Chapter 3

Abstraction

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

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.

3.1 Abstraction in OCaml

3.1.1 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 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.

3.1.2 Invariants

Abstraction has further implications beyond the ability to replace one implementation with another. In particular, abstraction allows us to preserve invariants on types.

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[1].

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.

[1]We ignore overflow for the sake of simplicity

3.1.3 The meaning of types

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

In the simply typed lambda calculus types only represent particular data representations. For example, `Bool` → `Bool` represents functions that take a boolean and produce a boolean. In this setting, the purpose of types is to prevent meaningless expressions which have no defined behaviour according to the semantics of the language. Types cannot be used to represent concepts beyond these semantics.

This prevents certain kinds of bugs but the majority of bugs in a program are not related to the semantics of the language but to the intended semantics of the program. For example, a program might be using integers to represent peoples' heights, and whilst it would be a bug if a value intending to represent a height was in fact a function, a more likely mistake would be for a person's height to be represented by a negative integer. Although the language's semantics define how this negative integer should behave for all the operations which could be performed on a height (e.g. addition) it is still not an acceptable value for a height.

Abstraction extends the possible meanings of types to include arbitrary invariants on values. This allows types to represent concepts within the intended semantics of the program. For example, abstraction can create a type that represents people's heights, and ensure that all values of this type are acceptable values for a person's height.

3.1.4 Phantom types

The `Positive.t` type in the earlier example represented not just the data representation (an integer) but also an invariant (that the integer be positive). Using higher-kinded types we can take the idea of types as invariants even further.

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.

One problem 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
```

3.1.5 The meaning of types (continued)

Just as abstraction allows types to represent more than just a particular data representation, higher-kinded abstraction allows types to represent an even wider set of concepts. Base-kinded abstraction restricts types to directly representing invariants on values, with each type corresponding to particular set of values. With higher-kinded abstraction, types can represent more general concepts without a direct correspondence to values.

For example, the `readonly` type in the above example represents the general concept of "read-only mode". There are no actual values of this type since it does not directly correspond to a property of values themselves. The `File.t` type can then be parameterized by this concept in order to represent file handles for read-only files.

Further types, (e.g. channels) may also be parameterized by the same concept, allowing types to express relationships between these values (e.g. a function which takes a file and produces a channel of the same mode).

3.1.6 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 F. 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 it only works for types of kind $\star$.

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 F types, including:

$\forall\alpha$::$\star$. Bool $\rightarrow$ α $\rightarrow$ α $\rightarrow$ α

$\forall\alpha$::$\star$.$\forall\beta$::$\star$. Bool $\rightarrow$ α $\rightarrow$ β $\rightarrow$ $\exists\gamma$::$\star$.γ

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 $\exists\alpha.\alpha \times (\alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{string})$:

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

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

```
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 F:

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

3.1.7 Example: lightweight static capabilities

To illustrate the kind of invariants that can be enforced using higher-kinded abstraction, 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 because 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 parameterized 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. 3.1. 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.

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:

```
 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 3.1: Implementation of the safe array interface

```
let set = Array.unsafe_set

let get = Array.unsafe_get
```

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

3.2 Abstraction in System F...

3.2.1 Existential types

The abstract types in OCaml's module system correspond to existential types in System F. 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 F. 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 α =
      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,
       \langle 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 \rangle
  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 F:

```
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 1.3.1), we can see that the type which is packed (List Nat) is not present in the type of the package (∃α::⋆.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.

... Teaser: System Fω

When defining NatSetImpl, we use a meta variable α to quantify over all possible instantiations:

```
NatSetImpl α =
      α
      × (α → Bool)
      × (Nat → α → Bool)
      × (Nat → α → α)
      × (Nat → α → α)
      × (α → List Nat);
```

Later, we will System Fω (lecture 5), in which we can express NatSetImpl as a higher-kinded type, here a type-level abstraction which takes a type α of kind $\star$ and returns a type, NatSetImpl α:

```
NatSetImpl =
  λα::⋆.
      α
```

```
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 3.2: A set interface

```
    × (α → Bool)
    × (Nat → α → Bool)
    × (Nat → α → α)
    × (Nat → α → α)
    × (α → List Nat);
```

3.2.2 Relational abstraction

In the previous sections we have been quite loose in our description of abstraction. We have talked about abstraction as invariance under change of implementation, but we have not made this notion precise.

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

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. 3.2. 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.

Two implementations of this interface are shown in Fig. 3.3 and Fig. 3.4– 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

```
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 3.3: 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 3.4: Tree implementation of the set interface

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:

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

  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:

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

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:

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

`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:

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

`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:

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

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

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

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 <u>equivalent</u> sets when given <u>equivalent</u> 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 <u>related</u> results when given <u>related</u> inputs. This leads us to the much stronger relation:

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

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.

The relational interpretation

The table in Fig. 3.5 compares 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 as an interpretation of types as relations.

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

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

`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`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $(\text{mem}_{list}\, x\, i\ =\ \text{mem}_{tree}\, y\, j)$
`val` `add:`	
`t -> int -> t`	$\forall x : t_{list}.\ \forall y : t_{tree}.\ \forall i : Int.\ \forall j : Int.$ $\sigma(x, y) \Rightarrow (i = j) \Rightarrow$ $\sigma(\text{add}_{list}\, x\, i,\ \text{add}_{tree}\, y\, j)$
`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 3.5: Types and relations for the set interface

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

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

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

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

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

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

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

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

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

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

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

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

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

The relational interpretation 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.

A relational definition of abstraction

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

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

$$\forall B_1 \ldots \forall B_n.\ \forall x : (\exists \alpha. T[\alpha, B_1, \ldots, B_n]).\ \forall y : (\exists \alpha. T[\alpha, B_1, \ldots, B_n]).$$

$$x = y \quad \Leftrightarrow \quad \begin{aligned}&\exists \gamma.\ \exists \delta.\ \exists \sigma \subset \gamma \times \delta.\\ &\quad \exists u : T[\gamma, B_1, \ldots, B_n].\ \exists v : T[\delta, B_1, \ldots, B_n].\\ &\qquad x = \text{pack } \gamma,\ u \text{ as } \exists \alpha. T[\alpha, B_1, \ldots, B_n]\\ &\qquad \wedge\ y = \text{pack } \delta,\ v \text{ as } \exists \alpha. T[\alpha, B_1, \ldots, B_n]\\ &\qquad \wedge\ [\![T]\!][\sigma, =_{B_1}, \ldots, =_{B_n}](u,\ v)\end{aligned}$$

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.

3.2.3 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, the ability of abstraction to preserve invariants on types.

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:

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

This means that if u preserves an invariant on type γ – represented by relation ρ – and u is packed up into an existential type then 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.

3.2.4 Identity extension

The relational interpretation 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 $\alpha_1, \ldots, \alpha_n$, if we substitute equality relations (=) for a type's free variables we get the equality relation of that type:

$$\begin{array}{l} \forall A_1. \ldots \forall A_n.\ \forall x : T[A_1, \ldots, A_n].\ \forall y : T[A_1, \ldots, A_n]. \\ \quad (x =_{T[A_1, \ldots, A_n]} y) \quad \Leftrightarrow \quad [\![T]\!][=_{A_1}, \ldots, =_{A_n}](x,\ y) \end{array}$$

This property of the relational interpretation is known as identity extension. The abstraction property of existentials follow as a direct consequence of identity extension.

Identity extension is a key property of System F which is often overlooked when discussing the soundness of extensions to it. The traditional approach of proving soundness by showing preservation and progress (Wright and Felleisen

[1994]) does not demonstrate that a system maintains identity extension, and thus ignores the key question of whether abstraction is preserved by that system.

In the next chapter we will look at another consequence of identity extension: parametricity.