Overloading and Bounded Polymorphism*

Geoffrey S. Smith

TR 89-1054
November 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

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

Overloading and Bounded Polymorphism*

Geoffrey S. Smith
Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853

November 9, 1989

Abstract

We present an extension of the Hindley/Milner polymorphic type system that deals with overloading. The system uses a kind of bounded polymorphic type to describe the nonuniform polymorphism resulting from the use of overloaded identifiers.

We consider the type inference problem for our system and show that it is undecidable. Restrictions are proposed to cope with this limitation.

1 Introduction

In mathematics, the study of a proof of a theorem may reveal that the theorem can be generalized. For example, in analyzing a proof of a theorem about the natural numbers, we might realize that the only property of the natural numbers needed is that they form a well-ordered set. This realization would enable us to generalize our theorem to any well-ordered set.

To exploit this useful principle of generalization, we rewrite our theorems so that they refer to abstract mathematical objects (e.g. well-ordered sets, topological spaces) that have certain properties but whose identity is not fixed. Poincaré describes the process thus:

> ...mathematics is the art of giving the same name to different things...When the language is well-chosen, we are astonished

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

to learn that all the proofs made for a certain object apply immediately to many new objects; there is nothing to change, not even the words, since the names have become the same. [Poi13, p. 375]

In computer science, similarly, the study of a correctness proof of an algorithm may reveal that the algorithm works for many types of objects. For example, an exponentiation function might work on any type whose members can be 'multiplied' by an associative operation $*$ and that contains an identity 1.

This principle of generalization goes by the name of *polymorphism* in computer science [Pri87], except that polymorphic type systems normally do not know anything about correctness proofs: they allow a function to accept inputs of all types that 'look' correct, in the sense that the operations needed are defined on the input types. Also, it is common to encumber the concept of polymorphism with irrelevant implementation considerations—e.g. it is usually demanded that a polymorphic function execute the same machine code, regardless of the types of its inputs [CW85].

Following Poincaré, we can say that the basis for polymorphism is the use of the same name for different objects, allowing programs that use these names to apply to many types of inputs. For example, we allow the identifiers *cons* and $*$ to refer to many different functions, depending on the input type. More precisely, we say that *cons* and $*$ have many types and that they have a meaning corresponding to each of their types:

$$cons : \forall\alpha.\alpha \rightarrow seq(\alpha) \rightarrow seq(\alpha)$$

describes the types of *cons* and

$$* : int \rightarrow int \rightarrow int$$

$$* : real \rightarrow real \rightarrow real$$

$$* : complex \rightarrow complex \rightarrow complex$$

gives some of the types of $*$. Because the types of *cons* are all described by a single type formula, *cons* is said to exhibit *parametric polymorphism.* The types of $*$, in contrast, cannot be uniformly described: they require many type formulas. For this reason $*$ is said to be *overloaded.*

Traditionally, parametric polymorphism has been regarded as more genuine and significant than overloading, which is often referred to as *ad-hoc* polymorphism. (See, for example, [CW85].) But with respect to the generalization principle, which seems to be the true foundation of polymorphism, there is no essential difference.

A key aspect of this work, then, is to elevate overloading to the same status as parametric polymorphism. A difficulty to be overcome is that, when a function inherits polymorphism from the overloaded identifiers that it uses, the function's types will not have a uniform description by a single type formula. For this reason, we use *bounded polymorphic* type formulas to describe implicitly the types of such functions. For example, we would say that the type of the exponentiation function is

$$expon : \forall\alpha \textbf{ with } (* : \alpha \to \alpha \to \alpha, 1 : \alpha) .\ \alpha \to nat \to \alpha$$

to indicate that *expon* has type

$$\alpha \to nat \to \alpha$$

for all choices of α that satisfy the constraints in the **with** clause.

Our type system is fundamentally the same as that proposed by Wadler and Blott in the appendix of [WB89], but was developed largely independently.

The organization of the rest of the paper is as follows. Sections 2, 3, 4, and 5 present the language and its type system. Section 6 contains undecidability results about type checking. Section 7 describes an application of our results to logic. Section 8 discusses type inference.

2 The Language and its Types

In order to study polymorphism in the presence of overloading in as clean a setting as possible, we take as our language the *core-ML* of Damas and Milner [DM82]. Given a set of *identifiers* (x, y, a, $\leq$,...), 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 sets of *type variables* $(\alpha, \beta, \gamma, \ldots)$ and *type constructors* (χ, *int*, *char*, *set*, *seq*,...), we define the τ-types by

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

A type constructor χ may have 0 arguments, in which case the parentheses are omitted. A variable-free τ-type is called a *ground* type.

Next we define the σ-types by

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n \textbf{ with } x_1 : \tau_1, \ldots, x_m : \tau_m \,.\, \tau$$

The α_i are the *quantified variables*, the $x_i : \tau_i$ are the *constraints*, and τ is the *body* of the type. If there are no quantified variables, the $\forall$ is omitted. If there are no constraints, the **with** is omitted. The x_i's in the constraints are required to be overloaded identifiers.

We now establish some notation for substitutions on τ-types. A substitution is a set of simultaneous replacements for type variables:

$$S = \{\alpha_1, \ldots, \alpha_n := \tau_1, \ldots, \tau_n\}$$

We denote the application of substitution S to type τ by τS, and the composition of substitutions S and T by ST. We say that τ is a *substitution instance* of τ' if $\tau' S = \tau$ for some S. We say that τ and τ' are *unifiable* if $\tau S = \tau' S$ for some S.

A less familiar notion is that of the *least common generalization* [Rey70], also known as the *least general predecessor* [McC84], which is the dual to the most general unification. We say that τ is a *least common generalization* of τ_1 and τ_2 if two conditions are met:

1. τ_1 and τ_2 are substitution instances of τ, and
2. τ is a substitution instance of any other type τ' having τ_1 and τ_2 as substitution instances.

3 Type Assumption Sets

An expression is type-checked relative to a finite set A of *type assumptions* of the form $x : \sigma$ which gives types to the free identifiers of the expression. Hence the form of a typing judgement is

$$A \vdash e : \sigma$$

which asserts that from assumptions A it follows that e has type σ.

Our assumption sets are unusual in that they are not limited to at most one assumption per identifier. If an assumption set contains more than one type assumption about an identifier x, then x is said to be *overloaded.* For example, an assumption set might contain

$$\leq : int \rightarrow int \rightarrow bool$$

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

to overload $\leq$. The first assumption defines $\leq$ on integers, and the second says that whenever $\leq$ is defined on a type α, it is also defined on the type $seq(\alpha)$ (presumably using lexicographic ordering). Note that the second assumption gives rise to infinitely many instances of $\leq$.

As a means of capturing any common structure among the type assumptions for an overloaded identifier, it is convenient to compute the least common generalization of the bodies of the assumptions. For example, the (universally quantified) least common generalization of the types of $\leq$ above is

$$\forall\alpha.\alpha \rightarrow \alpha \rightarrow bool$$

which shows that in all of its instances $\leq$ takes two arguments of the same type and returns a *bool*. We will refer to this as the *lcg* of $\leq$.

Suppose we were to add another assumption about $\leq$ to the above list:

$$\leq : \forall\alpha.seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool.$$

The idea might be that any two sequences can be compared by length, regardless of the type of the elements. But consider a program that uses $\leq$ to compare two sequences of integers; which of the two possible meanings is intended? To rule out this sort of ambiguity, we require that in the *initial* assumption set there be at most one way of deriving any type for an overloaded identifier. For convenience, we actually use a property that is somewhat stronger than necessary: we require that the types given to each overloaded identifier be pairwise *nonoverlapping*. We say that σ and σ' *overlap* if (after renaming to eliminate common bound variables) their bodies are unifiable.

In addition, we require that the initial assumption set have no assumptions with free type variables.

4 Typing Rules

Figure 1 gives the typing rules for our language. Recall that the notation A_x denotes the assumption set obtained by deleting from A any assumptions about x.

$$\text{(taut)} \quad A \vdash x : \sigma \qquad \text{if } x : \sigma \in A$$

$$(\to\text{-intro}) \quad \frac{A_x \cup \{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'}$$

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

$$\text{(let)} \quad \frac{\begin{array}{l} A \vdash e : \sigma \\ A_x \cup \{x : \sigma\} \vdash e' : \tau \end{array}}{A \vdash \textbf{let } x = e \textbf{ in } e' : \tau}$$

$$(\forall\text{-intro}) \quad \frac{A \cup C \vdash e : \tau}{A \vdash e : \forall \bar{\alpha} \textbf{ with } C\,.\ \tau} \qquad \bar{\alpha} \text{ not free in } A$$

$$(\forall\text{-elim}) \quad \frac{\begin{array}{l} A \vdash e : \forall \bar{\alpha} \textbf{ with } C\,.\ \tau \\ A \vdash x : \tau'\{\bar{\alpha} := \bar{\tau}''\}, \text{ for each } x : \tau' \text{ in } C \end{array}}{A \vdash e : \tau\{\bar{\alpha} := \bar{\tau}''\}}$$

Figure 1: The Typing Rules

The first four rules are identical to rules in [DM82]; only the $\forall$-intro and $\forall$-elim rules are new. Notice also that the **let** rule follows normal block structure, which means that a program cannot overload an identifier locally. Hence the only overloading in the language is that present in the initial assumption set.

As an example, let LEq α be an abbreviation for $\alpha \to \alpha \to bool$. Let

$$A = \left\{ \begin{array}{l} \leq : \text{LEq } char, \\ \leq : \forall \alpha \textbf{ with } \leq : \text{LEq } \alpha\,.\ \text{LEq } seq(\alpha), \\ c : seq(seq(char)) \end{array} \right\}$$

Also let

$$A' = A_f \cup \{f : \forall \alpha \textbf{ with } \leq : \text{LEq } \alpha\,.\ \alpha \to bool\}$$

Figure 2 gives a derivation of the typing

$$A \vdash \textbf{let } f = \lambda x.x \leq x \textbf{ in } f(c) : bool.$$

1.	$A \cup \{\leq : \text{LEq } \alpha, x : \alpha\}$	$\vdash$	$x : \alpha$ (taut)
2.	"	$\vdash$	$\leq : \alpha \rightarrow \alpha \rightarrow \mathit{bool}$ (taut)
3.	"	$\vdash$	$\leq (x) : \alpha \rightarrow \mathit{bool}$ ($\rightarrow$-elim on 2 and 1)
4.	"	$\vdash$	$x \leq x : \mathit{bool}$ ($\rightarrow$-elim on 3 and 1)
5.	$A \cup \{\leq : \text{LEq } \alpha\}$	$\vdash$	$\lambda x.x \leq x : \alpha \rightarrow \mathit{bool}$ ($\rightarrow$-intro on 4)
6.	A	$\vdash$	$\lambda x.x \leq x : \forall\alpha \ \textbf{with} \leq : \text{LEq } \alpha\,.\ \alpha \rightarrow \mathit{bool}$ ($\forall$-intro on 5)
7.	A'	$\vdash$	$f : \forall\alpha \ \textbf{with} \leq : \text{LEq } \alpha\,.\ \alpha \rightarrow \mathit{bool}$ (taut)
8.	"	$\vdash$	$\leq : \text{LEq } \mathit{char}$ (taut)
9.	"	$\vdash$	$\leq : \forall\alpha \ \textbf{with} \leq : \text{LEq } \alpha\,.\ \text{LEq } \mathit{seq}(\alpha)$ (taut)
10.	"	$\vdash$	$\leq : \text{LEq } \mathit{seq}(\mathit{char})$ ($\forall$-elim on 9 and 8 with $\alpha := \mathit{char}$)
11.	"	$\vdash$	$\leq : \text{LEq } \mathit{seq}(\mathit{seq}(\mathit{char}))$ ($\forall$-elim on 9 and 10 with $\alpha := \mathit{seq}(\mathit{char})$)
12.	"	$\vdash$	$f : \mathit{seq}(\mathit{seq}(\mathit{char})) \rightarrow \mathit{bool}$ ($\forall$-elim on 7 and 11 with $\alpha := \mathit{seq}(\mathit{seq}(\mathit{char}))$)
13.	"	$\vdash$	$c : \mathit{seq}(\mathit{seq}(\mathit{char}))$ (taut)
14.	"	$\vdash$	$f(c) : \mathit{bool}$ ($\rightarrow$-elim on 12 and 13)
15.	A	$\vdash$	$\textbf{let } f = \lambda x.x \leq x \textbf{ in } f(c) : \mathit{bool}$ (let on 6 and 14)

Figure 2: An example type derivation

Observe that whenever a typing $A \vdash e : \sigma$ has been derived, the derivation may be extended using the $\forall$-elim and $\forall$-intro rules. To describe the types that can be obtained by such extensions, we define the instance relation $\geq_A$ on types. We say that $\forall\bar{\beta}$ **with** C' . τ' is an *instance* of $\forall\bar{\alpha}$ **with** C . τ with respect to A, written

$$\forall\bar{\alpha} \ \textbf{with}\ C\ .\ \tau \geq_A \forall\bar{\beta}\ \textbf{with}\ C'\ .\ \tau',$$

if the β_i are not free in A and there is some substitution $S = \{\bar{\alpha} := \bar{\rho}\}$ such that

1. $\tau S = \tau'$ and
2. $A \vdash x : \tau''$ for each $x : \tau''$ in $CS - C'$.

The result of this definition is that $\sigma \geq_A \sigma'$ iff for all e,

$$A \vdash e : \sigma \text{ implies } A \vdash e : \sigma'.$$

5 Well-typed Expressions

When is a program e well-typed with respect to an assumption set A? The obvious condition is that

$$A \vdash e : \sigma, \text{ for some } \sigma.$$

However, this condition is too weak. For example, suppose

$$A = \{+ : \mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int}, + : \mathit{real} \rightarrow \mathit{real} \rightarrow \mathit{real}, c : \mathit{char}, c : \mathit{bool}\}.$$

Then

$$A \vdash c + c : \forall\alpha \ \textbf{with}\ + : \alpha \rightarrow \alpha \rightarrow \alpha, c : \alpha\ .\ \alpha.$$

But, intuitively, $c + c$ should not be well-typed with respect to A, because the constraints in the deduced type are not satisfiable.

So instead we say that e is well-typed with respect to A iff e can be given a τ-type, i.e., iff

$$A \vdash e : \tau, \text{ for some } \tau.$$

A peculiarity of this definition is that if A is as above, then

$$\textbf{let}\ f = c + c\ \textbf{in}\ 17$$

is well-typed with respect to A, because f is never used.

6 Limitations on Type Checking

The ability of bounded polymorphic types to express the fact that an overloaded identifier x has a certain type provided that x itself has some other type is quite powerful, as seen in the example of $\leq$. Unfortunately, it also causes difficulties in type checking.

Theorem 1 *Given a set A of nonoverlapping type assumptions and a ground assumption $x : \tau$, it is undecidable whether*

$$A \vdash x : \tau.$$

Proof: We reduce PCP, the Post Correspondence Problem [HU79], to this problem. We present the reduction by means of an example. Suppose that the PCP instance we are given is:

$$\begin{array}{ll} x_1 = 10 & y_1 = 101 \\ x_2 = 011 & y_2 = 11 \\ x_3 = 101 & y_3 = 011 \end{array}$$

Assume that there are type constants 0, 1, and t_i for all i, $1 \leq i \leq 3$. Let

$$A = \left\{ \begin{array}{l} f : (1 \rightarrow 0) \rightarrow (1 \rightarrow 0 \rightarrow 1) \rightarrow t_1, \\ f : (0 \rightarrow 1 \rightarrow 1) \rightarrow (1 \rightarrow 1) \rightarrow t_2, \\ f : (1 \rightarrow 0 \rightarrow 1) \rightarrow (0 \rightarrow 1 \rightarrow 1) \rightarrow t_3, \\ f : \forall \alpha, \beta, \gamma \textbf{ with } f : \alpha \rightarrow \beta \rightarrow \gamma \,. \\ \qquad (1 \rightarrow 0 \rightarrow \alpha) \rightarrow (1 \rightarrow 0 \rightarrow 1 \rightarrow \beta) \rightarrow (t_1 \rightarrow \gamma), \\ f : \forall \alpha, \beta, \gamma \textbf{ with } f : \alpha \rightarrow \beta \rightarrow \gamma \,. \\ \qquad (0 \rightarrow 1 \rightarrow 1 \rightarrow \alpha) \rightarrow (1 \rightarrow 1 \rightarrow \beta) \rightarrow (t_2 \rightarrow \gamma), \\ f : \forall \alpha, \beta, \gamma \textbf{ with } f : \alpha \rightarrow \beta \rightarrow \gamma \,. \\ \qquad (1 \rightarrow 0 \rightarrow 1 \rightarrow \alpha) \rightarrow (0 \rightarrow 1 \rightarrow 1 \rightarrow \beta) \rightarrow (t_3 \rightarrow \gamma), \\ g : \forall \alpha, \gamma \textbf{ with } f : \alpha \rightarrow \alpha \rightarrow \gamma \,.\ int \end{array} \right\}$$

Now consider whether

$$A \vdash g : int.$$

This will be true iff $f : \alpha \rightarrow \alpha \rightarrow \gamma$ for some α and γ. But the first six rules allow f to have only types of the form $\alpha \rightarrow \beta \rightarrow \gamma$, where α is obtained by concatenating various x_i's and β is obtained by concatenating the corresponding y_i's. (Recall that $\rightarrow$ is right associative.) Hence $g : int$ iff the PCP instance has a solution. (The γ components of the types of f are

included merely to ensure that the assumptions in A are nonoverlapping.)
QED

The rule for g in the above proof seems unreasonable: knowing that we want $g : \mathit{int}$ gives us no clue as to what α and γ should be. Furthermore, there might be many satisfying instantiations of α and γ, which suggests that the meaning of g of type *int* is somehow ambiguous.

These considerations lead us to introduce the concept of a *coupled* type: a type $\forall\bar{\alpha}$ **with** C . τ is *coupled* if every type variable occurring in C occurs also in τ.

If A contains only coupled types, we can verify a ground assumption $x : \tau$ by "working backwards". For example, if

$$A = \{c : \mathit{char}, c : \forall\alpha \ \mathbf{with}\ c : \alpha \ . \ \mathit{seq}(\alpha)\}$$

then

$$\begin{gathered} A \vdash c : \mathit{seq}(\mathit{seq}(\mathit{char})) \\ \Updownarrow \\ A \vdash c : \mathit{seq}(\mathit{char}) \\ \Updownarrow \\ A \vdash c : \mathit{char}. \end{gathered}$$

More formally, we verify a ground assumption $x : \tau$ as follows:

$B := \{x : \tau\}$;
while $B \neq \{\}$ **do**
  choose $y : \tau'$ in B;
  find an assumption $y : \forall\bar{\alpha}$ **with** C . τ'' in A
    such that for some S, $\tau''S = \tau'$;
  if none, then answer 'no';
  $B := (B - \{y : \tau'\}) \cup CS$;
od
answer 'yes'.

Because of the restriction to nonoverlapping and coupled assumptions, there is always a unique assumption to apply next, and B always contains only ground types. Notice, however, that if

$$A = \{c : \forall\alpha \ \mathbf{with}\ c : \mathit{seq}(\alpha) \ . \ \alpha\}$$

then an infinite branch will be generated when we attempt to verify $c : \mathit{char}$, so our procedure is not guaranteed to terminate. Surprisingly, this defect turns out to be unavoidable.

Theorem 2 *Given a set A of coupled, nonoverlapping type assumptions and a ground assumption $h : \tau$, it is undecidable whether*

$$A \vdash h : \tau.$$

Proof: We reduce the Halting Problem [HU79] to this problem.

Given is a Turing Machine $M = (Q, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_H\})$ where

- Q is the set of states,
- $\{0,1\}$ is the input alphabet,
- $\{0,1,B\}$ is the tape alphabet,
- $\delta : Q \times \{0,1,B\} \to Q \times \{0,1,B\} \times \{L,R\}$,
- q_0 is the start state,
- B is the blank symbol,
- q_H is the unique halting state.

The idea is to encode instantaneous descriptions (IDs) of M as types and to define a set of type assumptions A so that

$$A \vdash h : \tau$$

iff τ encodes an ID from which M eventually halts. We need just one identifier h, a binary type constructor $\to$, and a set of type constants $Q \cup \{0,1,B,\varepsilon\}$.

We encode IDs as follows:

$01Bq10$ is encoded as $(B \to 1 \to 0 \to \varepsilon) \to q \to (1 \to 0 \to \varepsilon)$.

In general, for α_1 and α_2 in $\{0,1,B\}^*$,

$\alpha_1 q \alpha_2$ is encoded as $\overline{\alpha_1^{\text{reverse}}} \to q \to \overline{\alpha_2}$,

where if $\alpha = X_1 X_2 \ldots X_n$, $n \geq 0$, then $\overline{\alpha} = X_1 \to X_2 \to \cdots \to X_n \to \varepsilon$.

Next, we observe that M eventually halts from ID iff either

- ID is a halting ID, or
- M eventually halts from the successor of ID.

We include assumptions in A to deal with these two possibilities. First, to deal with halting IDs, we include

$$h : \forall \alpha, \beta . \alpha \rightarrow q_H \rightarrow \beta .$$

Next we want to express "$h : ID$ if h : successor(ID)". Suppose $\delta(q, X) = (q', Y, R)$. We model this by including

$$h : \alpha \rightarrow q \rightarrow (X \rightarrow \beta) \textbf{ if } h : (Y \rightarrow \alpha) \rightarrow q' \rightarrow \beta ,$$

which in our official syntax is

$$h : \forall \alpha, \beta \textbf{ with } h : (Y \rightarrow \alpha) \rightarrow q' \rightarrow \beta . \ \alpha \rightarrow q \rightarrow (X \rightarrow \beta).$$

In addition, if $X = B$ then we also include

$$h : \alpha \rightarrow q \rightarrow \varepsilon \textbf{ if } h : (Y \rightarrow \alpha) \rightarrow q' \rightarrow \varepsilon .$$

Similarly, if $\delta(q, X) = (q', Y, L)$, we include

$$h : (Z \rightarrow \alpha) \rightarrow q \rightarrow (X \rightarrow \beta) \textbf{ if } h : \alpha \rightarrow q' \rightarrow (Z \rightarrow Y \rightarrow \beta),$$

and, if $X = B$,

$$h : (Z \rightarrow \alpha) \rightarrow q \rightarrow \varepsilon \textbf{ if } h : \alpha \rightarrow q' \rightarrow (Z \rightarrow Y \rightarrow \varepsilon).$$

It follows, then, that

$$A \vdash h : \varepsilon \rightarrow q_0 \rightarrow \varepsilon \text{ iff } M \text{ halts on } \varepsilon .$$

All the type assumptions used are coupled and nonoverlapping.
QED

Given these negative results, we need to restrict types further if we are to achieve decidability. A simple way of doing this is to say that a type σ is a *shrinking* type if each constraint in σ has a type formula shorter than the body of σ. If we limit the initial assumption set A to coupled, nonoverlapping, and shrinking type assumptions, then infinite branches cannot arise, and so we can decide whether

$$A \vdash x : \tau$$

for a ground type τ by using the method of working backwards described above.

7 A Logical Digression

Our type assumptions can be rephrased as Horn clauses [Llo84]:

$$x : \forall\alpha \textbf{ with } x : \alpha, y : \alpha \rightarrow \alpha \,.\; set(\alpha)$$

is equivalent to

$$T_x(set(\alpha)) \leftarrow T_x(\alpha) \wedge T_y(\alpha \rightarrow \alpha).$$

So if A^* denotes the translation of A, then

$$A \vdash x : \tau$$

iff

$$A^* \cup \{\text{false} \leftarrow T_x(\tau)\} \text{ is unsatisfiable.}$$

Furthermore, the properties of being coupled and nonoverlapping are applicable in the obvious way to Horn clauses.

Corollary 3 *Satisfiability of coupled, nonoverlapping Horn clauses is undecidable.*

8 Principal Typing and Type Inference

An especially desirable property of our type system is that an expression, if it has any types at all, has a *principal type* that describes the form of all possible types of the expression. More precisely, we say that a type σ is a *principal* type for e with respect to A if $A \vdash e : \sigma$ and if $\sigma \geq_A \sigma'$ for any type σ' such that $A \vdash e : \sigma'$.

It is perhaps surprising that, in spite of the undecidability results of Section 6, the straightforward modification of Milner's algorithm W given in Figure 3 can be used to infer principal types for our language. $W(A, e)$ returns a triple (S, B, τ), such that

$$(A \cup B)S \vdash e : \tau,$$

where set B contains type assumptions describing all the uses of overloaded identifiers in e.

$W(A, e)$ is defined by cases:

1. e is x

   if x is overloaded and the *lcg* of x is $\forall\bar{\alpha}.\tau$,

   return $(\{\}, \{x : \tau\{\bar{\alpha} := \bar{\beta}\}\}, \tau\{\bar{\alpha} := \bar{\beta}\})$ where $\bar{\beta}$ are new

   else if $(x : \forall\alpha_1, \ldots, \alpha_n$ **with** $C\,.\ \tau)\epsilon A$,

   /* Note that n may be 0 */

   return $(\{\}, C\{\bar{\alpha} := \bar{\beta}\}, \tau\{\bar{\alpha} := \bar{\beta}\})$ where $\bar{\beta}$ are new

   else fail.

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

   let $(S, B, \tau) = W(A_x \cup \{x : \alpha\}, e')$ where α is new;

   return $(S, B, \alpha S \to \tau)$.

3. e is $e'e''$

   let $(S, B, \tau) = W(A, e')$;

   let $(S', B', \tau') = W(AS, e'')$;

   let $S'' = \mathit{unify}(\tau S', \tau' \to \alpha)$ where α is new;

   return $(SS'S'', B \cup B', \alpha S'')$.

4. e is **let** $x = e'$ **in** e''

   let $(S, B, \tau) = W(A, e')$;

   let $(\sigma, B') = \mathit{close}(AS, BS, \tau)$;

   let $(S', B'', \tau') = W(A_x S \cup \{x : \sigma\}, e'')$;

   return $(SS', B' \cup B'', \tau')$.

Figure 3: Algorithm W

Function *close* used in case 4 of the algorithm takes as input A, B, and τ such that

$$A \cup B \vdash e : \tau \text{ for some } e$$

and applies the $\forall$-intro rule as much as possible to quantify τ, yielding a quantified type σ. As many of the assumptions in B as possible are moved into the constraints of σ; any leftover assumptions in B are collected into a set B'. Finally, (σ, B') is returned. (We will give a precise description below.)

After calling $W(A, e)$, we obtain a principal type for e as follows: since the initial assumption set A has no free type variables, we may apply *close* to obtain the principal typing

$$A \vdash e : \forall \bar{\alpha} \textbf{ with } BS \,.\, \tau,$$

where $\bar{\alpha}$ are the type variables occurring in BS or in τ.

The reason that our undecidability results do not prevent us from inferring principal types is that principal types are not necessarily very useful, because of the following simple corollary to theorem 1:

Corollary 4 *The instance relation $\geq_A$ is undecidable, even if A is restricted to coupled, nonoverlapping, and shrinking assumptions.*

Proof: Remove the assumption about g from the assumption set A used in the proof of theorem 1. The resulting assumption set, call it A', is coupled, nonoverlapping, and shrinking. Now consider whether

$$\forall \alpha, \gamma \textbf{ with } f : \alpha \to \alpha \to \gamma \,.\, int \geq_{A'} int.$$

QED

In view of this corollary, we see that if the principal type of an expression is uncoupled, then we cannot decide what the types of the expression are. Unfortunately, simple programs can have uncoupled principal types. For example,

$$A' \vdash (\lambda x.22)(\lambda y.f(y)(y)) : \forall \alpha, \gamma \textbf{ with } f : \alpha \to \alpha \to \gamma \,.\, int,$$

where A' is the assumption set from the above corollary.

To deal with this problem, we propose that uncoupled types should be disallowed. To do this, we weaken the $\forall$-intro rule by allowing it to generate only coupled types. The result of this is that function *close* will have to request explicit type information from the programmer whenever it would otherwise return an uncoupled type.

We now give the precise definition of $close(A, B, \tau)$: In the (usual) case where A has no free type variables,

1. Verify and remove all ground assumptions from B, failing if any verification fails.

2. Let $\bar{\alpha}$ be the variables in τ.

3. If any variable β not among the $\bar{\alpha}$ occurs in B, ask the programmer to replace it by some expression $\varphi(\bar{\alpha})$.

4. Return ($\forall\bar{\alpha}$ **with** B' . $\tau, \{\}$), where B' is the rewritten B.

In the general case, where A has free type variables (arising from nested function definitions such as $\lambda x.$let $y = f(x)$ **in** y),

1. Verify and remove all ground assumptions from B, failing if any verification fails.

2. Let $\bar{\alpha}$ be the variables in B or in τ that are not free in A, and such that each constraint $x : \tau'$ in B either has only α_i's free in it or has no α_i's free in it.

3. If any α_i does not occur in τ, ask the programmer to replace it by some expression φ all of whose variables occur in τ.

4. Return ($\forall\bar{\alpha}'$ **with** B' . $\tau, B - B'$), where $\bar{\alpha}'$ is the subset of $\bar{\alpha}$ occurring in τ, and B' is the set of rewritten constraints having α_i's free.

With this modification, we will always infer coupled types. Still there remains a final difficulty—knowing a coupled type for a program is not sufficient to determine whether the program is well-typed. For example, if we infer that

$$A \vdash e : \forall\alpha, \gamma \textbf{ with } f : \alpha \rightarrow \alpha \rightarrow \gamma \,.\; \alpha \rightarrow \gamma,$$

then it is undecidable whether e has any τ-types. A solution to this problem is to require that the programmer assign ground types to α and γ, in accordance with the belief that in the main program all types should be determined.

9 Conclusions

Our system extends the framework of Hindley/Milner polymorphism to incorporate overloading. We see that it is necessary to restrict the types of overloaded identifiers to coupled, shrinking types if type checking is to be decidable. Only partial type inference can be performed; on occasion it is necessary to have some explicit type information from the programmer. We look forward to implementing these ideas in the Polya programming language in order to gain practical experience into how often such explicit type information is required.

10 Acknowledgements

I am indebted to Dennis Volpano, David Gries, and the members of the Polya project for many helpful discussions which contributed greatly to this work.

References

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[DM82] Luis Damas and Robin Milner. Principle type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Lei83] Daniel Leivant. Polymorphic type inference. In *10th ACM Symposium on Principles of Programming Languages*, pages 88–98, Austin, Texas, 1983.

[Llo84] John Wylie Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin/New York, 1984.

[McC84] Nancy McCracken. The typechecking of programs with implicit type structure. In *Semantics of Data Types*, pages 301–315. Lecture Notes in Computer Science 173, 1984.

[Poi13] Henri Poincaré. *The Foundations of Science.* The Science Press, Lancaster, Pennsylvania, 1913. Authorized English translation by G. B. Halsted.

[Pri87] Jan F. Prins. *Partial Implementations in Program Derivation.* PhD thesis, Cornell University, August 1987.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[WB89] 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.