Programming Languages
Polymorphism
Robert Harper
Spring, 1999

Definitions

Type inference interacts with definitions in an interesting way.

Suppose we add top-level definitions to the language, so that we can write

```
def I = fun f(x) is x end;
eval I(I)(3);
```

We need two different reconstructions of `I` for this to be well-formed!

Substitution Principle

Principle: treat defined variables as if they were implicitly replaced by their definitions everywhere.

This is called the "by name", or "substitutive", interpretation of definitions.

Thus `I(I)(3)` is treated as if it were

```
(fun f(x) is x end)(fun f(x) is x end)(3)
```


Definitions by Name

The "by name" interpretaton ensures that that every occurrence gets a "fresh" reconstruction, suitable for its use at that occurrence.

The "by name" interpretation ensures that defined variables are **polymorphic**, as in ML.

Thus `I(I)(3)` reconstructs to

```
(fun f(x:int→int):int→int is x end)
  (fun f(x:int):int is x end)
    (3)
```

In effect `I` is used at different types.

Definitions by Name

But what happens if the definition has an effect?

```
def X = (print "hello"; fun f(x) is x end);
eval X(X)(3);
def Y = ref nil;
eval (Y := [1]; not (hd (!Y)));
```


Definitions by Name

In the first case, we print "hello" twice, not once! Every **use** of `X` causes "hello" to be printed. The **definition** has no effect.

In the second case, we allocate **two** different reference cells, one for each **use** of `Y`. The first contains `[1]`, the second contains `nil`. The application of `hd` fails due to a match exception.

Definitions by Value

Contrast with the **by value**, or **sequential**, intepretation in which the definition is evaluated **once** when bound. Each use retrieves the value.

In the above examples, "hello" would be printed once (when `X` is defined), but the `ref` example goes wrong!

1. Only one `ref` cell is allocated (when `Y` is defined).

2. We store `[1]`in that cell, but retrieve it as a list of booleans.

By Name vs. By Value

The "by name" interpretation is **not** equivalent to the "by value" interpretation if the definition has an effect.

That is, if the definition has no effect, then the "by name" and "by value" interpretations coincide. (You need an effect to detect replication.)

Polymorphism depends on the "by name" interpretation.

The "by value" interpretation is also useful for specifying the order of evaluation ("do this, then do that").

But the "by value" interpretation is incompatible with polymorphism!

By Name vs.By Value

One approach is to clearly distinguish two definition mechanisms:

1. Polymorphic = by name interpretation.

2. Monomorphic = by value interpretation.

By Name vs. By Value

But there is some redundancy in the case of non-effecting expressions, a possible source of problems.

```
def I = fun f(x) is x end
val J = fun f(x) is x end
eval I(I)(3)   (* OK *)
eval J(J)(3)   (* Not OK *)
```


The Value Restriction

We can reconcile the two views, by adopting the following conventions:

1. There is only one definition mechanism.

2. If the definition is effect-free, then it can be polymorphic.

3. If the definition is not effect-free, then it is monomorphic.

This is the ML solution.

The Value Restriction

But it is undecidable whether or not an expression has an effect!

For example, `f(); print "hello"` prints "hello" iff `f()` halts.

Solution: use a conservative approximation to effect-free expressions which is easy to check and easy to remember.

Simplest idea: every value (variable, constant, lambda) is manifestly effect-free. Restrict polymorphism to values.

The Value Restriction

This rule-of-thumb works out OK about 75% of the time.

It rules out examples such as these, which suppress polymorphism:

```
val I' = I o I
val l = nil @ nil
val f = map (fn x => (x,x))
```

We really need a better idea! (Suggestions are welcome.)

Implementing Polymorphism

In practice we don't actually substitute the definition for the defined variable.

- Wastes space for the copies.
- Wastes time for re-checking each copy.

Implementing Polymorphism

A nasty example in ML:

```
let val x =
  let val y = ...
    let val z = (nil, nil)
    in ...
  in (y, y) end
in (x, x) end
```


Polymorphic Types

The principal reconstruction property tells us that every copy of an expression reconstructs to an instance of the principal reconstruction.

This suggests that we build the principal reconstruction once, and arrange to instantiate it at the points of use.

This is called **polymorphism**, which was invented by John Reynolds and Jean-Yves Girard (for very different reasons).

Polymorphic Types

First, let's drop the distinction between types and type schemes. We'll call these **monotypes**. (We'll often write s, t, and u for type variables from now on.)

A **polymorphic type**, or **polytype**, σ, is either a monotype, τ, or a **quantified type** $\forall t.\sigma$.

That is, polytypes have the form $\forall t_1 \ldots \forall t_n.\tau$, where τ is a monotype.

Examples:
$\forall t.t\,\texttt{list} \rightarrow t\,\texttt{list}$
$\forall t.\forall u.(t \rightarrow u) \rightarrow t\,\texttt{list} \rightarrow u\,\texttt{list}$

Polymorphic Terms

We add two new term forms to MinML,

1. Polymorphic abstraction: $\Lambda\, t.e$ (also written $\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}$).
2. Polymorphic instantiation: $e\,[\tau]$ (also written $\texttt{inst}(e,\tau)$).

A polymorphic abstraction is a function that takes a type as argument. A polymorphic instantiation applies such a function to a type.

Examples

The polymorphic identity function, with type $\forall t.t \rightarrow t$:

```
Fun t in fn (x:t) in x end
```

The polymorphic reverse function, with type $\forall t.t\,\texttt{list} \rightarrow t\,\texttt{list}$:

```
Fun t in
  fun rev (l:t list):t list is
    if null(l) then nil
    else cons (hd l, rev (tl l)) fi
  end
end
```


Static Semantics

The static semantics is a lot like MinML, except that we must introduce another context to take account of the "active" type variables.

Typing judgement: $\Gamma \vdash_\Delta e : \sigma$, where Δ is a set of type variables.

Write $\Delta \vdash \tau$ to mean $\mathsf{TV}(\tau) \subseteq \Delta$.

Invariant: $\mathsf{TV}(\Gamma) \cup \mathsf{TV}(e) \cup \mathsf{TV}(\sigma) \subseteq \Delta$.

Static Semantics

New typing rules:

$$\frac{\Gamma \vdash_{\Delta \cup \{t\}} e : \sigma}{\Gamma \vdash_{\Delta} \texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end} : \forall t.\sigma}$$

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

The others remain the same, except that you add a Δ subscript everywhere.

Example

$$\vdash_{\emptyset} \texttt{Fun}\, t\, \texttt{in}\, \texttt{fn}\, x{:}t\, \texttt{in}\, x\, \texttt{end}\, \texttt{end} : \forall t.t{\to}t,$$

since

$$x{:}t \vdash_{\{t\}} x : t$$

$\vdash_{\emptyset}$ `Fun t in fun rev (l:t list):t list is ...end` : $\forall t.t\,\texttt{list}{\to}t\,\texttt{list}$,

since

$$\texttt{rev}{:}t\,\texttt{list}, \texttt{l}{:}t\,\texttt{list} \vdash_{\{t\}} \ldots : t\,\texttt{list}$$

Dynamic Semantics

Type abstractions are values.

New primitive instruction

$$(\Lambda\, t.e)\ [\tau] \mapsto [\tau/t]e$$

Check progress and preservation (it's quite easy)!

Polymorphic Definitions

Polymorphic definitions are reducible to polymorphic abstraction and instantiation.

To process `def` x `=` u,

1. Determine the principal reconstruction $e^* : \tau^*$ of u with unconstrained type variables $\alpha_1, \ldots, \alpha_n$.

2. Bind x to the term

   $$\Lambda\, \alpha_1 \ldots \Lambda\, \alpha_n.e^*$$

   with the polymorphic type

   $$\forall\alpha_1 \ldots \forall\alpha_n.\tau^*$$

Polymorphic Definitions

Whenever you see a use of a defined variable x with polytype $\forall t_1 \ldots \forall t_n . \tau$,

1. Apply it to enough schematic type variables to reduce to a monotype: `inst(inst(`x`,`α_1`)...,`α_n`)`.

2. Solve constraints for α_1, ..., α_n to find the type arguments for x.

Polymorphic Definitions

Example: `def I = fun f(x) is x end`.

Bind `I` to the polymorphic function

```
Fun t in fun f(x:t):t is x end
```

The expression `I(I)(3)` becomes

```
I[int->int](I[int])(3)
```


Polymorphic Definitions

Example: `def map = fun map (f, l) is ...end`.

Bind `map` to the polymorphic function

```
Fun s in
  Fun t in
    fun map (f : s->t, l : s list): t list is
      ...
    end
```

Uses of `map` become instances, as in

```
map [int] [string] (Int.toString, [1,2,3,4])
```


Polymorphic Definitions

Observe that polymorphic instantiation occurs at run-time!

That is,

$$\begin{aligned} \mathtt{I\,[int{\to}int]} &\mapsto \mathtt{fun}\, f\, (x\mathtt{:int{\to}int})\mathtt{:int{\to}int\ is}\ x\ \mathtt{end} \\ \mathtt{I\,[int]} &\mapsto \mathtt{fun}\, f\, (x\mathtt{:int})\mathtt{:int\ is}\ x\ \mathtt{end} \end{aligned}$$

This makes it possible to take advantage of type information at run-time (but we are not exploiting that possibility here.)

Introduction to Parametricity

Polymorphic functions enjoy a property called **parametricity**, which states that they behave "uniformly" with respect to their type instances.

Very roughly speaking, this means that all instances implement the same "algorithm".

Parametricity

Example:

```
def rev =
  Fun t in
    fun rev (l:t list):t list is
      if null(l) then nil else cons (hd l, rev (tl l)) fi
    end
  end
```

All instances `rev[`τ`]` run the same algorithm, no matter how τ is chosen.

Parametricity

Putting it another way, a polymorphic type severely constrains the possible programs of that type, precisely because the code has to be parametric.

For example, what could be the elements of the type $\forall t.t \rightarrow t$?

When applied to τ and $v : \tau$, its value (if any) must be of type τ.

Since τ is arbitrary, this cuts down the possibilities enormously!

Parametricity

The elements of type $\forall t.t \rightarrow t$ might be

- A divergent (looping) expression.
- The identity function which, given τ and $v : \tau$, returns v.
- Nothing else!

Parametricity

Why are there no other possibilities? A genuine proof is difficult, but informally we can reason as follows.

If $f : \forall t.t \to t$, then f must behave the same way on **any** instance τ.

This means that given τ and given $v : \tau$, apply(f [τ],v) must yield a value of type τ, if it yields a value at all.

Clearly it can diverge, or yield v itself (by any circuitous route).

It cannot yield, say, 3! This would only make sesne if $\tau =$ int; otherwise, the code would be ill-typed.

Parametricity

What can we say about a function f with type $\forall t.int \to t$?

It must diverge!

Every instance inst(f,τ) must be a function of type $int \to \tau$.

By uniformity, the same code must be executed for each instance. Therefore apply(inst(f,τ),3) must yield a value of type τ, for any type τ you choose.

This is impossible! Therefore f must diverge when applied!

Parametricity

What is the principal type of this function?

```
fun loop (x) is loop(x) end
```

Performing type reconstruction, we obtain the polytype $\forall s.\forall t.s \to t$, or, in ML notation, 'a->'b.

This makes sense because loop neither uses its argument nor constraints its result in any non-trivial manner.

Note that loop(e) has **any** type whatsoever!

Parametricity

What are the values of type $\forall t.t\,\texttt{list} \to t\,\texttt{list}$?

Parametricity

What are the values of type $\forall t.t\,\texttt{list}{\rightarrow}t\,\texttt{list}$?

- The divergent function.
- Any permutation function, including `rev` and `id`.
- Any "spine-manipulating" function, *e.g.*, returning a sublist.

Why? Such a function can only manipulate the spine of the list!

Implementing Parametric Polymorphism

Parametricity imposes significant burdens on an implementation!

In a monomorphic language we would have two separate identity functions $\texttt{id}_{\texttt{float}}$ and $\texttt{id}_{\texttt{int}}$.

- $\texttt{id}_{\texttt{float}}$ might take an argument in FPR0 and return its result in FPR1.
- $\texttt{id}_{\texttt{int}}$ might take an argument in IR0 and return its result in IR1.

Implementing Parametric Polymorphism

How might the **polymorphic** identity function be implemented?

- `id[float]` should have the "same algorithm" as `id[int]`, at least as far as the programmer is concerned.
- Must work uniformly for all possible type instances $\texttt{id}[\tau]$! How is this possible?

Implementing Parametric Polymorphism

How might the polymorphic subscript operation be implemented?

- `sub` has type $\forall t.t\,\texttt{array*int}{\rightarrow}t$.

Problem: what is the offset of the ith element of a polymorphic array?

In a monomorphic language we would have separate function $\texttt{sub}_\tau$ for each τ.

Implementing Parametric Polymorphism

Where is the ith element in a τ `array`?

- If τ is `float` and `float array`'s are an aligned sequences of doublewords, then the ith element is at offset $8 \times i$.
- If τ is `int` and `int array`'s are sequences of words, then the ith element is at offset $4 \times i$.
- How can we find the ith element uniformly in the type of the array?

Implementing Parametric Polymorphism

Two main methods:

1. Boxing: represent "large" (non-word) values by pointers into the heap, which are "small" (word-sized).
2. Type analysis: perform run-time type dispatch to select appropriate code.

Real implementations use a combination of methods.

Boxing

Naïve boxing strategy: if a value will not fit in a word, allocate a heap cell and represent it by a pointer.

- Integers and booleans are unboxed (not allocated).
- Floats are boxed.
- Closures and tuples are boxed.

Invariant: the size of a value at run-time is always one word.

Boxing

This solves the argument/return problem: always take one word in IR0 and return one word in IR1 (say).

(More complex methods are possible, but this is the general idea.)

This solves the array subscript problem: the offset of the ith element is always $4 \times i$.

Boxing

But boxing all values is costly!

- Floating point operations require at least two load/store's! (One to fetch the argument(s), the other to allocate the result.)

- Floating point arrays can no longer be represented as aligned sequences of doublewords. Extra level of indirection is forced.

Worse, boxing violates the pay-as-you-go principle! You're paying the cost of polymorphism, even if your code is completely monomorphic!

Type Analysis

Another method: explicitly analyze type arguments at run-time.

- `id[int]` evaluates to $\texttt{id}_{\texttt{int}}$;
  `id[float]` evaluates to $\texttt{id}_{\texttt{float}}$.

- In general $\texttt{id}[\tau]$ evaluates to $\texttt{id}_\tau$, the identity function for values of type τ.

Type Analysis

Similarly, we may use type analysis to select the appropriate subscript operation.

- `sub[int]` evaluates to $\texttt{sub}_{\texttt{int}}$;
  `sub[float]` evaluates to $\texttt{sub}_{\texttt{float}}$.

- In general $\texttt{sub}[\tau]$ evaluates to a specialized subscript operation for arrays with elements of type τ.

Type Analysis

Type analysis is also costly:

- Must pass type arguments to polymorphic functions.

- Must perform type analysis on calls.

But you pay the cost only if you use the feature!

- `sub[float]` can be simplified at compile time to $\texttt{sub}_{\texttt{float}}$.

- Type dispatch is only needed in the dynamic case, where the type argument is not known until run-time.

Summary

Polymorphism allows the definition of "generic" operations.

- Enhances code sharing and reuse substantially.
- Gives a rigorous semantics to "templates": no macros required.

Parametricity = independence of type instantiation.

Summary

Polymorphism has an implementation cost.

- Boxing to ensure universal representations of values.
- Type analysis to avoid same.

With care the cost is proportional to the use.