Lecture Notes on Polymorphism

15-411: Compiler Design
Frank Pfenning

Lecture 24
November 19, 2015

1 Introduction

Polymorphism in programming languages refers to the possibility that a function or data structure can accommodate data of different types. There are two principal forms of polymorphism: *ad hoc polymorphism* and *parametric polymorphism*. Ad hoc polymorphism allows a function to compute differently, based on the type of the argument. Parametric polymorphism means that a function behaves uniformly across the various types [Rey74].

In C0, the equality == and disequality != operators are ad hoc polymorphic: they can be applied to small types (int, bool, $\tau*$, $\tau[]$, and also char, which we don't have in L4), and they behave differently at different types (32 bit vs 64 bit comparisons). A common example from other languages are arithmetic operators so that $e_1 + e_2$ could be addition of integers or floating point numbers or even concatenation of strings. Type checking should resolve the ambiguities and translate the expression to the correct internal form.

The language extension of void$*$ is a (somewhat borderline) example of parametric polymorphism, as long as we do not add a construct hastype(τ, e) or eqtype(e_1, e_2) into the language and as long as the execution does not raise a dynamic tag exception. It should therefore be considered somewhat borderline parametric, since implementations must treat it uniformly but a dynamic tag error depends on the run-time type of a polymorphic value.

Generally, whether polymorphism is parametric depends on all the details of the language definition. The importance of parametricity for data abstraction in language implementations cannot be overstated. Failure of parametricity often means failure of data abstraction: an implementation of a generic data structure cannot necessarily be replaced by another one (even if it is correct!) without breaking a client.

2 Parametric Polymorphism

The prototypical example of a parametric function is the identity function, $\lambda x.\, x : \alpha \to \alpha$. In C0, we might write this as

```
a id(a x) {
  return x;
}
```

which interprets the undefined type name a as a type variable whose scope is the current function. The projection function, which ignores its second argument, would be

```
a proj(a x, b y) {
  return x;
}
```

with both a and b as type variables. From this we extract an abstract form of definition

$$\begin{array}{lcl} id & : & \forall a.\, (a) \to a \\ proj & : & \forall a, b.\, (a, b) \to a \end{array}$$

When type-checking the body of a function, the free variables in the function definition are treated like new basic types. In particular, they are *not* subject to instantiation, since in the end the function has to work for *all* types. To account for this we allow a new form of declaration a : type in our typecontext Γ.

When type-checking the use of a polymorphic function, we can instantiated the type variables to other types. For example,

```
if (id(true)) return id(id(4));
```

should be well-typed. In order to formalize this we will need a *substitution* θ for the (quantified) type variables from the definition of a function, using concrete types and other type variables declared in the context. We write

$$\Gamma \vdash \theta : (a_1, \ldots, a_k)$$

if θ substitutes types that are well-formed in Γ for the type variables $a_1, \ldots, a_k$. Furthermore, we write $\theta(\tau)$ for the result of applying the substitution θ to the type τ.

Our typing rule then shapes up as follows:

$$\frac{\begin{array}{l} f : \forall a_1, \ldots, a_k.(\tau_1, \ldots, \tau_n) \to \tau \\ \Gamma \vdash \theta : (a_1, \ldots, a_k) \\ \Gamma \vdash e_1 : \theta(\tau_1) \\ \cdots \\ \Gamma \vdash e_n : \theta(\tau_n) \end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : \theta(\tau)}$$

Note that there is a single substitution θ, so the type variables $a_1, \ldots, a_n$ must be instantiated consistently for all arguments and the result. For example:

$$\frac{\begin{array}{l} id : \forall a.\,(a) \to a \\ \cdot \vdash (\mathsf{int}/a) : (a) \\ \cdot \vdash 4 : \mathsf{int} \end{array}}{\cdot \vdash id(4) : \mathsf{int}}$$

where $4 : \mathsf{int}$ arises from $4 : (\mathsf{int}/a)(a)$.

3 Generic Data Structures

In a first-order imperative language, the main use of polymorphism is for generic data structures. For example, we may want to have a stack with elements of arbitrary type a.

```
struct list_node<a> {
  a data;
  struct list_node<a>* next;
};

typedef struct list_node<a> list<a>;
```

During compilation, we would like to create parametric code, which works the same independently of the type a. If we restrict type variables to be instantiated to *small types* then we can allocate 8 bytes for a polymorphic field of a struct, which should always be enough room. During allocation, the polymorphic field will be initialized with 0, which by design represents the default value of all types.

In other languages we may *box* polymorphic data (replace them by a reference to the actual data), or *monomorphise* the whole program and compile multiple versions of a function.

The type parameter of the structure is indicated inside the angle brackets. Functions manipulating the structure would be correspondingly polymorphic. For example:

```
list<a>* cons(list<a>* p, a elem) {
  list<a>* q = alloc(list<a>);
  q->data = elem;
  q->next = p;
  return q;
}
```

4 Pairs

We can easily define a product type, which would usually be written as $a * b$ in a functional language.

```
struct prod<a,b> {
  a fst;
  b snd;
};

typedef struct prod<a,b>* prod<a,b>;

a fst(prod<a,b> p) {
  return p->fst;
}

b snd(prod<a,b> p) {
  return p->snd;
}

prod<a,b> pair(a x, b y) {
  prod<a,b> p = alloc(struct prod<a,b>);
  p->fst = x;
  p->snd = y;
  return p;
}
```

5 Function Pointers

Polymorphism in data structures is severely handicapped unless we can store function pointers. For example, a hash table may be parameterized by a type key for keys and a type a for the elements stored in the table. We store in the header functions to hash a key value, to compare keys, and extracting a key from an element.

```
struct ht_header<key,a> {
  int size;                 /* size >= 0 */
  int capacity;             /* capacity > 0 */
  list<a*>*[] table;        /* \length(table) == capacity */
  int (*hash)(key k);       /* hash function */
  bool (*key_equal)(key k1, key k2); /* key comparison */
  key (*elem_key)(a elem);  /* extracting key from element */
};
```

```
typedef struct ht_header<key,a> ht<key,a>;

a* ht_lookup(ht<key,a> H, key k)
//@requires is_ht(H);
{
  int i = (*H->hash)(k);
  list<a*>* p = H->table[i];
  while (p != NULL) {
      //@assert p->data != NULL;
      if ((*H->key_equal)((*H->elem_key)(*p->data), k))
        return p->data;
      else
        p = p->next;
  }
  /* not in list */
  return NULL;
}
```

6 Interactions With Other Language Features

The interactions between parametric and ad hoc polymorphism are often tricky. In C0 with parametric polymorphism, the main issue arises with equality. If we have e_1 == e_2 where e_1 and e_2 are of type a? If a stands for a small type, this might be feasible, but there is still a difference between 32-bit and 64-bit comparisons. Alternative, we could simply rule this out. This would suggest itself in particular in C0 with a type string, which is not subject to equality testing.

A general approach to interactions between ad hoc and parametric polymorphism are *type classes* as they are used in Haskell. In lecture, students proposed some extensions of the above so that polymorphism can be limited to type classes. Since I did not take any pictures of the blackboard at the time, these extensions are lost to posterity unless someone sends me some suggestions.

7 Type Inference

Often associated with parametric polymorphism is the idea of *type inference*. For the polymorphic part of the language, this actually presents rather few problems, since the scope of type variables is naturally delineated by function definitions. However, in C0 there is a problem with field selection, $e.f$. Since fields are global and can freely be shared between different structs, it will be difficult to disambiguate uses of the field names f and therefore the type of e.

8 Type Conversions and Coherence

Ad hoc polymorphism is often associated with (implicit) conversions between types. For example, in an expression $3 + x$ where $x : \mathsf{float}$ we might *promote* the integer 3 to a floating point number, since the other summand is a floating point number. There is a complicated set of rules in the definition of C [KR88] regarding such conversions between types, including integral types of varying sizes, pointers, and other numeric types like float or double.

Inside a compiler, such promotions should be turned into explicit operators, for example $\mathsf{itof}(3) + x$, where itof converts an integer to its floating point representation.

The problem with such implicit conversion is that it can easily lead to errors. The more complicated the rules in the language definition, the more likely it is to lead to errors which are often hard to find. Particularly pernicious are error arising from truncation of wider types to narrower ones, since they can remain undetected for a long time on smaller inputs. A language satisfies *coherence* if various legal ways of inserting type conversions always leads to the same answer [Rey91]. In such language the meaning of expressions is less dependent on arcane details. Nevertheless, overloading and implicit conversions ought to be viewed with suspicion.

References

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag.