6 *Nameless Representation of Terms*

In the previous chapter, we worked with terms "up to renaming of bound variables," introducing a general convention that bound variables can be renamed, at any moment, to enable substitution or because a new name is more convenient for some other reason. In effect, the "spelling" of a bound variable name is whatever we want it to be. This convention works well for discussing basic concepts and for presenting proofs cleanly, but for building an implementation we need to choose a single representation for each term; in particular, we must decide how occurrences of variables are to be represented. There is more than one way to do this:

1. We can represent variables symbolically, as we have done so far, but replace the convention about implicit renaming of bound variables with an operation that explicitly replaces bound variables with "fresh" names during substitution as necessary to avoid capture.

2. We can represent variables symbolically, but introduce a general condition that the names of all bound variables must all be different from each other and from any free variables we may use. This convention (sometimes called the *Barendregt convention*) is more stringent than ours, since it does not allow renaming "on the fly" at arbitrary moments. However, it is not stable under substitution (or beta-reduction): since substitution involves copying the term being substituted, it is easy to construct examples where the result of substitution is a term in which some λ-abstractions have the same bound variable name. This implies that each evaluation step involving substitution must be followed by a step of renaming to restore the invariant.

3. We can devise some "canonical" representation of variables and terms that does not require renaming.

The system studied in this chapter is the pure untyped lambda-calculus, λ (Figure 5-3). The associated OCaml implementation is `fulluntyped`.

4. We can avoid substitution altogether by introducing mechanisms such as *explicit substitutions* (Abadi, Cardelli, Curien, and Lévy, 1991a).

5. We can avoid *variables* altogether by working in a language based directly on combinators, such as *combinatory logic* (Curry and Feys, 1958; Barendregt, 1984)—a variant of the lambda-calculus based on combinators instead of procedural abstraction—or Backus' functional language FP (1978).

Each scheme has its proponents, and choosing between them is somewhat a matter of taste (in serious compiler implementations, there are also performance considerations, but these do not concern us here). We choose the third, which, in our experience, scales better when we come to some of the more complex implementations later in the book. The main reason for this is that it tends to fail catastrophically rather than subtly when it is implemented wrong, allowing mistakes to be detected and corrected sooner rather than later. Bugs in implementations using named variables, by contrast, have been known to manifest months or years after they are introduced. Our formulation uses a well-known technique due to Nicolas de Bruijn (1972).

6.1 Terms and Contexts

De Bruijn's idea was that we can represent terms more straightforwardly—if less readably—by making variable occurrences *point directly to* their binders, rather than referring to them by name. This can be accomplished by replacing named variables by natural numbers, where the number `k` stands for "the variable bound by the *k*'th enclosing λ." For example, the ordinary term `λx.x` corresponds to the *nameless term* `λ.0`, while `λx.λy. x (y x)` corresponds to `λ.λ. 1 (0 1)`. Nameless terms are also sometimes called *de Bruijn terms*, and the numeric variables in them are called *de Bruijn indices*.[1] Compiler writers use the term "static distances" for the same concept.

6.1.1 EXERCISE [⋆]: For each of the following combinators

```
c0 = λs. λz. z;
c2 = λs. λz. s (s z);
plus = λm. λn. λs. λz. m s (n z s);
fix = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y));
foo = (λx. (λx. x)) (λx. x);
```

write down the corresponding nameless term. □

1. Note on pronunciation: the nearest English approximation to the second syllable in "de Bruijn" is "brown," not "broyn."

Formally, we define the syntax of nameless terms almost exactly like the syntax of ordinary terms (5.3.1). The only difference is that we need to keep careful track of how many free variables each term may contain. That is, we distinguish the sets of terms with no free variables (called the 0-*terms*), terms with at most one free variable (1-*terms*), and so on.

6.1.2 DEFINITION [TERMS]: Let $\mathcal{T}$ be the smallest family of sets $\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots\}$ such that

1. $\mathtt{k} \in \mathcal{T}_n$ whenever $0 \le \mathtt{k} < n$;
2. if $\mathtt{t}_1 \in \mathcal{T}_n$ and $n > 0$, then $\lambda.\mathtt{t}_1 \in \mathcal{T}_{n-1}$;
3. if $\mathtt{t}_1 \in \mathcal{T}_n$ and $\mathtt{t}_2 \in \mathcal{T}_n$, then $(\mathtt{t}_1\ \mathtt{t}_2) \in \mathcal{T}_n$.

(Note that this is a standard inductive definition, except that what we are defining is a *family* of sets indexed by numbers, rather than a single set.) The elements of each $\mathcal{T}_n$ are called n-terms. □

The elements of $\mathcal{T}_n$ are terms with *at most* n free variables, numbered between 0 and $n-1$: a given element of $\mathcal{T}_n$ need not have free variables with all these numbers, or indeed any free variables at all. When $\mathtt{t}$ is closed, for example, it will be an element of $\mathcal{T}_n$ for every n.

Note that each (closed) ordinary term has just one de Bruijn representation, and that two ordinary terms are equivalent modulo renaming of bound variables iff they have the same de Bruijn representation.

To deal with terms containing free variables, we need the idea of a *naming context*. For example, suppose we want to represent $\lambda\mathtt{x}.\ \mathtt{y}\ \mathtt{x}$ as a nameless term. We know what to do with $\mathtt{x}$, but we cannot see the binder for $\mathtt{y}$, so it is not clear how "far away" it might be and we do not know what number to assign to it. The solution is to choose, once and for all, an assignment (called a naming context) of de Bruijn indices to free variables, and use this assignment consistently when we need to choose numbers for free variables. For example, suppose that we choose to work under the following naming context:

$$\begin{array}{rcl} \Gamma & = & \mathtt{x} \mapsto 4 \\ & & \mathtt{y} \mapsto 3 \\ & & \mathtt{z} \mapsto 2 \\ & & \mathtt{a} \mapsto 1 \\ & & \mathtt{b} \mapsto 0 \end{array}$$

Then $\mathtt{x}\ (\mathtt{y}\ \mathtt{z})$ would be represented as $4\ (3\ 2)$, while $\lambda\mathtt{w}.\ \mathtt{y}\ \mathtt{w}$ would be represented as $\lambda.\ 4\ 0$ and $\lambda\mathtt{w}.\lambda\mathtt{a}.\mathtt{x}$ as $\lambda.\lambda.6$.

Since the order in which the variables appear in Γ determines their numerical indices, we can write it compactly as a sequence.

6.1.3 DEFINITION: Suppose x_0 through x_n are variable names from $\mathcal{V}$. The naming context $\Gamma = x_n, x_{n-1}, \ldots x_1, x_0$ assigns to each x_i the de Bruijn index i. Note that the rightmost variable in the sequence is given the index 0; this matches the way we count λ binders—from right to left—when converting a named term to nameless form. We write $dom(\Gamma)$ for the set $\{x_n, \ldots, x_0\}$ of variable names mentioned in Γ. □

6.1.4 EXERCISE [⋆⋆⋆ ↛]: Give an alternative construction of the sets of n-terms in the style of Definition 3.2.3, and show (as we did in Proposition 3.2.6) that it is equivalent to the one above. □

6.1.5 EXERCISE [RECOMMENDED, ⋆⋆⋆]:

1. Define a function $removenames_\Gamma(t)$ that takes a naming context Γ and an ordinary term t (with $FV(t) \subseteq dom(\Gamma)$) and yields the corresponding nameless term.

2. Define a function $restorenames_\Gamma(t)$ that takes a nameless term t and a naming context Γ and produces an ordinary term. (To do this, you will need to "make up" names for the variables bound by abstractions in t. You may assume that the names in Γ are pairwise distinct and that the set $\mathcal{V}$ of variable names is ordered, so that it makes sense to say "choose the first variable name that is not already in $dom(\Gamma)$.")

This pair of functions should have the property that

$$removenames_\Gamma(restorenames_\Gamma(t)) = t$$

for any nameless term t, and similarly

$$restorenames_\Gamma(removenames_\Gamma(t)) = t,$$

up to renaming of bound variables, for any ordinary term t. □

Strictly speaking, it does not make sense to speak of "some $t \in \mathcal{T}$"—we always need to specify how many free variables t might have. In practice, though, we will usually have some fixed naming context Γ in mind; we will then abuse the notation slightly and write $t \in \mathcal{T}$ to mean $t \in \mathcal{T}_n$, where n is the length of Γ.

6.2 Shifting and Substitution

Our next job is defining a substitution operation ($[k \mapsto s]t$) on nameless terms. To to this, we need one auxiliary operation, called "shifting," which renumbers the indices of the free variables in a term.

When a substitution goes under a λ-abstraction, as in $[1 \mapsto \mathtt{s}](\lambda.2)$ (i.e., $[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{x})$, assuming that 1 is the index of x in the outer context), the context in which the substitution is taking place becomes one variable longer than the original; we need to increment the indices of the free variables in s so that they keep referring to the same names in the new context as they did before. But we need to do this carefully: we can't just shift every variable index in s up by one, because this could also shift *bound* variables within s. For example, if $\mathtt{s} = 2\ (\lambda.0)$ (i.e., $\mathtt{s} = \mathtt{z}\ (\lambda\mathtt{w}.\mathtt{w})$, assuming 2 is the index of z in the outer context), we need to shift the 2 but not the 0. The shifting function below takes a "cutoff" parameter c that controls which variables should be shifted. It starts off at 0 (meaning all variables should be shifted) and gets incremented by one every time the shifting function goes through a binder. So, when calculating $\uparrow_c^d(\mathtt{t})$, we know that the term t comes from inside c-many binders in the original argument to $\uparrow^d$. Therefore all identifiers $\mathtt{k} < c$ in t are bound in the original argument and should not be shifted, while identifiers $\mathtt{k} \geq c$ in t are free and should be shifted.

6.2.1 DEFINITION [SHIFTING]: The d-place shift of a term t above cutoff c, written $\uparrow_c^d(\mathtt{t})$, is defined as follows:

$$\begin{array}{lcl} \uparrow_c^d(\mathtt{k}) & = & \begin{cases} \mathtt{k} & \text{if } k < c \\ \mathtt{k} + d & \text{if } k \geq c \end{cases} \\ \uparrow_c^d(\lambda.\mathtt{t}_1) & = & \lambda.\ \uparrow_{c+1}^d(\mathtt{t}_1) \\ \uparrow_c^d(\mathtt{t}_1\ \mathtt{t}_2) & = & \uparrow_c^d(\mathtt{t}_1)\ \uparrow_c^d(\mathtt{t}_2) \end{array}$$

We write $\uparrow^d(\mathtt{t})$ for $\uparrow_0^d(\mathtt{t})$. □

6.2.2 EXERCISE [⋆]:

1. What is $\uparrow^2(\lambda.\lambda.\ 1\ (0\ 2))$?
2. What is $\uparrow^2(\lambda.\ 0\ 1\ (\lambda.\ 0\ 1\ 2))$? □

6.2.3 EXERCISE [⋆⋆ ↛]: Show that if t is an n-term, then $\uparrow_c^d(\mathtt{t})$ is an $(n+d)$-term. □

Now we are ready to define the substitution operator $[\mathtt{j} \mapsto \mathtt{s}]\mathtt{t}$. When we use substitution, we will usually be interested in substituting for the *last* variable in the context (i.e., $\mathtt{j} = 0$), since that is the case we need in order to define the operation of beta-reduction. However, to substitute for variable 0 in a term that happens to be a λ-abstraction, we need to be able to substitute for the variable number numbered 1 in its body. Thus, the definition of substitution must work on an arbitrary variable.

6.2.4 DEFINITION [SUBSTITUTION]: The substitution of a term s for variable number j in a term t, written $[j \mapsto s]t$, is defined as follows:

$$\begin{array}{lcl} [j \mapsto s]k & = & \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases} \\ [j \mapsto s](\lambda.t_1) & = & \lambda.\, [j{+}1 \mapsto \uparrow^1(s)]t_1 \\ [j \mapsto s](t_1\ t_2) & = & ([j \mapsto s]t_1\ [j \mapsto s]t_2) \end{array}$$

□

6.2.5 EXERCISE [⋆]: Convert the following uses of substitution to nameless form, assuming the global context is $\Gamma = a,b$, and calculate their results using the above definition. Do the answers correspond to the original definition of substitution on ordinary terms from §5.3?

1. $[b \mapsto a]\ (b\ (\lambda x.\lambda y.b))$
2. $[b \mapsto a\ (\lambda z.a)]\ (b\ (\lambda x.b))$
3. $[b \mapsto a]\ (\lambda b.\ b\ a)$
4. $[b \mapsto a]\ (\lambda a.\ b\ a)$

□

6.2.6 EXERCISE [⋆⋆ ↛]: Show that if s and t are n-terms and $j \le n$, then $[j \mapsto s]t$ is an n-term. □

6.2.7 EXERCISE [⋆ ↛]: Take a sheet of paper and, without looking at the definitions of substitution and shifting above, regenerate them. □

6.2.8 EXERCISE [RECOMMENDED, ⋆⋆⋆]: The definition of substitution on nameless terms should agree with our informal definition of substitution on ordinary terms. (1) What theorem needs to be proved to justify this correspondence rigorously? (2) Prove it. □

6.3 Evaluation

To define the evaluation relation on nameless terms, the only thing we need to change (because it is the only place where variable names are mentioned) is the beta-reduction rule, which must now use our new nameless substitution operation.

The only slightly subtle point is that reducing a redex "uses up" the bound variable: when we reduce $((\lambda x.t_{12})\ v_2)$ to $[x \mapsto v_2]t_{12}$, the bound variable x disappears in the process. Thus, we will need to renumber the variables of

the result of substitution to take into account the fact that x is no longer part of the context. For example:

$$(\lambda.1\ 0\ 2)\ (\lambda.0) \longrightarrow 0\ (\lambda.0)\ 1 \qquad \text{(not } 1\ (\lambda.0)\ 2\text{)}.$$

Similarly, we need to shift the variables in v_2 up by one before substituting into t_{12}, since t_{12} is defined in a larger context than v_2. Taking these points into account, the beta-reduction rule looks like this:

$$(\lambda.t_{12})\ v_2 \longrightarrow \uparrow^{-1}([0 \mapsto \uparrow^1(v_2)]t_{12}) \qquad \text{(E-AppAbs)}$$

The other rules are identical to what we had before (Figure 5-3).

6.3.1 Exercise [⋆]: Should we be worried that the negative shift in this rule might create ill-formed terms containing negative indices? □

6.3.2 Exercise [⋆⋆⋆]: De Bruijn's original article actually contained two different proposals for nameless representations of terms: the deBruijn *indices* presented here, which number lambda-binders "from the inside out," and *de Bruijn levels,* which number binders "from the outside in." For example, the term λx. (λy. x y) x is represented using deBruijn indices as λ. (λ. 1 0) 0 and using deBruijn levels as λ. (λ. 0 1) 0. Define this variant precisely and show that the representations of a term using indices and levels are isomorphic (i.e., each can be recovered uniquely from the other). □