10 *An ML Implementation of Simple Types*

The concrete realization of $\lambda_{\rightarrow}$ as an ML program follows the same lines as our implementation of the untyped lambda-calculus in Chapter 7. The main addition is a function `typeof` for calculating the type of a given term in a given context. Before we get to it, though, we need a little low-level machinery for manipulating contexts.

10.1 Contexts

Recall from Chapter 7 (p. 85) that a `context` is just a list of pairs of variable names and `binding`s:

```
type context = (string * binding) list
```

In Chapter 7, we used contexts just for converting between named and nameless forms of terms during parsing and printing. For this, we needed to know just the names of the variables; the `binding` type was defined as a trivial one-constructor datatype carrying no information at all:

```
type binding = NameBind
```

To implement the typechecker, we will need to use the context to carry typing assumptions about variables. We support this by adding a new constructor called `VarBind` to the `binding` type:

```
type binding =
    NameBind
  | VarBind of ty
```

The implementation described here corresponds to the simply typed lambda-calculus (Figure 9-1) with booleans (8-1). The code in this chapter can be found in the `simplebool` implementation in the web repository.

Each VarBind constructor carries a typing assumption for the corresponding variable. We keep the old NameBind constructor in addition to VarBind, for the convenience of the printing and parsing functions, which don't care about typing assumptions. (A different implementation strategy would be to define two completely different context types—one for parsing and printing and another for typechecking.)

The typeof function uses a function addbinding to extend a context ctx with a new variable binding (x,bind); since contexts are represented as lists, addbinding is essentially just cons:

```
let addbinding ctx x bind = (x,bind)::ctx
```

Conversely, we use the function getTypeFromContext to extract the typing assumption associated with a particular variable i in a context ctx (the file information fi is used for printing an error message if i is out of range):

```
let getTypeFromContext fi ctx i =
  match getbinding fi ctx i with
      VarBind(tyT) → tyT
    | _ → error fi
      ("getTypeFromContext: Wrong kind of binding for variable "
       ^ (index2name fi ctx i))
```

The match provides some internal consistency checking: under normal circumstances, getTypeFromContext should always be called with a context where the i^{th} binding is in fact a VarBind. In later chapters, though, we will add other forms of bindings (in particular, bindings for *type variables*), and it is possible that getTypeFromContext will get called with the wrong kind of variable. In this case, it uses the low-level error function to print a message, passing it an info so that it can report the file position where the error occurred.

```
val error : info → string → 'a
```

The result type of the error function is the variable type 'a, which can be instantiated to any ML type (this makes sense because it is never going to return anyway: it prints a message and halts the program). Here, we need to assume that the result of error is a ty, since that is what the other branch of the match returns.

Note that we look up typing assumptions by *index,* since terms are represented internally in nameless form, with variables represented as numerical indices. The getbinding function simply looks up the i^{th} binding in the given context:

```
val getbinding : info → context → int → binding
```

Its definition can be found in the `simplebool` implementation on the book's web site.

10.2 Terms and Types

The syntax of types is transcribed directly into an ML datatype from the abstract syntax in Figures 8-1 and 9-1.

```
type ty =
    TyBool
  | TyArr of ty * ty
```

The representation of terms is the same as we used for the untyped lambda-calculus (p. 84), just adding a type annotation to the `TmAbs` clause.

```
type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
```

10.3 Typechecking

The typechecking function `typeof` can be viewed as a direct translation of the typing rules for $\lambda_{\rightarrow}$ (Figures 8-1 and 9-1), or, more accurately, as a transcription of the inversion lemma (9.3.1). The second view is more accurate because it is the inversion lemma that tells us, for every syntactic form, exactly what conditions must hold in order for a term of this form to be well typed. The typing rules tell us that terms of certain forms are well typed under certain conditions, but by looking at an individual typing rule, we can never conclude that some term is *not* well typed, since it is always possible that another rule could be used to type this term. (At the moment, this may appear to be a difference without a distinction, since the inversion lemma follows so directly from the typing rules. The difference becomes important, though, in later systems where proving the inversion lemma requires more work than in $\lambda_{\rightarrow}$.)

```
let rec typeof ctx t =
  match t with
    TmTrue(fi) →
      TyBool
```

```
| TmFalse(fi) →
    TyBool
| TmIf(fi,t1,t2,t3) →
   if (=) (typeof ctx t1) TyBool then
     let tyT2 = typeof ctx t2 in
     if (=) tyT2 (typeof ctx t3) then tyT2
     else error fi "arms of conditional have different types"
   else error fi "guard of conditional not a boolean"
| TmVar(fi,i,_) → getTypeFromContext fi ctx i
| TmAbs(fi,x,tyT1,t2) →
    let ctx' = addbinding ctx x (VarBind(tyT1)) in
    let tyT2 = typeof ctx' t2 in
    TyArr(tyT1, tyT2)
| TmApp(fi,t1,t2) →
    let tyT1 = typeof ctx t1 in
    let tyT2 = typeof ctx t2 in
    (match tyT1 with
        TyArr(tyT11,tyT12) →
          if (=) tyT2 tyT11 then tyT12
          else error fi "parameter type mismatch"
      | _ → error fi "arrow type expected")
```

A couple of details of the OCaml language are worth mentioning here. First, the OCaml equality operator = is written in parentheses because we are using it in prefix position, rather than its normal infix position, to facilitate comparison with later versions of `typeof` where the operation of comparing types will need to be something more refined than simple equality. Second, the equality operator computes a *structural* equality on compound values, not a *pointer* equality. That is, the expression

```
let t  = TmApp(t1,t2) in
let t' = TmApp(t1,t2) in
(=) t t'
```

is guaranteed to yield `true`, even though the two instances of `TmApp` bound to `t` and `t'` are allocated at different times and live at different addresses in memory.