4 An ML Implementation of Arithmetic Expressions

Working with formal definitions such as those in the previous chapter is often easier when the intuitions behind the definitions are "grounded" by a connection to a concrete implementation. We describe here the key components of an implementation of our language of booleans and arithmetic expressions. (Readers who do not intend to work with the implementations of the typecheckers described later can skip this chapter and all later chapters with the phrase "ML Implementation" in their titles.)

The code presented here (and in the implementation sections throughout the book) is written in a popular language from the ML family (Gordon, Milner, and Wadsworth, 1979) called *Objective Caml,* or *OCaml* for short (Leroy, 2000; Cousineau and Mauny, 1998). Only a small subset of the full OCaml language is used; it should be easy to translate the examples here into most other languages. The most important requirements are automatic storage management (garbage collection) and easy facilities for defining recursive functions by pattern matching over structured data types. Other functional languages such as Standard ML (Milner, Tofte, Harper, and MacQueen, 1997), Haskell (Hudak et al., 1992; Thompson, 1999), and Scheme (Kelsey, Clinger, and Rees, 1998; Dybvig, 1996) (with some pattern-matching extension) are fine choices. Languages with garbage collection but without pattern matching, such as Java (Arnold and Gosling, 1996) and pure Scheme, are somewhat heavy for the sorts of programming we'll be doing. Languages with neither, such as C (Kernighan and Ritchie, 1988), are even less suitable.[1]

The code in this chapter can be found in the `arith` implementation in the web repository, `http://www.cis.upenn.edu/~bcpierce/tapl`, along with instructions on downloading and building the implementations.

1. Of course, tastes in languages vary and good programmers can use whatever tools come to hand to get the job done; you are free to use whatever language you prefer. But be warned: doing manual storage management (in particular) for the sorts of symbol processing needed by a typechecker is a tedious and error-prone business.

4.1 Syntax

Our first job is to define a type of OCaml values representing terms. OCaml's datatype definition mechanism makes this easy: the following declaration is a straightforward transliteration of the grammar on page 24.

```
type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

The constructors `TmTrue` to `TmIsZero` name the different sorts of nodes in the abstract syntax trees of type `term`; the type following `of` in each case specifies the number of subtrees that will be attached to that type of node.

Each abstract syntax tree node is annotated with a value of type `info`, which describes where (what character position in which source file) the node originated. This information is created by the parser when it scans the input file, and it is used by printing functions to indicate to the user where an error occurred. For purposes of understanding the basic algorithms of evaluation, typechecking, etc., this information could just as well be omitted; it is included here only so that readers who wish to experiment with the implementations themselves will see the code in exactly the same form as discussed in the book.

In the definition of the evaluation relation, we'll need to check whether a term is a numeric value:

```
let rec isnumericval t = match t with
    TmZero(_) → true
  | TmSucc(_,t1) → isnumericval t1
  | _ → false
```

This is a typical example of recursive definition by pattern matching in OCaml: `isnumericval` is defined as the function that, when applied to `TmZero`, returns true; when applied to `TmSucc` with subtree `t1` makes a recursive call to check whether `t1` is a numeric value; and when applied to any other term returns `false`. The underscores in some of the patterns are "don't care" entries that match anything in the term at that point; they are used in the first two clauses to ignore the `info` annotations and in the final clause to match any `term` whatsoever. The `rec` keyword tells the compiler that this is a recursive function definition—i.e., that the reference to `isnumericval` in its body

refers to the function now being defined, rather than to some earlier binding with the same name.

Note that the ML code in the above definition has been "prettified" in some small ways during typesetting, both for ease of reading and for consistency with the lambda-calculus examples. For instance, we use a real arrow symbol (→) instead of the two-character sequence ->. A complete list of these prettifications can be found on the book's web site.

The function that checks whether a term is a value is similar:

```
let rec isval t = match t with
    TmTrue(_)  → true
  | TmFalse(_) → true
  | t when isnumericval t  → true
  | _ → false
```

The third clause is a "conditional pattern": it matches any term `t`, but only so long as the boolean expression `isnumericval t` yields `true`.

4.2 Evaluation

The implementation of the evaluation relation closely follows the single-step evaluation rules in Figures 3-1 and 3-2. As we have seen, these rules define a *partial* function that, when applied to a term that is not yet a value, yields the next step of evaluation for that term. When applied to a value, the result of the evaluation function yields no result. To translate the evaluation rules into OCaml, we need to make a decision about how to handle this case. One straightforward approach is to write the single-step evaluation function `eval1` so that it raises an exception when none of the evaluation rules apply to the term that it is given. (Another possibility would be to make the single-step evaluator return a `term option` indicating whether it was successful and, if so, giving the resulting term; this would also work fine, but would require a little more bookkeeping.) We begin by defining the exception to be raised when no evaluation rule applies:

```
exception NoRuleApplies
```

Now we can write the single-step evaluator itself.

```
let rec eval1 t = match t with
    TmIf(_,TmTrue(_),t2,t3) →
      t2
  | TmIf(_,TmFalse(_),t2,t3) →
      t3
  | TmIf(fi,t1,t2,t3) →
```

```
    let t1' = eval1 t1 in
    TmIf(fi, t1', t2, t3)
| TmSucc(fi,t1) →
    let t1' = eval1 t1 in
    TmSucc(fi, t1')
| TmPred(_,TmZero(_)) →
    TmZero(dummyinfo)
| TmPred(_,TmSucc(_,nv1)) when (isnumericval nv1) →
    nv1
| TmPred(fi,t1) →
    let t1' = eval1 t1 in
    TmPred(fi, t1')
| TmIsZero(_,TmZero(_)) →
    TmTrue(dummyinfo)
| TmIsZero(_,TmSucc(_,nv1)) when (isnumericval nv1) →
    TmFalse(dummyinfo)
| TmIsZero(fi,t1) →
    let t1' = eval1 t1 in
    TmIsZero(fi, t1')
| _ →
    raise NoRuleApplies
```

Note that there are several places where we are constructing terms from scratch rather than reorganizing existing terms. Since these new terms do not exist in the user's original source file, their `info` annotations are not useful. The constant `dummyinfo` is used as the `info` annotation in such terms. The variable name `fi` (for "file information") is consistently used to match `info` annotations in patterns.

Another point to notice in the definition of `eval1` is the use of explicit `when` clauses in patterns to capture the effect of metavariable names like `v` and `nv` in the presentation of the evaluation relation in Figures 3-1 and 3-2. In the clause for evaluating `TmPred(_,TmSucc(_,nv1))`, for example, the semantics of OCaml patterns will allow `nv1` to match any term whatsoever, which is not what we want; adding `when (isnumericval nv1)` restricts the rule so that it can fire only when the term matched by `nv1` is actually a numeric value. (We could, if we wanted, rewrite the original inference rules in the same style as the ML patterns, turning the implicit constraints arising from metavariable names into explicit side conditions on the rules

$$\frac{\texttt{t}_1 \text{ is a numeric value}}{\texttt{pred (succ t}_1\texttt{)} \longrightarrow \texttt{t}_1} \qquad \text{(E-PredSucc)}$$

at some cost in compactness and readability.)

Finally, the `eval` function takes a term and finds its normal form by repeatedly calling `eval1`. Whenever `eval1` returns a new term $\texttt{t}'$, we make a recur-

sive call to `eval` to continue evaluating from `t'`. When `eval1` finally reaches a point where no rule applies, it raises the exception `NoRuleApplies`, causing `eval` to break out of the loop and return the final term in the sequence.[2]

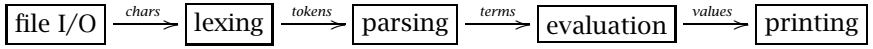
```
let rec eval t =
  try let t' = eval1 t
      in eval t'
  with NoRuleApplies → t
```

Obviously, this simple evaluator is tuned for easy comparison with the mathematical definition of evaluation, not for finding normal forms as quickly as possible. A somewhat more efficient algorithm can be obtained by starting instead from the "big-step" evaluation rules in Exercise 4.2.2.

4.2.2 EXERCISE [RECOMMENDED, ⋆⋆⋆ ↛]: Change the definition of the `eval` function in the `arith` implementation to the big-step style introduced in Exercise 3.5.17. □

4.3 The Rest of the Story

Of course, there are many parts to an interpreter or compiler—even a very simple one—besides those we have discussed explicitly here. In reality, terms to be evaluated start out as sequences of characters in files. They must be read from the file system, processed into streams of tokens by a lexical analyzer, and further processed into abstract syntax trees by a parser, before they can actually be evaluated by the functions that we have seen. Furthermore, after evaluation, the results need to be printed out.



Interested readers are encouraged to have a look at the on-line OCaml code for the whole interpreter.

2. We write `eval` this way for the sake of simplicity, but putting a `try` handler in a recursive loop is not actually very good style in ML.

4.2.1 EXERCISE [⋆⋆]: Why not? What is a better way to write `eval`? □