Polymorphism and Type Inference

Brigitte Pientka

April 14, 2007

So far we have mainly been concerned with assigning a type to a given piece of code or checking that an expression has a given type. In these notes we will explain how to infer a type for a given expression. We will not only be able to infer some type for a given expression, but the principal type, i.e. the most general type.

Before we discuss type inference more deeply, we will first concentrate on the role of polymorphic types and type variables. For example, we can check that the function

$$\mathsf{double} = \mathsf{fn}\ f \Rightarrow \mathsf{fn}\ x \Rightarrow f(f(x))$$

has the type : $(\alpha \to \alpha) \to \alpha \to \alpha$. Intuitively this means we can use this function in multiple ways for example, when we use it in

$$\mathsf{double}\ (\mathsf{fn}\ x \Rightarrow x + 2\ )\ 3$$

the type variable α will get instantiated to int. When we use it in

$$\mathsf{double}\ (\mathsf{fn}\ x \Rightarrow x\ )\ \mathsf{false}$$

the type variable α will be instantiated to bool. We would like to emphasize that the same code will get executed in both expression.

1 Type variables

We will begin by clarifying the use of type-variables. First, let us extend the definition for types with type variables:

$$\text{Types } T \ ::= \ \mathsf{int} \mid \mathsf{bool} \mid T_1 \to T_2 \mid T_1 \times T_2 \mid \alpha$$

What do we mean by "instantiating a type variable α with a concrete type bool in the type $\alpha \to \alpha$?" – Essentially we mean we apply a substitution to the type $\alpha \to \alpha$ which replaces all free occurrences of the type variable α with the type bool. Defining substitution is in fact straightforward:

$$
\begin{array}{lcl}
[T/\alpha](\alpha) & = & T \\
[T/\alpha](\beta) & = & \beta \\
[T/\alpha](\mathsf{int}) & = & \mathsf{int} \\
[T/\alpha](\mathsf{bool}) & = & \mathsf{bool} \\
[T/\alpha](T_1 \times T_2) & = & [T/\alpha]T_1 \times [T/\alpha]T_2 \\
[T/\alpha](T_1 \to T_2) & = & [T/\alpha]T_1 \to [T/\alpha]T_2
\end{array}
$$

We will write σ for the simultaneous substitution $[T_1/\alpha_1, \ldots T_n/\alpha_n]$.

A crucial property of type substitutions is that they preserve the validity of typing statements: If an expression e has type T and σ is a type substitution, then expression e will also have type $[\sigma]T$.

Theorem 1.1 *If $\Gamma \vdash e : T$ and σ is a type substitution then $[\sigma]\Gamma \vdash e : [\sigma]T$.*

1.1 Two views of type variables

Suppose we have an expression e which has type T in a context Γ, i.e. $\Gamma \vdash e : T$, where T and Γ may possibly contain type variables. Then we can ask two different questions:

1. Are *all* substitution instances of e well-typed? That is for every type substitution σ, we have $[\sigma]\Gamma \vdash e : [\sigma]T$.

2. Is *some* substitution instance of e well-typed? That is we can find a type substitution σ, such that $[\sigma]\Gamma \vdash e : [\sigma]T$.

The first answer implies that type variables are held abstract during type checking. They are merely place holders which can be instantiated with any concrete type. For example:

$$\mathsf{fn}\ f \Rightarrow \mathsf{fn}\ x \Rightarrow f(f(x)) \quad \text{has type } \ (\alpha \to \alpha) \to \alpha \to \alpha$$

By replacing α with bool, we still maintain that the function is well typed, and in fact has exactly the same typing derivation! Holding type variables abstract in such a way leads to *parametric polymorphism*. Type variables are used to encode the fact that a term can be used in many concrete contexts with different concrete types.

In the second view, the original term e may not even be well typed. what we want to know is whether we can instantiated the type variables such that it will be well-typed. For example, we could ask whether there exists an instantiation for the type variables β_1 and β_2 s.t. the term

$$\mathsf{fn}\ f \Rightarrow \mathsf{fn}\ x \Rightarrow f(f(x)) \quad \text{has type } \beta_1 \to \beta_2$$

well-typed.

The answer to this question would be yes, if we instantiate β_1 with $(\alpha \to \alpha)$ and β_2 with $\alpha \to \alpha$. Recall that function types are parenthesized to the right.

To illustrate let us consider a few more examples:

1. Does there exist and instantiation for the type variables β s.t. we have

$$\mathsf{fn}\ x \Rightarrow x + 1 \quad \text{has type} \quad \beta\ ?$$

   Yes, if we instantiate *beta* to $\mathsf{int} \to \mathsf{int}$.

2. Does there exist and instantiation for the type variables β s.t. we have

$$\mathsf{fn}\ x \Rightarrow x \quad \text{has type} \quad \beta\ ?$$

   Yes, if we instantiate β to $\mathsf{int} \to \mathsf{int}$ or we choose β to be $\mathsf{bool} \to \mathsf{bool}$ or we choose β to be $\alpha \to \alpha$.

3. Does there exist and instantiation for the type variables β s.t. we have

$$\mathsf{fn}\ x \Rightarrow x + 1 \quad \text{has type} \quad \beta \to \mathsf{bool}\ ?$$

   The answer is no.

The answer to whether there exists an instantiation for the type variables in T such that an expression e has type T is not necessarily unique as the examples show. There may be many possible instantiations. A principal solution is an instantiation σ for the free type variables in $\Gamma \vdash e : T$ s.t. any other solution can be obtained from it.

2 Type inference

When inferring the type for an expression e we essentially ask whether there exists an instantiation for the type variable β s.t. e has type β. Informally, the reasoning process can be split in two phases: *inferring a type* for an expression and *checking that certain constraints are satisfied*. For example, to infer the type for $\mathsf{if}\ 3 = 1\ \mathsf{then}\ 55\ \mathsf{else}\ 44$ we *infer the type* T for $3 = 1$, the type T_1 for 55 and the type T_2 for 44. In the second phase we *check* that certain conditions are satisfied, namely $T = \mathsf{bool}$ and $T_1 = T_2$.

An expression e has a type T, if we infer some type T for e and certain constraints are satisfied. We will concentrate first on constraint generation, and then discuss how to solve constraints.

2.1 Constraint generation

Constraints are either true (written as tt, or consist of an equality constraints on types (written as $T_1 = T_2$, or if C_1 and C_2 are constraints, then so is the conjunction $C_1 \wedge C_2$.

$$\text{Constraints}\ \ C \ \ := \mathsf{tt} \mid C_1 \wedge C_2 \mid T_1 = T_2$$

To satisfy the constraint $C_1 \wedge \ldots \wedge C_n$ all the individual constraints C_i must be satisfied.

How do constraints arise? Let us consider as an example, how we infer a type T for an if-expression $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ we will argue as follows:

1. Infer the type T' for expression e.
2. Infer the type T_1 for expression e_1
3. Infer the type T_2 for expression e_2

Then $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ will have T_1 if $T' = \mathsf{bool} \wedge T_1 = T_2$, i.e. the listed constraints can be satisfied.

$\Gamma \vdash e \Rightarrow T/C$ Infer type T for expression e in the typing environment Γ modulo the constraints C

Ultimately, we want that expression e will have type T if the constraints C are satisfied. For now however, we will just generate constraints. We will consider the typing rules individually.

Numbers and booleans and variables First the rules for number, booleans, and variables. For these expressions there are no constraints which need to be satisfied.

$$\frac{}{\Gamma \vdash n \Rightarrow \mathsf{int}/\mathsf{tt}}\ \text{T-NUM} \qquad \frac{x{:}T \in \Gamma}{\Gamma \vdash x \Rightarrow T/\mathsf{tt}}\ \text{T-VAR}$$

$$\frac{}{\Gamma \vdash \mathsf{true} \Rightarrow \mathsf{bool}/\mathsf{tt}}\ \text{T-TRUE} \qquad \frac{}{\Gamma \vdash \mathsf{false} \Rightarrow \mathsf{bool}/\mathsf{tt}}\ \text{T-FALSE}$$

If-expressions and primitive operations Next, the rule for if-expressions. It follows our recipe from above.

$$\frac{\Gamma \vdash e \Rightarrow T/C \qquad \Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow T_1/C \wedge C_1 \wedge C_2 \wedge T = \mathsf{bool} \wedge T_1 = T_2}\ \text{T-IF}$$

Similarly, we can define rules for our arithmetic operations.

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash e_1 + e_2 \Rightarrow \mathsf{int}/C_1 \wedge C_2 \wedge T_1 = \mathsf{int} \wedge T_2 = \mathsf{int}}\ \text{T-PLUS}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash e_1 = e_2 \Rightarrow \mathsf{bool}/C_1 \wedge C_2 \wedge T_1 = T_2}\ \text{T-EQ}$$

Let-expressions The rule for let-expressions is straightforward.

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma, x{:}T_1 \vdash e_2 \Rightarrow T/C_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \Rightarrow T/C_1 \wedge C_2}\ \text{T-LET}$$

Functions The most interesting case arises for functions, since we lack the information necessary to determine the type of the input argument. This is resolved by generating a new type variable α.

$$\frac{\Gamma, x{:}\alpha \vdash e \Rightarrow T/C}{\Gamma \vdash \mathsf{fn}\ x \Rightarrow e\ \Rightarrow \alpha \to T/C}\ \text{T-FN}$$

Before we continue, let us consider how we can infer a type for the function $\mathsf{fn}\ x \Rightarrow x+1$. We simplify a little bit along the way and writing instead of $C \wedge \mathsf{T}$ just C.

$$\dfrac{\dfrac{\dfrac{}{x{:}\alpha \vdash x \Rightarrow \alpha/\mathsf{tt}}\ \text{T-VAR} \quad \dfrac{}{x{:}\alpha \vdash 1 \Rightarrow \mathsf{int}/\mathsf{tt}}\ \text{T-NUM}}{x{:}\alpha \vdash x+1 \Rightarrow \mathsf{int}/\mathsf{tt} \wedge \mathsf{tt} \wedge \alpha = \mathsf{int} \wedge \mathsf{int} = \mathsf{int}}\ \text{T-PLUS}}{\cdot \vdash \mathsf{fn}\ x \Rightarrow x+1\ \Rightarrow \alpha \to \mathsf{int}/\mathsf{tt} \wedge \mathsf{tt} \wedge \alpha = \mathsf{int} \wedge \mathsf{int} = \mathsf{int}}\ \text{T-FN}$$

Simplifying a little bit the constraints, this essentially means $\mathsf{fn}\ x \Rightarrow x+1$ will have type $\alpha \to \mathsf{int}$ if we can satisfy $\alpha = \mathsf{int}$. Note, that our typing derivation will now always succeed! It may only happen that our constraints cannot be satisfied. For example, $\mathsf{fn}\ x \Rightarrow x + \mathsf{true}$ will lead to the constraints $\alpha = \mathsf{int} \wedge \mathsf{int} = \mathsf{bool}$. Obviously these constraints cannot be satisfied. We give a general constraint solving algorithm in the next section. For now, let us continue with the typing rules.

Application Let us consider application next. We recursively infer the type for e_1 and e_2, and must impose certain constraints on their type respectively. We introduce a type variable α to extract the return type of the function described by T_1.

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \alpha/C_1 \wedge C_2 \wedge T_1 = (T_2 \to \alpha)}\ \text{T-APP}$$

Let us briefly summarize. We have so far seen how to collect constraints during type checking. This process will always succeed. However an expression is only well-typed, if we can collect some constraints C (a process which always succeeds) and we can solve the constraints C (a process which may fail).

2.2 Solving typing constraints

The general question we are answering in this section is whether there exists an instantiation for the type variables in the constraint C s.t. all constraints occurring in C are satisfied. For example,

- $\alpha = \mathsf{int} \wedge \alpha \to \beta = \mathsf{int} \to \mathsf{bool}$ can be satisfied by instantiating α with int and β with bool.
- $\alpha_1 \to \alpha_2 = \mathsf{int} \to \beta \wedge \beta = \mathsf{bool}$ can be satisfied by instantiating α_1 with int and β with bool, and α_2 with bool.

- $\alpha_1 \to \alpha_2 = \mathsf{int} \to \beta \wedge \beta = \alpha_2 \to \alpha_2$ cannot be satisfied! The first constraint suggests that $\alpha_2 = \beta$, but the second suggests $\beta = \alpha_2 \to \alpha_2$! We can never find an instantiation for β s.t. $\beta = \beta \to \beta$!!

Solving constraints of this form is done via *unification*. In general, we say two types T_1 and T_2 are *unifiable* if there exists an instantiation σ for the free type variables in T_1 and T_2 s.t. $[\sigma]T_1 = [\sigma]T_2$, i.e $[\sigma]T_1$ is syntactically equal to $[\sigma]T_2$.

Unification is a general algorithm to determine whether two objects can be made syntactically equal. We will present here a version which will only test whether a set of constraints is unifiable. It will stop, if we have simplified our constraints to tt.

$$
\begin{array}{lcll}
C \wedge \mathsf{tt} & \Longrightarrow & C & \\
C \wedge \mathsf{int} = \mathsf{int} & \Longrightarrow & C & \\
C \wedge \mathsf{bool} = \mathsf{bool} & \Longrightarrow & C & \\
C \wedge (T_1 \to T_2) = (S_1 \to S_2) & \Longrightarrow & C \wedge T_1 = S_1 \wedge T_2 = S_2 & \\
C \wedge \alpha = T & \Longrightarrow & [T/\alpha]C & \text{provided that } \alpha \notin \mathsf{FV}(T) \\
C \wedge T = \alpha & \Longrightarrow & [T/\alpha]C & \text{provided that } \alpha \notin \mathsf{FV}(T)
\end{array}
$$

We can solve the constraints C by transforming C using the rules above. We terminate, when no rules is applicable anymore. If we can transform C to the true constraint tt, i.e. $C \Longrightarrow^* \mathsf{tt}$, otherwise we fail. We also not that solving a constraints C using the unification rules always terminates, either showing that C is unifiable or it is not unifiable.

2.3 Type inference in practice

In practice, type inference algorithms do not strictly separate the two phases of generating constraints and checking constraints, but rather check the constraints eagerly. This however requires that we propagate instantiations for type-variables.

3 Polymorphism and Let-expressions

The type inference algorithm described above can be easily generalized to provide ML-style polymorphism, also known as let-polymorphism. Let us consider again the example from the introduction, where we defined the function double as follows:

$$\mathsf{double} = \mathsf{fn}\ f \Rightarrow \mathsf{fn}\ x \Rightarrow f(f(x))$$

Since the type we infer for this function is polymorphic, we should be able to use this function in two ways:

(a) $\mathsf{double}\ (\mathsf{fn}\ x \Rightarrow x + 2\ )\ 3$

(b) $\mathsf{double}\ (\mathsf{fn}\ x \Rightarrow x\ )\ \mathsf{false}$

However, what will happen when we try to use the same double function with both booleans and numbers? In other words, will we be able to type-check the following piece of code?

$$\begin{array}{ll} \mathsf{let} & d = \mathsf{fn}\ f \Rightarrow \mathsf{fn}\ x \Rightarrow f(f(x)) \\ & x = d\ (\mathsf{fn}\ x \Rightarrow x + 2\ )\ 3 \\ & y = d\ (\mathsf{fn}\ x \Rightarrow x\ )\ \mathsf{false} \\ \mathsf{in} & (x, y)\ \ \mathsf{end} \end{array}$$

The answer is in fact no. This is best explained at a simpler example. What will happen when we try to infer a type for the following expression

$$\mathsf{let}\ f = \mathsf{fn}\ x \Rightarrow x\ \ \mathsf{in}\ (f\ 3,\ f\ \mathsf{true})\ \mathsf{end}?$$

We will first infer the type of $\mathsf{let}\ f = x\ \mathsf{in}\quad \mathsf{end}$ as $\alpha \to \alpha$. Next, we need to infer the type for $(f\ 3,\ f\ \mathsf{true})$ in the typing environment where we assume that f has type $\alpha \to \alpha$. For the expression $f\ 3$ we will infer a type β_0 together with the constraints $\alpha \to \alpha = \mathsf{int} \to \beta_0$, as demonstrated by the following derivation:

$$\frac{\overline{f{:}\alpha \to \alpha \vdash f \Rightarrow \alpha \to \alpha/\mathsf{tt}} \qquad \overline{f{:}\alpha \to \alpha \vdash 3 \Rightarrow \mathsf{int}/\mathsf{tt}}}{f{:}\alpha \to \alpha \vdash (f\ 3) \Rightarrow \beta_0/\alpha \to \alpha = \mathsf{int} \to \beta_0}$$

Next, let us consider the expression $(f\ \mathsf{true})$. For this expression we will infer a type β_1 together with the constraints $\alpha \to \alpha = \mathsf{bool} \to \beta_1$, as demonstrated by the following derivation:

$$\frac{\overline{f{:}\alpha \to \alpha \vdash f \Rightarrow \alpha \to \alpha/\mathsf{tt}} \qquad \overline{f{:}\alpha \to \alpha \vdash \mathsf{true} \Rightarrow \mathsf{bool}/\mathsf{tt}}}{f{:}\alpha \to \alpha \vdash (f\ \mathsf{true}) \Rightarrow \beta_1/\alpha \to \alpha = \mathsf{bool} \to \beta_1}$$

In order for the tuple $(f\ 3,\ f\ \mathsf{true})$ to have some type $\beta_0 \times \beta_1$ we must satisfy the constraints:

$$\alpha \to \alpha = \mathsf{int} \to \beta_0 \wedge \alpha \to \alpha = \mathsf{bool} \to \beta_1$$

But this is impossible since α is supposed to be int and bool at the same time! What went wrong? The problem is that we use the same type variable α for both uses of the function f. By doing so we impose a constraint which is too strong. It requires us to use f in the same way, not in different ways. What we'd like is to break this connection, i.e. we would like to associate different types with the variable f. In other words, every time we use f we can instantiate the type variable α as we need.

How can this problem be fixed? – The simplest solution is to change the rule for let-expressions. Recall the rule for let.

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}$$

Instead of calculating the type of e_1 and then using the type in inferring the type of e_2, we will just re-calculate the type of e_1 every time we need it. We will just substitute for any occurrence of x in the expression e_2 the expression e_1

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash [e_1/x]e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}$$

You may wonder whether it is necessary to type-check e_1 in the first premise of this typing rule. The answer is yes. If x does not occur in e_2 we will actually never type check e_1. This is considered bad, since we would like to ensure that every expression is type-checked without considering special cases of whether a bound variable is used or not.

There is however another more serious objection to this proposal. If x is used many times within the body of e_2, then we will re-type-check e_1 multiple times. Since the right hand side itself can contain let-bindings, this typing rule can cause the type-checker to perform an amount of work that is exponential in the size of the original term! To avoid re-type-checking, practical implementations actually use a more clever though equivalent reformulation of the typing rules. To type-check the term $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$ we infer the type T_1 of e_1. This can be for example done by using the constraint typing rules to calculate the type S_1 and a set of constraints C_1, and then solving the constraints C_1 to obtain an instantiation σ for the free type variables. We can then obtain the principal type T_1 for e_1 by applying σ to S_1. The type T_1 may still contain free type variables. Next, we *generalize* over the free type variables in T_1. If $\alpha_1, \ldots \alpha_n$ are the free type variables in T_1, then we write $\forall\alpha_1 \ldots \forall\alpha_n.T_1$ for the *principal type scheme* of the expression e_1. Finally, we continue to type check e_2 in the extended context $\Gamma, x{:}\forall\alpha_1 \ldots \forall\alpha_n.T_1$.

$$\frac{\Gamma \vdash e_1 : T_1 \quad \mathsf{generalize}(T_1) = \forall\alpha_1 \ldots \forall\alpha_n.T_1 \quad \Gamma, x{:}\forall\alpha_1 \ldots \forall\alpha_n.T_1 \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}$$

When using an assumption such as $x{:}\forall\alpha_1 \ldots \forall\alpha_n.T_1$ we can re-instantiate all the type variables α_i appropriately as often as we need to.

The algorithm is much more efficient that the simplistic approach of substituting e_1 into the body of e_2. SML's type inference algorithm essentially builds on these described ideas. In practice, type inference is linear in the size of the input program. However, the worst-case complexity of type inference is still exponential as shown by Mairson and independently by Kfoury, Tiuryn and Urzyczyn in 1990. The example they constructed involves using deeply nested sequences of lets in the right-hand sides, which cause the types to grow exponentially.

In a full-blown programming language with let-polymorphism, we need to be a bit careful when we try to combine polymorphism with side-effects. Consider the following SML code:

```
let
  val r = ref (fn x => x)
in
  r := (fn x => x + 1) ; (!r) true
end
```

Using the algorithm sketched above we would compute the type of `fn x => x` to be α `=>` α. Hence the type of `r` is (α `=>` α) `ref`. Since we have type variables, we will generalize and continue to type-check the body with the assumption that `r` is $\forall\alpha.$(α `=>` α) `ref`. Then the type-checker will consider the body of the let-expression. Clearly, the update of storing the function `fn x => x+1` in the reference cell `r` is ok. Unfortunately, also the expression `(!r) true` is ok, since we only know that `r` is a cell which contains polymorphic functions of type α `=>` α. By having generalized over the type variables, we now can use this reference cell with different instantiations for α! But this is clearly WRONG! We have updated the function which is stored in the reference cell `r`, and when we read from `r`, we do not get a polymorphic function, but a function of type `int => int`, which cannot be applied to `true`!

The problem is that the typing rules have gotten out of sync with the evaluation rules, and do not properly approximate anymore the runtime behavior! The typing rules see two uses of the reference cell and analyze them under different assumptions. But at run time only one reference cell is actually allocated. The fix used in many programming languages is to restrict the generalization of type variables in the let-expression let $x = e_1$ in e_2 end. Only when the expression e_1 is a syntactic value, we are allowed to generalize. This is often called the *value restriction*.

The value restriction solves our problem at some cost in expressiveness. we can no longer write programs in which the right-hand side of let-expressions can both perform some interesting computation and be assigned a polymorphic type. Surprisingly, this makes hardly any difference in practice.

In SML, the problem arises when you try to type-check for example the following code:

```
- let  val r = ref (fn x => x) in  r end;
```

SML will print back to you a warning saying

```
stdIn:15.2-15.40 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = ref fn : (?.X1 -> ?.X1) ref
```

This indicates that the use of `r` with multiple types is disallowed. Once you use `r` its type will be fixed.