CPSC 311: Definition of Programming Languages: Polymorphism

17-polymorphism

DRAFT

Joshua Dunfield
University of British Columbia

October 29, 2015

1 What is polymorphism?

In a language with polymorphism (*poly* = many; *morph* = form), some features of the language can operate with *multiple types*. "Some features" and "can operate with" are deliberately vague: there are many kinds of polymorphism, and a given language might allow one kind for some language fatures, under some circumstances, and another kind of polymorphism in others.

2 Kinds of polymorphism

In 1967, Christopher Strachey (who made important contributions to programming language semantics, *and* designed a key ancestor of C) distinguished two kinds of polymorphism:

- parametric polymorphism, and
- *ad hoc* polymorphism.

A further kind of polymorphism (quite likely the kind you've used the most) is *subtype polymorphism*, also called *inclusion polymorphism*. Perhaps ill-advisedly, I'm going to discuss subtype polymorphism when we (almost certainly) discuss subtyping later in 311. (At that point, I might try to argue that subtype polymorphism is a special case of *ad hoc* polymorphism.)

2.1 Examples of parametric polymorphism

In parametric polymorphism, types include *type variables* that can be *instantiated*.

(see 17-poly.sml)

To understand these types, we should really write the *quantifiers* that SML (implicitly) puts around these types. For example, `identity_function` has type

$$\forall\alpha.\,(\alpha \to \alpha) \qquad \text{“for all types } \alpha, \ldots\text{”}$$

That is, any code that calls `identity_function` can provide something of any type it chooses, and will (if evaluation results in a value!) get back something of that same type.

```
identity_function 5;
identity_function (1, 2);
```

In the first line above, `5` has SML type `int`, so SML *instantiates* α with `int`, resulting in the type

$$(\texttt{int} \to \texttt{int})$$

Applying a function of type $(\texttt{int} \to \texttt{int})$ to an `int` results in an `int`, so `identity_function 5` has type `int`.

A larger example is `map_list`, which has the polymorphic type

$$\forall\alpha.\ \big(\forall\beta.\ (\alpha \to \beta) \to (\alpha\ \texttt{list}) \to (\beta\ \texttt{list})\big)$$

This type says: if you pick types α and β (which, like meta-variables in typing rules, might or might not be *different* types), and pass (first) a function of type $\alpha \to \beta$ and (second) a list whose elements all have type α, then the value returned by calling `map_list` (if that call returns at all) will be a list whose elements are of type β.

(illustrate with `map_list make_pair` from 17-poly.sml)

The reason this is called *parametric* polymorphism is that the types α and β don't matter: the implementation of `map_list` doesn't care what types you instantiate α and β with. In fact, in SML it is *impossible* for `map_list` to know which types α and β have been instantiated with!

If you try to do something that depends on α having a particular type, SML will infer a "less polymorphic" type instead:

```
val unpoly_map_list = fn : (bool -> 'b) -> bool list -> 'b list
```

The fact that a parametrically-polymorphic function *cannot* inspect its argument's type means that we can prove "parametricity properties", such as:

> If a function has type $\forall\alpha.\ (\alpha \to \alpha)$, and it is applied to a value v of some type A, and that application evaluates to a value, then the resulting value *is exactly* v.

Or, suppose a function has type $\forall\alpha.\ \big((\alpha * \alpha) \to \alpha\big)$. It could return the first part of the pair, or the second part. Could it do anything else?

Turning the question around (sideways?): What functions *besides* `map_list` have `map_list`'s type?

2.2 Examples of *ad hoc* polymorphism

A common form of *ad hoc* polymorphism is *operator overloading*: in many languages, a single + operator works on more than one type of argument. For example, in SML, + works on both `int`s and `real`s (though not on `string`, and not on one `int` and one `real`).

2.3 Polymorphism in untyped languages

Is Racket polymorphic? The answer depends on whether we take "type" in the (vague) definition above to mean a static type (perhaps defined through typing rules), or whether we consider it more informally, so that, say, `3` and `#false` in Racket are of different types, even though Racket has no type system to stop you from compiling a program like `(+ 3 #false)`.

- If we require "type" to mean a static type, then Racket is not polymorphic because, in a sense, it has *only one type*: the type of "s-expressions", which includes numbers, `#true` and `#false`, functions (`lambda`), lists, and everything else.

  This claim is sometimes phrased as "dynamic 'typing' is *really* just *unityping*", a "unityped" language being a (statically) typed language with only one (*uni-*) type. Thus, Carnegie Mellon University's Bob Harper:

  > "Dynamic typing is but a special case of static typing, one that limits, rather than liberates... Something can hardly be *opposed* to that of which it is but a trivial special case." (from a 2011 blog post)

  Conor McBride (who is *also* a type-systems researcher, and who might thus be expected to agree with Harper) responded to this idea as follows:

  > ". . . in much the way that a punch in the face is a special case of dinner"

  That is, a punch may meet some very literal definition of dinner, but it doesn't meet any *useful* definition of dinner.

  My opinion is that, presuming that "type" only means a static type, Harper's claim is true—but McBride is correct in implying that it's not particularly illuminating: Even if Harper's preference for (statically) typed languages is *entirely correct*, repeating that "dynamic typing is a special case of static typing" tells us nothing about *why* programmers might prefer "dynamically-typed" (or untyped, or "unityped") languages.

- If we say that *any* precise organization of code and/or data into subcategories is "typing", then `#true` and `#false` can be called "booleans", (**lambda** `(x) x`) can be called a "function", and so on. Then Racket is certainly polymorphic, because many functions that you can write in Racket—for example, (**lambda** `(x) x`)—work on many different kinds of Racket "types".