Seminar Types and Programming Languages

WS 02/03

Type Reconstruction

Sven Woop

Abstract

As standard type checking depends on type annotation (all lambda abstractions need to be type annotated) we will show an algorithm that allows typechecking without any type annotation. This *type reconstruction algorithm* is capable of calculating a *principal type* for a term in which some or all of these type annotations are omited. We only consider type reconstruction for simple types and not for records as example.

This paper is a summary of the chapter 22 of the book *Types and Programming Languages* by Benjamin C. Pierce. It was created as the draft of the Seminar "Types and Programming Languages (WS02/03)" at the Programming Systems Lab of Prof. Smolka at the Saarland University.

April 28, 2003
set with $\LaTeX\,2_\varepsilon$

email:
woop@ps.uni-sb.de

Contents

1 Simple Typed Lambda Calculus

We assume that you are familiar with the simple typed lambda calculus. In the following we will use a lambda calculus with booleans, natural numbers, a let and if-contruct. The let construct will only be used in the end where we discuss let polymorphism. We have alambda and let with and without type annotation. Now we will give a short grammar describing the notation we use the following chapters.

Let Var be an arbitrary infinite countable set of variables then we define the set of terms the following:

$t, t_1, ... :=$	$x \in Var$	(Variables)
	$\mid t_1\ t_2$	(Application)
	$\mid \lambda x : T.\ t$	(Abstraction)
	$\mid \lambda x.\ t$	(Abstraction without type annotation)
	$\mid$ true $\mid$ false	(Booleans)
	$\mid$ **if** t_1 **then** t_2 **else** t_3	(If)
	$\mid 0 \mid 1 \mid ...$	(Natural Numbers)
	$\mid$ succ $\mid$ pred $\mid$ iszero	(Operations on Natural Numbers)
	$\mid$ **let** x : $T_1 = t_1$ **in** t_2 **end**	(Let)
	$\mid$ **let** $x = t_1$ **in** t_2 **end**	(Let without type annotation)

Let $TyVar$ be an arbitrary infinite countable set of type variables, then we define the set $Types$ of types as follows:

$T, T_1, ... :=$	$X \in TyVar$	(Type Variable)
	$\mid T_1 \rightarrow T_2$	(Functional Type)
	$\mid Bool$	(Booleans)
	$\mid Nat$	(Natural Numbers)

We are not interested in the operational semantics, but only in the typing rules. Therefore we present them:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{Ty-Var}$$

$$\frac{\Gamma \vdash t_1 : T_2 \rightarrow T_3 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_3}\ \text{Ty-App}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \rightarrow T_2}\ \text{Ty-Abs}$$

$$\Gamma \vdash true : Bool \quad \Gamma \vdash false : Bool \quad \text{Ty-Bool}$$

$$\frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : T}\ \text{Ty-If}$$

$$\Gamma \vdash 0 : Nat \quad \Gamma \vdash 1 : Nat \quad \dots \quad \text{Ty-Nat}$$

$$\frac{\Gamma \vdash t_1 : Nat}{\Gamma \vdash \text{iszero } t_1 : Bool}$$

$$\frac{\Gamma \vdash t_1 : Nat}{\Gamma \vdash \text{succ } t_1 : Nat}$$

$$\frac{\Gamma \vdash t_1 : Nat}{\Gamma \vdash \text{pred } t_1 : Nat}$$

$$\frac{\Gamma, x : T_1 \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \textbf{let } x : T_1 = t_1 \textbf{ in } t_2 \textbf{ end} : T_2} \quad \text{Ty-Let}$$

Notice that the standard typing rules can not handle the lambda and let contruct without type annotation. For the let construct this would in principle be possible, but without the possibility of recursive definitions.

2 Type Variables and Substitution

In the previous chapter we assumed that there is a set of *type variables.* These variables are just placeholders for some particular types whose exact identities we do not care about. We analyse the question whether a term gets welltyped if we instantiate the type variables with some other type. More precisely we apply a substitution $\sigma \in TyVar \to Types$ to the type annotations of a term t. Such a substitution is called a *type substitution.* Applying this mapping to a particular type T we obtain an instance σT. For example, if we apply the substitution $\sigma = [X := Bool]$ to the type $X \to X$ we obtain $\sigma(X \to X) = Bool \to Bool$. The application of a substitution to a type is defined the obvious way:

$$\begin{array}{rll} \sigma(X) = & T & \text{if } (X,T) \in \sigma \\ \sigma(X) = & X & \text{if } (X,T) \notin \sigma \\ \sigma(Nat) = & Nat & \\ \sigma(Bool) = & Bool & \\ \sigma(T_1 \to T_2) = & \sigma T_1 \to \sigma T_2 & \end{array}$$

As we have no construct in our language that *binds* a type variable we need not to care about *variable capturing.* We extend the substitution pointwise to variable contexts:

$$\sigma(x_1 : T_1, ..., x_n : T_n) \quad = \quad (x_1 : \sigma T_1, ..., x_n : \sigma T_n)$$

Similarly a substitution is applied to a term by applying it to all type annotations in it. A very important property of the type substitution is that every well typed term t is also well typed if we apply any type assignment to it. The following theorem makes this more precise.

Theorem 2.1. [Preservation of typing under type substitution]: If σ is any type substitution and $\Gamma \vdash t : T$ then we also have $\sigma\Gamma \vdash \sigma t : \sigma T$.
Proof: Straightforward induction on typing derivations.

3 Two views of Type Variables

Assume that t is a term containing type variables and Γ is an associated context than we can ask two different questions:

- Do we have $\sigma\Gamma \vdash \sigma t : T$ for all σ and some T.
- Can we find a σ and a T such that $\sigma\Gamma \vdash \sigma t : T$.

The theorem 2.1 says us that each well typed term has the first property. For example, the term

$$\lambda f : X \to X.\lambda a : X.f(f(a))$$

has type $(X \to X) \to X \to X$ and whenever we replace X by a concrete type T the instance

$$\lambda f : T \to T.\lambda a : T.f(f(a))$$

is well typed. As you see this first property directly leads to *parametric polymorphism*, where each type variable encodes the fact that a term can be used in many concrete contexts with different types. Later we will say a little bit more about parametric polymorphism.
A term that fulfills the second property has not necessarily to be well typed. The term

$$\lambda f : Y.\lambda a : X.f(f(a))$$

is not typable as it stands, but if we replace Y by $Nat \to Nat$ and X by Nat, we obtain

$$\lambda f : Nat \to Nat.\lambda a : Nat.f(f(a))$$

of type $(Nat \to Nat) \to Nat \to Nat$. We can also replace Y by $X \to X$ and obtain the term

$$\lambda f : X \to X.\lambda a : X.f(f(a))$$

which is welltyped and contains type variables. This last term is a *most general* instance of $\lambda f : Y.\lambda a : X.f(f(a))$ in the sense that it makes the smallest commitment about the values of type variables that yields a welltyped term.
Looking for valid instantiations of type variables leads to the idea of *type reconstruction* which is sometimes also called *type inference*. The idea behind this is that the compiler fills in the type information that the programmer left out. Of course may the programmer leave out all type information and write just in the syntax of untyped lambda calculus.

Definition 3.1. Let Γ be a context and t a term. A solution for (Γ, t) is a pair (σ, T) such that $\sigma\Gamma \vdash \sigma t : T$.

4 Constraint-Based Typing

Now we present an algorithm that given a pair (Γ, t) calculates a set of constraints - equations between type expressions - that must be satisfied by any solution for (Γ, t). The intuition behind this algorithm is just the same as the standard type checking algorithm. The only difference is that instead of directly checking constraints between subtypes we record them for later consideration. The reason why we are not able to check directly all constraints the typing rules do, is that we don't know the typ assignments (we first have to compute them). For example, if we have an application $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$. Instead of checking that t_1 has the form $T_2 \to T_{12}$ and returning T_{12} as the type of the application we do the following. We introduce a *fresh type variable* X and record the constraint $T_1 = T_2 \to X$ and return X as the type of the application.

Definition 4.1. A *constraint set* C is a set of equations $S = T$. A substitution σ is said to unify an equation $S = T$, if σS and σT are identical. The substitution σ *unifies* (or *satisfies*) C if it unifies every equation in C.

Definition 4.2. The following inference rules define the *constraint typing relation* $\Gamma \vdash t : T \mid C$. Informally $\Gamma \vdash t : T \mid C$ can be read as "The term t has type T under assumptions Γ whenever the constraints C are satisfied."
Let X be a fresh type variable:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid \emptyset} \quad \text{CT-Var}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid C}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2 \mid C} \quad \text{CT-Abs}$$

$$\frac{\Gamma, x : X \vdash t_2 : T_2 \mid C}{\Gamma \vdash \lambda x.t_2 : X \to T_2 \mid C} \quad \text{CT-Abs'}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma \vdash t_2 : T_2 \mid C_2}{\Gamma \vdash t_1\ t_2 : X \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}} \quad \text{CT-App}$$

$$\Gamma \vdash \mathit{true} : \mathit{Bool} \quad \Gamma \vdash \mathit{false} : \mathit{Bool} \quad \text{CT-Bool}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma \vdash t_2 : T_2 \mid C_2 \quad \Gamma \vdash t_3 : T_3 \mid C_3}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : T_2 \mid C_1 \cup C_2 \cup C_3 \cup \{T_1 = \mathit{Bool}, T_2 = T_3\}} \quad \text{CT-If}$$

$$\Gamma \vdash 0 : \mathit{Nat} \quad \Gamma \vdash 1 : \mathit{Nat} \quad \ldots \quad \text{CT-Nat}$$

$$\frac{\Gamma \vdash t : T \mid C}{\Gamma \vdash \text{iszero}\ t : \mathit{Bool} \mid C \cup \{T = \mathit{Nat}\}}$$

$$\frac{\Gamma \vdash t : T \mid C}{\Gamma \vdash \text{succ}\ t : \mathit{Nat} \mid C \cup \{T = \mathit{Nat}\}}$$

$$\frac{\Gamma \vdash t : T \mid C}{\Gamma \vdash \text{pred}\ t : \mathit{Nat} \mid C \cup \{T = \mathit{Nat}\}}$$

It is important, that the type variable X choosen in rule CT-Abs' is a fresh variable. This means that the variable occured nowhere else. The rule for the let construct will be shown later as it is a little bit more complicated.
When read from bottom to top, the constraint typing rules determine a straightforward procedure that, given Γ and t, calculates T and C such that $\Gamma \vdash t : T|C$. In contrast to the normal typing algorithm this one never fails, in the sense that for every Γ and t there are always some T and C such that $\Gamma \vdash t : T \mid C$. Moreover T and C are uniquely determined by Γ and t (modulo renaming of introduced fresh type variables).

The idea of the contraint typing rules is that given a term t and a context Γ, we can check whether t is typable under Γ by first collecting the constraints C that must be satisfied by any solution. The algorithm yields us an result type S which may share type variables with C. To find solutions we now look for substitutions σ that satisfy C. For each such σ, the type σS is a possible type of t. If we find no such substitutions we are sure, that t cannot be instantiated in such a way that it es well typed.
For example, the constraint set generated by the algorithm for the term $t = \lambda x : X \to Y.x(0)$ is $\{Nat \to Z = X \to Y\}$ and the associated result type is $(X \to Y) \to Z$. The substitution $\sigma = [X := Nat, Z := Bool, Y := Bool]$ makes the equation $Nat \to Z = X \to Y$ into an identity, so we know that $\sigma((X \to Y) \to Z) = (Nat \to Bool) \to Bool$ is a possible type for t. This is formalized by the following definition:

Definition 4.3. Suppose that $\Gamma \vdash t : S \mid C$. A solution for (Γ, t, S, C) is a pair (σ, T) such that σ satisfies C and $\sigma S = T$.

Example 4.4. Before we continue we will do a little example. We will apply the contraint typing rules to the term $\lambda x.\ \lambda y.\ \lambda z.\ (x\ z)\ (y\ z)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash x : X \mid \emptyset \qquad \Gamma' \vdash z : Z \mid \emptyset}{\Gamma' \vdash (x\ z) : X_1 \mid C_1 = \{X = Z \to X_1\}} \qquad \dfrac{\Gamma' \vdash y : Y \mid \emptyset \qquad \Gamma' \vdash z : Z \mid \emptyset}{\Gamma' \vdash (y\ z) : X_2 \mid C_2 = \{Y = Z \to X_2\}}}{\Gamma' = \Gamma, x : X, y : Y, z : Z \vdash (x\ z)\ (y\ z) : X_3 \mid C = C_1 \cup C_2 \cup \{X_1 = X_2 \to X_3\}}}{\Gamma, x : X, y : Y \vdash \lambda z.\ (x\ z)\ (y\ z) : Z \to X_3 \mid C}}{\Gamma, x : X \vdash \lambda y.\ \lambda z.\ (x\ z)\ (y\ z) : Y \to Z \to X_3 \mid C}}{\Gamma \vdash \lambda x.\ \lambda y.\ \lambda z.\ (x\ z)\ (y\ z) : X \to Y \to Z \to X_3 \mid C}$$

We derived the set $C = \{X = Z \to X_1, Y = Z \to X_2, X_1 = X_2 \to X_3\}$ of constraints and the type $S = X \to Y \to Z \to X_3$. As you see the type S shares type variables with the constraints C.

As you can see there are two different ways of finding a solution for a pair (Γ, t). The first one is the formal definition 3.1. This means to find a substitution σ and a type T such that σt is welltyped of type T.
The other way is more algorithmic. First apply the constraint typing rules and then find a solution for these according to definition 4.3. The last part of finding a solution to this set of constraint is described later.
It is clear that the solutions according to definition 3.1 are the most general ones. This means that each possible typing is a solution. As our algorithm has to calculate all these solutions and no more we have to show the equivalence

between both caracterizations. More formal we have to show that each solution to (Γ, t) is also a solution to (Γ, t, S, C) and vice versa. The proof is given in section 22.3.5 and 22.3.7 in Pierce's book and we will only present the theorems to be proved.

Theorem 4.5. : Suppose that $\Gamma \vdash t : S | C$. If (σ, T) is a solution for (Γ, t, S, C), then it is also a solution for (Γ, t).

Theorem 4.6. : Suppose $\Gamma \vdash t : S \mid C$. If (σ, T) is a solution for (Γ, t) then there is a solution (σ', T) for (Γ, t, S, C) such that $\sigma'|_{dom(\Gamma)} = \sigma$.

5 Unification

To calculate solutions to the constraint sets, we use the idea due to Hindley (1969), Milner (1978) and Robinson (1971) of *unification*. A fundamental property of the unification algorithm is, that all solutions can be represented in one most general solution as described later.

The following algorithm is a simple unification algorithm which is specialized to our needs.

```
1.  unify(C) = if C = $\emptyset$ then []
2.             else let { S = T } $\cup$ C' = C in
3.                 if S = T then
4.                     unify(C')
5.                 else if X = T and X $\notin$ FV(T) then
6.                     unify($[X := T]$C') $\circ$ $[X := oT]$
7.                 else if S = X and $X \notin FV(S)$ then
8.                     unify($[X := S]$C') $\circ$ $[X := S]$
9.                 else if $S = S_1 \to S_2$ and $T = T_1 \to T_2$ then
10.                    unify($C' \cup \{S_1 = T_1, S_2 = T_2\}$)
11.                else
12.                    fail
```

The notation in the line 2 means, that we take a constraint $S = T$ out of the set C of remaining constraints. The side conditions $X \notin FV(T)$ and $X \notin FV(S)$ in the 5th and 7th line are known as the *occur check*. Their effect is to prevent the algorithm from generating a solution involving a cyclic substitution like $X := (X \to X)$, which makes no sense if we are taking about finite type expressions. If we expand our language to include *infinity* type expressions, i.e. recursive types, then the occur check can be omitted.

Definition 5.1. A substitution σ is *less specific* (or *more general*) than a substitution σ', written $\sigma \sqsubseteq \sigma'$, if $\sigma' = \gamma \circ \sigma$ for some substitution γ.

Definition 5.2. A *principal unifier* (or sometimes *most general unifier*) for a constraint set C is a substitution σ that satisfies C and such that $\sigma \sqsubseteq \sigma'$ for every substitution σ' satisfying C.

Theorem 5.3. The unification algorithm `unify` always terminates, failing when given a nonunifiable constraint set as input and otherwise returning a principal unifier.

Proof. See section 22.4.5 in Pierce's book. □

Note that theorem 5.3 guarantees the existence of a principal unifier for a constraint set C if C can be unified.

6 Principal Types

We remarked above that if there is some way to instantiate the type variables in a term so that it becomes typable, then there is a *most general* or *principal* way of doing so.

Definition 6.1. A *principal solution* for (Γ, t, S, C) is a solution (σ, T) such that whenever (σ', T') is also a solution for (Γ, t, S, C) we have $\sigma \sqsubseteq \sigma'$. When (σ, T) is a principal solution, we call T a *principal type* of t under Γ.

Theorem 6.2. If (Γ, t, S, C) has any solution, then it has a principal one. The unification algorithm can be used to determine whether (Γ, t, S, C) has a solution and if so to calculate a principal one.

Proof. By definition of a solution for (Γ, t, S, C) and the properties of unification. □

Example 6.3. We continue the last example of $\lambda x.\ \lambda y.\ \lambda z.\ (x\ z)\ (y\ z)$ by solving the achieved set $C = \{X = Z \to X_1, Y = Z \to X_2, X_1 = X_2 \to X_3\}$ of constraints by unification to obtain the most principal type. Applying the unification algorithm yields $\sigma = \{X := Z \to X_2 \to X_3, Y := Z \to X_2, X_1 := X_2 \to X_3\}$ as most general unifier for C. The principal type for the term is then $\sigma S = (Z \to X_2 \to X_3) \to (Z \to X_2) \to Z \to X_3$.

7 Let-Polymorphism

The term *polymorphism* refers to a range of language mechanisms that allow a single part of a program to be used with different types in different contexts. The type reconstruction algorithm shown above can be generalized to provide a simple form of polymorphism known as *let-polymorphism*.
The motivation for let-polymorphism arises from examples like the following. Suppose we define and use a simple function *double*, which applies its first argument twice to its second:

let double $= \lambda f : Nat \to Nat.\ \lambda a : Nat.\ f(f(a))$
in double $(\lambda x : Nat.\ x + 1)\ 0$ **end**

As we want to apply `double` to a function of type $Nat \to Nat$ we choose type annotations that gives it the type $(Nat \to Nat) \to (Nat \to Nat)$. We can alternatively define `double` so that it can be used to double a boolean function, but we *cannot* use the same `double` function with both booleans and numbers, although the function body is nearly the same (except the type annotations). Even the following does not work:

let double $= \lambda f : X \to X.\ \lambda a : X.\ f(f(a))$
in double $(\lambda x : Nat.\ x + 1)\ 0;$
  double $(\lambda x : Bool.\ x)\ true$
end

The reason is, that the first application of the double function introduces the constraint $X \to X = Nat \to Nat$ and the second one the constraint $X \to X = Bool \to Bool$. But these constraints are unsatisfiable and therefore the

whole program is untypable.
The problem is, that in both applications of `double` the type variable X is the same. Therefore X is first bound to the type Nat and then to $Bool$ which doesn't work.
The most popular solution to this problem is to introduce a typescheme at the definition of the `double` function.

```
let double = $\lambda f : X.\, \lambda a : Y.\, f(f(a))$
in double $(\lambda x : Nat.\, x + 1)\ 0$;
   double $(\lambda x : Bool.\, x)\ true$
end
```

In this example the function `double` gets the following type scheme: $(\{X, Y, Z, U\}, U, \{X = Y \to Z, X = Z \to U\})$. As you can see the typescheme is a tuple of the type variables which are introduced by the constraint typing rules, the type of the term and the constraints.
If we access the `double` function first we instantiate the variables X, Y, Z, U to new instances X', Y', Z', U' in the type and the constraints. Accessing it a second time we use other instances, such that the program above gets typable.
The following rules formalize the use of type schemes. Let be $X_1, ..., X_n$ the free type variables of T and C that do not occur in Γ (the ones that are introduced by the constraint typing rules) and the $X, X'_1, ..., X'_n$ be fresh type variables:

$$\frac{\Gamma, x : T_x \vdash t_1 : T \mid C \quad \Gamma, x : (\{X_1, ..., X_n\}, T, C \cup \{T_x = T\}) \vdash t_2 : T' \mid C'}{\Gamma \vdash \textbf{let } x : T_x = t_1 \textbf{ in } t_2 \textbf{ end} : T' \mid C'}$$

$$\frac{\Gamma, x : X \vdash t_1 : T \mid C \quad \Gamma, x : (\{X, X_1, ..., X_n\}, T, C \cup \{X = T\}) \vdash t_2 : T' \mid C'}{\Gamma \vdash \textbf{let } x = t_1 \textbf{ in } t_2 \textbf{ end} : T' \mid C'}$$

$$\frac{x : (\{X_1, ..., X_n\}, T, C) \in \Gamma \quad \sigma = [X_1 := X'_1, ..., X_n := X'_n]}{\Gamma \vdash x : \sigma T \mid \sigma C}$$

As you see the second rule introduces new instances of the type variables $X_1, ..., X_n$ and instantiates the type T as well as the constraints C.

In practise this algorithm is almost linear in the size of the input program. It therefore came as a significant surprise when Kfoury, Tiuryn and Urzyczyn (1990) and independently Marison (1990) showed that its worst-case complexity is still exponential. The example they constrcuted involves using deeply nested sequences of lets in the right-hand side of other lets to build expressions whose types grow double exponentionally larger than the expressions themselves. The following programm for example is well typed but it takes a very long time to typecheck it:

```
let $f_0$ = λ x. (x,x) in
    let $f_1$ = λ x. $f_0(f_0 x)$ in
        let $f_2$ = λ x. $f_1(f_1 x)$ in
            let $f_3$ = λ x. $f_2(f_2 x)$ in
                let $f_4$ = λ x. $f_3(f_3 x)$ in
                    let $f_5$ = λ x. $f_4(f_4 x)$ in
                        $f_5$ (λ x.x)
end end end end end end
```

The reason why the typechecking of this program is exponential is the following. The typescheme of the function f_1 has more than 2 constraints as both application introduce exactly one constraint. In the definition of f_n the constraints of f_{n-1} are always instantiated two times and we get another 2 constraints from the 2 applications. It yields that the number of constraints in level n is greater then two times the number of constraints in level $n-1$. As a consequence the constraint typing rules generate exponentially many constraints. As the unification algorithm is linear in the number of constraints, the whole typechecking is exponential.

The above rule for the let construct has one simple error we still have to fix. Consider the following program (using references to generate side effects):

```
let r = ref (λx : X. x)
in r := (λx : Nat. x + 1);
   (!r) true
end
```

The program first allocates a reference for a function of type $X \to X$. Then a function of type $Nat \to Nat$ is saved under the reference. But in the next line we apply this function to a boolean value which is not safe. The program is welltyped, as we introduce for both occurences of r different type variables.
The reason why this error occurs is that our rules don't care about side effects, like references. To handle this problem we simple introduce typschemes *only for values* (e.g. functions). In the above example we had not introduced a typscheme as ref $(\lambda x : x)$ is no value, as it can be evaluated further. Doing so the above program is not well-typed.
This concept of value restriction solves our problem with type safety, at some cost in expressiveness. We can no longer write programs which in the right-hand side of let expressions we can both, perform some interesting computation and be assigned a polymorphic type scheme. What is surprising is that this restriction makes hardly any difference in practice.

8 Notes

Notions of principal types for the lambda-calculus go back at least to the work of Curry in the 1950s (Curry and Feys, 1958). An algorithm for calculating principal types based on Curry's ideas was given by Hindley (1969); similar algorithms were discovered independently by Morris (1968) and Milner (1978). In the world of propositional logic, the ideas go back still further, perhaps to Tarski in the 1920s. Additional historical remarks on principal types can be found in Hindley (1997).
Unification (Robinson, 1971) is fundamental to many areas of computer science. Thorough instructions can be found, for example, in Baader and Nipkow (1998), Baader and Siekmann (1994), and Lassez and Plotkin (1991).
ML-style let-polymorphism was first described by Miler (1978). A number of type reconstruction algorithms have been proposed, notably the classic *Algorithm W* of Damas and Milner (1982).
Principal types should not be confused with the similar notion of *principal typings.* The difference is that, when we calculate principal types, the context Γ

and term t are considered as inputs to the algorithm, while the principal type T is the output. An algorithm for calculating principal typings takes just t as input and yields both Γ and T as outputs, which means also to calculate the minimal assumptions about the types of the free variables in t.
Principal typings are useful in supporting separate compilation and smartest recompilation, performing incremental type inference, and pinpointing type errors. Unfortunately many languages, in particular ML, have principal types but not principal typings, see Jim (1996).
Extending type reconstruction to handle *recursive types* has been shown not to pose significant difficulties (Huet, 1975, 1976). The only difference from the algorithm presented in this paper appears in the definition of unification, where we omit the *occur check.* If we disable the occur check and use a graph representation, like done in high performance unifiers, we can also compute regular solutions which correspond to recursive types.

References

[1] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[2] Haskell B. Curry and Robert Feys. *Combinatory Logic, Volume 1.* 1958, Second Edition 1968.

[3] J. Roger Hindley. *The principal type-scheme of an object in combinatory logic.* 1969.

[4] J. Roger Hindley. *Basic Simple Type Theory.* Cambridge University Press Press, 1986.

[5] Luis Damas and Robin Milner. *Principal type schemes for functional programs.* Springer LNCS 431, 1990.

[6] Trevor Jim. *What are principal typings and what are they good for?* 1996.

[7] Pawel Urzyczyn Kfoury Assaf J., Jerzy Tiuryn. *ML typability is DEXPTIME-complete.* 1996.