] \arrow[ul] \arrow[dl] & \mathit{bool} \arrow[u] \\
 & \zeta & & & & \theta & &
\end{tikzcd}$$

body: $\mathit{seq}(\delta_1) \rightarrow \mathit{seq}(\gamma_1) \rightarrow \iota$

Figure 6: Collapsed Components of *lexicographic*

$\leq : \tau \rightarrow \tau \rightarrow \mathit{bool}$

$$\begin{tikzcd}
 & \tau & \\
 & \kappa & \\
\delta_1 \arrow[uur] \arrow[ur] & & \gamma_1 \arrow[uul] \arrow[ul]
\end{tikzcd}$$

body: $\mathit{seq}(\delta_1) \rightarrow \mathit{seq}(\gamma_1) \rightarrow \mathit{bool}$

Figure 7: Result of Shrinking ι, υ, λ, ζ, ϕ, μ, and θ

Refer again to Figure 6. The body $\mathit{seq}(\delta_1) \rightarrow (\mathit{seq}(\gamma_1) \rightarrow \iota)$ 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 $[\iota := \mathit{bool}]$ 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 7.

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 $\mathit{seq}(\delta_1) \rightarrow \mathit{seq}(\gamma_1) \rightarrow \mathit{bool}$ 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

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

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

Figure 8: Function *close*

have violated the overloading constraint $\leq : \tau \rightarrow \tau \rightarrow bool$. 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

$$\{\leq : \tau \rightarrow \tau \rightarrow bool\}$$

and the body

$$seq(\tau) \rightarrow seq(\tau) \rightarrow bool.$$

At last the simplification process is finished; we can now apply ($\forall$-intro) to produce the principal type

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

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

The complete function *close* is given in Figure 8. One important aspect of *close* that has not been mentioned is its use of *transitive reductions*. 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 [11, 13]. 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

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

Overloadings of this form are captured by the following definition.

Definition 4 *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,*

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

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 [15] that the satisfiability problem is NP-complete.

In our system, which has both overloading and subtyping, the restriction to overloading by constructors is enough to make the satisfiability problem decidable [11]. But to get an efficient algorithm, it will be necessary to restrict the subtype relation.

7 Conclusion

This paper gives a clean extension of the Hindley/Milner type system that incorporates overloading and subtyping. The algorithms in this paper have been implemented and tried on a number of examples, usually producing the expected types. For example, the type inferred for *mergesort* is

$$\forall\alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool\,.\, seq(\alpha) \rightarrow seq(\alpha)$$

This experience gives confidence that this approach has the potential to be useful in practice.

7.1 Acknowledgements

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

References

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

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

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

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

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

[6] 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.

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

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

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

[10] 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.

[11] Geoffrey S. Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, August 1991.

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

[13] 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.

[14] 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.

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