Supplementary Notes on Parametric Polymorphism

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 11
September 30, 2003

After an excursion into advanced control constructs, we return to the basic questions of type systems in the next couple of lectures. The first one addresses a weakness of the language we have presented so far: every expression has exactly one type. Some functions (such as the identity function `fn x => x`) should clearly be applicable at more than one type. We call such function *polymorphic*. We later distinguish two principal forms of polymorphism, namely *parameteric* and *ad hoc* polymorphism.

Briefly, a polymorphic construct is said to be *parametric* if it behaves the same at all its types. The identity function is an example of a function that is parametric in this sense. A function such as addition also has more than one type, at least `+ : int * int -> int` and `+ : float * float -> float`, but the function behaves differently at these two types: one implementation manipulates floating point representations the other integers.

Besides pure functions, there are many data structure (such as lists) whose element types should be arbitrary. We achieved this so far by making lists *primitive* in the language, but this trick does not extend when we try to write interesting programs over lists. For example, the following `map` function is clearly too specalized.

```
rec map : (int -> bool) -> int list -> bool list =>
  fn f : int -> bool =>
    fn l : int list =>
      case l
        of nil => nil[bool]
         | cons(x,l') => cons(f(x),map f l')
```

It should work for any $f : \tau \rightarrow \sigma$, $l : \tau$ `list` and return a result of type σ `list`. The importance of this kind of generic programming varies from language to language and application to application. It has always been considered central in functional programming in order to avoid unnecessary code duplication. In object-oriented programming it does not appear as critical, because subtyping and the class hierarchy allow some form of polymorphic programming. Nonetheless, the Java language has recently decided to add "generics" to its next revision—we will discuss later how this relates to parametric polymorphism as we present it here.

There are different ways to approach polymorphism. In its *intrinsic* form we allow polymorphic functions, but we are careful to engineer the language so that every function still has a unique type. This may sound contradictory, but it is in fact possible with a suitable extension of the expression language. In its *extrinsic* form, we allow an expresson to have multiple types, but we ensure that there is a *principal type* that subsumes (in a suitable sense) all other types an expression might have. The polymorphism of ML is extrinsic; nonetheless, we present it in its intrinsic form first.

The idea is to think of the `map` function above not only takes f and l as arguments, but also the type τ and σ. Fortunately, this does not mean we actually have to pass them at run-time, as we discuss later. We write `Fn` t `=>` e for a function that take a *type* as an argument. The (bound) type variable t stands for that argument in the body, e. The type of such a function is written a $\forall t.\tau$, where τ is the type of the body. To apply a function e to a type argument τ (called *instantiation*), we write e `[`τ`]`. We also introduce a short, mathematical notation for functions that are not recursive, called λ-abstraction.

Concrete	Abstract	Mathematical
`All t.` τ	`All`$(t.\tau)$	$\forall t.\tau$
`Fn` t `=>` e	`Fn`$(t.e)$	$\Lambda t.e$
e `[`τ`]`	`Inst`(e, τ)	$e[\tau]$
`fn` x`:`τ `=>` e	`fn`$(\tau, x.e)$	$\lambda x{:}\tau.\ e$

Using this notation, we can rewrite the example above.

```
Fn t => Fn s =>
rec map : (t -> s) -> t list -> s list =>
  fn f : t -> s =>
    fn l : t list =>
      case l
        of nil => nil[s]
         | cons(x,l') => cons(f(x),map f l')
```

In order to formalize the typing rules, recall the judgment τ type. So far, this judgment was quite straightforward, with rules such as

$$\frac{\tau_1 \text{ type} \quad \tau_2 \text{ type}}{\texttt{arrow}(\tau_1, \tau_2) \text{ type}} \qquad \frac{\tau \text{ type}}{\texttt{list}(\tau) \text{ type}} \qquad \frac{}{\texttt{int type}}$$

Now, types may contain type variables. An example is the type of the identity function, which is $\forall t.t \to t$, or the type of the `map` function, which is $\forall t.\forall s.(t \to s) \to \texttt{list}(t) \to \texttt{list}(s)$. So the typing judgment becomes *hypothetical*, that is, we may reason from assumption t type for variables t. In all the rules above, they are simply propagated (we show the example of the function type). In addition, we have new rule for universal quantification.

$$\frac{\tau_1 \text{ type} \quad \tau_2 \text{ type}}{\texttt{arrow}(\tau_1, \tau_2) \text{ type}} \qquad \frac{\Gamma, t \text{ type} \vdash \tau \text{ type}}{\Gamma \vdash \texttt{All}(t.\tau) \text{ type}}$$

In addition, the notion of hypothetical judgments yields the rule for type variables

$$\frac{}{\Gamma_1, t \text{ type}, \Gamma_2 \vdash t \text{ type}}$$

and a substitution property.

Lemma 1 (Type Substitution in Types)
If $\Gamma_1 \vdash \tau$ type *and* Γ_1, t type$, \Gamma_2 \vdash \sigma$ type *then* $\Gamma_1, \{\tau/t\}\Gamma_2 \vdash \{\tau/t\}\sigma$ type*.*

This is the idea behind higher-order abstract syntax and hypothetical judgments, applied now to the language of types. Note that even though we wrote Γ above, only assumptions of the form t type will actually be relevant to the well-formedness of types.

Now we can present the typing rules proper.

$$\frac{\Gamma, t \text{ type} \vdash e : \sigma}{\Gamma \vdash \texttt{Fn}(t.e) : \texttt{All}(t.\sigma)}$$

$$\frac{\Gamma \vdash e : \texttt{All}(t.\sigma) \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \texttt{Inst}(e, \tau) : \{\tau/t\}\sigma}$$

Let us consider the example of the polymorphic identity function to understand the substitution taking place in the last rule. You should read this derivation bottom-up to understand the process of type-checking.

$$\begin{array}{rcl} t\ \mathsf{type}, x{:}t & \vdash & x : t \\ t\ \mathsf{type} & \vdash & \mathtt{fn}(t, x.x) : \mathtt{arrow}(t,t) \\ \cdot & \vdash & \mathtt{Fn}(t.\mathtt{fn}(t, x.x)) : \mathtt{All}(t.\mathtt{arrow}(t,t)) \end{array}$$

If we abbreviate the identity function by id then it must be instantiated by (apply to) a type before it can be applied to an expression argument.

$$\begin{array}{rcl} \cdot & \vdash & id : \forall t.t \to t \\ \cdot & \vdash & id\,[\mathtt{int}] : \mathtt{int} \to \mathtt{int} \\ \cdot & \vdash & id\,[\mathtt{int}]\,3 : \mathtt{int} \end{array}$$

$$\begin{array}{rcl} \cdot & \vdash & id : \forall t.t \to t \\ \cdot & \vdash & id\,[\mathtt{bool}] : \mathtt{bool} \to \mathtt{bool} \\ \cdot & \vdash & id\,[\mathtt{bool}]\,\mathtt{true} : \mathtt{bool} \end{array}$$

$$\begin{array}{rcl} \cdot & \vdash & id : \forall t.t \to t \\ \cdot & \vdash & id\,[\mathtt{int}] : \mathtt{int} \to \mathtt{int} \\ \cdot & \nvdash & id\,[\mathtt{int}]\,\mathtt{true} : \mathtt{int} \end{array}$$

Using mathematical notation:

$$\begin{array}{rcl} t\ \mathsf{type}, x{:}t & \vdash & x : t \\ t\ \mathsf{type} & \vdash & \lambda x{:}t.\ x : t \to t \\ \cdot & \vdash & \Lambda t.\ \lambda x{:}t.\ x : \forall t.t \to t \end{array}$$

As should be clear from these rules, assumptions of the form t type also must appear while typing expression, since expressions contain types. Therefore, we need a second substitution property:

Lemma 2 (Type Substitution in Expressions)
If $\Gamma_1 \vdash \tau$ type *and* Γ_1, t type, $\Gamma_2 \vdash e : \sigma$ *then* $\Gamma_1, \{\tau/t\}\Gamma_2 \vdash \{\tau/t\}e : \{\tau/t\}\sigma$.

Note that we must substitute into Γ_2, because the type variable t may occur in some declaration $x{:}\sigma$ in Γ_2.

In the operational semantics we have a choice on whether to declare a type abstraction `Fn` t `=>` e to be a value, or to reduce e. Intuitively, the latter cannot get stuck because t is a *type variable* not an ordinary variable, and therefore is never needed in evaluation. Even though it seems consistent, we know if now language that supports such evaluation in the presence of free type variables. This decision yields the following rules:

$$\frac{}{\texttt{Fn}(t.e)\ \ \mathsf{value}}$$

$$\frac{}{\texttt{Inst}(\texttt{Fn}(t.e), \tau) \mapsto \{\tau/t\}e} \qquad \frac{e \mapsto e'}{\texttt{Inst}(e, \tau) \mapsto \texttt{Inst}(e', \tau)}$$

From this it is routine to prove the progress and preservation theorems. For preservation, we need the type substitution lemmas stated earlier in this lecture. For progress, we need a new value inversion property.

Lemma 3 (Polymorphic Value Inversion)
If $\cdot \vdash v : \texttt{All}(t.\tau)$ *and* $v\ \ \mathsf{value}$ *then* $v = \texttt{Fn}(t.e')$ *for some* e'.

Theorem 4 (Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof: By rule induction on the transition derivation for e. In the case of the reduction of a polymorphic function to a type argument, we need the type substitution property. ■

Theorem 5 (Progress)
If $\cdot \vdash e : \tau$ *then either*

(i) $e\ \ \mathsf{value}$*, or*

(ii) $e \mapsto e'$ *for some* e'

Proof: By rule induction on the typing derivation for e. We need polymorphic value inversion to show that all cases for a type instantiation are covered. ■

In our language the polymorphism is *parametric*, which means that the operation of a polymorphic function is independent of the type that it is applied to. Formalizing this observation requires some advanced techniques that we will not discuss in this course.

This can be contrasted with *ad hoc* polymorphism, in which the function may compute differently at different types. For example, if the function $+$ is overloaded, so it has type $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$ and also type $\texttt{float} \times \texttt{float} \rightarrow \texttt{float}$, then we need to have two different implementations of the function. Another example may be a `toString` function whose behavior depends on the type of the argument.

Parametric polymorphism can often be implemented in a way that avoids carrying types at run-time. This is important because we do not want polymorphic functions to be inherently less efficient than ordinary functions. ML has the property that all polymorphic functions are parametric with polymorphic equality as the only exception. Ignoring polymorphic equality, this means we can avoid carrying type information at run-time. In practice, some time information is usually retained in order to support garbage collection or some optimization. How to best implement polymorphic languages is still an area of active research.

ML-style polymorphism is not quite as general as the one described here. This is so that polymorphic type inference remains decidable and has *principal types*. See [Ch 20.2] for a further discussion. We will return to the issue of type inference later in this course.

Parametric polymorphism, even in the restricted form in which it is present in ML, can be dangerous when the language also has effects such as mutable references. The most straightforward rules for polymorphism in its extrinsic form (where expressions have multiple types) are

$$\frac{\Gamma, t\ \mathsf{type} \vdash e : \sigma \quad e\ \mathsf{value}}{\Gamma \vdash e : \mathtt{All}(t.\sigma)}$$

$$\frac{\Gamma \vdash e : \mathtt{All}(t.\sigma) \quad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash e : \{\tau/t\}\sigma}$$

The only change to the previous system is that we do not allow types in expressions, and that the expression e remains the same for type abstraction and application. In this system, when the language includes effects, the generalization rule must be restricted to values or it will be unsound. The prototypical example is the following ML code:

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

Even though `(fn x => x)` $: \alpha \rightarrow \alpha$ we can not conclude that $r : \forall\alpha.(\alpha \rightarrow \alpha)\mathsf{ref}$. If that were allowed, both the assignment to r and the dereferencing of r would be well-typed, even though the code obviously violates types safety. In our rule above, the restriction of e to values would rule out this generalation (`ref (fn x => x)` is not a value).