9

Polymorphism and generic programming

What happens when the edict of intention runs up against the edict of irredundancy? The edict of intention calls for expressing clearly the intended types over which functions operate, so that the language can provide help by checking that the types are used consistently. We've heeded that edict, for example, in our definition of the higher-order function `map` from the previous chapter, repeated here:

```
# let rec map (f : int -> int) (xs : int list) : int list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> f hd :: (map f tl) ;;
val map : (int -> int) -> int list -> int list = <fun>
```

The `map` function is tremendously useful for a wide variety of operations over integer lists. It seems natural to apply the same idea to other kinds of lists as well. For instance, we may want to define a function to double all of the elements of a `float list` or implement the `prods` function from Section 7.3.1 to take the products of pairs of integers in a list of such pairs. Using `map` we can try

```
# let double = map (fun x -> 2. *. x) ;;
Line 1, characters 33-34:
1 | let double = map (fun x -> 2. *. x) ;;
                                     ^
Error: This expression has type int but an expression was expected
  of type
         float
# let prods = map (fun (x, y) -> x * y) ;;
Line 1, characters 21-27:
1 | let prods = map (fun (x, y) -> x * y) ;;
                         ^^^^^^
Error: This pattern matches values of type 'a * 'b
       but a pattern was expected which matches values of type int
```

but we run afoul of the typing constraints on `map`, which can only apply functions of type `int -> int`, and not `float -> float` or `int * int -> int`.

Of course, we can implement a version of `map` for lists of these types as well:

```
# let rec map_float_float (f : float -> float)
#                         (xs : float list)
#                       : float list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> f hd :: (map_float_float f tl) ;;
val map_float_float : (float -> float) -> float list -> float list
  = <fun>

# let rec map_intpair_int (f : int * int -> int)
#                         (xs : (int * int) list)
#                       : int list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> f hd :: (map_intpair_int f tl) ;;
val map_intpair_int : (int * int -> int) -> (int * int) list -> int
  list =
  <fun>
```

This is where we run up against the edict of irredundancy: we've written the same code three times now, once for each set of argument types.

What we'd like is a way to map functions over lists *generically*, while still obeying the constraint that whatever type the list elements are, they are appropriate to apply the function to; and whatever type the function returns, the map returns a list of elements of that type.

9.1 Polymorphism

The solution to this quandary is found in POLYMORPHISM. In a language with polymorphism, like OCaml, functions can apply generically to values from any type, so long as they do so *consistently and systematically*, as the various versions of `map` above do. Nonetheless, we'd still like to keep the advantages of strong static typing, so that code can be checked for this consistency and systematicity. Then what should the type of a polymorphic version of the map function be?

We can get a hint of the answer by taking advantage of OCaml's type inference process, first introduced in Section 4.2.1. The type inference process combines all of the type constraints implicit in the use of typed functions together with all of the constraints in explicit typings to compute the types for all of the expressions in a program. For instance, in the definition

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
```

it follows from the fact that the + function is applied to x that x must have the same type as the argument type for +, that is, int. Similarly, since succ x is calculated as the output of the + function, it must have the same type as +'s output type, again int. Since succ's argument is of type int and output is of type int, its type must be int -> int. And in fact that is the type OCaml reports for it, even though no explicit typings were provided.

Propagating type information in this way results in a fully instantiated type int -> int for the succ function. But what if there aren't enough constraints in the code to yield a fully instantiated type? The IDENTITY FUNCTION id, which just returns its argument unchanged, is an example:

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

Since x is never involved in any applications in the definition of id, there are no type constraints on it. All that we can conclude is that whatever type x is – call it α – the id function must take values of type α as argument and return values of type α as output. That is, id must be of type α -> α.

The id function doesn't have a fully instantiated type. It is a POLYMORPHIC FUNCTION, with a POLYMORPHIC TYPE. The term *polymorphic* means *many forms*; the id function can take arguments of many forms and operate on them similarly.

As the type inference process has indicated in the REPL output, to express polymorphic types, we need to extend the type expression language. We use TYPE VARIABLES to specify that *any* type can be used. We write type variables as identifiers with a prefixed quote mark – 'a, 'b, 'c, and so forth – and conventionally read them as their corresponding Greek letter – α (alpha), β (beta), γ (gamma) – as we've done above. Notice that OCaml has reported a polymorphic type for id, namely, 'a -> 'a (read, "α to α"). This type makes the claim, "for *any* type α, if id is applied to an argument of type α it returns a value of type α."

9.2 Polymorphic map

Returning to the map function, we wanted a way to map functions over lists generically. If we just remove the typings in the definition of map, it would seem that we could have just such a function, a polymorphic version of map.

```
# let rec map f xs =
#   match xs with
#   | [] -> []
```

```
#   | hd :: tl -> f hd :: (map f tl) ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

This function performs the same computation as the previous version of `map`, just without any of the explicit type constraints enforced. The function `f` is applied to elements of `xs` and returns elements that appear in the result list, so the type of the argument of `f` must be the type of the elements of `xs` and the type of the result of `f` must be the type of the elements of the returned list *simply as a consequence of the structure of the code.*

Happily, the type inference process that OCaml uses – developed by Roger Hindley (Figure 9.1) and Robin Milner (Figure 1.7) – infers these constraints automatically, concluding that `map`, like `id`, has a polymorphic type, which the OCaml type inference system has inferred and reported as `('a -> 'b) -> 'a list -> 'b list`. This type expresses the constraint that "for *any* types α and β, if `map` is applied to a function from α values to β values, it will return a function that when given a list of α values returns a list of β values."

Figure 9.1: J. Roger Hindley (1939–), codeveloper with Robin Milner (Figure 1.7) of the Hindley-Milner type inference algorithm that OCaml relies on for inferring the most general polymorphic types for expressions.

This polymorphic version of `map` can be used to implement `double` and `prods` as above. In each case, the types for these functions are themselves properly inferred by instantiating the type variables of the polymorphic `map` type. [1]

[1] Note the use of partial application in these examples.

```
# let double = map (fun x -> 2. *. x) ;;
val double : float list -> float list = <fun>
# let prods = map (fun (x, y) -> x * y) ;;
val prods : (int * int) list -> int list = <fun>
```

As inferred by OCaml, `double` takes a `float list` argument and returns a `float list`, and `prods` takes an `(int * int) list` argument and returns an `int list`.

9.3 Regaining explicit types

By taking advantage of polymorphism in OCaml, we've satisfied the edict of irredundancy by defining a polymorphic version of `map`. Unfortunately, we seem to have forgone the edict of intention, since we are no longer explicitly providing information about the intended type for `map`.

But by using the additional expressivity provided by type variables, we can express the intended typing for `map` explicitly.

```
# let rec map (f : 'a -> 'b) (xs : 'a list) : 'b list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> f hd :: (map f tl) ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

The type variables make clear the intended constraints among `f`, `xs`, and the return value `map f xs`.

Problem 60

For each of the following types construct an expression for which OCaml would infer that type. For example, for the type `bool * bool`, the expression `true, true` would be a possible answer. (The idea in this exercise is not that the expressions be practical or do anything useful; they need only have the requested type. But no cheating by using explicit typing annotations with the `:` operator!)

1. `bool * bool -> bool`
2. `'a list -> bool list`
3. `('a * 'b -> 'a) -> 'a -> 'b -> 'a`
4. `int * 'a * 'b -> 'a list -> 'b list`
5. `bool -> unit`
6. `'a -> ('a -> 'b) -> 'b`
7. `'a -> 'a -> 'b`

Exercise 61

Define polymorphic versions of `fold` and `filter`, providing explicit polymorphic typing information.

Problem 62

For each of the following definitions of a function `f`, give its most general type (as would be inferred by OCaml) or explain briefly why no type exists for the function.

1.
```
let f x =
  x +. 42. ;;
```
2.
```
let f g x =
  g (x + 1) ;;
```
3.
```
let f x =
  match x with
  | [] -> x
  | h :: t -> h ;;
```
4.
```
let rec f x a =
  match x with
  | [] -> a
  | h :: t -> h (f t a) ;;
```
5.
```
let f x y =
  match x with
  | (w, z) -> if w then y z else w ;;
```
6.
```
let f x y =
  x y y ;;
```
7.
```
let f x y =
  x (y y) ;;
```
8.
```
let rec f x =
  match x with
  | None
  | Some 0 -> None
  | Some y -> f (Some (y - 1)) ;;
```
9.
```
let f x y =
  if x then [x]
  else [not x; y] ;;
```

9.4 *The List library*

One way, perhaps the best, for satisfying the edict of irredundancy is to avoid writing the same code twice by *not writing the code even once*, instead taking advantage of code that someone else has already written. OCaml, like many modern languages, comes with a large set of libraries (packaged as modules, which we'll cover in Chapter 12) that provide a wide range of functionality. The `List` module in particular provides exactly the higher-order list processing functions presented in this and the previous chapter as polymorphic functions. The documentation for the `List` module gives typings and descriptions for lots of useful list processing functions. For instance, the module provides the map, fold, and filter abstractions of Chapter 8, described in the documentation as

- `map : ('a -> 'b) -> 'a list -> 'b list`

  `map f [a1; ...; an]` applies function `f` to `a1`, ..., `an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.[2]

- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

  `fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.

- `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

  `fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`. Not tail-recursive.

- `filter : ('a -> bool) -> 'a list -> 'a list`

  `filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.

They can be invoked as `List.map`, `List.fold_left`, and so forth. The library provides many other useful functions, including

- `append : 'a list -> 'a list -> 'a list`

  Concatenate two lists. Same as the infix operator `@`....

- `partition : ('a -> bool) -> 'a list -> 'a list * 'a list`

  `partition p l` returns a pair of lists `(l1, l2)`, where `l1` is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`. The order of the elements in the input list is preserved.

The `List` library has further functions for sorting, combining, and transforming lists in all kinds of ways.

Although these functions are built into OCaml through the `List` library, it's still useful to have seen how they are implemented and

[2] We'll come back to the issue of tail recursion in Section 16.2.2.

why they have the types they have. In particular, it makes clear that the power of list processing via higher-order functional programming doesn't require special language constructs; they arise from the interactions of simple language primitives like first-class functions and structured data types.

Problem 63

Provide an implementation of the `List.map` function over a list using only a call to `List.fold_right` over the same list, or provide an argument for why it's not possible to do so.

Problem 64

Provide an implementation of the `List.fold_right` function using only a call to `List.map` over the same list, or provide an argument for why it's not possible to do so.

Problem 65

In the `list` module, OCaml provides a function `partition : ('a -> bool) -> 'a list -> 'a list * 'a list`. According to the OCaml documentation, "`partition p l` returns a pair of lists `(l1, l2)`, where l1 is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`. The order of the elements in the input list is preserved."

For example, we can use this to divide a list into two new ones, one containing the even numbers and one containing the odd numbers:

```
# List.partition (fun n -> n mod 2 = 0)
#                [1; 2; 3; 4; 5; 6; 7] ;;
- : int list * int list = ([2; 4; 6], [1; 3; 5; 7])
```

As described above, the `List` module provides the `partition` function of type `('a -> bool) -> 'a list -> 'a list * 'a list`. Give your own definition of `partition`, implemented directly without the use of any library functions except for those in the `Stdlib` module.

Exercise 66

Define a function `permutations : 'a list -> 'a list list`, which takes a list of values and returns a list containing every permutation of the original list. For example,

```
# permutations [1; 2; 3] ;;
- : int list list =
[[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
```

It doesn't matter what order the permutations appear in the returned list. Note that if the input list is of length n, then the answer should be of length $n!$ (that is, the factorial of n). Hint: One way to do this is to write an auxiliary function, `interleave : int -> int list -> int list list`, that yields all interleavings of its first argument into its second. For example:

```
# interleave 1 [2; 3] ;;
- : int list list = [[1; 2; 3]; [2; 1; 3]; [2; 3; 1]]
```

9.5 *Problem section: Function composition*

The COMPOSITION of two unary functions `f` and `g` is the function that applies `f` to the result of applying `g` to its argument.

For example, suppose you're given a list of pairs of integers, where we think of each pair as containing a number and a corresponding weight. We'd like to compute the WEIGHTED SUM of the numbers, that is, the sum of the numbers where each has been weighted according to (that is, multiplied by) its weight. Recall the `sum` function from Exercise 44 and the `prods` function from Section 7.3.1. The weighted average of a pair-list can be computed by applying the `sum` function to the result of applying the `prods` function to the list. Thus, `weighted_-sum` is just the composition of `sum` and `prods`.

Problem 67

Provide an OCaml definition for a higher-order function `@+` that takes two functions as arguments and returns their composition. The function should have the following behavior:

```
# let weighted_sum = sum @+ prods ;;
val weighted_sum : (int * int) list -> int = <fun>
# weighted_sum [(1, 3); (2, 4); (3, 5)] ;;
- : int = 26
```

Notice that by naming the function `@+`, it is used as an infix, right-associative operator. See the operator table in the OCaml documentation for further information about the syntactic properties of operators. When defining the function itself, though, you'll want to use it as a prefix operator by wrapping it in parentheses, as `(@+)`.

Problem 68

What is the type of the `@+` function?

9.6 *Problem section: Credit card numbers and the Luhn check*

Here's an interesting bit of trivia: Not all credit card numbers are well-formed. The final digit in a 16-digit credit card number is in fact a CHECKSUM, a digit that is computed from the previous 15 by a simple algorithm.

The algorithm used to generate the checksum is called the LUHN CHECK. To calculate the correct final checksum digit used in a 16-digit credit card number, you perform the following computation on the first 15 digits of the credit card number:

Figure 9.2: A sample credit card

1. Take all of the digits in an odd-numbered position (the leftmost digit being the first, not the zero-th digit, hence an odd-numbered one) and double them, subtracting nine if the doubling is greater than nine (called "casting out nines").

   As an example, we'll use the (partial) credit card number from the card in Figure 9.2:

   $$\underline{4}2\underline{7}5 \quad \underline{3}1\underline{5}6 \quad \underline{0}3\underline{7}2 \quad \underline{5}4\underline{9}x$$

   Here, the odd-numbered digits (4, 7, 3, 5, 0, 7, 5, and 9) have been underlined. We double them and cast out nines to get 8, 5, 6, 1, 0, 5, 1, and 9.

2. Add all of the even position digits and the doubled odd position digits together. For the example above, the sum would be

   $$(2+5+1+6+3+2+4)+(8+5+6+1+0+5+1+9) = 23+35 = 58 \quad .$$

3. The checksum is then the digit that when added to this sum makes it a multiple of ten. In the example above the checksum would be 2, since adding 2 to 58 generates 60, which is a multiple of 10. Thus, the sequence 4275 3156 0372 5492 is a valid credit card number, but changing the last digit to any other makes it invalid. (In particular, the final 3 in the card in Figure 9.2 is not the correct checksum!)

Problem 69

Define an explicitly recursive polymorphic function `odds` to extract the elements at odd-numbered indices in a list, where the indices are counted starting with 1, so that

```
# let cc = [4; 2; 7; 5; 3; 1; 5; 6; 0; 3; 7; 2; 5; 4; 9] ;;
val cc : int list = [4; 2; 7; 5; 3; 1; 5; 6; 0; 3; 7; 2; 5; 4; 9]
# odds cc ;;
- : int list = [4; 7; 3; 5; 0; 7; 5; 9]
```

Exercise 70

What is the type of the `odds` function?

In addition to `odds`, it will be useful to have a function `evens` that extracts the elements at even-numbered indices in a list.

Problem 71

Define `evens` to extract the elements at even-numbered indices in a list, where the indices are counted starting with 1, so that

```
# let cc = [4; 2; 7; 5; 3; 1; 5; 6; 0; 3; 7; 2; 5; 4; 9] ;;
val cc : int list = [4; 2; 7; 5; 3; 1; 5; 6; 0; 3; 7; 2; 5; 4; 9]
# evens cc ;;
- : int list = [2; 5; 1; 6; 3; 2; 4]
```

The process of doubling a number and "casting out nines" is easy to implement as well. Here is some code to do that:

```
# let doublemod9 (n : int) : int =
#   (n * 2 - 1) mod 9 + 1 ;;
val doublemod9 : int -> int = <fun>
```

Finally, it will be useful to have a function to sum a list of integers.

Problem 72

Implement the function `sum` using the tail-recursive `List` module function `fold_left`.

All the parts are now in place to implement the Luhn check algorithm.

Problem 73

Implement a function `luhn` that takes a list of integers and returns the check digit for that digit sequence. (You can assume that it is called with a list of 15 integers.) For instance, for the example above

```
# luhn cc ;;
- : int = 2
```

You should feel free to use the functions `evens`, `odds`, `doublemod9`, `sum`, and any other OCaml library functions that you find useful and idiomatic.

9.7 Weak type variables

The `List` module provides polymorphic `hd` and `tl` functions for extracting the head and tail of a list.

Exercise 74

What are the types of the `hd` and `tl` functions? See if you can determine them without looking them up.

These can be composed to allow, for instance, extracting the head of the tail of a list, that is, the list's second item.

```
# let second = List.hd @+ List.tl ;;
val second : '_weak1 list -> '_weak1 = <fun>
```

This definition works,

```
# second [1; 2; 3] ;;
- : int = 2
```

but why did the typing of `second` have those oddly named type variables?

Type variables like `'_weak1` (with the initial underscore) are WEAK TYPE VARIABLES, not true type variables. They maintain their polymorphism only temporarily, until the first time they are applied. Weak type variables arise because in certain situations OCaml's type inference can't figure out how to express the most general types and must resort to this fallback approach.

When a function with these weak type variables is applied to arguments with a specific type, the polymorphism of the function disappears. Having applied `second` to an `int list`, OCaml further instantiates the type of `second` to *only* apply to `int list` arguments, losing its polymorphism. We can see this in two ways, first by checking its type directly,

```
# second ;;
- : int list -> int = <fun>
```

and second by attempting to apply it to a list of another type,

```
# second [1.0; 2.1; 3.2] ;;
Line 1, characters 8-11:
1 | second [1.0; 2.1; 3.2] ;;
            ^^^
Error: This expression has type float but an expression was
  expected of type
         int
```

To correct the problem, you can of course add in specific typing information

```
# let second : float list -> float =
#   List.hd @+ List.tl ;;
val second : float list -> float = <fun>
```

but this provides no polymorphism. Alternatively, you can provide a full specification of the call pattern in the definition rather than the partial application that was used above:

```
# let second x = (List.hd @+ List.tl) x ;;
val second : 'a list -> 'a = <fun>
```

which gives OCaml sufficient hints to infer types more generally. Of course, in this case, the composition operator isn't really helping. We might as well have defined `second` more directly as

```
# let second x = List.hd (List.tl x) ;;
val second : 'a list -> 'a = <fun>
```

For the curious, if you want to see what's going on in detail, you can check out the discussion in the section "A function obtained through partial application is not polymorphic enough" in the OCaml FAQ.

9.8 Supplementary material

- Lab 3: Polymorphism and record types