Chapter 6

Abstraction and parametricity

Type structure is a syntactic discipline for maintaining levels of abstraction – John Reynolds, “Types, Abstraction and Parametric Polymorphism”

6.1 Abstraction

Abstraction, also known as information hiding, is fundamental to computer science. When faced with creating and maintaining a complex system, the interactions of different components can be simplified by hiding the details of each component’s implementation from the rest of the system.

Details of a component’s *implementation* are hidden by protecting it with an *interface*. An interface describes the information which is exposed to other components in the system. Abstraction is maintained by ensuring that the rest of the system is invariant to changes of implementation that do not affect the interface.

6.1.1 Modules

The most powerful form of abstraction in OCaml is achieved using the *module system*. The module system is basically its own language within OCaml, consisting of modules and module types. All OCaml definitions (e.g. values, types, exceptions, classes) live within modules, so the module system’s support for abstraction includes support for abstraction of any OCaml definition.

Structures

A structure creates a module from a collection of OCaml definitions. For example, the following defines a module with the definitions of a simple implementation of a set of integers:

```
module IntSet = struct

  type t = int list

  let empty = []

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

  let equal_member (x : int) (y : int) =
    x = y

  let rec mem x = function
    | [] -> false
    | y :: rest ->
        if (equal_member 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 (equal_member x y) then rest
        else y :: (remove x rest)

  let to_list t = t

end
```

The module `IntSet` uses lists of integers to represent sets of integers. This is indicated by the inclusion of a type `t` defined as an alias to `int list`. The implementation provides the basic operations of sets as a collection of functions that operate on these `int lists`.

The components of a structure are accessed using the `.` operator. For example, the following creates a set containing 1, 2 and 3.

```
let one_two_three : IntSet.t =
  IntSet.add 1 (IntSet.add 2 (IntSet.add 3 IntSet.empty))
```

A structure's components can also be made available using `open` to avoid needing to repeatedly use the `.` operator:

```
open IntSet
```

```
let one_two_three : t =
  add 1 (add 2 (add 3 empty))
```

There is also a scoped opening syntax to temporarily make a structure's components available without the `.` operator:

```
let one_two_three : IntSet.t =
  IntSet.(add 1 (add 2 (add 3 empty)))
```

Structures can be built from other structures using `include`. For example, we can build a structure containing all the components of `IntSet` as well as a `singleton` function:

```
module IntSetPlus = struct
  include IntSet

  let singleton x = add x empty
end
```

Signatures

Signatures are interfaces for structures. They are a kind of module type, and the most general signature is automatically inferred for a structure definition. The signature inferred for our `IntSet` structure is as follows:

```
sig
  type t = int list
  val empty : 'a list
  val is_empty : 'a list -> bool
  val equal_member : int -> int -> bool
  val mem : int -> int list -> bool
  val add : int -> int list -> int list
  val remove : int -> int list -> int list
  val to_list : 'a -> 'a
end
```

We can use a signature to hide components of the structure, and also to expose a component with a restricted type. For example, we can remove the `equal_member` function, and restrict `empty`, `is_empty` and `to_list` to only operate on `int lists`:

```
module IntSet : sig
  type t = int list
  val empty : int list
  val is_empty : int list -> bool
  val mem : int -> int list -> bool
  val add : int -> int list -> int list
  val remove : int -> int list -> int list
  val to_list : int list -> int list
end = struct
  ...
end
```

For convenience, we can name the signature using a `module type` declaration:

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

module IntSet : IntSetS = struct
  ...
end
```

Abstract types

The above definition of `IntSet` still exposes the fact that our sets of integers are represented using `int list`. This means that code outside of the module may rely on the fact that our sets are lists of integers. For example,

```
let print_set (s : IntSet.t) : unit =
  let rec loop = function
    | x :: xs -> print_int x; print_string " "; loop xs
    | [] -> ()
  in
    print_string "{ ";
    loop s;
    print_string "}"
```

Such code is correct, but it will break if we later decide to use a different representation for our sets of integers.

In order to prevent this, we must make the type alias `IntSet.t` into an *abstract type*, by hiding its definition as an alias of `int list`. This gives us the following definition:

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

module IntSet : IntSetS = struct
  ...
end
```

Observe that we also change `int list` in the types of the functions to `t` (except for the result of `to_list`).

Now that the type is abstract, code outside of `IntSet` can only pass the set values around and use the functions in `IntSet` to create new ones, it cannot use values of type `IntSet.t` in any other way because the it cannot see the type's definition.

This means that the implementation of `IntSet` can be replaced with a more efficient one (perhaps based on binary trees), safe in the knowledge that the change will not break any code outside of `IntSet`.

Compilation Units

In OCaml, every source file defines a structure (e.g. "foo.ml" defines a module named `Foo`). The signature for these modules is defined in a corresponding interface file (e.g. "foo.mli" defines the signature of the `Foo` module). Note that all such compilation units in a program must have a unique name.

6.1.2 Abstraction in System Fω

The abstract types in OCaml's module system correspond to existential types in System Fω. Just like abstract types, existentials can pack together operations on a shared type, without exposing the definition of that type. As an example we will implement our `IntSet` with an abstract type using existentials in System Fω. For convenience, we will use natural numbers instead of integers and use simpler, less efficient, implementations of the set operations.

First, we create a type constructor and some functions for dealing with the products that represent the structures we are implementing:

```
NatSetImpl =
  λα::*.
      α
      × (α → Bool)
      × (Nat → α → Bool)
      × (Nat → α → α)
      × (Nat → α → α)
      × (α → List Nat);

empty = Λα::*.λs:NatSetImpl α.π₁ s;
is_empty = Λα::*.λs:NatSetImpl α.π₂ s;
```

```
mem = Λα::*.λs:NatSetImpl α.π₃ s;
add = Λα::*.λs:NatSetImpl α.π₄ s;
remove = Λα::*.λs:NatSetImpl α.π₅ s;
to_list = Λα::*.λs:NatSetImpl α.π₆ s;
```

Now we can create our implementation of sets of naturals, and give it the type corresponding to the abstract `IntSet` signature using `pack`:

```
nat_set_package =
  pack List Nat,⟨
        nil [Nat],
         isempty [Nat],
         λn:Nat.fold [Nat] [Bool]
           (λx:Nat.λy:Bool.or y (equal_nat n x))
           false,
         cons [Nat],
         λn:Nat.fold [Nat] [List Nat]
           (λx:Nat.λl:List Nat
              if (equal_nat n x) [List Nat] l (cons [Nat] x l))
           (nil [Nat]),
         λl:List Nat.l ⟩
  as ∃α::*.NatSetImpl α;
```

By opening `nat_set_package` as `nat_set` in the environment using `open`

```
open nat_set_package as NatSet, nat_set;
```

we are able to write `one_two_three` in System Fω:

```
one_two_three =
  (add [NatSet] nat_set) one
    ((add [NatSet] nat_set) two
      ((add [NatSet] nat_set) three
        (empty [NatSet] nat_set)));
```

If we look at the typing rules for existentials (Section 2.3.1), we can see that the type which is packed (`List Nat`) is not present in the type of the package ($\exists\alpha$`::*.NatSetImpl` α) – it is replaced by a fresh type variable (α). As with OCaml's abstract types, this means code outside of `nat_set_package` can only pass the set values around and use the functions in `nat_set_package` to create new ones, it cannot use values of type α in any other way, because the it cannot see the type's definition.

This means that we can replace `nat_set_impl` with a more efficient implementation, safe in the knowledge that the change will not break code using `nat_set_package`.

6.1.3 Existential types in OCaml

We have seen that OCaml's module system provides abstraction for all OCaml definitions. This includes abstract types, which are closely related to existential types in System Fω. However, OCaml also provides more direct support for existential types within its core language. This can sometimes be more convenient than using the module system, which is quite verbose, but only works for types of kind `*`.

Type inference for general existential types is undecidable. As an illustration, consider the following OCaml function:

```
fun p x y -> if p then x else y
```

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

$$\forall\alpha::*.\ \text{Bool} \to \alpha \to \alpha \to \alpha$$

$$\forall\alpha::*.\forall\beta::*.\ \text{Bool} \to \alpha \to \beta \to \exists\gamma::*.\gamma$$

and none of these types is more general than the rest, so we require some annotations in order to type-check programs involving existentials. The required annotations include explicit `pack` and `open` statements, as well as explicitly specifying the type of the existential created by a `pack` statement.

Rather than directly using `open` and `pack` with type annotations, existential types in OCaml are provided through sum types. The constructors of the sum type act as `pack` statements in expressions, and `open` statements in patterns. The declaration of a sum type includes specifying the types of its constructors arguments, which provide us with the required type annotations for `pack` statements.

The following definition defines a type corresponding to $\exists\alpha.\alpha \times (\alpha \to \alpha) \times (\alpha \to \text{string})$:

```
type t = E : 'a * ('a -> 'a) * ('a -> string) -> t
```

Building a value using the `E` constructor corresponds to the `pack` operation of System Fω:

```
let ints = E(0, (fun x -> x + 1), string_of_int)
let floats = E(0.0, (fun x -> x +. 1.0), string_of_float)
```

Destructing a value using the `E` constructor with `let` or `match` corresponds to the `open` operation of System Fω:

```
let E(z, s, p) = ints in
  p (s (s z))
```

6.2 Parametricity

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.

6.2.1 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.

6.2.2 Parametricity in System Fω

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.π_1 s;
is_empty = Λγ::*.Λα::*.λs:SetImpl γ α.π_2 s;
mem = Λγ::*.Λα::*.λs:SetImpl γ α.π_3 s;
add = Λγ::*.Λα::*.λs:SetImpl γ α.π_4 s;
remove = Λγ::*.Λα::*.λs:SetImpl γ α.π_5 s;
to_list = Λγ::*.Λα::*.λs:SetImpl γ α.π_6 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 2.3), 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.

6.2.3 Universal types in OCaml

In addition to supporting universal types through abstract types in functor arguments, OCaml also provides more direct support for universal types within its core language. This is more convenient than using the module system, which is quite verbose, but only works for types of kind `*`.

ML polymorphism provides simple universal types. As discussed in Chapter 3, ML polymorphism can be inferred without any type annotations. For example, the polymorphic identity function, which has type $\forall\alpha.\alpha \rightarrow \alpha$:

```
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 $(\forall\alpha.\texttt{List}\alpha \rightarrow \texttt{Int}) \rightarrow \texttt{Int}$.

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:

```
∀α::*. (α → Int) → α → α → Int

∀α::*.∀β::*. (∀γ::*. γ → Int) → α → β → Int
```

and none of these types is more general than the rest, 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 $\forall\alpha.\texttt{List}\,\alpha \rightarrow \mathit{Int}$:

```
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]
```

6.3 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 $\mathtt{List}(\mathtt{Int} \times \mathtt{Int})$. This involves unifying the following two types:

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

There are many possible solutions to this unification, including:

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

None of which is more general than the others.

6.3.1 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 $\mathbf{F}$ 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
```

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

```
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.

6.4 Relational abstraction and relational parametricity

In the previous sections we have been quite loose in our description of abstraction and parametricity. We have talked about abstraction as invariance under change of implementation, and parametricity as uniformity of behaviour, but we have not made these notions precise.

We can give precise descriptions of parametricity and abstraction using relations between types. To keep things simple we will restrict ourselves to System F for this discussion.

6.4.1 Changing set implementations

We have talked about abstraction in terms of a system being invariant under a change of a component's implementation that does not affect the component's interface. In order to give a precise definition to abstraction, we must consider what it means to change a component's implementation without affecting its interface.

For example, consider the interface for sets of integers in Fig. 6.1. This is a reduced version of the set interface used earlier in the chapter, with the addition of the `if_empty` function. The `if_empty` function takes a set and two values as arguments, if the set is empty it returns the first argument, otherwise it returns the second argument.

```
type t

val empty : t

val is_empty : t -> bool

val mem : t -> int -> bool

val add : t -> int -> t

val if_empty : t -> 'a -> 'a -> 'a
```

Figure 6.1: A set interface

Two implementations of this interface are shown in Fig. 6.2 and Fig. 6.3–one based on lists, the other based on ordered trees.

We would like to know that swapping one implementation with the other will not affect the rest of our program. In other words, how do we show that switching between these implementations will not affect the interface?

Relations between types

If the `t` types in our two implementations both represent sets then there must be some relation between these that describes how sets in one representation can be represented in the other representation.

In other words, given a set represented as a list and a set represented as a tree there must be a relation that tells us if they represent the same set. For example, the list `[]` and the tree `Empty` both represent the empty set. Similarly the lists `[1; 2]` and `[2; 1]`, and the trees `Node(Node(Empty, 1, Empty), 2, Empty)` and `Node(Empty, 1, Node(Empty, 2, Empty))` all represent a set containing `1` and `2`.

Throughout this chapter we shall use relations of the following form:

$$(x : A, y : B).\phi[x, y]$$

where A and B are System F types, and $\phi[x, y]$ is a logical formula involving x and y.

We will not overly concern ourselves with the particular choice of logic used for these formulae, but we will assume the existence of certain kinds of term:

- We will assume the that we have basic logical connectives:

$$\phi ::= \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi$$

- We will assume that we have universal quantification over terms, types and relations:

$$\phi ::= \forall x : A.\phi \mid \forall \alpha.\phi \mid \forall R \subset A \times B.\phi$$

```
type t_list = int list

let empty_list = []

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

let rec mem_list x = function
  | [] -> false
  | y :: rest ->
      if x = y then true
      else mem_list x rest

let add_list x t =
  if (mem_list x t) then t
  else x :: t

let if_empty_list t x y =
  match t with
  | [] -> x
  | _ -> y
```

Figure 6.2: List implementation of the set interface

```
type t_tree =
     | Empty
     | Node of t_tree * int * t_tree

let empty_tree = Empty

let is_empty_tree = function
   | Empty -> true
   | _ -> false

let rec mem_tree x = function
   | Empty -> false
   | Node(l, y, r) ->
        if x = y then true
        else if x < y then mem_tree x l
        else mem_tree x r

let rec add_tree x t =
     match t with
     | Empty -> Node(Empty, x, Empty)
     | Node(l, y, r) as t ->
          if x = y then t
          else if x < y then Node(add_tree x l, y, r)
          else Node(l, y, add_tree x r)

let if_empty_tree t x y =
   match t with
   | Empty -> x
   | _ -> y
```

Figure 6.3: Tree implementation of the set interface

and similarly for existential quantification:

$$\phi \ ::= \ \exists x : A.\phi \mid \exists \alpha.\phi \mid \exists R \subset A \times B.\phi$$

- We will assume that we can apply a relation to terms:

$$\phi ::= R(t, u)$$

- We will assume that we have equality on terms at a given type:

$$\phi ::= (t =_A u)$$

  which represents the equational theory of System F (e.g. beta equalities, eta equalities).

In the case of our set implementations we leave the precise logical formula as an exercise for the reader, and will simply refer to the relation as σ.

Relations between values

To show that the values in our implementations implement the same interface, we must show that they have the same behaviour in terms of the sets being represented. For each value, this equivalence of behaviour can be represented by a relation. Considering each of the values in our set interface in turn:

`empty` The `empty` values of our implementations behave the same if they represent the same set. More precisely:

$$\sigma(\text{empty}_{list}, \text{empty}_{tree})$$

where σ is the relation between sets as lists and sets as trees.

`is_empty` The `is_empty` values behave the same if they agree about which sets are empty. They should return `true` on the same sets and `false` on the same sets. More precisely:

$$\begin{array}{l} \forall x : t_{list}. \forall y : t_{tree}. \\ \quad \sigma(x, y) \Rightarrow (\text{is\_empty}_{list}\, x \ = \ \text{is\_empty}_{tree}\, y) \end{array}$$

`mem` The `mem` values behave the same if they agree about which integers are members of which sets. Their results should be the equivalent when given the same sets and integers. More precisely:

$$\begin{array}{l} \forall x : t_{list}. \forall y : t_{tree}. \forall i : \mathit{Int}. \forall j : \mathit{Int}. \\ \quad \sigma(x, y) \Rightarrow (i = j) \Rightarrow (\text{mem}_{list}\, x\, i \ = \ \text{mem}_{tree}\, y\, j) \end{array}$$

`add` The relation for `add` values is similar to that for `mem` values, except that instead of requiring that the results be equivalent we require that they represent the same set:

$$\begin{aligned}&\forall x : t_{list}.\, \forall y : t_{tree}.\, \forall i : \mathit{Int}.\, \forall j : \mathit{Int}.\\ &\quad \sigma(x, y) \Rightarrow (i = j) \Rightarrow \sigma(\mathrm{add}_{list}\, x\, i,\ \mathrm{add}_{tree}\, y\, j)\end{aligned}$$

`if_empty` The relation for `if_empty` is more complicated than the others. We might be tempted to use the relation:

$$\begin{aligned}&\forall \gamma.\, \forall \delta.\\ &\quad \forall x : t_{list}.\, \forall y : t_{tree}.\, \forall a : \gamma.\, \forall b : \gamma.\, \forall c : \delta.\, \forall d : \delta.\\ &\qquad \sigma(x, y) \Rightarrow (a = c) \Rightarrow (b = d) \Rightarrow\\ &\qquad\quad (\mathrm{if\_empty}_{list}\, x\, a\, b\ =\ \mathrm{if\_empty}_{tree}\, y\, c\, d)\end{aligned}$$

which would ensure that the behaviour was the same for calls like:

```
if_empty t 5 6
```

where `t` is a value representing a set. However, it would not ensure equivalent behaviour for calls such as:

```
if_empty t t (add t 1)
```

where the second and third arguments are also sets. In this case, we do not want to guarantee that our `if_empty` implementations will produce *equivalent* sets when given *equivalent* inputs, since a set represented as a list will never be equivalent to a set represented as a tree. Instead we would like to guarantee that our implementations will produce *related* results when given *related* inputs. This leads us to the much stronger relation:

$$\begin{aligned}&\forall \gamma.\, \forall \delta.\, \forall \rho \subset \gamma \times \delta.\\ &\quad \forall x : t_{list}.\, \forall y : t_{tree}.\, \forall a : \gamma.\, \forall b : \gamma.\, \forall c : \delta.\, \forall d : \delta.\\ &\qquad \sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow\\ &\qquad\quad \rho(\mathrm{if\_empty}_{list}\, x\, a\, b,\ \mathrm{if\_empty}_{tree}\, y\, c\, d)\end{aligned}$$

that must be satisfied by our implementations of `if_empty`. This condition ensures that all relations will be preserved by `if_empty`, including equality between integers and σ between sets.

The existence of the relation σ along with demonstrations that each of the five relations above hold, is sufficient to demonstrate that our two implementations implement the same interface. One can safely be replaced by the other, without affecting any of the other components in the system. By generalising this approach we can produce a precise definition of abstraction.

val empty:	
`t`	$\sigma(\text{empty}_{list},\ \text{empty}_{tree})$
val is_empty:	
`t -> bool`	$\forall x : t_{list}. \forall y : t_{tree}.$ $\sigma(x, y) \Rightarrow (\text{is\_empty}_{list}\, x \ = \ \text{is\_empty}_{tree}\, y)$
val mem:	
`t -> int -> bool`	$\forall x : t_{list}. \forall y : t_{tree}. \forall i : \mathit{Int}. \forall j : \mathit{Int}.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\text{mem}_{list}\, x\, i \ = \ \text{mem}_{tree}\, y\, j)$
val add:	
`t -> int -> t`	$\forall x : t_{list}. \forall y : t_{tree}. \forall i : \mathit{Int}. \forall j : \mathit{Int}.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $\sigma(\text{add}_{list}\, x\, i,\ \text{add}_{tree}\, y\, j)$
val if_empty:	
`t -> 'a -> 'a -> 'a`	$\forall \gamma. \forall \delta. \forall \rho \subset \gamma \times \delta.$ $\forall x : t_{list}. \forall y : t_{tree}. \forall a : \gamma. \forall b : \gamma. \forall c : \delta. \forall d : \delta.$ $\sigma(x, y) \Rightarrow \rho(a,\ c) \Rightarrow \rho(b,\ d) \Rightarrow$ $\rho(\text{if\_empty}_{list}\, x\, a\, b,\ \text{if\_empty}_{tree}\, y\, c\, d)$

Figure 6.4: Types and relations for the set interface

6.4.2 Relational substitution

The table in Fig. 6.4 compare the types of each of the values in our set interface with the relations that they must satisfy. From this we can see that the type of the value completely determines the relation:

- Every `t` in the type produces as σ in the relation.
- Every free type variable (e.g. `int`, `bool`) in the type produces an equality in the relation.
- Every `->` in the type produces an implication in the relation.
- Every universal quantification over types in the type produces a universal quantification over relations in the relation.

We can represent this translation of types into relations, as a substitution of relations for type variables.

Given a type T with free variables $\vec{\alpha} = \alpha_1, \dots, \alpha_n$ and relations $\vec{\rho} = \rho_1 \subset A_1 \times B_1, \dots, \rho_n \subset A_n \times B_n$, we define the relation $T[\vec{\rho}] \subset T[\vec{A}] \times T[\vec{B}]$ as follows:

- if T is α_i then $T[\vec{\rho}] = \rho_i$

- if T is $T' \times T''$ then

$$\begin{array}{rcl} T[\vec{\rho}] & = & (x : T[\vec{A}],\ y : T[\vec{B}]). \\ & & \quad T'[\vec{\rho}](fst(x),\ fst(y)) \\ & & \quad \wedge\ T''[\vec{\rho}](snd(x),\ snd(y)) \end{array}$$

- if T is $T' + T''$ then

$$\begin{array}{rcl} T[\vec{\rho}] & = & (x : T[\vec{A}],\ y : T[\vec{B}]). \\ & & \quad \exists u' : T'[\vec{A}].\ \exists v' : T'[\vec{B}]. \\ & & \qquad x = inl(u')\ \wedge\ y = inl(v') \\ & & \qquad \wedge\ T'[\vec{\rho}](u',\ v') \\ & & \vee \\ & & \quad \exists u'' : T''[\vec{A}].\ \exists v'' : T''[\vec{B}]. \\ & & \qquad x = inr(u'')\ \wedge\ y = inr(v'') \\ & & \qquad \wedge\ T''[\vec{\rho}](u'',\ v'') \end{array}$$

- if T is $T' \to T''$ then

$$\begin{array}{rcl} T[\vec{\rho}] & = & (f : T[\vec{A}],\ g : T[\vec{B}]). \\ & & \quad \forall u : T'[\vec{A}].\ \forall v : T'[\vec{B}]. \\ & & \qquad T'[\vec{\rho}](u,\ v) \Rightarrow T''[\vec{\rho}](f\,u,\ g\,v) \end{array}$$

- if T is $\forall\beta.T'$ then

$$\begin{array}{rcl} T[\vec{\rho}] & = & (x : T[\vec{A}],\ y : T[\vec{B}]). \\ & & \quad \forall\gamma.\ \forall\delta.\ \forall\rho' \subset \gamma \times \delta. \\ & & \qquad T'[\vec{\rho}, \rho'](x[\gamma],\ y[\delta]) \end{array}$$

- if T is $\exists\beta.T'$ then

$$\begin{array}{rcl} T[\vec{\rho}] & = & (x : T[\vec{A}],\ y : T[\vec{B}]). \\ & & \quad \exists\gamma.\ \exists\delta.\ \exists\rho' \subset \gamma \times \delta. \\ & & \qquad \exists u : T'[\vec{A}, \gamma].\ \exists v : T'[\vec{B}, \delta]. \\ & & \qquad\quad x = \text{pack } \gamma,\ u \text{ as } T[\vec{A}] \\ & & \qquad\quad \wedge\ y = \text{pack } \delta,\ v \text{ as } T[\vec{B}] \\ & & \qquad\quad \wedge\ T'[\vec{\rho}, \rho'](u,\ v) \end{array}$$

Using this substitution, the relation that our two set implementations must satisfy to show that they are implementing the same interface can be written:

$$\begin{aligned}&(\alpha\\ &\ \times (\alpha \to \gamma)\\ &\ \times (\alpha \to \beta \to \gamma)\\ &\ \times (\alpha \to \beta \to \alpha)\\ &\ \times (\forall \delta.\, \alpha \to \delta \to \delta \to \delta))[\sigma, =_{\text{Int}}, =_{\text{Bool}}](\text{set}_{list},\, \text{set}_{tree})\end{aligned}$$

where set_{list} and set_{tree} are products containing the implementations of set using lists and trees respectively.

6.4.3 Identity extension

The relational substitution can be thought of as representing equality between values of that type under the assumption that the substituted relations represent equality for values of the free type variables.

In particular, given a type T with free variables $\alpha_1, \dots, \alpha_n$, if we substitue equality relations (=) for a type's free variables we get the equality relation of that type:

$$\begin{aligned}&\forall \alpha_1. \dots \forall \alpha_n. \forall x : T. \forall y : T.\\ &\qquad (x =_T y) \quad \Leftrightarrow \quad T[=_{\alpha_1}, \dots, =_{\alpha_n}](x,\, y)\end{aligned}$$

This property of the relational substitution is known as *identity extension*.

6.4.4 A relational definition of abstraction

Using the relational substitution we can now give a precise meaning to the idea that existential types provide abstraction.

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

$$\forall \beta_1. \dots \forall \beta_n. \forall x : (\exists \alpha. T). \forall y : (\exists \alpha. T).$$

$$x = y \quad \Leftrightarrow \quad \begin{aligned}&\exists \gamma.\ \exists \delta.\ \exists \sigma \subset \gamma \times \delta.\\ &\quad \exists u : T[\gamma, \beta_1, \dots \beta_n].\ \exists v : T[\delta, \beta_1, \dots \beta_n].\\ &\qquad x = \text{pack}\ \gamma,\ u\ \text{as}\ T[\overrightarrow{A}]\\ &\qquad \wedge\ y = \text{pack}\ \delta,\ v\ \text{as}\ T[\overrightarrow{B}]\\ &\qquad \wedge\ T[\sigma, =_{\beta_1}, \dots, =_{\beta_n}](u,\, v)\end{aligned}$$

This formula can be read as: For two values x and y with existential type, if there is a way to view their implementation types (γ and δ) as representing the same thing – captured by the relation σ – and their implementations (u and v) behave the same with respect to σ, then x and y are equal: they will behave the same in all contexts.

This is the essence of abstraction: if two implementations behave the same with respect to some relation, then once they have been packed into an existential type they are indistinguishable.

The above abstraction property for existential types can be derived from identity extension applied to existentials by expanding out the relational substitution.

6.4.5 A relational definition of parametricity

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

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

$$\begin{array}{l}\forall\beta_1. \dots \forall\beta_n. \forall x : (\forall\alpha.T). \\ \quad \forall\gamma. \forall\delta. \forall\rho \subset \gamma \times \delta. \\ \qquad T[\rho, =_{\beta_1}, \dots, =_{\beta_n}](x[\gamma], x[\delta])\end{array}$$

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 substitution.

6.5 Invariants

Now that we have a precise description of abstraction, we can talk about the implications of abstraction beyond the ability to replace one implementation with another. In particular, abstraction allows us to preserve invariants on types, allowing types to represent more than just the representation of data.

Consider the following module:

```
module Positive : sig
  type t
  val zero : t
  val succ : t -> t
  val to_int : t -> int
end = struct
  type t = int
  let zero = 0
  let succ x = x + 1
  let to_int x = x
end
```

Here the abstract type `t` is represented by an `int`. However, we can also show that, thanks to abstraction, all values of type `t` will be positive integers[2].

Informally, this is because all values of type `t` must be created using either `zero` (which is a positive integer), or `succ` (which returns a positive integer when given a positive integer), so all values of type `t` must be positive integers.

The ability for types to represent invariants beyond their particular data representation is a key difference between languages with abstraction (e.g. System F) and languages without it (e.g the simply typed lambda calculus). It fundementally changes the notion of what a type is, greatly increasing their utility as a programming tool.

6.5.1 Preserving invarints

We can represent an invariant $\phi[x]$ on a type γ as a relation $\rho \subset \gamma \times \gamma$:

$$\rho(x : \gamma,\ y : \gamma) \quad = \quad (x = y)\ \wedge\ \phi[x]$$

Using this representation, $T[\rho](u, u)$ holds for some value u of type $T[\gamma]$ iff u preserves the invariant ϕ on type γ.

Given

- a type T with free variable α
- a type γ
- a value u of type $T[\gamma]$
- an expression E with free variable x such that if x has type β then E also has type β

it can be shown from the abstraction property of existentials that:

$$\begin{array}{l}\forall \rho \subset \gamma \times \gamma. \quad T[\rho](u, u) \Rightarrow \\ \qquad \rho\left(\begin{array}{l}\texttt{open (pack } \gamma,\ u\texttt{as } \exists\gamma.\ T[\gamma]\texttt{) as } x,\ \gamma \texttt{ in } E, \\ \qquad\qquad\qquad \texttt{open (pack } \gamma,\ u\texttt{as } \exists\gamma.\ T[\gamma]\texttt{) as } x,\ \gamma \texttt{ in } E\end{array}\right)\end{array}$$

This means that if u preserves an invariant on type γ – represented by relation ρ – then if u is packed up into an existential type the invariant will hold for any value of (the now abstract) type γ that are created from u.

In other words, if we can show that an implementation of an interface preserves an invariant on an abstract type, then that invariant holds for all values of that abstract type in the program.

[2]We ignore overflow for the sake of simplicity

6.5.2 Phantom types

The `Positive.t` type in the earlier example represented not just the data representation (an integer) but also an invaraint (that the integer be positive). Using higher-kinded types we can take the idea of types as invariants even further: we can create types for which there is no data representation – they only represent invariants.

Consider the following file I/O interface:

```
module File : sig
  type t
  val open_readwrite : string -> t
  val open_readonly : string -> t
  val read : t -> string
  val write : t -> string -> unit
end = struct
  type t = int
  let open_readwrite filename = ...
  let open_readonly filename = ...
  let read f = ...
  let write f s = ...
end
```

It allows files to be opened in either read-only or read-write mode, and it provides functions to read from and write to these files.

This interface is that it does not prevent you from trying to write to a file which was opened read-only. Instead, such attempts result in a *run-time* error:

```
# let f = File.open_readonly "foo" in
    File.write f "bar";;

Exception: Invalid_argument "write: file is read-only".
```

This is unfortunate, since such errors could easily be caught at *compile-time*, giving us more confidence in the correctness of our programs.

To detect these errors at compile-time we add a type parameter to the `File.t` type, which represents whether the file was opened in read-only or read-write mode. Each mode is represented by a type without a definition (`readonly` and `readwrite`). These types have no data representation – they only exist to represent invariants:

```
module File : sig
  type readonly
  type readwrite
  type 'a t
  val open_readwrite : string -> readwrite t
  val open_readonly : string -> readonly t
  val read : 'a t -> string
  val write : readwrite t -> string -> unit
```

```
end = struct
  type readonly
  type readwrite
  type 'a t = int
  let open_readwrite filename = ...
  let open_readonly filename = ...
  let read f = ...
  let write f s = ...
end
```

The return types of `open_readonly` and `open_readwite` are restricted to producing files whose type parameter represents the appropriate mode. Similarly, `write` is restricted to only operate on values of type `readwrite t`. This prevents the errors we are trying to avoid. However, `read` is polymorphic in the mode of the file to be read – it will operate on files opened in either mode.

Note that the `File.t` type is still defined as an integer. The type parameter is not actually used in the type's definition: it is a *phantom type*. Within the `File` module the type `readonly t` is equal to the type `readwrite t` – since they both equal `int`. However, thanks to abstraction, these types are not equal outside of the module and the invariant that files opened by `open_readonly` cannot be passed to `write` is preserved.

Using this interface, the previous example now produces a compiler-time error:

```
# let f = File.open_readonly "foo" in
    File.write f "bar";;

  Characters 51-52:
      File.write f "bar";;
                 ^
Error: This expression has type File.readonly File.t
       but an expression was expected of type
       File.readwrite File.t
       Type File.readonly is not compatible with type
       File.readwrite
```

6.5.3 Lightweight static capabilities

To illustrate the kind of invariants that can be enforced using phantom types, we will look at an example from the paper "Lightweight Static Capabilities" (Kiselyov and Shan [2007]).

Consider the following interface for an array type:

```
module Array : sig
  type 'a t = 'a array
  val length : 'a t -> int
  val set : 'a t -> int -> 'a -> unit
  val get   : 'a t -> int -> 'a
end
```

We can use this interface to try to write binary search of a sorted array:

```
let search cmp arr v =
  let rec look low high =
    if high < low then None
    else begin
      let mid = (high + low)/2 in
      let x = Array.get arr mid in
      let res = cmp v x in
        if res = 0 then Some mid
        else if res < 0 then look low (mid - 1)
        else look (mid + 1) high
    end
  in
    look 0 (Array.length arr)
```

This function takes a comparison function `cmp`, an array `arr` sorted according to `cmp` and a value `v`. If `v` is in `arr` then the function returns its index, otherwise it returns `None`.

However, if we try a few examples with this function, we find that there is a problem:

```
# let arr = [|'a';'b';'c';'d'|];;

val arr : char array = [|'a'; 'b'; 'c'; 'd'|]

# let test1 = search compare arr 'c';;

val test1 : int option = Some 2

# let test2 = search compare arr 'a';;

val test2 : int option = Some 0

# let test3 = search compare arr 'x';;

Exception: Invalid_argument "index out of bounds".
```

Our last example raises an `Invalid_argument` exception becuase we have tried to access an index outside the bounds of the array.

The problem is easy enough to fix – we need to change the last line to use the index of the last element of `arr` rather than its length:

```
    look 0 ((Array.length arr) - 1)
```

However, we would rather catch such mistakes at compile-time.

To prevent out-of-bounds accesses at compile-time, we add another type parameter to the array type, which represents the size of the array. We also replace `int` with an abstract type `index` for representing array indices. The `index` type is also parameterised by a size type, which indicates that the index is within the bounds of arrays of that size.

```
module BArray : sig
  type ('s,'a) t
  type 's index

  val last : ('s, 'a) t -> 's index
  val set : ('s,'a) t -> 's index -> 'a -> unit
  val get  : ('s,'a) t -> 's index -> 'a
end
```

The types of `set` and `get` ensure that only indices that are within the bounds of the array are allowed by enforcing the size parameter of the array and the index to match.

We could try to use sophisticated types to represent sizes – perhaps encoding the size using type-level arithmetic. This would allow us to represent relationships between the different sizes – for instance allowing us to represent one array being smaller than another. However, such sophistication comes with added complexity, so we will take a simpler approach: size types will be abstract.

We extend our array interface with a function `brand`:

```
type 'a brand =
  | Brand : ('s, 'a) t -> 'a brand
  | Empty : 'a brand

val brand : 'a array -> 'a brand
```

The `Brand` constructor contains a value of type `('s, 'a) t`, where `'s` is an existential type variable. In essence, the `brand` function takes a regular OCaml array and returns a combination of an abstract size type and a `t` value of that size.

Since the size of each branded array is abstract, we cannot use indices for one array to access a different array:

```
# let Brand x = brand [| 'a'; 'b'; 'c'; 'd'|] in
  let Brand y = brand [| 'a'; 'b'|] in
    get y (last x);;

    Characters 96-104:
      get y (last x);;
            ^^^^^^^^^
Error: This expression has type s#1 BArray.index
       but an expression was expected of type
```

```
        s#2 BArray.index
        Type s#1 is not compatible with type s#2
```

Finally, we add some functions to our interface for manipulating indices.

```
val zero : 's index
val last : ('s, 'a) t -> 's index

val index : ('s, 'a) t -> int -> 's index option
val position : 's index -> int

val middle : 's index -> 's index -> 's index

val next : 's index -> 's index -> 's index option
val previous : 's index -> 's index ->
                 's index option
```

Each of these functions must maintain the invariant that an index of type `'s index` is always valid for an array of type `('s, 'a) t`. For example, the `next` function, which takes an index and returns the index of the next element in the array, also takes an additional index parameter and will only return the new index if it is less than this additional index. This ensures that the new index lies between two existing indices, and is therefore a valid index.

The full implementation of this safe array interface is given in Fig. 6.5. We can use it to implement our binary search function without too many changes:

```
let bsearch cmp arr v =
  let open BArray in
  let rec look barr low high =
    let mid = middle low high in
    let x = get barr mid in
    let res = cmp v x in
      if res = 0 then Some (position mid)
      else if res < 0 then
        match previous low mid with
        | Some prev -> look barr low prev
        | None -> None
      else
        match next mid high with
        | Some next -> look barr next high
        | None -> None
  in
    match brand arr with
    | Brand barr -> look barr zero (last barr)
    | Empty -> None
```

This function is guaranteed not to make an out-of-bounds access to the array, giving us greater confidence in the correctness of its implementation.

```
 type ('s,'a) t = 'a array

 type 'a brand =
   | Brand : ('s, 'a) t -> 'a brand
   | Empty : 'a brand

let brand arr =
   if Array.length arr > 0 then Brand arr
   else Empty

type 's index = int

 let index arr i =
   if i > 0 && i < Array.length arr then Some i
   else None

let position idx = idx

let zero = 0
let last arr = (Array.length arr) - 1
let middle idx1 idx2 = (idx1 + idx2)/2

let next idx limit =
   let next = idx + 1 in
      if next <= limit then Some next
      else None

let previous limit idx =
   let prev = idx - 1 in
      if prev >= limit then Some prev
      else None

let set = Array.set

let get = Array.get
```

Figure 6.5: Implementation of the safe array interface

Abstraction is the key to this technique. Thanks to abstraction we know that if the implementation of `BArray` preserves our invariant then so must the entire program. In essence, we have reduced the problem of proving our invariant for the whole problem to proving our invariant for a small *trusted kernel* – if we trust the implementation of this kernel we can trust the entire program.

As an additional benefit, we can safely adjust our implementation to use the unsafe variants of OCaml's `get` and `set` primitives:

```
let set = Array.unsafe_set

let get = Array.unsafe_get
```

This means that our array accesses will not perform any runtime checks for out-of-bounds accesses: by using abstraction to preserve a safety invariant we are able to improve the performance of our programs.

6.6 Theorems for free

In the previous section we applyed 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 $u : T$:

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

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

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

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 $\forall\alpha.\mathrm{List}\,\alpha \to \mathrm{List}\,\alpha$:

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

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 instantiating ρ with such a relation, we obtain the following property:

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

Applying the relational substitution of `List` to a function's relation gives a relation representing mapping the function over the list:

$$(\mathrm{List}\,\alpha)[\langle g\rangle](xs : \mathit{List}\,A, ys : \mathit{List}\,B) \ = \ (\mathrm{map}\,g\,xs =_{List\ B} ys)$$

which means the formula can be reduced to:

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

This shows 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.

6.7 Practical limitations

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

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 identitiy function. However, the following three OCaml functions have that type and are not the identity function:

```
let f (x : 'a) : 'a =
  Printf.printf "Lanch 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 substitution for OCaml's function type is different from the relational substitution that we gave for System F. The relational substituion 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 identitiy function, we do know that if such a function returns a value then it will be equal to the function's input.

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:

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

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

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

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 existance 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.

Bibliography

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2014. ISBN 978-3-319-07150-3. doi: 10.1007/978-3-319-07151-0_8. URL http://dx.doi.org/10.1007/978-3-319-07151-0_8.

Oleg Kiselyov and Chung-chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7):79–104, 2007. doi: 10.1016/j.entcs.2006.10.039. URL http://dx.doi.org/10.1016/j.entcs.2006.10.039.

Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989. doi: 10.1145/99370.99404. URL http://doi.acm.org/10.1145/99370.99404.

Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8 (4):343–355, December 1995. ISSN 0892-4635.