7 *An ML Implementation of the Lambda-Calculus*

In this chapter we construct an interpreter for the untyped lambda-calculus, based on the interpreter for arithmetic expressions in Chapter 4 and on the treatment of variable binding and substitution in Chapter 6.

An executable evaluator for untyped lambda-terms can be obtained by a straightforward translation of the foregoing definitions into OCaml. As in Chapter 4, we show just the core algorithms, ignoring issues of lexical analysis, parsing, printing, and so forth.

7.1 Terms and Contexts

We can obtain a datatype representing abstract syntax trees for terms by directly transliterating Definition 6.1.2:

```
type term =
    TmVar of int
  | TmAbs of term
  | TmApp of term * term
```

The representation of a variable is a number—its de Bruijn index. The representation of an abstraction carries just a subterm for the abstraction's body. An application carries the two subterms being applied.

The definition actually used in our implementation, however, will carry a little bit more information. First, as before, it is useful to annotate every term with an element of the type `info` recording the file position where that term was originally found, so that error printing routines can direct the user (or even the user's text editor, automatically) to the precise point where the error occurred.

The system studied in most of this chapter is the pure untyped lambda-calculus (Figure 5-3). The associated implementation is `untyped`. The `fulluntyped` implementation includes extensions such as numbers and booleans.

```
type term =
    TmVar of info * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

Second, for purposes of debugging, it is helpful to carry an extra number on each variable node, as a consistency check. The convention will be that this second number will always contain the *total length* of the context in which the variable occurs.

```
type term =
    TmVar of info * int * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

Whenever a variable is printed, we will verify that this number corresponds to the actual size of the current context; if it does not, then a shift operation has been forgotten someplace.

One last refinement also concerns printing. Although terms are represented internally using de Bruijn indices, this is obviously not how they should be presented to the user: we should convert from the ordinary representation to nameless terms during parsing, and convert back to ordinary form during printing. There is nothing very hard about this, but we should not do it completely naively (for example, generating completely fresh symbols for the names of variables), since then the names of the bound variables in the terms that are printed would have nothing to do with the names in the original program. This can be fixed by annotating each abstraction with a string to be used as a hint for the name of the bound variable.

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * term
  | TmApp of info * term * term
```

The basic operations on terms (substitution in particular) do not do anything fancy with these strings: they are simply carried along in their original form, with no checks for name clashes, capture, etc. When the printing routine needs to generate a fresh name for a bound variable, it tries first to use the supplied hint; if this turns out to clash with a name already used in the current context, it tries similar names, adding primes until it finds one that is not currently being used. This ensures that the printed term will be similar to what the user expects, modulo a few primes.

The printing routine itself looks like this:

```
let rec printtm ctx t = match t with
    TmAbs(fi,x,t1) →
      let (ctx',x') = pickfreshname ctx x in
      pr "(lambda "; pr x'; pr ". "; printtm ctx' t1; pr ")"
  | TmApp(fi, t1, t2) →
      pr "("; printtm ctx t1; pr " "; printtm ctx t2; pr ")"
  | TmVar(fi,x,n) →
      if ctxlength ctx = n then
        pr (index2name fi ctx x)
      else
        pr "[bad index]"
```

It uses the datatype `context`,

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

which is just a list of strings and associated `bindings`. For the [redacted]
bindings themselves are completely trivial

```
type binding = NameBind
```

carrying no interesting information. Later on (in Chapter 10), v
other clauses of the `binding` type that will keep track of the
tions associated with variables and other similar information.

The printing function also relies on several lower-level funct
a string to the standard output stream; `ctxlength` returns tl
context; `index2name` looks up the string name of a variable f
The most interesting one is `pickfreshname`, which takes a con
string hint `x`, finds a name `x′` similar to `x` such that `x′` is not al
`ctx`, adds `x′` to `ctx` to form a new context `ctx′`, and returns bo
as a pair.

The actual printing function found in the `untyped` impleme
book's web site is somewhat more complicated than this on
account two additional issues. First, it leaves out as many paren
sible, following the conventions that application associates to t
bodies of abstractions extend as far to the right as possible. Sec
ates formatting instructions for a low-level *pretty printing* modu
`Format` library) that makes decisions about line breaking and i

7.2 Shifting and Substitution

The definition of shifting (6.2.1) can be translated almost symb
into OCaml.

```
let termShift d t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x>=c then TmVar(fi,x+d,n+d)
                    else TmVar(fi,x,n+d)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

The internal shifting $\uparrow^d_c$ (t) is here represented by a call to the inner function walk c t. Since d never changes, there is no need to pass it along to each call to walk: we just use the outer binding of d when we need it in the variable case of walk. The top-level shift $\uparrow^d$ (t) is represented by termShift d t. (Note that termShift itself is not marked recursive, since all it does is call walk once.)

Similarly, the substitution function comes almost directly from Definition 6.2.4:

```
let termSubst j s t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x=j+c then termShift c s else TmVar(fi,x,n)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

The substitution [j ↦ s]t of term s for the variable numbered j in term t is written as termSubst j s t here. The only difference from the original definition of substitution is that here we do all the shifting of s at once, in the TmVar case, rather than shifting s up by one every time we go through a binder. This means that the argument j is the same in every call to walk, and we can omit it from the inner definition.

The reader may note that the definitions of termShift and termSubst are very similar, differing only in the action that is taken when a variable is reached. The untyped implementation available from the book's web site exploits this observation to express both shifting and substitution operations as special cases of a more general function called tmmap. Given a term t and a function onvar, the result of tmmap onvar t is a term of the same shape as t in which every variable has been replaced by the result of calling onvar on that variable. This notational trick saves quite a bit of tedious repetition in some of the larger calculi; §25.2 explains it in more detail.

In the operational semantics of the lambda-calculus, the only place where substitution is used is in the beta-reduction rule. As we noted before, this rule actually performs several operations: the term being substituted for the bound variable is first shifted up by one, then the substitution is made, and then the whole result is shifted down by one to account for the fact that the bound variable has been used up. The following definition encapsulates this sequence of steps:

```
let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
```

7.3 Evaluation

As in Chapter 3, the evaluation function depends on an auxiliary predicate `isval`:

```
let rec isval ctx t = match t with
    TmAbs(_,_,_) → true
  | _ → false
```

The single-step evaluation function is a direct transcription of the evaluation rules, except that we pass a context `ctx` along with the term. This argument is not used in the present `eval1` function, but it is needed by some of the more complex evaluators later on.

```
let rec eval1 ctx t = match t with
    TmApp(fi,TmAbs(_,x,t12),v2) when isval ctx v2 →
      termSubstTop v2 t12
  | TmApp(fi,v1,t2) when isval ctx v1 →
      let t2' = eval1 ctx t2 in
      TmApp(fi, v1, t2')
  | TmApp(fi,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmApp(fi, t1', t2)
  | _ →
      raise NoRuleApplies
```

The multi-step evaluation function is the same as before, except for the `ctx` argument:

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

7.3.1 Exercise [Recommended, ★★★ ↛]: Change this implementation to use the "big-step" style of evaluation introduced in Exercise 5.3.8. □

7.4 Notes

The treatment of substitution presented in this chapter, though sufficient for our purposes in this book, is far from the final word on the subject. In particular, the beta-reduction rule in our evaluator "eagerly" substitutes the argument value for the bound variable in the function's body. Interpreters (and compilers) for functional languages that are tuned for speed instead of simplicity use a different strategy: instead of actually performing the substitution, we simply record an association between the bound variable name and the argument value in an auxiliary data structure called the *environment*, which is carried along with the term being evaluated. When we reach a variable, we look up its value in the current environment. This strategy can be modeled by regarding the environment as a kind of *explicit substitution*—i.e., by moving the mechanism of substitution from the meta-language into the object language, making it a part of the *syntax* of the terms manipulated by the evaluator, rather than an external operation on terms. Explicit substitutions were first studied by Abadi, Cardelli, Curien, and Lévy (1991a) and have since become an active research area.

Just because you've implemented something doesn't mean you understand it.
—Brian Cantwell Smith