CSE 341: Programming Languages

Dan Grossman

Spring 2008

Lecture 12— Parametric Polymorphism; Equivalence

Today

Two more "conceptual" topics

- Higher density of more abstract concepts as course progresses
- Think about the theory and how languages "fit together", not just how do I "code something up"

1. Parametric polymorphism
   - Also: Type constructors (e.g., ML's `list` and `option`)
2. Equivalence
   - When are two functions or other expressions "the same"

Parametric Polymorphism

Fancy phrase for "forall types" or sometimes "generics." In ML since mid-80s and now in Java, C#, VB, etc.

- (C++ templates are more like macros (later)).

In ML, there's an implicit "for all" at the beginning of any type with `'a`, `'b`, etc. Example:

```
('a * 'b) -> ('b * 'a)
```

really means:

```
forall 'a, 'b . (('a * 'b) -> ('b * 'a))
```

(though `forall` is just for lecture purposes; it is not in ML)

We can *instantiate* the *type variables* to get a *less general* type. For example, with `string` for `'a` and `int->int` for `'b` we get:

```
(string * (int -> int)) -> ((int->int) * string)
```

All the types

In principle, we could have a very flexible way of building types:

- *Base types* like `int`, `string`, `real`, ...
- *Compound types* like `t1 * t2` and `t1 -> t2` where `t1` and `t2` are *any type*
- *Polymorphic types* like `forall 'a. t` where `'a` can appear in `t`.

Would let you have types like

```
(forall 'a. 'a -> ('a*'a)) -> ((int*int) * (bool*bool))
```

Every language has limits; in ML there is no type like this.

The `forall` is always implicit and "all the way to the outside left", for example this *different type*:

```
('a -> ('a * 'a)) -> ((int * int) * (bool * bool))
```

Example

This code is fine, but ML disallows it to make *type inference* easier.

```
(* function f does _not_ type-check *)
fun f pairmaker = (pairmaker 7, pairmaker true)
val x = f (fn y => (y,y))
```

Versus Subtyping

Compare:

```
fun swap (x,y) = (y,x) (* ('a * 'b) -> ('b * 'a) *)
```

with:

```
class Pair {
  Object x;
  Object y;
  Pair(Object _x, Object _y) { x=_x; y=_y; }
  static Pair swap(Pair pr) {return new Pair(pr.y, pr.x);}
}
```

ML wins in two ways (for this example):

- Caller instantiates types, so doesn't need to cast fields of result
- Callee cannot return a pair of any two objects.

That's why Java added generics...

Java Generics

```
class Pair<T1,T2> {
    T1 x;
    T2 y;
    Pair(T1 _x, T2 _y) { x=_x; y=_y; }
    static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> pr) {
       return new Pair<T2,T1>(pr.y,pr.x);
    }
}
```

This really is a step forward despite the clutter, i.e., it is

```
fun swap (x,y) = (y,x)
```

with explicit types and other verbiage.

Containers

Parametric polymorphism is also ideal for functions over containers (lists, sets, hashtables, etc.) where elements have the same type.

Example: ML lists

```
val :: : ('a * ('a list)) -> 'a list (* infix is syntax *)
val map : (('a -> 'b) * ('a list)) -> 'b list
val sum : int list -> int
val fold : ('a * 'b -> 'b) -> ('a list) -> 'b
```

`list` is not a type; if `t` is a type, then `t list` is a type.

User-defined type constructors

Language-design: If something is useful for a built-in feature, it is useful for progerammer-defined stuff too.

So: Let programmers declare type constructors.

Examples:

```
datatype 'a non_mt_list = One of 'a
                        | More of 'a * ('a non_mt_list)
datatype ('a,'b) mytree =
   Leaf of 'a
 | Node of 'b * ('a,'b) mytree * ('a,'b) mytree
```

Example construction of values:

```
Node("hi",Leaf 17,Leaf 4)     (* (string,int) mytree *)
Node(14,Leaf "hi",Leaf "mom") (* (int,string) mytree *)
(* Node("hi",Leaf 17,Leaf true) *) (* doesn't type-check *)
```

What about lists?

Now *everything* about lists is syntactic sugar!

- Constuctors use funny (infix) syntax
- `[1,2,3]` syntax is built-in

But otherwise it is basically:

```
datatype 'a list = [] | :: of 'a * ('a list)
```

One last thing – not on the test

Polymorphism and mutation can be a dangerous combination.

```
val x = ref []         (* 'a list ref *)
val _ = x := ["hi"]  (* instantiate 'a with string *)
val _ = (hd(!x)) + 7 (* instantiate 'a with int -- bad!! *)
```

To prevent this, ML has "the value restriction": bindings can only get polymorphic types if they are initialized with values.

Alas, that means this does not work even though it should be fine:

```
val pr_list = List.map (fn x => (x,x))
```

But these all work:

```
val pr_list : int list -> (int*int) list =
   List.map (fn x => (x,x))
val pr_list = fun lst => List.map (fn x => (x,x)) lst
fun pr_list lst = List.map (fn x => (x,x)) lst
```

Equivalence

"Equivalence" is a fundamental programming concept

- Code maintenance (simplify code)
- Backward-compatibility (add new optional features)
- Program optimization (make faster without breaking it)
- Abstraction and strong interfaces (previous lecture)

But what does it mean for an expression (or program) $e1$ to be "equivalent" to expression $e2$?

Toward a definition

"Equivalence" really *depends on what is observable*.

- Two different sorting algorithms generally "are equivalent".
- But if one takes a second and the other takes a century?

In programming languages, we generally ignore *internal* differences like running time, private data structures used, etc.

- Otherwise too few things would be "equivalent" — we *want* to justify replacing code with "better (or at least as good) but equivalent"

A definition

Two functions are equivalent if they have the same observable behavior no matter how they are used anywhere in any program.

Given the same argument/environment:

- they produce the same result.
- they have the same (non)termination behavior.
- they mutate the same memory the same way.
- they do the same input/output.
- they raise the same exceptions.

Discouraging/forbidding 3, 4, and 5, helps ensure equivalence.

- For example, *if* you "stay functional" then `(f x) + (f x)` can be replaced by `(f x)*2` *without* consulting what `f` is bound to.
- (Side)-effects are often worth discouraging in any language.

Function equivalences

There are 3 very general things you can do with functions that produce equivalent code. Recognizing them (and their subtle caveats) can make you a better programmer.

1. Systematic renaming of variables
2. "Inlining" by replacing a function call with a body + substitutions
3. Unnecessary function wrapping

We will probably discuss these notions of equivalence and the notion of "free variables" later in the course.

Syntactic Sugar

When all expressions using one construct are totally equivalent to another more primitive construct, we say the former is "syntactic sugar".

- Makes language definition easier
- Makes language implementation easier

Examples:

- `e1 andalso e2` (define as a conditional)
- `if e1 then e2 else e3` (define as a case)
- tuples are really records with field names 1, 2, ...

Note: The error messages used to be even worse because the type-checker worked on a desugared version of your code.

Almost sugar

`#1 e` is not quite sugar because it works for pairs and triples

If we ignore types, then we have this equivalence too:

`let val p = e1 in e2 end` is just `(fn p => e2) e1`.