Chapter 10

Function Definitions and Values

In the language $\mathcal{L}\{\texttt{num str}\}$ we may perform calculations such as the doubling of a given expression, but we cannot express doubling as a concept in itself. To capture the general pattern of doubling, we abstract away from the particular number being doubled using a *variable* to stand for a fixed, but unspecified, number, to express the doubling of an arbitrary number. Any particular instance of doubling may then be obtained by substituting a numeric expression for that variable. In general an expression may involve many distinct variables, necessitating that we specify which of several possible variables is varying in a particular context, giving rise to a *function* of that variable.

In this chapter we will consider two extensions of $\mathcal{L}\{\texttt{num str}\}$ with functions. The first, and perhaps most obvious, extension is by adding *function definitions* to the language. A function is defined by binding a name to an abt with a bound variable that serves as the argument of that function. A function is *applied* by substituting a particular expression (of suitable type) for the bound variable, obtaining an expression.

The domain and range of defined functions are limited to the types `nat` and `str`, since these are the only types of expression. Such functions are called *first-order* functions, in contrast to *higher-order functions*, which permit functions as arguments and results of other functions. Since the domain and range of a function are types, this requires that we introduce *function types* whose elements are functions. Consequently, we may form functions of *higher type*, those whose domain and range may themselves be function types.

Historically the introduction of higher-order functions was responsible for a mistake in language design that subsequently was re-characterized as a feature, called *dynamic binding*. Dynamic binding arises from getting the definition of substitution wrong by failing to avoid capture. This makes the names of bound variables important, in violation of the fundamental principle of binding stating that the names of bound variables are unimportant.

10.1 First-Order Functions

The language $\mathcal{L}\{\texttt{num str fun}\}$ is the extension of $\mathcal{L}\{\texttt{num str}\}$ with function definitions and function applications as described by the following grammar:

Expr	e	::=	$\texttt{call}[f](e)$	$f(e)$ call
			$\texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e)$	$\texttt{fun}\, f(x_1:\tau_1):\tau_2 = e_2\ \texttt{in}\ e$ definition

The expression $\texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e)$ binds the function name f within e to the pattern $x_1.e_2$, which has parameter x_1 and definition e_2. The domain and range of the function are, respectively, the types τ_1 and τ_2. The expression $\texttt{call}[f](e)$ instantiates the binding of f with the argument e.

The statics of $\mathcal{L}\{\texttt{num str fun}\}$ defines two forms of judgement:

1. Expression typing, $e : \tau$, stating that e has type τ;
2. Function typing, $f(\tau_1) : \tau_2$, stating that f is a function with argument type τ_1 and result type τ_2.

The judgment $f(\tau_1) : \tau_2$ is called the *function header* of f; it specifies the domain type and the range type of a function.

The statics of $\mathcal{L}\{\texttt{num str fun}\}$ is defined by the following rules:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e) : \tau} \tag{10.1a}$$

$$\frac{\Gamma \vdash f(\tau_1) : \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{call}[f](e) : \tau_2} \tag{10.1b}$$

Function substitution, written $[\![x.e/f]\!]e'$, is defined by induction on the structure of e' much like the definition of ordinary substitution. However, a function name, f, is not a form of expression, but rather can only occur in

a call of the form `call[`f`](`e`)`. Function substitution for such expressions is defined by the following rule:

$$\overline{[\![x.e/f]\!]\texttt{call[}f\texttt{]}(e') = \texttt{let}([\![x.e/f]\!]e'\,x.e)} \tag{10.2}$$

At call sites to f with argument e', function substitution yields a `let` expression that binds x to the result of expanding any further calls to f within e'.

Lemma 10.1. *If* $\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau$ *and* $\Gamma, x_1 : \tau_2 \vdash e_2 : \tau_2$, *then* $\Gamma \vdash [\![x_1.e_2/f]\!]e : \tau$.

Proof. By induction on the structure of e'. □

The dynamics of $\mathcal{L}\{\texttt{num str fun}\}$ is defined using function substitution:

$$\overline{\texttt{fun[}\tau_1;\tau_2\texttt{]}(x_1.e_2; f.e) \mapsto [\![x_1.e_2/f]\!]e} \tag{10.3}$$

Since function substitution replaces all calls to f by appropriate `let` expressions, there is no need to give a rule for function calls.

The safety of $\mathcal{L}\{\texttt{num str fun}\}$ may be obtained as an immediate corollary of the safety theorem for higher-order functions, which we discuss next.

10.2 Higher-Order Functions

The syntactic and semantic similarity between variable definitions and function definitions in $\mathcal{L}\{\texttt{num str fun}\}$ is striking. This suggests that it may be possible to consolidate the two concepts into a single definition mechanism. The gap that must be bridged is the segregation of functions from expressions. A function name f is bound to an abstractor $x.e$ specifying a pattern that is instantiated when f is applied. To consolidate function definitions with expression definitions it is sufficient to *reify* the abstractor into a form of expression, called a *λ-abstraction*, written `lam[`τ_1`](`$x.e$`)`. Correspondingly, we must generalize application to have the form `ap(`e_1`;`e_2`)`, where e_1 is any expression, and not just a function name. These are, respectively, the introduction and elimination forms for the *function type*, `arr(`τ_1`;`τ_2`)`, whose elements are functions with domain τ_1 and range τ_2.

The language $\mathcal{L}\{\texttt{num str} \to\}$ is the enrichment of $\mathcal{L}\{\texttt{num str}\}$ with function types, as specified by the following grammar:

Type	τ	::=	$\texttt{arr}(\tau_1;\tau_2)$ $\quad \tau_1 \to \tau_2$	function
Expr	e	::=	$\texttt{lam}[\tau](x.e)$ $\quad \lambda(x{:}\tau.e)$	abstraction
			$\texttt{ap}(e_1;e_2)$ $\quad e_1(e_2)$	application

Functions are now "first class" in the sense that a function is an expression of function type.

The statics of $\mathcal{L}\{\texttt{num str} \to\}$ is given by extending Rules (6.1) with the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{arr}(\tau_1;\tau_2)} \tag{10.4a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{10.4b}$$

Lemma 10.2 (Inversion). *Suppose that $\Gamma \vdash e : \tau$.*

1. *If $e = \texttt{lam}[\tau_1](x.e)$, then $\tau = \texttt{arr}(\tau_1;\tau_2)$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.*
2. *If $e = \texttt{ap}(e_1;e_2)$, then there exists τ_2 such that $\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau)$ and $\Gamma \vdash e_2 : \tau_2$.*

Proof. The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies, and that the premises of the rule in question provide the required result. □

Lemma 10.3 (Substitution). *If $\Gamma, x : \tau \vdash e' : \tau'$, and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$.*

Proof. By rule induction on the derivation of the first judgement. □

The dynamics of $\mathcal{L}\{\texttt{num str} \to\}$ extends that of $\mathcal{L}\{\texttt{num str}\}$ with the following additional rules:

$$\frac{}{\texttt{lam}[\tau](x.e)\ \texttt{val}} \tag{10.5a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{10.5b}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1} \tag{10.5c}$$

These rules specify a call-by-name discipline for function application. It is a good exercise to formulate a call-by-value discipline as well.

Theorem 10.4 (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. The proof is by induction on rules (10.5), which define the dynamics of the language.

Consider rule (10.5c),

$$\frac{}{\mathtt{ap(lam[}\tau_2\mathtt{]}(x.e_1);e_2) \mapsto [e_2/x]e_1}\ .$$

Suppose that $\mathtt{ap(lam[}\tau_2\mathtt{]}(x.e_1);e_2) : \tau_1$. By Lemma 10.2 on the facing page $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 10.3 on the preceding page $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 10.5 (Canonical Forms). *If e val and $e : \mathtt{arr}(\tau_1;\tau_2)$, then $e = \mathtt{lam[}\tau_1\mathtt{]}(x.e_2)$ for some x and e_2 such that $x : \tau_1 \vdash e_2 : \tau_2$.*

Proof. By induction on the typing rules, using the assumption e val. □

Theorem 10.6 (Progress). *If $e : \tau$, then either e is a value, or there exists e' such that $e \mapsto e'$.*

Proof. The proof is by induction on rules (10.4). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (10.4b). By induction either e_1 val or $e_1 \mapsto e_1'$. In the latter case we have $\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)$. In the former case, we have by Lemma 10.5 that $e_1 = \mathtt{lam[}\tau_2\mathtt{]}(x.e)$ for some x and e. But then $\mathtt{ap}(e_1;e_2) \mapsto [e_2/x]e$. □

10.3 Evaluation Dynamics and Definitional Equivalence

An inductive definition of the evaluation judgement $e \Downarrow v$ for $\mathcal{L}\{\mathtt{num\ str} \to\}$ is given by the following rules:

$$\frac{}{\mathtt{lam[}\tau\mathtt{]}(x.e) \Downarrow \mathtt{lam[}\tau\mathtt{]}(x.e)} \tag{10.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam[}\tau\mathtt{]}(x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v} \tag{10.6b}$$

It is easy to check that if $e \Downarrow v$, then v val, and that if e val, then $e \Downarrow e$.

Theorem 10.7. *$e \Downarrow v$ iff $e \mapsto^* v$ and v val.*

Proof. In the forward direction we proceed by rule induction on Rules (10.6). The proof makes use of a *pasting lemma* stating that, for example, if $e_1 \mapsto^* e_1'$, then $\mathtt{ap}(e_1; e_2) \mapsto^* \mathtt{ap}(e_1'; e_2)$, and similarly for the other constructs of the language.

In the reverse direction we proceed by rule induction on Rules (7.1). The proof relies on a *converse evaluation lemma*, which states that if $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$. This is proved by rule induction on Rules (10.5). □

Definitional equivalence for the call-by-name dynamics of $\mathcal{L}\{\mathtt{num}\ \mathtt{str} \to\}$ is defined by a straightforward extension to Rules (7.11).

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{10.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \to \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2') : \tau} \tag{10.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{10.7c}$$

Definitional equivalence for call-by-value requires a small bit of additional machinery. The main idea is to restrict Rule (10.7a) to require that the argument be a value. However, to be fully expressive, we must also widen the concept of a value to include all variables that are in scope, so that Rule (10.7a) would apply even when the argument is a variable. The justification for this is that in call-by-value, the parameter of a function stands for the value of its argument, and not for the argument itself. The call-by-value definitional equivalence judgement has the form

$$\Xi\, \Gamma \vdash e_1 \equiv e_2 : \tau,$$

where Ξ is the finite set of hypotheses $x_1\ \mathsf{val}, \ldots, x_k\ \mathsf{val}$ governing the variables in scope at that point. We write $\Xi \vdash e\ \mathsf{val}$ to indicate that e is a value under these hypotheses, so that, for example, $\Xi, x\ \mathsf{val} \vdash x\ \mathsf{val}$.

The rule of definitional equivalence for call-by-value are similar to those for call-by-name, modified to take account of the scopes of value variables. Two illustrative rules are as follows:

$$\frac{\Xi, x\ \mathsf{val}\, \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\, \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{10.8a}$$

$$\frac{\Xi \vdash e_1\ \mathsf{val}}{\Xi\, \Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau}. \tag{10.8b}$$

10.4 Dynamic Scope

The dynamics of function application given by Rules (10.5) is defined only for expressions without free variables. When a function is called, the argument is substituted for the function parameter, ensuring that the result remains closed. Moreover, since substitution of closed expressions can never incur capture, the scopes of variables are not disturbed by the dynamics, ensuring that the principles of binding and scope described in Chapter 1 are respected. This treatment of variables is called *static scoping*, or *static binding*, to contrast it with an alternative approach that we now describe.

Another approach, called *dynamic scoping*, or *dynamic binding*, is sometimes advocated as an alternative to static binding. Evaluation is defined for expressions that may contain free variables. Evaluation of a variable is undefined; it is an error to ask for the value of an unbound variable. Function call is defined similarly to dynamic binding, *except* that when a function is called, the argument *replaces* the parameter in the body, possibly *incurring*, rather than avoiding, capture of free variables in the argument. (As we will explain shortly, this behavior is considered to be a feature, not a bug!)

The difference between replacement and substitution may be illustrated by example. Let e be the expression $\lambda\,(x{:}\mathtt{str}.\,y + |x|)$ in which the variable y occurs free, and let e' be the expression $\lambda\,(y{:}\mathtt{str}.\,f(y))$ with free variable f. If we *substitute* e for f in e' we obtain an expression of the form

$$\lambda\,(y'{:}\mathtt{str}.\,\lambda\,(x{:}\mathtt{str}.\,y + |x|)(y')),$$

where the bound variable, y, in e has been renamed to some fresh variable y' so as to avoid capture. If we instead *replace* f by e in e' we obtain

$$\lambda\,(y{:}\mathtt{str}.\,\lambda\,(x{:}\mathtt{str}.\,y + |x|)(y))$$

in which y is no longer free: it has been captured during replacement.

The implications of this seemingly small change to the dynamics of $\mathcal{L}\{\to\}$ are far-reaching. The most obvious implication is that the language is not type safe. In the above example we have that $y : \mathtt{nat} \vdash e : \mathtt{str} \to \mathtt{nat}$, and that $f : \mathtt{str} \to \mathtt{nat} \vdash e' : \mathtt{str} \to \mathtt{nat}$. It follows that $y : \mathtt{nat} \vdash [e/f]e' : \mathtt{str} \to \mathtt{nat}$, but it is easy to see that the result of replacing f by e in e' is ill-typed, regardless of what assumption we make about y. The difficulty, of course, is that the bound occurrence of y in e' has type $\mathtt{str}$, whereas the free occurrence in e must have type $\mathtt{nat}$ in order for e to be well-formed.

One way around this difficulty is to ignore types altogether, and rely on run-time checks to ensure that bad things do not happen, despite the

evident failure of safety. (See Chapter 20 for a full exploration of this approach.) But even if we ignore worries about safety, we are still left with the serious problem that the names of bound variables matter, and cannot be altered without changing the meaning of a program. So, for example, to use expression e', one must bear in mind that the parameter, f, occurs within the scope of a binder for y, a fact that is not revealed by the type of e' (and certainly not if one disregards types entirely!) If we change e' so that it binds a different variable, say z, then we must correspondingly change e to ensure that it refers to z, and not y, in order to preserve the overall behavior of the system of two expressions. This means that e and e' must be developed in tandem, violating a basic principle of modular decomposition. (For more on dynamic scope, please see Chapter 35.)

10.5 Notes

Nearly all programming languages provide some form of function definition mechanism of the kind illustrated here. The main point of the present account is to demonstrate that a more natural, and more powerful, approach is to separate the generic concept of a definition from the specific concept of a function. Function types codify the general notion in a systematic manner that encompasses function definitions as a special case, and moreover, admits passing functions as arguments and returning them as results without special provision. The essential contribution of Church's λ-calculus [20] was to take the notion of function as primary, and indeed to point out that nothing more is needed to obtain a fully expressive programming language. The defining feature of *functional* programming languages is precisely that functions are first-class values that can be handled without special provision or restriction. This, too, is a central feature of *object-oriented* languages: objects consist of methods acting on private data, and are nothing more than functions combined with storage effects.

Chapter 11

Gödel's System T

The language $\mathcal{L}\{\mathtt{nat} \to\}$, better known as *Gödel's System T*, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}\{\mathtt{num\,str}\}$, which equips the naturals with some arbitrarily chosen arithmetic primitives, the language $\mathcal{L}\{\mathtt{nat} \to\}$ provides a general mechanism, called *primitive recursion*, from which these primitives may be defined. Primitive recursion captures the essential inductive character of the natural numbers, and hence may be seen as an intrinsic termination proof for each program in the language. Consequently, we may only define *total* functions in the language, those that always return a value for each argument. In essence every program in $\mathcal{L}\{\mathtt{nat} \to\}$ "comes equipped" with a proof of its termination. While this may seem like a shield against infinite loops, it is also a weapon that can be used to show that some programs cannot be written in $\mathcal{L}\{\mathtt{nat} \to\}$. To do so would require a master termination proof for every possible program in the language, something that we shall prove does not exist.

11.1 Statics

The syntax of $\mathcal{L}\{\mathtt{nat}\to\}$ is given by the following grammar:

Type	τ	::=	$\mathtt{nat}$	naturals
			$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$ function
Expr	e	::=	x	x variable
			$\mathtt{z}$	$\mathtt{z}$ zero
			$\mathtt{s}(e)$	$\mathtt{s}(e)$ successor
			$\mathtt{natrec}(e;e_0;x.y.e_1)$	$\mathtt{natrec}\,e\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\}$ recursion
			$\mathtt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.\,e)$ abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$ application

We write $\overline{n}$ for the expression $\mathtt{s}(\ldots\mathtt{s}(\mathtt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression $\mathtt{natrec}(e;e_0;x.y.e_1)$ is called *primitive recursion*. It represents the e-fold iteration of the transformation $x.y.e_1$ starting from e_0. The bound variable x represents the predecessor and the bound variable y represents the result of the x-fold iteration. The "`with`" clause in the concrete syntax for the recursor binds the variable y to the result of the recursive call, as will become apparent shortly.

Sometimes *iteration*, written $\mathtt{natiter}(e;e_0;y.e_1)$, is considered as an alternative to primitive recursion. It has essentially the same meaning as primitive recursion, except that only the result of the recursive call is bound to y in e_1, and no binding is made for the predecessor. Clearly iteration is a special case of primitive recursion, since we can always ignore the predecessor binding. Conversely, primitive recursion is definable from iteration, provided that we have product types (Chapter 13) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

The statics of $\mathcal{L}\{\mathtt{nat}\to\}$ is given by the following typing rules:

$$\frac{}{\Gamma, x : \mathtt{nat} \vdash x : \mathtt{nat}} \tag{11.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{11.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{11.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{natrec}(e;e_0;x.y.e_1) : \tau} \tag{11.1d}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \mathtt{lam}[\sigma](x.e) : \mathtt{arr}(\sigma;\tau)} \tag{11.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{11.1f}$$

As usual, admissibility of the structural rule of substitution is crucially important.

Lemma 11.1. *If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$, then $\Gamma \vdash [e/x]e' : \tau'$.*

11.2 Dynamics

The dynamics of $\mathcal{L}\{\mathtt{nat} \to\}$ adopts a call-by-name interpretation of function application, and requires that the successor operation evaluate its argument (so that values of type `nat` are numerals).

The closed values of $\mathcal{L}\{\mathtt{nat} \to\}$ are determined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{11.2a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}} \tag{11.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{11.2c}$$

The dynamics of $\mathcal{L}\{\mathtt{nat} \to\}$ is given by the following rules:

$$\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')} \tag{11.3a}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{11.3b}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{11.3c}$$

$$\frac{e \mapsto e'}{\mathtt{natrec}(e;e_0;x.y.e_1) \mapsto \mathtt{natrec}(e';e_0;x.y.e_1)} \tag{11.3d}$$

$$\frac{}{\mathtt{natrec}(\mathtt{z};e_0;x.y.e_1) \mapsto e_0} \tag{11.3e}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{natrec}(\mathtt{s}(e);e_0;x.y.e_1) \mapsto [e,\mathtt{natrec}(e;e_0;x.y.e_1)/x,y]e_1} \tag{11.3f}$$

Rules (11.3e) and (11.3f) specify the behavior of the recursor on $\mathtt{z}$ and $\mathtt{s}(e)$. In the former case the recursor evaluates e_0, and in the latter case the variable x is bound to the predecessor, e, and y is bound to the (unevaluated) recursion on e. If the value of y is not required in the rest of the computation, the recursive call will not be evaluated.

Lemma 11.2 (Canonical Forms). *If $e : \tau$ and e val, then*

1. *If $\tau = \mathtt{nat}$, then $e = \mathtt{s}(\mathtt{s}(\ldots \mathtt{z}))$ for some number $n \geq 0$ occurrences of the successor starting with zero.*
2. *If $\tau = \tau_1 \to \tau_2$, then $e = \lambda\,(x{:}\tau_1.\, e_2)$ for some e_2.*

Theorem 11.3 (Safety). 1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*
2. *If $e : \tau$, then either e val or $e \mapsto e'$ for some e'*

11.3 Definability

A mathematical function $f : \mathbb{N} \to \mathbb{N}$ on the natural numbers is *definable* in $\mathcal{L}\{\mathtt{nat} \to\}$ iff there exists an expression e_f of type $\mathtt{nat} \to \mathtt{nat}$ such that for every $n \in \mathbb{N}$,

$$e_f(\overline{n}) \equiv \overline{f(n)} : \mathtt{nat}. \tag{11.4}$$

That is, the numeric function $f : \mathbb{N} \to \mathbb{N}$ is definable iff there is an expression e_f of type $\mathtt{nat} \to \mathtt{nat}$ such that, when applied to the numeral representing the argument $n \in \mathbb{N}$, is definitionally equivalent to the numeral corresponding to $f(n) \in \mathbb{N}$.

Definitional equivalence for $\mathcal{L}\{\mathtt{nat} \to\}$, written $\Gamma \vdash e \equiv e' : \tau$, is the strongest congruence containing these axioms:

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau} \tag{11.5a}$$

$$\frac{}{\Gamma \vdash \mathtt{natrec}(\mathtt{z}; e_0; x.y.e_1) \equiv e_0 : \tau} \tag{11.5b}$$

$$\frac{}{\Gamma \vdash \mathtt{natrec}(\mathtt{s}(e); e_0; x.y.e_1) \equiv [e, \mathtt{natrec}(e; e_0; x.y.e_1)/x, y]e_1 : \tau} \tag{11.5c}$$

For example, the doubling function, $d(n) = 2 \times n$, is definable in $\mathcal{L}\{\mathtt{nat} \to\}$ by the expression $e_d : \mathtt{nat} \to \mathtt{nat}$ given by

$$\lambda\,(x{:}\mathtt{nat}.\mathtt{natrec}\, x\, \{\mathtt{z} \Rightarrow \mathtt{z} \mid \mathtt{s}(u)\ \mathtt{with}\ v \Rightarrow \mathtt{s}(\mathtt{s}(v))\}).$$

To check that this defines the doubling function, we proceed by induction on $n \in \mathbb{N}$. For the basis, it is easy to check that

$$e_d(\overline{0}) \equiv \overline{0} : \mathtt{nat}.$$

For the induction, assume that

$$e_d(\overline{n}) \equiv \overline{d(n)} : \mathtt{nat}.$$

Then calculate using the rules of definitional equivalence:

$$\begin{aligned} e_d(\overline{n+1}) &\equiv \mathtt{s}(\mathtt{s}(e_d(\overline{n}))) \\ &\equiv \mathtt{s}(\mathtt{s}(\overline{2 \times n})) \\ &= \overline{2 \times (n+1)} \\ &= \overline{d(n+1)}. \end{aligned}$$

As another example, consider the following function, called *Ackermann's function*, defined by the following equations:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)). \end{aligned}$$

This function grows very quickly. For example, $A(4,2) \approx 2^{65,536}$, which is often cited as being much larger than the number of atoms in the universe! Yet we can show that the Ackermann function is total by a lexicographic induction on the pair of argument (m,n). On each recursive call, either m decreases, or else m remains the same, and n decreases, so inductively the recursive calls are well-defined, and hence so is $A(m,n)$.

A *first-order primitive recursive function* is a function of type $\mathtt{nat} \to \mathtt{nat}$ that is defined using primitive recursion, but without using any higher order functions. Ackermann's function is defined so that it is not first-order primitive recursive, but is higher-order primitive recursive. The key is to showing that it is definable in $\mathcal{L}\{\mathtt{nat} \to\}$ is to observe that $A(m+1,n)$ iterates the function $A(m,-)$ for n times, starting with $A(m,1)$. As an auxiliary, let us define the higher-order function

$$\mathtt{it} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the λ-abstraction

$$\lambda\,(f{:}\mathtt{nat} \to \mathtt{nat}.\,\lambda\,(n{:}\mathtt{nat}.\,\mathtt{natrec}\,n\,\{\mathtt{z} \Rightarrow \mathtt{id} \mid \mathtt{s}(\_)\ \mathtt{with}\ g \Rightarrow f \circ g\})),$$

where $\mathtt{id} = \lambda\,(x{:}\mathtt{nat}.\,x)$ is the identity, and $f \circ g = \lambda\,(x{:}\mathtt{nat}.\,f(g(x)))$ the composition of f and g. It is easy to check that

$$\mathtt{it}(f)(\overline{n})(\overline{m}) \equiv f^{(n)}(\overline{m}) : \mathtt{nat},$$

where the latter expression is the n-fold composition of f starting with $\overline{m}$. We may then define the Ackermann function

$$e_a : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the expression

$$\lambda\,(m{:}\mathtt{nat}.\,\mathtt{natrec}\,m\,\{\mathtt{z} \Rightarrow \mathtt{succ} \mid \mathtt{s}(\_)\ \mathtt{with}\ f \Rightarrow \lambda\,(n{:}\mathtt{nat}.\,\mathtt{it}(f)(n)(f(\overline{1})))\}).$$

It is instructive to check that the following equivalences are valid:

$$e_a(\overline{0})(\overline{n}) \equiv \mathtt{s}(\overline{n}) \tag{11.6}$$

$$e_a(\overline{m+1})(\overline{0}) \equiv e_a(\overline{m})(\overline{1}) \tag{11.7}$$

$$e_a(\overline{m+1})(\overline{n+1}) \equiv e_a(\overline{m})(e_a(\mathtt{s}(\overline{m}))(\overline{n})). \tag{11.8}$$

That is, the Ackermann function is definable in $\mathcal{L}\{\mathtt{nat} \to\}$.

11.4 Undefinability

It is impossible to define an infinite loop in $\mathcal{L}\{\mathtt{nat} \to\}$.

Theorem 11.4. *If $e : \tau$, then there exists v val such that $e \equiv v : \tau$.*

Proof. See Corollary 49.11 on page 493. □

Consequently, values of function type in $\mathcal{L}\{\mathtt{nat} \to\}$ behave like mathematical functions: if $f : \sigma \to \tau$ and $e : \sigma$, then $f(e)$ evaluates to a value of type τ. Moreover, if $e : \mathtt{nat}$, then there exists a natural number n such that $e \equiv \overline{n} : \mathtt{nat}$.

Using this, we can show, using a technique called *diagonalization*, that there are functions on the natural numbers that are not definable in the $\mathcal{L}\{\mathtt{nat} \to\}$. We make use of a technique, called *Gödel-numbering*, that assigns a unique natural number to each closed expression of $\mathcal{L}\{\mathtt{nat} \to\}$. This allows us to manipulate expressions as data values in $\mathcal{L}\{\mathtt{nat} \to\}$, and hence permits $\mathcal{L}\{\mathtt{nat} \to\}$ to compute with its own programs.[1]

The essence of Gödel-numbering is captured by the following simple construction on abstract syntax trees. (The generalization to abstract binding trees is slightly more difficult, the main complication being to ensure

[1] The same technique lies at the heart of the proof of Gödel's celebrated incompleteness theorem. The undefinability of certain functions on the natural numbers within $\mathcal{L}\{\mathtt{nat} \to\}$ may be seen as a form of incompleteness similar to that considered by Gödel.

that α-equivalent expressions are assigned the same Gödel number.) Recall that a general ast, a, has the form $o(a_1, \ldots, a_k)$, where o is an operator of arity k. Fix an enumeration of the operators so that every operator has an index $i \in \mathbb{N}$, and let m be the index of o in this enumeration. Define the *Gödel number* $\ulcorner a \urcorner$ of a to be the number

$$2^m 3^{n_1} 5^{n_2} \ldots p_k^{n_k},$$

where p_k is the kth prime number (so that $p_0 = 2$, $p_1 = 3$, and so on), and $n_1, \ldots, n_k$ are the Gödel numbers of $a_1, \ldots, a_k$, respectively. This obviously assigns a natural number to each ast. Conversely, given a natural number, n, we may apply the prime factorization theorem to "parse" n as a unique abstract syntax tree. (If the factorization is not of the appropriate form, which can only be because the arity of the operator does not match the number of factors, then n does not code any ast.)

Now, using this representation, we may define a (mathematical) function $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that, for any $e : \mathtt{nat} \to \mathtt{nat}$, $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \mathtt{nat}$.[2] The determinacy of the dynamics, together with Theorem 11.4 on the facing page, ensure that f_{univ} is a well-defined function. It is called the *universal function* for $\mathcal{L}\{\mathtt{nat} \to\}$ because it specifies the behavior of any expression e of type $\mathtt{nat} \to \mathtt{nat}$. Using the universal function, let us define an auxiliary mathematical function, called the *diagonal function*, $d : \mathbb{N} \to \mathbb{N}$, by the equation $d(m) = f_{univ}(m)(m)$. This function is chosen so that $d(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \mathtt{nat}$. (The motivation for this definition will be apparent in a moment.)

The function d is not definable in $\mathcal{L}\{\mathtt{nat} \to\}$. Suppose that d were defined by the expression e_d, so that we have

$$e_d(\overline{\ulcorner e \urcorner}) \equiv e(\overline{\ulcorner e \urcorner}) : \mathtt{nat}.$$

Let e_D be the expression

$$\lambda\,(x{:}\mathtt{nat}.\,\mathtt{s}(e_d(x)))$$

of type $\mathtt{nat} \to \mathtt{nat}$. We then have

$$\begin{aligned} e_D(\overline{\ulcorner e_D \urcorner}) &\equiv \mathtt{s}(e_d(\overline{\ulcorner e_D \urcorner})) \\ &\equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner})). \end{aligned}$$

[2] The value of $f_{univ}(k)(m)$ may be chosen arbitrarily to be zero when k is not the code of any expression e.

But the termination theorem implies that there exists n such that $e_D(\ulcorner \overline{e_D} \urcorner) \equiv \overline{n}$, and hence we have $\overline{n} \equiv \mathtt{s}(\overline{n})$, which is impossible.

The function f_{univ} is computable (that is, one can write an interpreter for $\mathcal{L}\{\mathtt{nat} \to\}$), but it is not programmable in $\mathcal{L}\{\mathtt{nat} \to\}$ itself. In general a language $\mathcal{L}$ is *universal* if we can write an interpreter for $\mathcal{L}$ in the language $\mathcal{L}$ itself. The foregoing argument shows that $\mathcal{L}\{\mathtt{nat} \to\}$ is *not universal*. Consequently, there are computable numeric functions, such as the diagonal function, that cannot be programmed in $\mathcal{L}\{\mathtt{nat} \to\}$. Consequently, the universal function for $\mathcal{L}\{\mathtt{nat} \to\}$ cannot be programmed in the language. In other words, one cannot write an interpreter for $\mathcal{L}\{\mathtt{nat} \to\}$ in the language itself!

11.5 Notes

$\mathcal{L}\{\mathtt{nat} \to\}$ was introduced by Gödel in his study of proofs of proving the consistency of arithmetic [32]. In this paper Gödel showed how to "compile" proofs in arithmetic into well-typed terms of the language $\mathcal{L}\{\mathtt{nat} \to\}$, and thereby that consistency of arithmetic is equivalent to the termination (more precisely, normalization) of programs in $\mathcal{L}\{\mathtt{nat} \to\}$. This was perhaps the first programming language whose design was directly influenced by consideration of verification (of termination) of its programs.

Chapter 12

Plotkin's PCF

The language $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, also known as *Plotkin's* ***PCF***, integrates functions and natural numbers using *general recursion*, a means of defining self-referential expressions. In contrast to $\mathcal{L}\{\mathtt{nat} \to\}$ expressions in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ may not terminate when evaluated; consequently, functions are partial (may be undefined for some arguments), rather than total (which explains the "partial arrow" notation for function types). Compared to $\mathcal{L}\{\mathtt{nat} \to\}$, the language $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ moves the termination proof from the expression itself to the mind of the programmer. The type system no longer ensures termination, which permits a wider range of functions to be defined in the system, but at the cost of admitting infinite loops when the termination proof is either incorrect or absent.

The crucial concept embodied in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is the *fixed point* characterization of recursive definitions. In ordinary mathematical practice one may define a function f by *recursion equations* such as these:

$$\begin{aligned} f(0) &= 1 \\ f(n+1) &= (n+1) \times f(n) \end{aligned}$$

These may be viewed as simultaneous equations in the variable, f, ranging over functions on the natural numbers. The function we seek is a *solution* to these equations—a function $f : \mathbb{N} \to \mathbb{N}$ such that the above conditions are satisfied. We must, of course, show that these equations have a unique solution, which is easily shown by mathematical induction on the argument to f.

The solution to such a system of equations may be characterized as the fixed point of an associated functional (operator mapping functions to

functions). To see this, let us re-write these equations in another form:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Re-writing yet again, we seek f such that

$$f : n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Now define the *functional* F by the equation $F(f) = f'$, where

$$f' : n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Note well that the condition on f' is expressed in terms of the argument, f, to the functional F, and not in terms of f' itself! The function f we seek is then a *fixed point* of F, which is a function $f : \mathbb{N} \to \mathbb{N}$ such that $f = F(f)$. In other words f is defined to the $fix(F)$, where fix is an operator on functionals yielding a fixed point of F.

Why does an operator such as F have a fixed point? Informally, a fixed point may be obtained as the limit of series of approximations to the desired solution obtained by iterating the functional F. This is where partial functions come into the picture. Let us say that a partial function, ϕ on the natural numbers, is an *approximation* to a total function, f, if $\phi(m) = n$ implies that $f(m) = n$. Let $\bot : \mathbb{N} \rightharpoonup \mathbb{N}$ be the totally undefined partial function—$\bot(n)$ is undefined for every $n \in \mathbb{N}$. Intuitively, this is the "worst" approximation to the desired solution, f, of the recursion equations given above. Given any approximation, ϕ, of f, we may "improve" it by considering $\phi' = F(\phi)$. Intuitively, ϕ' is defined on 0 and on $m + 1$ for every $m \geq 0$ on which ϕ is defined. Continuing in this manner, $\phi'' = F(\phi') = F(F(\phi))$ is an improvement on ϕ', and hence a further improvement on ϕ. If we start with $\bot$ as the initial approximation to f, then pass to the limit

$$\lim_{i \geq 0} F^{(i)}(\bot),$$

we will obtain the least approximation to f that is defined for every $m \in \mathbb{N}$, and hence is the function f itself. Turning this around, if the limit exists, it must be the solution we seek.

This fixed point characterization of recursion equations is taken as a primitive concept in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$—we may obtain the least fixed point of *any*

functional definable in the language. Using this we may solve any set of recursion equations we like, with the proviso that there is no guarantee that the solution is a *total* function. Rather, it is guaranteed to be a *partial* function that may be undefined on some, all, or no inputs. This is the price we pay for expressive power—we may solve all systems of equations, but the solution may not be as well-behaved as we might like. It is our task as programmers to ensure that the functions defined by recursion are total—all of our loops terminate.

12.1 Statics

The abstract binding syntax of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is given by the following grammar:

Type	τ	::=	$\mathtt{nat}$	naturals
		$\mathtt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial function
Expr	e	::=	x	variable
		$\mathtt{z}$	$\mathtt{z}$	zero
		$\mathtt{s}(e)$	$\mathtt{s}(e)$	successor
		$\mathtt{ifz}(e;e_0;x.e_1)$	$\mathtt{ifz}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}$	zero test
		$\mathtt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.e)$	abstraction
		$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
		$\mathtt{fix}[\tau](x.e)$	$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e$	recursion

The expression $\mathtt{fix}[\tau](x.e)$ is called *general recursion*; it is discussed in more detail below. The expression $\mathtt{ifz}(e;e_0;x.e_1)$ branches according to whether e evaluates to $\mathtt{z}$ or not, binding the predecessor to x in the case that it is not.

The statics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \tag{12.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z}:\mathtt{nat}} \tag{12.1b}$$

$$\frac{\Gamma \vdash e:\mathtt{nat}}{\Gamma \vdash \mathtt{s}(e):\mathtt{nat}} \tag{12.1c}$$

$$\frac{\Gamma \vdash e:\mathtt{nat} \quad \Gamma \vdash e_0:\tau \quad \Gamma, x:\mathtt{nat} \vdash e_1:\tau}{\Gamma \vdash \mathtt{ifz}(e;e_0;x.e_1):\tau} \tag{12.1d}$$

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e):\mathtt{parr}(\tau_1;\tau_2)} \tag{12.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{12.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e) : \tau} \tag{12.1g}$$

Rule (12.1g) reflects the self-referential nature of general recursion. To show that $\mathtt{fix}[\tau](x.e)$ has type τ, we *assume* that it is the case by assigning that type to the variable, x, which stands for the recursive expression itself, and checking that the body, e, has type τ under this very assumption.

The structural rules, including in particular substitution, are admissible for the static semantics.

Lemma 12.1. *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

12.2 Dynamics

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined by the judgements e val, specifying the closed values, and $e \mapsto e'$, specifying the steps of evaluation. We will consider a call-by-name dynamics for function application, and require that the successor evaluate its argument.

The judgement e val is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{12.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{s}(e)\ \mathsf{val}} \tag{12.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{12.2c}$$

The bracketed premise on Rule (12.2b) is to be included for the *eager* interpretation of the sucessor operation, and omitted for the *lazy* interpretation. (See Section 12.4 on page 112 for more on this choice, which is further elaborated in Chapter 39).

The transition judgement $e \mapsto e'$ is defined by the following rules:

$$\left\{\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}\right\} \tag{12.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e;e_0;x.e_1) \mapsto \mathtt{ifz}(e';e_0;x.e_1)} \tag{12.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \mapsto e_0} \tag{12.3c}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \mapsto [e/x]e_1} \tag{12.3d}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{12.3e}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{12.3f}$$

$$\frac{}{\mathtt{fix}[\tau](x.e) \mapsto [\mathtt{fix}[\tau](x.e)/x]e} \tag{12.3g}$$

The bracketed Rule (12.3a) is to be included for an eager interpretation of the successor, and omitted otherwise. Rule (12.3g) implements self-reference by substituting the recursive expression itself for the variable x in its body. This is called *unwinding* the recursion.

Theorem 12.2 (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.

Proof. The proof of preservation is by induction on the derivation of the transition judgement. Consider Rule (12.3g). Suppose that $\mathtt{fix}[\tau](x.e) : \tau$. By inversion and substitution we have $[\mathtt{fix}[\tau](x.e)/x]e : \tau$, from which the result follows directly by transitivity of the hypothetical judgement. The proof of progress proceeds by induction on the derivation of the typing judgement. For example, for Rule (12.1g) the result follows immediately since we may make progress by unwinding the recursion. □

Definitional equivalence for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, written $\Gamma \vdash e_1 \equiv e_2 : \tau$, is defined to be the strongest congruence containing the following axioms:

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{z};e_0;x.e_1) \equiv e_0 : \tau} \tag{12.4a}$$

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \equiv [e/x]e_1 : \tau} \tag{12.4b}$$

$$\frac{}{\Gamma \vdash \mathtt{fix}[\tau](x.e) \equiv [\mathtt{fix}[\tau](x.e)/x]e : \tau} \tag{12.4c}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau} \tag{12.4d}$$

These rules are sufficient to calculate the value of any closed expression of type `nat`: if $e : \mathtt{nat}$, then $e \equiv \overline{n} : \mathtt{nat}$ iff $e \mapsto^* \overline{n}$.

12.3 Definability

General recursion is a very flexible programming technique that permits a wide variety of functions to be defined within $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. The drawback is that, in contrast to primitive recursion, the termination of a recursively defined function is not intrinsic to the program itself, but rather must be proved extrinsically by the programmer. The benefit is a much greater freedom in writing programs.

General recursive functions are definable from general recursion and non-recursive functions. Let us write $\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e$ for a recursive function within whose body, $e : \tau_2$, are bound two variables, $y : \tau_1$ standing for the argument and $x : \tau_1 \to \tau_2$ standing for the function itself. The dynamic semantics of this construct is given by the axiom

$$\frac{}{\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e(e_1) \mapsto [\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e, e_1/x, y]e}\,.$$

That is, to apply a recursive function, we substitute the recursive function itself for x and the argument for y in its body.

Recursive functions may be defined in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ using a combination of recursion and functions, writing

$$\mathtt{fix}\, x{:}\tau_1 \rightharpoonup \tau_2\, \mathtt{is}\, \lambda\,(y{:}\tau_1.\, e)$$

for $\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e$. It is a good exercise to check that the static and dynamic semantics of recursive functions are derivable from this definition.

The primitive recursion construct of $\mathcal{L}\{\mathtt{nat} \to\}$ is defined in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ using recursive functions by taking the expression

$$\mathtt{natrec}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\, \mathtt{with}\, y \Rightarrow e_1\}$$

to stand for the application, $e'(e)$, where e' is the general recursive function

$$\mathtt{fun}\, f(u{:}\mathtt{nat}){:}\tau\, \mathtt{is}\, \mathtt{ifz}\, u\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow [f(x)/y]e_1\}.$$

The static and dynamic semantics of primitive recursion are derivable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ using this expansion.

In general, functions definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ are partial in that they may be undefined for some arguments. A partial (mathematical) function, $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$, is *definable* in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ iff there is an expression $e_\phi : \mathtt{nat} \rightharpoonup \mathtt{nat}$ such that $\phi(m) = n$ iff $e_\phi(\overline{m}) \equiv \overline{n} : \mathtt{nat}$. So, for example, if ϕ is the totally undefined function, then e_ϕ is any function that loops without returning whenever it is called.

It is informative to classify those partial functions ϕ that are definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. These are the so-called *partial recursive functions*, which are defined to be the primitive recursive functions augmented by the *minimization* operation: given $\phi(m,n)$, define $\psi(n)$ to be the least $m \geq 0$ such that (1) for $m' < m$, $\phi(m',n)$ is defined and non-zero, and (2) $\phi(m,n) = 0$. If no such m exists, then $\psi(n)$ is undefined.

Theorem 12.3. *A partial function ϕ on the natural numbers is definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ iff it is partial recursive.*

Proof sketch. Minimization is readily definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, so it is at least as powerful as the set of partial recursive functions. Conversely, we may, with considerable tedium, define an evaluator for expressions of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Consequently, $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ does not exceed the power of the set of partial recursive functions. □

Church's Law states that the partial recursive functions coincide with the set of effectively computable functions on the natural numbers—those that can be carried out by a program written in any programming language currently available or that will ever be available.[1] Therefore $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

The universal function, ϕ_{univ}, for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is the partial function on the natural numbers defined by

$$\phi_{univ}(\ulcorner e \urcorner)(m) = n \text{ iff } e(\overline{m}) \equiv \overline{n} : \mathtt{nat}.$$

In contrast to $\mathcal{L}\{\mathtt{nat} \rightarrow\}$, the universal function ϕ_{univ} for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is partial (may be undefined for some inputs). It is, in essence, an interpreter that, given the code $\overline{\ulcorner e \urcorner}$ of a closed expression of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$, simulates the dynamic semantics to calculate the result, if any, of applying it to the $\overline{m}$, obtaining $\overline{n}$. Since this process may not terminate, the universal function is not defined for all inputs.

By Church's Law the universal function is definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. In contrast, we proved in Chapter 11 that the analogous function is *not* definable in $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ using the technique of diagonalization. It is instructive to examine why that argument does not apply in the present setting. As in Section 11.4 on page 102, we may derive the equivalence

$$e_D(\overline{\ulcorner e_D \urcorner}) \equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner}))$$

[1] See Chapter 19 for further discussion of Church's Law.

for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. The difference, however, is that this equation is not inconsistent! Rather than being contradictory, it is merely a proof that the expression $e_D(\ulcorner \overline{e_D} \urcorner)$ does not terminate when evaluated, for if it did, the result would be a number equal to its own successor, which is impossible.

12.4 Co-Natural Numbers

The dynamics of the successor operation on natural numbers may be taken to be either eager or lazy, according to whether the predecessor of a successor is required to be a value. The eager interpretation represents the standard natural numbers in the sense that if $e : \mathtt{nat}$ and e val, then e evaluates to a numeral. The lazy interpretation, however, admits non-standard "natural numbers," such as

$$\omega = \mathtt{fix}\, x{:}\mathtt{nat}\, \mathtt{is}\, \mathtt{s}(x).$$

The "number" ω evaluates to $\mathtt{s}(\omega)$. This "number" may be thought of as an infinite stack of successors, since whenever we peel off the outermost successor we obtain the same "number" back again. The "number" ω is therefore larger than any other natural number in the sense that one may reach zero by repeatedly taking the predecessor of a natural number, but any number of predecessors on ω leads back to ω itself.

As the scare quotes indicate, it is stretching the terminology to refer to ω as a natural number. Instead one should distinguish a new type, called `conat`, of *lazy natural numbers*, of which ω is an element. The prefix "co-" indicates that the co-natural numbers are "dual" to the natural numbers in the following sense. The natural numbers are inductively defined as the *least* type such that if $e \equiv \mathtt{z} : \mathtt{nat}$ or $e \equiv \mathtt{s}(e') : \mathtt{nat}$ for some $e' : \mathtt{nat}$, then $e : \mathtt{nat}$. Dually, the co-natural numbers may be regarded as the *largest* type such that if $e : \mathtt{conat}$, then either $e \equiv \mathtt{z} : \mathtt{conat}$, or $e \equiv \mathtt{s}(e') : \mathtt{nat}$ for some $e' : \mathtt{conat}$. The difference is that $\omega : \mathtt{conat}$, because ω is definitionally equivalent to its own successor, whereas it is not the case that $\omega : \mathtt{nat}$, according to these definitions.

The duality between the natural numbers and the co-natural numbers is developed further in Chapter 17, wherein we consider the concepts of inductive and co-inductive types. Eagerness and laziness in general is discussed further in Chapter 39.

12.5 Notes

The language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is inspired by Plotkin's PCF [76]. Plotkin's original motivation was to analyze the relationship between denotational and operational semantics, but many subsequent studies have used PCF as a jumping off point for discussing numerous issues in language design.