Semi-explicit polymorphic parameters

Leo White, Jane Street

Abstract

Garrigue and Remy[1] described a system for higher-order polymorphism that forms the basis of the support for polymorphic methods in OCaml. This can be used for higher-rank polymorphism, but the result is not convenient. However, the same underlying ideas can be used to directly support polymorphic parameters. We describe these underlying ideas and give typing rules for polymorphic parameters. Based on these rules, we've implemented polymorphic parameters in our branch of OCaml and have been using them in production for almost a year.

1 Polymorphic methods in OCaml

OCaml supports polymorphic methods in objects. For example,

```
let o = object method id : 'a. 'a -> 'a = fun x -> x end
val o : < id : 'a. 'a -> 'a >
```

This is a form of higher-order polymorphism. For instance, we can represent a list of identity functions with the type `< id : 'a. 'a -> 'a > list`. It can be used for functions with polymorphic parameters, however the result is not convenient. For example,

```
let int_and_string (id : < f : 'a. 'a -> 'a >) =
  id#f 7, id#f "seven"

let sevens =
  int_and_string
    (object method f : 'a. 'a -> 'a = fun x -> x end)
```

Both the type annotations in that example are needed for it to type-check. The annotation on the function parameter is reasonable, but needing a type annotation on every call-site makes this approach too inconvenient for regular use. OCaml also provides polymorphic record fields:

```
type id = { f : 'a. 'a -> 'a }

let int_and_string id =
  id.f 7, id.f "seven"
```

```
let sevens =
  int_and_string {f = fun x -> x}
```

These avoid the type annotations but require a type definition and an explicit record construction at the call-site. In this talk, we'll describe how we added direct support for polymorphic parameters, enabling this example to be written as just:

```
let int_and_string (id : 'a. 'a -> 'a) =
  id 7, id "seven"

let sevens =
  int_and_string (fun x -> x)
```

2 Inference

Supporting polymorphic parameters means supporting the construction and elimination of functions with polymorphic parameters. The construction part is easy. We require all functions with polymorphic parameters to be explicitly annotated. Giving us a simple typing rule:

$$\frac{\Gamma\,;\; x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda(x : \sigma).\, e : \sigma \to \tau}$$

Where σ is a polymorphic type scheme and τ is an ordinary type. It's pretty easy to see that this rule presents no difficulties for principal type inference.

Elimination, on the other hand, is tricky. Our aim is to avoid syntactically distinguishing function applications involving polymorphic parameters from ordinary function applications. This is what will let us write:

```
int_and_string (fun x -> x)
```

However, we also cannot guess the types of polymorphic parameters. Given a higher-order function like:

```
let with_id f = f (fun x -> x)
```

we cannot infer `f` as being a function with polymorphic parameters as that would require inferring higher-order types, which is undecidable.

That means we can only allow a function with polymorphic parameters to be applied when the type of that function is not being guessed – when the type is no longer the subject of type inference. Any function application where the type of the function is still being inferred will be assumed to be an ordinary function application.

2.1 Generalization and sharing

It might seem that distinguishing types currently being inferred from those that are not being guessed runs counter to the usual unification based inference of

Hindley-Milner (HM) type systems. However, inference in HM systems already has an operation with this same constraint: *generalization*.

When inferring whether a unification variable can be generalized to a universal type variable, we must be sure that all constraints which might affect the unification variable have already been solved for. In the standard declarative typing rules, this constraint appears as the side condition on the Gen rule:

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma}$$

If the type variable α does not appear free in the context, then all constraints that might influence it are within e and have already been considered. Thus it can be safely generalized.

The insight of Garrigue and Remy[1] is that we can extend this approach beyond type variables and generalization. Rather than just tracking which type variables are shared with the context, we track all type expressions that are shared with the context.

2.2 Label polymorphism

In order to track which types are shared with the context we need to give these types a notion of identity. This is done through the addition of *labels*. Type constructors where sharing is of interest are annotated with a label:

$$\tau ::= \ \ldots \mid \tau \to_i \tau$$

and type schemes are extended with label polymorphism:

$$\varsigma ::= \ \ldots \mid \forall i.\varsigma$$

A function arrow whose type is not shared with the context – and so is no longer the subject of inference – can now be represented as:

$$\forall i.\ \sigma \to_i \tau$$

Allowing us to write the principal elimination rule for functions with polymorphic parameters:

$$\frac{\Gamma \vdash e_1 : \forall i.\ \sigma \to_i \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}$$

Note that in practice we do not show labels or label polymorphism to the user. They are detectable only through optional warnings that indicate that a particular application of a function with polymorphic parameters is not principal.

3 Polymorphic parameters for OCaml

The syntax and typing rules for a simple HM system with polymorphic parameters is shown in the online appendix[3]. We've extended OCaml with support for polymorphic parameters based on those rules. OCaml's existing support for polymorphic records is based on the similar rules in[1], although sharing is tracked using an efficient implementation based on the notion of levels[2] rather than actual named labels. That made it very easy to reuse the same mechanism for parameters, and the entire implementation took less than a day. We've been using it in production at Jane Street for almost a year now with no reported issues.

References

[1] Garrigue, Jacques, and Didier Rémy. "Semi-explicit first-class polymorphism for ML." Information and computation 155.1-2 (1999): 134-169.

[2] Oleg Kiselyov. "Efficient and Insightful Generalization". http://okmij.org/ftp/ML/generalization.html.

[3] https://gist.github.com/lpw25/fa04a679d186325c0f2025fc2c495f3b