Computer Science 15–312
Polymorphism

Robert Harper

Spring, 2000

1 Introduction

MinML is an *explicitly-typed* language. The abstract syntax is defined to have sufficient type information to ensure that all expressions have a unique type. In particular the types of the parameters of a function must be chosen when the function is defined.

While this is not itself a serious problem, it does expose a significant weakness in the MinML type system. For example, there is no way to define a generic procedure for composing two functions whose domain and range match up appropriately. Instead we must define a separate composition operation for each choice of types for the functions being composed. Here is one composition function

```
fun _ (f:string->int):(char->string)->(string->int) is
  fun _ (g:char->string):string->int is
    fun _ (x:string):int is apply(f, apply(g, x)),
```

and here is another

```
fun _ (f:float->double):(int->float)->(int->double) is
  fun _ (g:int->float):int->double is
    fun _ (x:int):double is apply(f, apply(g, x)).
```

The annoying thing is that both versions of function composition execute the same way; they differ only in the choice of types of the functions being composed. This is rather irksome, and very quickly gets out of hand in practice. Statically typed languages have long been criticized for precisely this reason. Fortunately this inflexibility is not an inherent limitation of statically typed languages, but rather a limitation of the particular type system we have given to MinML. A rather straightforward extension is sufficient to provide the kind of flexibility that is essential for a practical language. This extension is called *polymorphism*.

While ML has had such a type system from its inception (circa 1978), few other languages have followed suit. Notably the Java language suffers from this limitation (but the difficulty is mitigated somewhat in the presence of subtyping). Plans are in the works, however, for adding polymorphism (called *generics*) to the Java language. A compiler for this extension, called Generic Java, is already available.

2 Polymorphic MinML

Polymorphic MinML, or PolyMinML, is an extension of MinML with the ability to define *polymorphic functions*. Informally, a polymorphic function is a function that takes a *type* as argument and yields a *value* as result. The type parameter to a polymorphic function represents an *unknown*, or *generic*, type, which can be instantiated by applying the function to a specific type. The types of polymorphic functions are called *polymorphic types*, or *polytypes*.

A significant design decision is whether to regard polymorphic types as "first-class" types, or whether they are, instead, "second-class" citizens. Polymorphic functions in ML are second-class — they cannot be passed as arguments, returned as results, or stored in data structures. The only thing we may do with polymorphic values is to bind them to identifiers with a `val` or `fun` binding. Uses of such identifiers are automatically instantiated by an implicit polymorphic instantiation. The alternative is to treat polymorphic functions as first-class values, which can be used like any other value in the language. Here there are no restrictions on how they can be used, but you should be warned that doing so precludes using type inference to perform polymorphic abstraction and instantiation automatically.

We'll set things up for second-class polymorphism by explicitly distinguishing polymorphic types from monomorphic types. The first-class case can then be recovered by simply conflating polytypes and monotypes.

Abstract Syntax

The abstract syntax of PolyMinML is defined by the following extension to the MinML grammar:

$$
\begin{array}{llcl}
\textit{Polytypes} & \sigma & ::= & \tau \mid \forall t(\sigma) \\
\textit{Monotypes} & \tau & ::= & \ldots \mid t \\
\textit{Expressions} & e & ::= & \ldots \mid \texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end} \mid \texttt{inst(}e\texttt{,}\tau\texttt{)} \\
\textit{Values} & v & ::= & \ldots \mid \texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}
\end{array}
$$

The variable t ranges over a set of *type variables*, which are written ML-style `'a`, `'b`, and so on in examples. In the polytype $\forall t(\sigma)$ the type variable t is bound in σ; we do not distinguish between polytypes that differ only in the names of bound variables. Since the quantifier can occur only at the outermost level, in ML it is left implicit. An expression of the form $\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}$ is a *polymorphic function* with parameter t and body e. The variable t is bound within e. An expression of the form $\texttt{inst(}e\texttt{,}\tau\texttt{)}$ is a *polymorphic instantiation* of the polymorphic function e at *monotype* τ. Notice that we may *only* instantiate polymorphic functions with monotypes. In examples we write $f\texttt{[}\tau\texttt{]}$ for polymorphic instantiation, rather than the more verbose $\texttt{inst(}f\texttt{,}\tau\texttt{)}$.

We write $\mathrm{FTV}(()\tau)$ (respectively, $\mathrm{FTV}(()\sigma)$, $\mathrm{FTV}(()e)$) for the set of free type variables occurring in τ (respectively, σ, e). Capture-avoiding substitution of a monotype τ for free occurrences of a type variable t in a polytype σ (resp., monotype τ', expression e) is written $[\tau/t]\sigma$ (resp., $[\tau/t]\tau'$, $[\tau/t]e$).

Static Semantics

The static semantics of PolyMinML is a straightforward extension to that of MinML. One significant change, however, is that we must now keep track of the scopes of type variables, as well as ordinary variables. In the static semantics of MinML a typing judgement had the form $\Gamma \vdash e : \tau$, where Γ is a context assigning types to ordinary variables. Only those variables in $\mathrm{dom}\,\Gamma$ may legally occur in e. For PolyMinML we must introduce an additional context, Δ, which is a set of type variables, those that may legally occur in the types and expression of the judgement.

The static semantics consists of rules for deriving the following two judgements:

$$
\begin{array}{ll}
\Delta \vdash \tau\ \mathsf{ok} & \tau \textit{ is a well-formed type in } \Delta \\
\Gamma \vdash_\Delta e : \sigma & e \textit{ is a well-formed expression of type } \sigma \textit{ in } \Gamma \textit{ and } \Delta
\end{array}
$$

The rules for validity of types are as follows:

$$\frac{t \in \Delta}{\Delta \vdash t \;\mathsf{ok}} \qquad \frac{}{\Delta \vdash \texttt{int}\;\mathsf{ok}} \qquad \frac{}{\Delta \vdash \texttt{bool}\;\mathsf{ok}}$$

$$\frac{\Delta \vdash \tau_1 \;\mathsf{ok} \quad \Delta \vdash \tau_2 \;\mathsf{ok}}{\Delta \vdash \tau_1 \to \tau_2 \;\mathsf{ok}}$$

$$\frac{\Delta \cup \{\, t \,\} \vdash \sigma \;\mathsf{ok} \quad t \notin \Delta}{\Delta \vdash \forall t(\sigma) \;\mathsf{ok}}$$

The auxiliary judgement $\Delta \vdash \Gamma$ is defined by the following rule:

$$\frac{\Delta \vdash \Gamma(x) \;\mathsf{ok}\; (\forall x \in \mathrm{dom}(\Gamma))}{\Delta \vdash \Gamma \;\mathsf{ok}}.$$

The rules for deriving typing judgements $\Gamma \vdash_\Delta e : \sigma$ are as follows. We assume that $\Delta \vdash \Gamma$ ok, $\Delta \vdash \sigma$ ok, $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$, and $\mathrm{FTV}(()e) \subseteq \Delta$. We give only the rules specific to PolyMinML; the remaining rules are those of MinML, augmented with a set Δ of type variables.

$$\frac{\Gamma \vdash_{\Delta \cup \{\, t \,\}} e : \sigma \quad t \notin \Delta}{\Gamma \vdash_\Delta \texttt{Fun}\, t \,\texttt{in}\, e \,\texttt{end} : \forall t(\sigma)}$$

$$\frac{\Gamma \vdash_\Delta e : \forall t(\sigma) \quad \Delta \vdash \tau \;\mathsf{ok}}{\Gamma \vdash_\Delta \texttt{inst}(e,\tau) : [\tau/t]\sigma}$$

For example, here is the polymorphic composition function in PolyMinML:

```
Fun t in
  Fun u in
    Fun v in
      fun _(f:u->v):(t->u)->(t->v) is
        fun _(g:t->u):t->v is
          fun _(x:t):v is apply(f, apply(g, x))
```

It is easy to check that it has type

$$\forall t(\forall u(\forall v((u \to v) \to (t \to u) \to (t \to v)))).$$

We will need the following technical lemma stating that typing is preserved under instantiation:

Lemma 1 (Instantiation)
If $\Gamma \vdash_{\Delta \cup \{\, t \,\}} e : \sigma$, where $t \notin \Delta$, and $\Delta \vdash \tau$ ok, then $[\tau/t]\Gamma \vdash_\Delta [\tau/t]e : [\tau/t]\sigma$.

The proof is by induction on typing, and involves no new ideas beyond what we have already seen.

We will also have need of the following canonical forms lemma:

Lemma 2 (Canonical Forms)
If $v : \forall t(\sigma)$, then $v = \mathtt{Fun}\, t\, \mathtt{in}\, e\, \mathtt{end}$ for some t and e such that $\emptyset \vdash_{\{t\}} e : \sigma$.

This is proved by a straightforward analysis of the typing rules.

Dynamic Semantics

The dynamic semantics of PolyMinML is a simple extension of that of MinML. We need only add the following two SOS rules:

$$\frac{}{\mathtt{inst(Fun}\, t\, \mathtt{in}\, e\, \mathtt{end,}\tau\mathtt{)} \mapsto [\tau/t]e} \qquad \frac{e \mapsto e'}{\mathtt{inst(}e\mathtt{,}\tau\mathtt{)} \mapsto \mathtt{inst(}e'\mathtt{,}\tau\mathtt{)}}$$

It is then a simple matter to prove safety for this language.

Theorem 3 (Preservation)
If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.

The proof is by induction on evaluation.

Theorem 4 (Progress)
If $e : \sigma$, then either e is a value or there exists e' such that $e \mapsto e'$.

As before, this is proved by induction on evaluation.

First-Class Polymorphism

The syntax given above describes an ML-like treatment of polymorphism, *albeit* one in which polymorphic abstraction and instantiation is explicit, rather than implicit, as it is in ML. To obtain the first-class variant of PolyMinML, we simply ignore the distinction between poly- and mono-types, regarding them all as simply types. Everything else remains unchanged, including the proofs of progress and preservation.

With first-class polymorphism we may consider types such as

$$\forall t(t \to t) \to \forall t(t \to t),$$

which cannot be expressed in the ML-like fragment. This is the type of functions that accept a polymorphic function as argument and yield a polymorphic function (of the same type) as result. If f has the above type, then

f`(Fun` t `in fun _ (`x`:`t`):`t `is` x `end end)` is well-formed. However, the application f`(fun _ (`x`:int):int is` $+(x,1)$ `end)` is ill-formed, because the successor function does not have type $\forall t(t{\rightarrow}t)$. The requirement that the argument be polymorphic is a significant restriction on how f may be used!

Contrast this with the following type (which does lie within the ML-like fragment):

$$\forall t((t{\rightarrow}t){\rightarrow}(t{\rightarrow}t)).$$

This is the type of polymorphic functions that, for each type t, accept a function on t and yield another function on t. If g has this type, the expression `inst(`g`,int)(succ)` is well-formed, since we first instantiate g at `int`, then apply it to the successor function.

The situation gets more interesting in the presence of data structures such as lists and reference cells. It is a worthwhile exercise to consider the difference between the types $\forall t(\sigma)$ `list` and $\forall t(\sigma$ `list`$)$ for various choices of σ. Note once again that the former type cannot be expressed in ML, whereas the latter can.

Recall the following counterexample to type soundness for the early version of ML without the so-called value restriction:

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

A simple check of the polymorphic typing rules reveals that this is a well-formed expression, provided that the value restriction is suspended. Of course, it “gets stuck” during evaluation by attempting to add `1` to `true`.

Using the framework of explicit polymorphism, I will argue that the superficial plausibility of this example (which led to the unsoundness in the language) stems from a failure to distinguish between these two types:

1. The type $\forall t(t{\rightarrow}t$ `ref`$)$ of polymorphic functions yielding reference cells containing a function from a type to itself.

2. The type $\forall t(t{\rightarrow}t)$ `ref` of reference cells containing polymorphic functions yielding a function from a type to itself.

(Notice the similarity to the distinctions discussed above.) For this example to be well-formed, we rely on an inconsistent reading of the example. At the point of the `val` binding we are treating `r` as a value of the latter type, namely a reference cell containing a polymorphic function. But in the body of the `let` we are treating it as a value of the former type, a polymorphic function yielding a reference cell. We cannot have it both ways at once!

To sort out the error let us make the polymorphic instantiation and abstraction explicit. Here's one rendering:

```
let
    val r : All 'a (('a -> 'a) ref) =
      Fun 'a in ref (fn x:'a => x) end
 in
    r[int] := (fn x:int => x+1) ; (!(r[bool]))(true)
end
```

Notice that we have made the polymorphic abstraction explicit, and inserted corresponding polymorphic instantiations. This example is type correct, and hence (by the proof of safety above) sound. But notice that it allocates *two* reference cells, not *one*! Recall that polymporphic functions are values, and the binding of `r` is just such a value. Each of the two instances of `r` executes the body of this function separately, each time allocating a new reference cell. Hence the unsoundness goes away!

Here's another rendering that is, in fact, ill-typed (and should be, since it "gets stuck"!).

```
let
    val r : (All 'a ('a -> 'a)) ref =
      ref (Fun 'a in fn x:'a => x end)
 in
    r := (fn x:int => x+1) ; (!r)[bool](true)
end
```

The assignment to `r` is ill-typed because the successor is not sufficiently polymorphic. The retrieval and subsequent instantiation and application is type-correct, however. If we change the program to

```
let
    val r : (All 'a ('a -> 'a)) ref =
      ref (Fun 'a in fn x:'a => x end)
 in
    r := (Fun 'a in fn x:'a => x end) ; (!r)[bool](true)
end
```

then the expression is well-typed, and behaves sanely, precisely because we have assigned to `r` a sufficiently polymorphic function.

3 ML-style Type Inference

ML-style type inference may be viewed as a translation from the implicitly-typed syntax of ML to the explicitly-typed syntax of PolyMinML. Specifically, the type inference mechanism performs the following tasks:

- Attaching type labels to function arguments and results.
- Inserting polymorphic abstractions for declarations of polymorphic type.
- Inserting polymorphic instantiations whenever a polymorphic declared variable is used.

Thus in ML we may write

```
val I : 'a -> 'a = fn x => x
val n : int = I(I)(3)
```

This stands for the PolyMinML declarations[1]

$$\texttt{val I : } \forall t(t \to t) \texttt{ = Fun}\; t \;\texttt{in}\; \texttt{fun}\; \_\; \texttt{(}x\texttt{:}t\texttt{):}t \;\texttt{is}\; x \;\texttt{end}\;\texttt{end}$$
$$\texttt{val n : int = inst(}I\texttt{,int}\to\texttt{int)(inst(}I\texttt{,int))(3)}$$

Here we apply the polymorphic identity function to itself, then apply the result to 3. The identity function is explicitly abstracted on the type of its argument and result, and its domain and range types are made explicit on the function itself. The two occurrences of `I` in the ML code are replaced by instantiations of `I` in the PolyMinML code, first at type $\texttt{int}\to\texttt{int}$, the second at type `int`.

With this in mind we can now explain the "value restriction" on polymorphism in ML. Referring to the example of the previous section, the type inference mechanism of ML generates the first rendering of the example give above in which the type of the reference cell is $\forall t((t \to t)\, \texttt{ref})$. As we've seen, when viewed in this way, the example is not problematic, *provided that* polymorphic abstractions are seen as values. For in this case the two instances of `r` generate two distinct reference cells, and no difficulties arise. Unfortunately, ML does not treat polymorphic abstractions as values! Only one reference cell is allocated, which, in the absence of the value restriction, would lead to unsoundness.

[1] We've not equipped PolyMinML with a declaration construct, but you can see from the example how this might be done.

Why does the value restriction save the day? In the case that the polymorphic expression is not a value (in the ML sense) the polymorphic abstraction that is inserted by the type inference mechanism *changes a non-value into a value*! This changes the semantics of the expression (as we've seen, from allocating one cell, to allocating two different cells), which violates the semantics of ML itself.[2] However, if we limit ourselves to values in the first place, then the polymorphic abstraction is only ever wrapped around a value, and no change of semantics occurs. Therefore[3], the insertion of polymorphic abstraction doesn't change the semantics, and everything is safe. The example above involving reference cells is ruled out, because the expression `ref (fn x => x)` is not a value, but such is the nature of the value restriction.

4 Parametricity

Our original motivation for introducing polymorphism was to enable more programs to be written — those that are "generic" in one or more types, such as the composition function give above. The idea is that if the behavior of a function *does not* depend on a choice of types, then it is useful to be able to define such "type oblivious" functions in the language. Once we have such a mechanism in hand, it can also be used to ensure that a particular piece of code *can not* depend on a choice of types by insisting that it be polymorphic in those types. In this sense polymorphism may be used to impose restrictions on a program, as well as to allow more programs to be written.

The restrictions imposed by requiring a program to be polymorphic underlie the often-observed experience when programming in ML that if the types are correct, then the program is correct. Roughly speaking, since the ML type system is polymorphic, if a function type checks with a polymorphic type, then the strictures of polymorphism vastly cut down the set of well-typed programs with that type. Since the intended program is one these (by the hypothesis that its type is "right"), you're much more likely to have written it if the set of possibilities is smaller.

The technical foundation for these remarks is called *parametricity*. The goal of this section is to give an account of parametricity for PolyMinML. To keep the technical details under control, we will restrict attention to the

[2]One could argue that the ML semantics is incorrect, which leads to a different language.

[3]This would need to be proved, of course.

ML-like (prenex) fragment of PolyMinML. It is possibly to generalize to first-class polymorphism, but at the expense of considerable technical complexity. Nevertheless we will find it necessary to gloss over some technical details, but wherever a "pedagogic fiction" is required, I will point it out. To start with, it should be stressed that the following *does not apply* to languages with mutable references!

4.1 Informal Discussion

We will begin with an informal discussion of parametricity based on a "seat of the pants" understanding of the set of well-formed programs of a type.

Suppose that a function value f has the type $\forall t(t \to t)$. What function could it be?

1. It could diverge when instantiated — f `[`τ`]` goes into an infinite loop. Since f is polymorphic, its behavior cannot depend on the choice of τ, so in fact f `[`τ'`]` diverges for all τ' if it diverges for τ.

2. It could converge when instantiated at τ to a function g of type $\tau \to \tau$ that loops when applied to an argument v of type τ — *i.e.*, g`(`v`)` runs forever. Since f is polymorphic, g must diverge on *every* argument v of type τ if it diverges on *some* argument of type τ.

3. It could converge when instantiated at τ to a function g of type $\tau \to \tau$ that, when applied to a value v of type τ returns a value v' of type τ. Since f is polymorphic, g cannot depend on the choice of v, so v' must in fact be v.

Let us call cases (1) and (2) *uninteresting*. The foregoing discussion suggests that the only *interesting* function f of type $\forall t(t \to t)$ is the polymorphic identity function.

Suppose that f is an interesting function of type $\forall t(t)$. What function could it be? A moment's thought reveals that it cannot be interesting! That is, every function f of this type must diverge when instantiated, and hence is uninteresting. In other words, there are no interesting values of this type — it is essentially an "empty" type.

For a final example, suppose that f is an interesting function of type $\forall t(t\,\texttt{list} \to t\,\texttt{list})$. What function could it be?

1. The identity function that simply returns its argument.

2. The constantly-`nil` function that always returns the empty list.

3. A function that drops some elements from the list according to a predetermined (data-independent) algorithm — *e.g.*, always drops the first three elements of its argument.

4. A permutation function that reorganizes the elements of its argument.

The characteristic that these functions have in common is that their behavior is entirely determined by the *spine* of the list, and is independent of the *elements* of the list. For example, f cannot be the function that drops all "even" elements of the list — the elements might not be numbers! The point is that the type of f is polymorphic in the element type, but reveals that the argument is a list of unspecified elements. Therefore it can only depend on the "list-ness" of its argument, and never on its contents.

In general if a polymorphic function behaves the same at every type instance, we say that it is *parametric* in that type. In PolyMinML all polymorphic functions are parametric. In Standard ML most functions are, except those that involve *equality types*. The equality function is *not* parametric because the equality test depends on the type instance — testing equality of integers is different than testing equality of floating point numbers, and we cannot test equality of functions. Such "pseudo-polymorphic" operations are said to be *ad hoc*, to contrast them from *parametric*.

How can parametricity be exploited? As we will see later, parametricity is the foundation for data abstraction in a programming language. To get a sense of the relationship, let us consider a classical example of exploiting parametricity, the *polymorphic Church numerals*. Let N be the type $\forall t(t\rightarrow(t\rightarrow t)\rightarrow t)$. What are the interesting functions of the type N? Given any type τ, and values $z : \tau$ and $s : \tau\rightarrow\tau$, the expression

$$f\,\texttt{[}\tau\texttt{]}\,\texttt{(}z\texttt{)}\,\texttt{(}s\texttt{)}$$

must yield a value of type τ. Moreover, it must behave uniformly with respect to the choice of τ. What values could it yield? The only way to build a value of type τ is by using the element z and the function s passed to it. A moment's thought reveals that the application must amount to the n-fold composition

$$s\texttt{(}s\texttt{(}\ldots s\texttt{(}z\texttt{)}\ldots\texttt{))}.$$

That is, the elements of N are in 1-to-1 correspondence with the natural numbers.

Let us write $\overline{n}$ for the polymorphic function of type N representing the natural number n, namely the function

```
Fun t in
    fn z:t in
       fn s:t->t in
          s(s(... s)...))
       end
    end
end
```

where there are n occurrences of s in the expression. Observe that if we instantiate $\overline{n}$ at the built-in type `int` and apply the result to 0 and `succ`, it evaluates to the number n. In general we may think of performing an "experiment" on a value of type N by instantiating it at a type whose values will constitute the observations, the applying it to operations z and s for performing the experiment, and observing the result.

Using this we can calculate with Church numerals. Let us consider how to define the addition function on N. Given $\overline{m}$ and $\overline{n}$ of type N, we wish to compute their sum $\overline{m+n}$, also of type N. That is, the addition function must look as follows:

```
fn m:N in
   fn n:N in
      Fun t in
          fn z:t in
             fn s:t->t in
                ...
             end
          end
      end
   end
end
```

The question is: how to fill in the missing code? Think in terms of experiments. Given m and n of type N, we are to yield a value that when "probed" by supplying a type t, an element z of that type, and a function s on that type, must yield the $(m+n)$-fold composition of s with z. One way to do this is to "run" m on t, z, and s, yielding the m-fold composition of s with z, then "running" n on this value and s again to obtain the n-fold composition of s with the n-fold composition of s with z — the desired answer. Here's the code:

```
fn m:N in
   fn n:N in
      Fun t in
          fn z:t in
             fn s:t->t in
                n[t](m[t](z)(s))(s)
             end
          end
      end
   end
end
```

To see that it works, instantiate the result at τ, apply it to z and s, and observe the result.

4.2 Relational Parametricity

In this section we give a more precise formulation of parametricity. The main idea is that polymorphism implies that certain equations between expressions must hold. For example, if $f : \forall t(t \rightarrow t)$, then f must be *equal to* the identity function, and if $f : N$, then f must be *equal to* some Church numeral $\overline{n}$. To make the informal idea of parametricity precise, we must clarify what we mean by equality of expressions.

The main idea is to define equality in terms of "experiments" that we carry out on expressions to "test" whether they are equal. The valid experiments on an expression are determined solely by its type. In general we say that two closed *expressions* of a type τ are equal iff either they both diverge, or they both converge to equal *values* of that type. Equality of closed values is then defined based on their type. For integers and booleans, equality is straightforward: two values are equal iff they are identical. The intuition here is that equality of numbers and booleans is directly observable. Since functions are "infinite" objects (when thought of in terms of their input/output behavior), we define equality in terms of their behavior when applied. Specifically, two functions f and g of type $\tau_1 \rightarrow \tau_2$ are equal iff whenever they are applied to equal arguments of type τ_1, they yield equal results of type τ_2.

More formally, we make the following definitions. First, we define equality of closed expressions of type τ as follows:

$$e \cong_{exp} e' : \tau \quad \textit{iff} \quad e \mapsto^* v \Leftrightarrow e' \mapsto^* v.$$

Notice that if e and e' both diverge, then they are equal expressions in this sense. For closed values, we define equality by induction on the structure of monotypes:

$$\begin{array}{rcl} v \cong_{val} v' : \texttt{bool} & \textit{iff} & v = v' = \texttt{true} \textit{ or } v = v' = \texttt{false} \\ v \cong_{val} v' : \texttt{int} & \textit{iff} & v = v' = n \textit{ for some } n \geq 0 \\ v \cong_{val} v' : \tau_1 \rightarrow \tau_2 & \textit{iff} & v_1 \cong_{val} v_1' : \tau_1 \textit{ implies } v\texttt{(}v_1\texttt{)} \cong_{exp} v'\texttt{(}v_1'\texttt{)} : \tau_2 \end{array}$$

The following lemma states two important properties of this notion of equality.

Lemma 5

1. *Expression and value equivalence are reflexive, symmetric, and transitive.*

2. *Expression equivalence is a* congruence*: we may replace any sub-expression of an expression e by an equivalent sub-expression to obtain an equivalent expression.*

So far we've considered only equality of closed expressions of monomorphic type. The definition is made so that it readily generalizes to the polymorphic case. The idea is that when we quantify over a type, we are not able to say *a priori* what we mean by equality at that type, precisely because it is "unknown". Therefore we *also* quantify over all possible notions of equality to cover all possible interpretations of that type. Let us write $R : \tau \leftrightarrow \tau'$ to indicate that R is a binary relation between valeus of type τ and τ'.

Here is the definition of equality of polymorphic values:

$$v \cong_{val} v' : \forall t(\sigma) \quad \textit{iff} \quad \textit{for all } \tau \textit{ and } \tau', \textit{ and all } R : \tau \leftrightarrow \tau',\ v\ \texttt{[}\tau\texttt{]} \cong_{exp} v'\ \texttt{[}\tau'\texttt{]} : \sigma$$

where we take equality at the type variable t to be the relation R (*i.e.*, $v \cong_{val} v' : t$ iff $v\,R\,v'$).

There is one important *proviso*: when quantifying over relations, we must restrict attention to what are called *admissible* relations, a sub-class of relations that, in a suitable sense, respects computation. Most natural choices of relation are admissible, but it is possible to contrive examples that are not. The rough-and-ready rule is this: a relation is admissible iff it is closed under "partial computation". Evaluation of an expression e to a value proceeds through a series of intermediate expressions $e \mapsto e_1 \mapsto e_2 \mapsto \cdots e_n$. The expressions e_i may be thought of as "partial computations" of e, stopping points along the way to the value of e. If a relation relates corresponding partial computations of e and e', then, to be admissible, it must also relate e and e' — it cannot relate all partial computations, and then refuse to relate the complete expressions. We will not develop this idea any further, since to do so would require the formalization of partial computation. I hope that this informal discussion suffices to give the idea.

The following is Reynolds' Parametricity Theorem:

Theorem 6 (Parametricity)
If $e : \sigma$ is a closed expression, then $e \cong_{exp} e : \sigma$.

This may seem obvious, until you consider that the notion of equality between expressions of polymorphic type is very strong, requiring equivalence under *all possible* relational interpretations of the quantified type.

Using the Parametricity Theorem we may prove a result we stated informally above.

Theorem 7
If $f : \forall t(t \to t)$ is an interesting value, then $f \cong_{val} id : \forall t(t \to t)$, where id is the polymorphic identity function.

Proof: Suppose that τ and τ' are monotypes, and that $R : \tau \leftrightarrow \tau'$. We wish to show that

$$f\ [\tau] \cong_{exp} id\ [\tau'] : t \to t,$$

where equality at type t is taken to be the relation R.

Since f (and id) are interesting, there exists values f_τ and $id_{\tau'}$ such that

$$f\ [\tau] \mapsto^* f_\tau$$

and

$$id\ [\tau'] \mapsto^* id_{\tau'}.$$

We wish to show that

$$f_\tau \cong_{val} id_{\tau'} : t \to t.$$

Suppose that $v_1 \cong_{val} v_1' : t$, which is to say $v_1\, R\, v_1'$ since equality at type t is taken to be the relation R. We are to show that

$$f_\tau\,(v_1) \cong_{exp} id_{\tau'}\,(v_1') : t$$

By the assumption that f is interesting (and the fact that id is interesting), there exists values v_2 and v_2' such that

$$f_\tau\,(v_1) \mapsto^* v_2$$

and

$$id_{\tau'}(v_1') \mapsto^* v_2'.$$

By the definition of id, it follows that $v_2' = v_1'$ (it's the identity function!). We must show that $v_2 \, R \, v_1'$ to complete the proof.

Now define the relation $R' : \tau \leftrightarrow \tau$ to be the set $\{ (v, v) \mid v \, R \, v_1' \}$. Since $f : \forall t(t \rightarrow t)$, we have by the Parametricity Theorem that $f \cong_{val} f : \forall t(t \rightarrow t)$, where equality at type t is taken to be the relation R'. Since $v_1 \, R \, v_1'$, we have by definition $v_1 \, R' \, v_1$. Using the definition of equality of polymorphic type, it follows that

$$f_\tau(v_1) \cong_{exp} id_{\tau'}(v_1) : t.$$

Hence $v_2 \, R \, v_1'$, as required. ■

You might reasonably wonder, at this point, what the relationship $f \cong_{val} id : \forall t(t \rightarrow t)$ has to do with f's execution behavior. It is a general fact, which we will not attempt to prove, that equivalence as we've defined it yields results about execution behavior. For example, if $f : \forall t(t \rightarrow t)$, we can show that for every τ and every $v : \tau$, $f \, [\tau](v)$ evaluates to v. By the preceding theorem $f \cong_{val} id : \forall t(t \rightarrow t)$. Suppose that τ is some monotype and $v : \tau$ is some closed value. Define the relation $R : \tau \leftrightarrow \tau$ by

$$v_1 \, R \, v_2 \text{ iff } v_1 = v_2 = v.$$

Then we have by the definition of equality for polymorphic values

$$f \, [\tau](v) \cong_{exp} id \, [\tau](v) : t,$$

where equality at t is taken to be the relation R. Since the right-hand side terminates, so must the left-hand side, and both must yield values related by R, which is to say that both sides must evaluate to v.