Chapter 4

Parametricity

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies. – Philip Wadler, "Theorems for free!"

Polymorphism allows a single piece of code to be instantiated with multiple types. Polymorphism is parametric when all of the instances behave uniformly. This is in contrast to ad-hoc polymorphism, where values can behave differently depending on which type they are being instantiated with.

Parametricity can be thought of as the dual to abstraction. Where abstraction hides details about an implementation from the outside world, parametricity hides details about the outside world from an implementation.

4.1 Parametricity in OCaml

4.1.1 Universal types in OCaml

Universal types are the most common form of parametricity in OCaml. As discussed in Chapter 2, ML polymorphism provides simple universal types without any type annotations. For example, the polymorphic identity function, which has type $\forall\alpha.\alpha \to \alpha$:

```
let f x = x
```

However, this only provides rank-1 or prenex polymorphism, which means that all universal quantifiers ($\forall$) must appear at the out-most position (i.e. at the very beginning of the type expression). It does not support higher-rank universal types such as $(\forall\alpha.\mathtt{List}\,\alpha \to \mathtt{Int}) \to \mathtt{Int}$.

Type inference for higher-rank universal types is undecidable in general. As an illustration, consider the following OCaml function:

```
fun f x y -> f x + f y
```

This expression could have a number of System Fω types, including:

$\forall\alpha::\star.\ (\alpha \to \texttt{Int}) \to \alpha \to \alpha \to \texttt{Int}$

$\forall\alpha::\star.\forall\beta::\star.\ (\forall\gamma::\star.\ \gamma \to \texttt{Int}) \to \alpha \to \beta \to \texttt{Int}$

and none of these types is more general than the others, so we require some annotations in order to type-check programs involving higher-rank universals. The required annotations include explicit type abstraction and type application statements, as well as explicitly specifying the type of the universal created by a type abstraction.

Rather than directly using type application and type abstraction with type annotations, universal types in OCaml are provided through record types (similar support is also available through object types). Constructing the record type acts as a type abstraction statement, and destructing the record using a pattern or projection acts as a type application statement. The declaration of a record type includes specifying the types of its fields, which provide us with the required type annotations for type abstraction statements.

The following definition defines a type corresponding to $\forall\alpha.\texttt{List}\,\alpha \to \mathit{Int}$:

```
type t = { f : 'a. 'a list -> int }
```

Building a value of type `t` corresponds to a type abstraction operation:

```
let len = {f = List.length}
```

Destructing a value of type `t` using a pattern or projection corresponds to type application:

```
let g r = r.f [1; 2; 3] + r.f [1.0; 2.0; 3.0]
```

4.1.2 Functors

The most powerful form of parametricity in OCaml is provided by functors. Functors are functions that operate on modules. Since modules can contain any OCaml definition, functors can be parametric on any OCaml definition.

As an example, we will extend our `IntSet` module to a `Set` module that works uniformly on any module that matches the appropriate signature.

For convenience, we will define a module type for the signature that we expect of arguments to the functor:

```
module type Eq = sig
  type t
  val equal : t -> t -> bool
end
```

This signature says that we expect the module to contain a type `t` and a function `equal` for comparing two values of type `t` for equality.

We also create a signature of the functor's result:

```
module type SetS = sig
  type t
  type elt
  val empty : t
  val is_empty : t -> bool
  val mem : elt -> t -> bool
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val to_list : t -> elt list
end
```

Now we define our `Set` functor as follows:

```
module Set (E : Eq) : SetS with type elt := E.t = struct

  type t = E.t list

  let empty = []

  let is_empty = function
    | [] -> true
    | _ -> false

  let rec mem x = function
    | [] -> false
    | y :: rest ->
        if (E.equal x y) then true
        else mem x rest

  let add x t =
    if (mem x t) then t
    else x :: t

  let rec remove x = function
    | [] -> []
    | y :: rest ->
        if (E.equal x y) then rest
        else y :: (remove x rest)

  let to_list t = t

end
```

Note that we have specified the result of the functor to have the signature `SetS with type elt := E.t`. This signature is the same as `SetS` but with the `elt` type component removed and all occurrences of it replaced by `E.t`. If we had wanted to leave the `elt` type in the signature, but changed from an abstract type to an alias for `E.t`, we could instead have used `SetS with type elt = E.t`.

We can apply the functor to recreate `IntSet`:

```
module IntEq = struct
  type t = int
  let equal (x : int) (y : int) =
    x = y
end

module IntSet = Set(IntEq)
```

Since the type `E.t` is abstract within the body of `Set`, the implementation of `Set` can only pass these values around and compare them using `E.equals`, it cannot use values of type `E.t` in any other way because the it cannot see the type's definition. This means that the behaviour of `Set` cannot depend on the particular type used for `E.t`: it must behave uniformly on any `Eq` module that it is applied to.

4.2 Higher-kinded polymorphism

Modules and functors allow abstraction and parametricity for both types (with kind $*$) and type constructors (with kind $* \to *$). However, the existential and universal polymorphism in the core language only supports types with kind $*$. The reason for this is that type-checking higher-kinded polymorphism, requires type annotations on type application as well as type abstraction. The module system includes such annotations, but the core language avoids all such annotations as too verbose.

As an example, consider trying to type-check applications of a function with type $\forall F :: * \to *. \forall \alpha :: *. F\ \alpha \to (F\ \alpha \to \alpha) \to \alpha$ to a value with type $\texttt{List}(\texttt{Int} \times \texttt{Int})$. This involves unifying the following two types:

$$F\,\alpha \quad \sim \quad \texttt{List}(\texttt{Int} \times \texttt{Int})$$

There are many possible solutions to this unification, including:

$$\begin{array}{ll}
F = \texttt{List} & \alpha = \texttt{Int} \times \texttt{Int} \\
F = \Lambda\beta.\texttt{List}(\beta \times \beta) & \alpha = \texttt{Int} \\
F = \Lambda\beta.\texttt{List}(\texttt{Int} \times \texttt{Int}) &
\end{array}$$

None of which is more general than the others.

4.2.1 Lightweight higher-kinded polymorphism

It is possible to restrict higher-kinded polymorphism so that it can be type-checked without type annotations on type applications. By restricting the type functions which can be used to a set of functions for which each function maps to a different set of types. In other words, a set **F** of functions such that:

$$\forall F, G \in \mathbf{F}. \quad F \neq G \quad \Rightarrow \quad \forall t. F(t) \neq G(t)$$

This is how Haskell provides higher-kinded polymorphism: it only supports type constructors which create fresh types. For example, Haskell has no equivalent of the OCaml type[1]:

```
type 'a t = ('a * 'a) list
```

This restricted form of higher-kinded polymorphism can also be used in OCaml with an encoding based on defunctionalisation (Reynolds [1972]). For example, the following type definition:

```
type lst = List
type opt = Option

type ('a, 'f) app =
  | Lst : 'a list -> ('a, lst) app
  | Opt : 'a option -> ('a, opt) app
```

Allows us to represent `'a list` as `('a, lst) app` and `'a option` as `('a, opt) app`. Then we can use polymorphism in the second parameter of `app` to encode higher-kinded polymorphism over `list` and `option`. For example:

```
type 'f map = {
  map: 'a 'b. ('a -> 'b) -> ('a, 'f) app -> ('b, 'f) app;
}

let lmap : lst map = {map = fun f (Lst l) -> Lst (List.
    map f l)}
let omap : opt map = {map = fun f (Opt o) -> Opt (Option.
    map f o)}

let f : 'b map -> (int, 'b) app -> (string, 'b) app =
  fun m c -> m.map (fun x -> "Int: " ^ (string_of_int x))
      c

let l = f lmap (Lst [1; 2; 3])
let o = f omap (Opt (Some 6))
```

The `higher` library (Yallop and White [2014]), extends this idea by providing a general `app` type that any type constructor can be injected into.

[1]A Haskell type synonym can be created which looks like this OCaml definition, however it is very different because it cannot be abstracted

4.3 Parametricity in System Fω

4.3.1 Universal types

OCaml functors with abstract types in their arguments correspond to universal types in System Fω. This is not surprising: universal types are the fundamental concept of the polymorphic lambda calculus (System F), which is intended to capture the essence of parametric polymorphism.

As an example we will implement our `Set` functor using universals in System Fω.

First, for convenience, we create a type constructor and some functions for dealing with the products that represent the structures of both the argument and result structures of `Set`:

```
EqImpl =
  λγ::⋆.γ → γ → Bool;

equal = Λγ::⋆.λs:EqImpl γ.s;

SetImpl =
  λγ::⋆.λα::⋆.
      α
      × (α → Bool)
      × (γ → α → Bool)
      × (γ → α → α)
      × (γ → α → α)
      × (α → List γ);

empty = Λγ::⋆.Λα::⋆.λs:SetImpl γ α.π1 s;
is_empty = Λγ::⋆.Λα::⋆.λs:SetImpl γ α.π2 s;
mem = Λγ::⋆.Λα::⋆.λs:SetImpl γ α.π3 s;
add = Λγ::⋆.Λα::⋆.λs:SetImpl γ α.π4 s;
remove = Λγ::⋆.Λα::⋆.λs:SetImpl γ α.π5 s;
to_list = Λγ::⋆.Λα::⋆.λs:SetImpl γ α.π6 s;
```

Now we can create our implementation of sets, using Λ to give it a universal type corresponding to the type of the `Set` functor.

```
set_package =
  Λγ::⋆. λeq:EqImpl γ.
    pack List γ,
       ⟨ nil [γ],
         isempty [γ],
         λn:γ.fold [γ] [Bool]
           (λx:γ.λy:Bool.or y (equal [γ] eq n x))
           false,
         cons [γ],
         λn:γ.fold [γ] [List γ]
```

```
            (λx:γ.λl:List γ.
               if (equal [γ] eq n x) [List γ] l (cons [γ]
                  x l))
            (nil [γ]),
          λl:List γ.l ⟩
     as ∃α::*.SetImpl γ α;
```

If we look at the typing rules for universals (Section 1.3), we can see that the type parameter of Λ is a fresh type variable. As with abstract types in the parameters of OCaml functors, this means the body of `set_package` can only pass these values around and compare them using `eq`, it cannot use values of type γ in any other way because the it cannot see the type's definition.

This means that the behaviour of `set_package` cannot depend on the particular type used for γ: it must behave uniformly on any type that it is applied to.

4.3.2 Relational parametricity

We can also use the relational interpretation to give a precise meaning to the idea that universal types provide parametricity.

Given a type T with free variables $\alpha, \beta_1, \ldots, \beta_n$:

$$\begin{array}{l}\forall B_1.\ldots\forall B_n.\ \forall x : (\forall \alpha.T[\alpha, B_1, \ldots, B_n]).\\ \quad\quad \forall\gamma.\ \forall\delta.\ \forall\rho \subset \gamma \times \delta.\\ \quad\quad\quad\quad [\![T]\!][\rho, =_{B_1}, \ldots, =_{B_n}](x[\gamma],\ x[\delta])\end{array}$$

This formula can be read as: For a value x with universal type, two types γ and δ, and any way of viewing γ and δ as representing the same thing – captured by a relation ρ – $x[\gamma]$ will behave the same as $x[\delta]$ with respect to ρ.

This is the essence of parametricity: a value with universal type will behave uniformly for any types it is applied to.

The above parametricity property for universal types can be derived from identity extension applied to universals by expanding out the relational interpretation.

4.3.3 Theorems for free

In the previous chapter we applied the abstraction property to a particular kind of relation to show that abstraction guarantees code outside of a component preserves invariants. Similarly, we can apply the parametricity property to particular relations to show that parametricity guarantees code inside of a component behaves in certain ways.

For example, applying parametricity to the type $\forall\alpha.\alpha \to \alpha$, gives us the following formula:

$$\begin{array}{l}\forall f : (\forall \alpha.\alpha \to \alpha).\\ \quad \forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.\\ \quad\quad \forall u : \gamma.\ \forall v : \delta.\\ \quad\quad\quad \rho(u,\ v) \Rightarrow \rho(f[\gamma]\,u,\ f[\delta]\,v)\end{array}$$

By defining a relation is_u to represent being equal to a value $u : T$:

$$\text{is}_u(x : T, y : T) \quad = \quad (x =_T u)\ \wedge\ (y =_T u)$$

and using it to instantiate ρ, we obtain the following formula:

$$\begin{array}{l}\forall f : (\forall \alpha.\alpha \to \alpha).\\ \quad \forall \gamma.\forall u : \gamma.\\ \quad\quad \text{is}_u(u, u) \Rightarrow \text{is}_u(f[\gamma]\,u,\ f[\gamma]\,u)\end{array}$$

which can be reduced to:

$$\begin{array}{l}\forall f : (\forall \alpha.\alpha \to \alpha).\\ \quad \forall \gamma.\forall u : \gamma.\\ \quad\quad f[\gamma]\,u\ =_\gamma u\end{array}$$

This shows that any value f of type $\forall \alpha.\alpha \to \alpha$ must be the identity function. Properties like this, which use parametricity to give guarantees about the behaviour of all values of a given type, are often called free theorems after an influential paper on the subject (Wadler [1989]).

As a second example, we can apply parametricity to the type $\forall \alpha.\text{List}\,\alpha \to \text{List}\,\alpha$:

$$\begin{array}{l}\forall f : (\forall \alpha.\text{List}\,\alpha \to \text{List}\,\alpha).\\ \quad \forall \gamma.\ \forall \delta.\ \forall \rho \subset \gamma \times \delta.\\ \quad\quad \forall u : \text{List}\,\gamma.\ \forall v : \text{List}\,\delta.\\ \quad\quad\quad [\![\text{List}\,\alpha]\!][\rho](u,\ v) \Rightarrow [\![\text{List}\,\alpha]\!][\rho](f[\gamma]\,u,\ f[\delta]\,v)\end{array} \tag{4.1}$$

Applying the relational interpretation to the System F encoding of `List` gives the following equation:

$$\begin{array}{l}[\![\text{List}\,\alpha]\!][\rho] =\\ \quad (x : \text{List}\,A,\ y : \text{List}\,B).\\ \quad\quad \forall \gamma.\ \forall \delta.\ \forall \rho' \subset \gamma \times \delta.\\ \quad\quad\quad \forall n : \gamma.\ \forall m : \delta.\\ \quad\quad\quad \forall c : A \to \gamma \to \gamma.\ \forall d : B \to \delta \to \delta.\\ \quad\quad\quad\quad \rho'(n,\ m) \Rightarrow\\ \quad\quad\quad\quad\quad (\forall a : A.\ \forall b : B.\ \forall u : \gamma.\ \forall v : \delta.\\ \quad\quad\quad\quad\quad\quad \rho(a, b) \Rightarrow \rho'(u, v) \Rightarrow \rho'(c\,a\,u,\ d\,b\,v)) \Rightarrow\\ \quad\quad\quad\quad\quad\quad\quad \rho'(x[\gamma]\,n\,c,\ y[\delta]\,m\,d)\end{array}$$

which can be reduced to a more manageable form by considering the cases of *nil* and *cons* separately:

$$\llbracket \mathrm{List}\,\alpha \rrbracket[\rho](x : \mathrm{List}\,A,\ y : \mathrm{List}\,B) = \\ \begin{cases} \rho(i,j)\ \wedge\ \llbracket \mathrm{List}\,\alpha \rrbracket[\rho](l,k), & x = cons_A\, i\, l \wedge y = cons_B\, j\, k \\ true, & x = nil_A \wedge y = nil_B \\ false, & otherwise \end{cases} \tag{4.2}$$

By defining a family of relations $\langle g \rangle$ to represent functions $g : A \to B$:

$$\langle g \rangle(x : A, y : B) \quad = \quad (g\, x =_B y)$$

and substituting it for ρ in 4.2, we obtain the following equation:

$$\llbracket \mathrm{List}\,\alpha \rrbracket[\langle g \rangle](x : \mathrm{List}\,A,\ y : \mathrm{List}\,B) = \\ \begin{cases} g\ i =_B j\ \wedge\ \llbracket \mathrm{List}\,\alpha \rrbracket[\langle g \rangle](l,k), & x = cons_A\, i\, l \wedge y = cons_B\, j\, k \\ true, & x = nil_A \wedge y = nil_B \\ false, & otherwise \end{cases}$$

This recursive equation may look quite familiar. It bears a striking resemblance to the definition of the standard `map` function for lists. In fact, we can use `map` to rewrite it as:

$$\llbracket \mathrm{List}\,\alpha \rrbracket[\langle g \rangle](x : \mathrm{List}\,A,\ y : \mathrm{List}\,B) = \\ \mathrm{map}[A][B]\, g\, x =_B y \tag{4.3}$$

Substituting 4.3 into 4.1 we get a free theorem for $\forall\alpha.\mathrm{List}\,\alpha \to \mathrm{List}\,\alpha$:

$$\begin{array}{l} \forall f : (\forall\alpha.\mathrm{List}\,\alpha \to \mathrm{List}\,\alpha). \\ \quad \forall\gamma.\ \forall\delta.\ \forall g : \gamma \to \delta \\ \quad\quad \forall u : \mathrm{List}\,\gamma.\ \forall v : \mathrm{List}\,\delta. \\ \quad\quad\quad \mathrm{map}[\gamma][\delta]\, g\, (f[\gamma]\, u) = f[\delta]\, (\mathrm{map}[\gamma][\delta]\, g\, u) \end{array}$$

An interesting corollary of this theorem is that any function f of type $\forall\alpha.\mathrm{List}\,\alpha \to \mathrm{List}\,\alpha$ is a "rearrangement" function: the output of f is a list whose elements all come from the input to f.

4.4 Practical limitations

The previous chapter and this one have described how abstraction and parametricity can give guarantees about program behaviour. However, these guarantees rely on some assumptions that do not necessarily hold in real programming languages.

4.4.1 Side-effects

In a pure language such as System Fω, a function accepts an input value and produces an output value. The relation between inputs and outputs completely defines the behaviour of a function. However, programming languages are not generally pure, they allow functions to perform side-effects. Such side-effects include:

- Printing to the console
- Raising exceptions
- Failing to terminate

Side-effects affect the guarantees that are provided by abstraction and parametricity. For example, we showed that parametricity ensures that all functions of type $\forall\alpha.\alpha \to \alpha$ are the identity function. However, the following three OCaml functions have that type and are not the identity function:

```
let f (x : 'a) : 'a =
  Printf.printf "Launch missiles\n";
  x

let f (x : 'a) : 'a = raise Exit

let rec f (x : 'a) : 'a = f x
```

The issue here is that the function type in OCaml is a different type from the function type in System F – since it supports side-effects. This means that the relational interpretation for OCaml's function type is different from the relational interpretation that we gave for System F. The relational interpretation for OCaml's function type should ensure not only that related inputs produce related outputs, but that the side-effects of the function preserve relations.

This has a number of consequences in practice:

- When replacing one implementation with another, we must ensure that the side-effects of the two implementations are also equivalent.
- When using abstraction to preserve an invariant, we must also ensure that the side effects preserve that invariant.
- When interpreting a "free theorem" we must consider the possible affect of side-effects on a function's behaviour.

For example, whilst we do not know that all functions of type $\forall\alpha.\alpha \to \alpha$ are the identity function, we do know that if such a function returns a value then it will be equal to the function's input.

4.4.2 Non-parametric values

Abstraction and parametricity only hold if there are no non-parametric polymorphic functions available in the environment.

For example, applying parametricity to the type $\forall\alpha.\alpha \to \alpha \to \text{Bool}$ gives us the following formula:

$$\begin{aligned}
&\forall f : (\forall\alpha.\alpha \to \alpha \to \text{Bool}).\\
&\quad\forall\gamma.\ \forall\delta.\ \forall\rho \subset \gamma \times \delta.\\
&\qquad\forall u : \gamma.\ \forall v : \delta.\ \forall u' : \gamma.\ \forall v' : \delta.\\
&\qquad\quad\rho(u,\ v) \Rightarrow \rho(u',\ v') \Rightarrow\\
&\qquad\qquad(f[\gamma]\,u\,u' =_{Bool} f[\delta]\,v\,v')
\end{aligned}$$

If we instantiate ρ with the trivial relation that is always true, then we get the following free theorem:

$$\begin{aligned}
&\forall f : (\forall\alpha.\alpha \to \alpha \to \text{Bool}).\\
&\quad\forall\gamma.\ \forall\delta.\\
&\qquad\forall u : \gamma.\ \forall v : \delta.\ \forall u' : \gamma.\ \forall v' : \delta.\\
&\qquad\quad(f[\gamma]\,u\,u' =_{Bool} f[\delta]\,v\,v')
\end{aligned}$$

This means that any parametric function of type $\forall\alpha.\alpha \to \alpha \to \text{Bool}$ must be a constant function: a function that returns the same value for all inputs.

However, OCaml provides the following built-in structural equality function:

```
val (=) : 'a -> 'a -> bool
```

Even though it has this type, it is not a constant function. This means that it is not parametric.

The existence of this function and several similar ones in OCaml, can break the guarantees provided by abstraction and parametricity. However, not all guarantees are broken, for instance the preservation of invariants is not affected by any of the built-in functions available in OCaml.

In practice, this means that such functions should only be used at known types. Using them on abstract types produces non-parametric code whose correctness may rely on details of an implementation that are not exposed in its interface.