Part IV

Function Types

Chapter 13

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 the concept of doubling itself. The general concept may be expressed by abstracting away from the expression being doubled, leaving behind just the pattern of doubling some fixed, but unspecified, number, represented by a *variable*. Specific instances of doubling are recovered by substituting an expression for the variable. A *function* is an expression with a designated free variable.

We consider two methods for permitting a function to be used more than once in an expression (for example, to double several different numbers). One method is through the introduction of *function definitions*, which give names to functions. An instance of the function is obtained by *applying* the function name to another expression, its *argument*. Each function has a domain and a range type, which in $\mathcal{L}\{\texttt{num str}\}$ must be either `num` or `str`. A function whose domain and range are base type is said to be a *first-order* function. A language in which functions are first-order and confined to function definitions is said to have *second-class* functions, since they are not values in the same sense as numbers or strings.

A more general method for supporting functions is as *first-class* values of *function type* whose domain and range are arbitrary types, including function types. A language with function types is said to be *higher-order*, in contrast to first-order, since it allows functions to be passed as arguments to and returned as results from other functions. Higher-order languages are surprisingly powerful, and they are, correspondingly, remarkably subtle, and have led to notorious design errors in programming languages.

13.1 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:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	$\texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{]}(x_1\texttt{.}e_2\texttt{;}f\texttt{.}e)$	$\texttt{fun}\, f(x_1\texttt{:}\tau_1)\texttt{:}\tau_2\texttt{=}e_2\,\texttt{in}\,e$
		|	$\texttt{call[}f\texttt{]}(e)$	$f(e)$

The variable f ranges over a distinguished class of variables, called *function names*. The expression $\texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{]}(x_1\texttt{.}e_2\texttt{;}f\texttt{.}e)$ binds the function name f within e to the pattern $x_1\texttt{.}e_2$, 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 $\texttt{call[}f\texttt{]}(e)$ instantiates the abstractor bound to f with the argument e.

The static semantics of $\mathcal{L}\{\texttt{num str fun}\}$ consists of judgements of the form $\Gamma \vdash e : \tau$, where Γ consists of hypotheses of one of two forms:

1. $x : \tau$, declaring the type of a variable x to be τ;
2. $f(\tau_1) : \tau_2$, declaring that f is a function name with domain τ_1 and range τ_2.

The second form of assumption is sometimes called a *function header*, since it resembles the concrete syntax of the first part of a function definition. The static semantics is defined in terms of these hypotheses by the following rules:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{]}(x_1\texttt{.}e_2\texttt{;}f\texttt{.}e) : \tau} \tag{13.1a}$$

$$\frac{\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau_1}{\Gamma, f(\tau_1) : \tau_2 \vdash \texttt{call[}f\texttt{]}(e) : \tau_2} \tag{13.1b}$$

The structural property of substitution takes an unusual form that matches the form of the hypotheses governing function names. The operation of *function substitution*, written $[\![x\texttt{.}e \,/\, f]\!]e'$, is inductively defined similarly to ordinary substitution, but bearing in mind that the function name, f, may only occur within e' as part of a function call. The rule governing such occurrences is given as follows:

$$\overline{[\![x\texttt{.}e \,/\, f]\!]\texttt{call[}f\texttt{]}(e') = \texttt{let}(e'\texttt{;}x\texttt{.}e)} \tag{13.2}$$

That is, at call sites to f, we bind x to e' within e to instantiate the pattern substituted for f.

Lemma 13.1. *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 dynamic semantics of $\mathcal{L}\{\texttt{num str fun}\}$ is easily defined using function substitution:

$$\frac{}{\texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e) \mapsto [\![x_1.e_2 \,/\, f]\!]e} \tag{13.3}$$

Observe that the use of function substitution eliminates all applications of f within e, so that no rule is required for evaluating them. This rule imposes either a *call-by-name* or a *call-by-value* application discipline according to whether the `let` binding is given a by-name or a by-value interpretation.

The safety of $\mathcal{L}\{\texttt{num str fun}\}$ may be proved separately, but it may also be obtained as a corollary of the safety of the more general language of higher-order functions, which we discuss next.

13.2 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 $\texttt{lam}[\tau_1](x.e)$. Corresponingly, we must generalize application to have the form $\texttt{ap}(e_1;e_2)$, where e_1 is any expression, and not just a function name. These are, respectively, the introduction and elimination forms for the *function type*, $\texttt{arr}(\tau_1;\tau_2)$, whose elements are functions with domain τ_1 and range τ_2.

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{arr}(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$
Expr	e	::=	$\texttt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		|	$\texttt{ap}(e_1;e_2)$	$e_1(e_2)$

The static semantics of $\mathcal{L}\{\texttt{num str} \to\}$ is given by extending Rules (9.1) with the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{arr}(\tau_1;\tau_2)} \tag{13.4a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{13.4b}$$

Lemma 13.2 (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 13.3 (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 dynamic semantics of $\mathcal{L}\{\texttt{num str} \to\}$ extends that of $\mathcal{L}\{\texttt{num str}\}$ with the following additional rules:

$$\frac{}{\texttt{lam}[\tau](x.e)\ \texttt{val}} \tag{13.5a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{13.5b}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1} \tag{13.5c}$$

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

Proof. The proof is by induction on rules (13.5), which define the dynamic semantics of the language.

Consider rule (13.5c),

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

Suppose that $\mathtt{ap}(\mathtt{lam}[\tau_2](x.e_1);e_2) : \tau_1$. By Lemma 13.2 on the preceding page $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 13.3 on the facing page $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 13.5 (Canonical Forms)**.** *If e val and $e : \mathtt{arr}(\tau_1;\tau_2)$, then $e = \mathtt{lam}[\tau_1](x.e_2)$ 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 13.6 (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 (13.4). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (13.4b). 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 13.5 that $e_1 = \mathtt{lam}[\tau_2](x.e)$ for some x and e. But then $\mathtt{ap}(e_1;e_2) \mapsto [e_2/x]e$. □

13.3 Evaluation Semantics and Definitional Equivalence

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

$$\frac{}{\mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.e)} \tag{13.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v} \tag{13.6b}$$

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

Theorem 13.7. *$e \Downarrow v$ iff $e \mapsto^* v$ and v val.*

Proof. In the forward direction we proceed by rule induction on Rules (13.6). 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 (4.1). 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 (13.5). □

Definitional equivalence for the call-by-name semantics of $\mathcal{L}\{\mathtt{num}\ \mathtt{str} \to\}$ is defined by a straightforward extension to Rules (10.9).

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{13.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \to \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) \equiv \mathtt{ap}(e_1';e_2') : \tau} \tag{13.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{13.7c}$$

Definitional equivalence for call-by-value requires a small bit of additional machinery. The main idea is to restrict Rule (13.7a) 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 (13.7a) 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:

$$\frac{\Xi, x\ \mathsf{val}\, \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\, \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{13.8a}$$

$$\frac{\Xi \vdash e_1\ \mathsf{val}}{\Xi\, \Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau}\ . \tag{13.8b}$$

13.4 Dynamic Scope

The dynamic semantics of function application given by Rules (13.5) is defined for closed expressions (those without free variables). Variables are never encountered during evaluation, because a closed expression will have been substituted for it before it is needed during evaluation. This accurately reflects the meaning of a variable as an unknown whose value may be specified by substitution. This treatment of variables is called *static scope*, or *static binding*, because it respects the statically determined scoping rules defined in Chapter 6.

Another evaluation strategy for $\mathcal{L}\{\rightarrow\}$ is sometimes considered as an alternative to static binding, called *dynamic scope*, or *dynamic binding*. The semantics of a dynamically scoped version of $\mathcal{L}\{\rightarrow\}$ is given by the same rules as for static binding, but altered in two crucial respects. First, evaluation is defined for *open* terms (those with free variables), as well as for closed terms. It is, however, an *error* to evaluate a variable; as with static scope, we must arrange that its binding is determined before its value is needed. Second, the binding of a variable is specified by a special form of substitution that *incurs*, rather than *avoids*, capture of free variables.

To avoid confusion, we will use the term *replacement* to refer to the capture-incurring form of substitution, which we write as $[x \leftarrow e_1]e_2$. As an example of replacement, let e be the expression $\lambda(x:\sigma.\, y)$ (with a free variable y), and let e' be the expression $\lambda(y:\tau.\, f(y))$, where f is a variable. The result of the substitution $[e/f]e'$ is the expression

$$\lambda(y':\tau.\, \lambda(x:\sigma.\, y)(y')),$$

in which the bound variable, y, has been renamed to y' to avoid confusion with the free variable, y, in e. The variable y remains free in the result. In contrast, the result of the replacement $[f \leftarrow e]e'$ is the expression

$$\lambda(y:\tau.\, \lambda(x:\sigma.\, y)(y)),$$

which has no free variables because the free y in e is captured by the binding for y in e'.

The implications of these alterations to the semantics of $\mathcal{L}\{\rightarrow\}$ are far-reaching. An immediate question suggested by the foregoing example is whether typing is preserved by replacement (as distinct from substitution). The answer is no! In the example if $\sigma \neq \tau$, then the result of replacement is not well-typed, even though both e and e' are well-typed (assuming $y : \tau$ and $f : \tau \rightarrow \tau'$). For this reason, dynamic scope is usually only considered

feasible for languages with only *one* type, so that such considerations do not arise.[1] An alternative is to consider a much richer type system that accounts for the types of the free variables in an expression; this possibility is explored in Chapter 34.

Setting aside these concerns, there is a further problem with dynamic scope that merits careful consideration, since it is closely tied to its purported advantages. The idea of dynamic scope is to make it convenient to parameterize a function by the values of one or more variables, without having to pass them as additional arguments. So, for example, a function $\lambda(x:\sigma.\, e)$ with y free is to be regarded as a family of functions, one for each choice of the parameter y. Using replacement, rather than substitution, allows the specification of a value for y to be determined by the context in which the function is used, rather than the context in which the function is introduced. (This is what gives rise to the terminology "dynamic scope.") Thus, in the example above, the meaning of the expression e is not fixed until after the replacement of f by e in e', at which point y is tied to the argument of the function e'. Whatever that turns out to be will determine the particular instance of e that will be used.

The chief difficulty with dynamic scope is that *the names of bound variables matter*. For example, consider the expression e'' given by $\lambda(y':\tau.\, f(y'))$. The expression e' is α-equivalent to e''; all we have done is to rename the bound variable from y to y'. The principles of binding and scope described in Chapter 6 state that these two expressions should be interchangeable in all situations, and indeed they are under static scope. However, with dynamic scope they behave quite differently. In particular, the replacement $[x \leftarrow e]e''$ results in the expression

$$\lambda(y':\tau.\, \lambda(x:\sigma.\, y)\,(y')),$$

which differs from the replacement $[x \leftarrow e]e'$, even though e' and e'' are α-equivalent.

From a programmer's perspective, the author of the expression e must be aware of the parameter naming conventions used by the author of e' (or e''). This does violence to any form of modularity or separation of concerns; the two pieces of code must be written in conjunction with each other, and this intimate relationship must be maintained as the code evolves. Experience shows that this is an impossible demand. For this reason, together with the difficulties with typing, dynamic scoping of variables is often treated with skepticism. However, there are other means of supporting

[1] See Chapter 22 for a discussion of useful programming languages with but one type.

essentially the same functionality, but without doing violence to the fundamental principles of binding and scope explained in Chapter 6. This concept, called *fluid binding*, is the subject of Chapter 34.

13.5 Exercises

Chapter 14

Gödel's System T

The language $\mathcal{L}\{\mathtt{nat} \to\}$, better known as *Gödel's System* $\boldsymbol{T}$, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, 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.

14.1 Statics

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{nat}$	$\texttt{nat}$
		|	$\texttt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$
Expr	e	::=	x	x
		|	$\texttt{z}$	$\texttt{z}$
		|	$\texttt{s}(e)$	$\texttt{s}(e)$
		|	$\texttt{rec}(e;e_0;x.y.e_1)$	$\texttt{rec}\,e\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x)\ \texttt{with}\ y \Rightarrow e_1\}$
		|	$\texttt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		|	$\texttt{ap}(e_1;e_2)$	$e_1(e_2)$

We write $\overline{n}$ for the expression $\texttt{s}(\ldots\texttt{s}(\texttt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression

$$\texttt{rec}(e;e_0;x.y.e_1)$$

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 "$\texttt{with}$" 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 $\texttt{iter}(e;e_0;y.e_1)$, 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 16) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

The static semantics of $\mathcal{L}\{\texttt{nat} \to\}$ is given by the following typing rules:

$$\frac{}{\Gamma, x : \texttt{nat} \vdash x : \texttt{nat}} \tag{14.1a}$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat}} \tag{14.1b}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat}} \tag{14.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}(e; e_0; x.y.e_1) : \tau} \tag{14.1d}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad x \# \Gamma}{\Gamma \vdash \mathtt{lam}[\sigma](x.e) : \mathtt{arr}(\sigma; \tau)} \tag{14.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{14.1f}$$

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

Lemma 14.1. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

14.2 Dynamics

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ 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 $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ are determined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{14.2a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}} \tag{14.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{14.2c}$$

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ is given by the following rules:

$$\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')} \tag{14.3a}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1; e_2) \mapsto \mathtt{ap}(e_1'; e_2)} \tag{14.3b}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e); e_2) \mapsto [e_2/x]e} \tag{14.3c}$$

$$\frac{e \mapsto e'}{\mathtt{rec}(e; e_0; x.y.e_1) \mapsto \mathtt{rec}(e'; e_0; x.y.e_1)} \tag{14.3d}$$

$$\frac{}{\texttt{rec(z;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \mapsto e_0} \tag{14.3e}$$

$$\frac{\texttt{s(}e\texttt{)}\ \mathsf{val}}{\texttt{rec(s(}e\texttt{);}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \mapsto [e,\texttt{rec(}e\texttt{;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)}/x,y]e_1} \tag{14.3f}$$

Rules (14.3e) and (14.3f) specify the behavior of the recursor on `z` and `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 14.2 (Canonical Forms)**.** *If $e : \tau$ and e val, then*

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

Theorem 14.3 (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'*

14.3 Definability

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

$$e_f(\overline{n}) \equiv \overline{f(n)} : \texttt{nat}. \tag{14.4}$$

That is, the numeric function $f : \mathbb{N} \to \mathbb{N}$ is definable iff there is a expression e_f of type $\texttt{nat} \to \texttt{nat}$ 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 $\mathcal{L}\{\texttt{nat} \to\}$, written $\Gamma \vdash e \equiv e' : \tau$, is the strongest congruence containing these axioms:

$$\frac{}{\Gamma \vdash \texttt{ap(lam[}\tau\texttt{]}\,(x\texttt{.}e_2)\texttt{;}e_1\texttt{)} \equiv [e_1/x]e_2 : \tau} \tag{14.5a}$$

$$\frac{}{\Gamma \vdash \texttt{rec(z;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \equiv e_0 : \tau} \tag{14.5b}$$

$$\frac{}{\Gamma \vdash \texttt{rec(s(}e\texttt{);}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \equiv [e,\texttt{rec(}e\texttt{;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)}/x,y]e_1 : \tau} \tag{14.5c}$$

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

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

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

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

where $\mathtt{id} = \lambda(x{:}\mathtt{nat}.\,x)$ is the identity, and $f \circ g = \lambda(x{:}\mathtt{nat}.\,f(g(x)))$ is 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

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

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

$$e_a(\overline{0})(\overline{n}) \equiv \mathtt{s}(\overline{n}) \tag{14.6}$$

$$e_a(\overline{m+1})(\overline{0}) \equiv e_a(\overline{m})(\overline{1}) \tag{14.7}$$

$$e_a(\overline{m+1})(\overline{n+1}) \equiv e_a(\overline{m})(e_a(\mathtt{s}(\overline{m}))(\overline{n})). \tag{14.8}$$

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

14.4 Non-Definability

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

Theorem 14.4. *If $e : \tau$, then there exists v val such that $e \equiv v : \tau$.*

Proof. See Corollary 50.9 on page 432. □

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 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 dynamic semantics, together with Theorem 14.4 on the preceding 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

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

[1] The same technique lies at the heart of the proof of Gödel's celebrated incompleteness theorem. The non-definability 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.

[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.

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

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

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

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

14.5 Exercises

1. Explore variant dynamic semantics for $\mathcal{L}\{\texttt{nat} \to\}$, both separately and in combination, in which the successor does not evaluate its argument, and in which functions are called by value.

Chapter 15

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 may for expressive power—we may solve all systems of equations, but the solution may not be as well-behaved as we might like it to be. It is our task as programmer's to ensure that the functions defined by recursion are total—all of our loops terminate.

15.1 Statics

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\mathtt{nat}$	$\mathtt{nat}$
		$\mid$	$\mathtt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$
Expr	e	::=	x	x
		$\mid$	$\mathtt{z}$	$\mathtt{z}$
		$\mid$	$\mathtt{s}(e)$	$\mathtt{s}(e)$
		$\mid$	$\mathtt{ifz}(e;e_0;x.e_1)$	$\mathtt{ifz}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}$
		$\mid$	$\mathtt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		$\mid$	$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$
		$\mid$	$\mathtt{fix}[\tau](x.e)$	$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e$

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{15.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{15.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{15.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}(e;e_0;x.e_1) : \tau} \tag{15.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]}(x.e) : \mathtt{parr}(\tau_1;\tau_2)} \tag{15.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{15.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix[}\tau\mathtt{]}(x.e) : \tau} \tag{15.1g}$$

Rule (15.1g) reflects the self-referential nature of general recursion. To show that $\mathtt{fix[}\tau\mathtt{]}(x.e)$ 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 15.1. *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

15.2 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:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{15.2a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}} \tag{15.2b}$$

$$\frac{}{\mathtt{lam[}\tau\mathtt{]}(x.e)\ \mathsf{val}} \tag{15.2c}$$

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

$$\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')} \tag{15.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e;e_0;x.e_1) \mapsto \mathtt{ifz}(e';e_0;x.e_1)} \tag{15.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \mapsto e_0} \tag{15.3c}$$

$$\frac{\texttt{s}(e)\ \mathsf{val}}{\texttt{ifz}(\texttt{s}(e);e_0;x.e_1) \mapsto [e/x]e_1} \tag{15.3d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{15.3e}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{15.3f}$$

$$\frac{}{\texttt{fix}[\tau](x.e) \mapsto [\texttt{fix}[\tau](x.e)/x]e} \tag{15.3g}$$

Rule (15.3g) implements self-reference by substituting the recursive expression itself for the variable x in its body. This is called *unwinding* the recursion.

Theorem 15.2 (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 (15.3g). Suppose that $\texttt{fix}[\tau](x.e) : \tau$. By inversion of typing we have $\texttt{fix}[\tau](x.e) : \tau \vdash [\texttt{fix}[\tau](x.e)/x]e : \tau$, 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 (15.1g) the result follows immediately since we may make progress by unwinding the recursion. □

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

$$\frac{}{\Gamma \vdash \texttt{ifz}(\tau;\texttt{z};e_0.x)e_1 \equiv e_0 : \tau} \tag{15.4a}$$

$$\frac{}{\Gamma \vdash \texttt{ifz}(\tau;\texttt{s}(e);e_0.x)e_1 \equiv [e/x]e_1 : \tau} \tag{15.4b}$$

$$\frac{}{\Gamma \vdash \texttt{fix}[\tau](x.e) \equiv [\texttt{fix}[\tau](x.e)/x]e : \tau} \tag{15.4c}$$

$$\frac{}{\Gamma \vdash \texttt{ap}(\texttt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau} \tag{15.4d}$$

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

15.3 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

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

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 ϕ, define $\psi(m)$ to be the least $n \geq 0$ such that (1) for $m < n$, $\phi(m)$ is defined and non-zero, and (2) $\phi(n) = 0$. If no such n exists, then $\psi(m)$ is undefined.

Theorem 15.3. *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 class 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 class of partial recursive functions. □

Church's Law states that the partial recursive functions coincide with the class 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 class 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 14 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 14.4 on page 120, 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 21 for further discussion of Church's Law.

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

15.4 Co-Natural Numbers

The evaluation strategy for the successor operation specified by Rules (15.3) ensures that the type `nat` is interpreted standardly as the type of natural numbers. This means that if $e : \texttt{nat}$ and e val, then e is definitionally equivalent to a numeral. In contrast the lazy interpretation of successor, obtained by omitting Rule (15.3a), and requiring that $\texttt{s}(e)$ val for any e, ruins this correspondence. The expression

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

evaluates to $\texttt{s}(\omega)$, which is a *value* of type `nat`. The "number" ω may be thought of as an infinite stack of successors, which is therefore larger than any finite stack of successors starting with zero. In other words ω is larger than any (finite) natural number, and hence can be regarded as an *infinite* "natural number."

Of course it is stretching the terminology to refer to ω as a number, much less as a natural number. Rather, we should say that the lazy interpretation of the successor operation gives rise to a distinct type, called the *lazy natural numbers*, or the *co-natural numbers*. The latter terminology arises from considering the co-natural numbers as "dual" to the ordinary natural numbers in the following sense. The standard natural numbers are inductively defined as the *least* type such that if $e \equiv \texttt{z} : \texttt{nat}$ or $e \equiv \texttt{s}(e') : \texttt{nat}$ for some $e' : \texttt{nat}$, then $e : \texttt{nat}$. Dually, the co-natural numbers may be regarded as the *largest* type such that if $e : \texttt{conat}$, then either $e \equiv \texttt{z} : \texttt{conat}$, or $e \equiv \texttt{s}(e') : \texttt{nat}$ for some $e' : \texttt{conat}$. The difference is that $\omega : \texttt{conat}$, because ω is definitionally equivalent to its own successor, whereas it is not the case that $\omega : \texttt{nat}$, according to these definitions.

The duality between the natural numbers and the co-natural numbers is developed further in Chapter 19, wherein we consider the concepts of inductive and co-inductive types. Eagerness and laziness in general is discussed further in Chapter 42.

15.5 Exercises

Part V

Finite Data Types

Chapter 16

Product Types

The *binary product* of two types consists of *ordered pairs* of values, one from each type in the order specified. The associated eliminatory forms are *projections*, which select the first and second component of a pair. The *nullary product*, or *unit*, type consists solely of the unique "null tuple" of no values, and has no associated eliminatory form. The product type admits both a *lazy* and an *eager* dynamics. According to the lazy dynamics, a pair is a value without regard to whether its components are values; they are not evaluated until (if ever) they are accessed and used in another computation. According to the eager dynamics, a pair is a value only if its components are values; they are evaluated when the pair is created.

More generally, we may consider the *finite product*, $\prod_{i \in I} \tau_i$, indexed by a finite set of *indices*, I. The elements of the finite product type are *I-indexed tuples* whose ith component is an element of the type τ_i, for each $i \in I$. The components are accessed by *I-indexed projection* operations, generalizing the binary case. Special cases of the finite product include *n-tuples*, indexed by sets of the form $I = \{ 0, \ldots, n-1 \}$, and *labelled tuples*, or *records*, indexed by finite sets of symbols. Similarly to binary products, finite products admit both an eager and a lazy interpretation.

16.1 Nullary and Binary Products

The abstract syntax of products is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`unit`	`unit`
		|	`prod`$(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$
Expr	e	::=	`triv`	$\langle\rangle$
		|	`pair`$(e_1;e_2)$	$\langle e_1, e_2\rangle$
		|	`proj[l]`(e)	$\mathtt{pr}_{\mathtt{l}}(e)$
		|	`proj[r]`(e)	$\mathtt{pr}_{\mathtt{r}}(e)$

The type `prod`$(\tau_1;\tau_2)$ is sometimes called the *binary product* of the types τ_1 and τ_2, and the type `unit` is correspondingly called the *nullary product* (of no types). We sometimes speak loosely of *product types* in such as way as to cover both the binary and nullary cases. The introductory form for the product type is called *pairing*, and its eliminatory forms are called *projections*. For the unit type the introductory form is called the *unit element*, or *null tuple*. There is no eliminatory form, there being nothing to extract from a null tuple.

The static semantics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \mathtt{triv} : \mathtt{unit}} \tag{16.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(e_1;e_2) : \mathtt{prod}(\tau_1;\tau_2)} \tag{16.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[l]}(e) : \tau_1} \tag{16.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[r]}(e) : \tau_2} \tag{16.1d}$$

The dynamic semantics of product types is specified by the following rules:

$$\frac{}{\mathtt{triv}\ \mathsf{val}} \tag{16.2a}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{pair}(e_1;e_2)\ \mathsf{val}} \tag{16.2b}$$

$$\left\{\frac{e_1 \mapsto e_1'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1';e_2)}\right\} \tag{16.2c}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1;e_2')}\right\} \tag{16.2d}$$

$$\frac{e \mapsto e'}{\mathtt{proj[l]}(e) \mapsto \mathtt{proj[l]}(e')} \tag{16.2e}$$

$$\frac{e \mapsto e'}{\mathtt{proj[r]}(e) \mapsto \mathtt{proj[r]}(e')} \tag{16.2f}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{proj[l]}(\mathtt{pair}(e_1;e_2)) \mapsto e_1} \tag{16.2g}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{proj[r]}(\mathtt{pair}(e_1;e_2)) \mapsto e_2} \tag{16.2h}$$

The bracketed rules and premises are to be omitted for a lazy semantics, and included for an eager semantics of pairing.

The safety theorem applies to both the eager and the lazy dynamics, with the proof proceeding along similar lines in each case.

Theorem 16.1 (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. Preservation is proved by induction on transition defined by Rules (16.2). Progress is proved by induction on typing defined by Rules (16.1). □

16.2 Finite Products

The syntax of finite product types is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{prod[}I\mathtt{]}(i \mapsto \tau_i)$	$\prod_{i\in I} \tau_i$
Expr	e	$::=$	$\mathtt{tuple[}I\mathtt{]}(i \mapsto e_i)$	$\langle e_i \rangle_{i\in I}$
		$\mid$	$\mathtt{proj[}I\mathtt{][}i\mathtt{]}(e)$	$e \cdot i$

For I a finite index set of size $n \geq 0$, the syntactic form $\mathtt{prod[}I\mathtt{]}(i \mapsto \tau_i)$ specifies an n-argument operator of arity $(0, 0, \ldots, 0)$ whose ith argument is the type τ_i. When it is useful to emphasize the tree structure, such an abt is written in the form $\prod \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$. Similarly, the syntactic form $\mathtt{tuple[}I\mathtt{]}(i \mapsto e_i)$ specifies an abt constructed from an n-argument

operator whose i operand is e_i. This may alternatively be written in the form $\langle i_0 : e_0, \ldots, i_{n-1} : e_{n-1} \rangle$.

The static semantics of finite products is given by the following rules:

$$\frac{(\forall i \in I)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \mathtt{tuple}[I]\,(i \mapsto e_i) : \mathtt{prod}[I]\,(i \mapsto \tau_i)} \tag{16.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}[I]\,(i \mapsto e_i) \quad j \in I}{\Gamma \vdash \mathtt{proj}[I]\,[j]\,(e) : \tau_j} \tag{16.3b}$$

In Rule (16.3b) the index $j \in I$ is a *particular* element of the index set I, whereas in Rule (16.3a), the index i ranges over the index set I.

The dynamic semantics of finite products is given by the following rules:

$$\frac{\{(\forall i \in I)\ e_i\ \mathsf{val}\}}{\mathtt{tuple}[I]\,(i \mapsto e_i)\ \mathsf{val}} \tag{16.4a}$$

$$\left\{ \frac{e_j \mapsto e_j' \quad (\forall i \neq j)\ e_i' = e_i}{\mathtt{tuple}[I]\,(i \mapsto e_i) \mapsto \mathtt{tuple}[I]\,(i \mapsto e_i')} \right\} \tag{16.4b}$$

$$\frac{e \mapsto e'}{\mathtt{proj}[I]\,[j]\,(e) \mapsto \mathtt{proj}[I]\,[j]\,(e')} \tag{16.4c}$$

$$\frac{\mathtt{tuple}[I]\,(i \mapsto e_i)\ \mathsf{val}}{\mathtt{proj}[I]\,[j]\,(\mathtt{tuple}[I]\,(i \mapsto e_i)) \mapsto e_j} \tag{16.4d}$$

Rule (16.4b) specifies that the components of a tuple are to be evaluated in *some* sequential order, without specifying the order in which they components are considered. It is straightforward, if a bit technically complicated, to impose a linear ordering on index sets that determines the evaluation order of the components of a tuple.

Theorem 16.2 (Safety)**.** *If $e : \tau$, then either e val or there exists e' such that $e' : \tau$ and $e \mapsto e'$.*

Proof. The safety theorem may be decomposed into progress and preservation lemmas, which are proved as in Section 16.1 on page 134. □

We may define nullary and binary products as particular instances of finite products by choosing an appropriate index set. The type `unit` may be defined as the product $\prod_{\_ \in \emptyset} \emptyset$ of the empty family over the empty index set, taking the expression $\langle\rangle$ to be the empty tuple, $\langle \emptyset \rangle_{\_ \in \emptyset}$. Binary products

$\tau_1 \times \tau_2$ may be defined as the product $\prod_{i \in \{1,2\}} \tau_i$ of the two-element family of types consisting of τ_1 and τ_2. The pair $\langle e_1, e_2 \rangle$ may then be defined as the tuple $\langle e_i \rangle_{i \in \{1,2\}}$, and the projections $\mathtt{pr_l}(e)$ and $\mathtt{pr_r}(e)$ are correspondingly defined, respectively, to be $e \cdot 1$ and $e \cdot 2$.

Finite products may also be used to define *labelled tuples*, or *records*, whose components are accessed by symbolic names. If $L = \{ l_1, \ldots, l_n \}$ is a finite set of symbols, called *field names*, or *field labels*, then the product type $\prod \langle l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1} \rangle$ has as values tuples of the form $\langle l_0 : e_0, \ldots, l_{n-1} : e_{n-1} \rangle$ in which $e_i : \tau_i$ for each $0 \leq i < n$. If e is such a tuple, then $e \cdot l$ projects the component of e labeled by $l \in L$.

16.3 Mutual Recursion

An important application of product types is to support *mutual recursion*. In Chapter 15 we used general recursion to define recursive functions, those that may "call themselves" when called. Product types support a natural generalization in which we may simultaneously define two or more functions, each of which may call the others, or even itself.

Consider the following recursion equations defining two mathematical functions on the natural numbers:

$$\begin{aligned} E(0) &= 1 \\ O(0) &= 0 \\ E(n+1) &= O(n) \\ O(n+1) &= E(n) \end{aligned}$$

Intuitively, $E(n)$ is non-zero iff n is even, and $O(n)$ is non-zero iff n is odd. If we wish to define these functions in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, we immediately face the problem of how to define two functions simultaneously. There is a trick available in this special case that takes advantage of the fact that E and O have the same type: simply define $\mathtt{eo}$ of type $\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$ so that $\mathtt{eo}(\overline{0})$ represents E and $\mathtt{eo}(\overline{1})$ represents O. (We leave the details as an exercise for the reader.)

A more general solution is to recognize that the definition of two mutually recursive functions may be thought of as the recursive definition of a pair of functions. In the case of the even and odd functions we will define the labelled tuple, e_{EO}, of type, τ_{EO}, given by

$$\prod \langle \mathtt{even} : \mathtt{nat} \to \mathtt{nat}, \mathtt{odd} : \mathtt{nat} \to \mathtt{nat} \rangle.$$

From this we will obtain the required mutually recursive functions as the projections $e_{EO} \cdot \texttt{even}$ and $e_{EO} \cdot \texttt{odd}$.

To effect the mutual recursion the expression e_{EO} is defined to be

$$\texttt{fix}\,\texttt{this}\!:\!\tau_{EO}\,\texttt{is}\,\langle\texttt{even}\!:\!e_E, \texttt{odd}\!:\!e_O\rangle,$$

where e_E is the expression

$$\lambda(\texttt{x}\!:\!\texttt{nat}.\,\texttt{ifz}\,\texttt{x}\,\{\texttt{z} \Rightarrow \texttt{s(z)} \mid \texttt{s(y)} \Rightarrow \texttt{this} \cdot \texttt{odd(y)}\}),$$

and e_O is the expression

$$\lambda(\texttt{x}\!:\!\texttt{nat}.\,\texttt{ifz}\,\texttt{x}\,\{\texttt{z} \Rightarrow \texttt{z} \mid \texttt{s(y)} \Rightarrow \texttt{this} \cdot \texttt{even(y)}\}).$$

The functions e_E and e_O refer to each other by projecting the appropriate component from the variable `this` standing for the object itself. The choice of variable name with which to effect the self-reference is, of course, immaterial, but it is common to use `this` or `self` to emphasize its role.

In the context of so-called *object-oriented* languages, labelled tuples of mutually recursive functions defined in this manner are called *objects*, and their component functions are called *methods*. Component projection is called *message passing*, viewing the component name as a "message" sent to the object to invoke the method by that name in the object. Internally to the object the methods refer to one another by sending a "message" to `this`, the canonical name for the object itself.

16.4 Exercises

Chapter 17

Sum Types

Most data structures involve alternatives such as the distinction between a leaf and an interior node in a tree, or a choice in the outermost form of a piece of abstract syntax. Importantly, the choice determines the structure of the value. For example, nodes have children, but leaves do not, and so forth. These concepts are expressed by *sum types*, specifically the *binary sum*, which offers a choice of two things, and the *nullary sum*, which offers a choice of no things. *Finite sums* generalize nullary and binary sums to permit an arbitrary number of cases indexed by a finite index set. As with products, sums come in both eager and lazy variants, differing in how values of sum type are defined.

17.1 Binary and Nullary Sums

The abstract syntax of sums is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`void`	`void`
		$\mid$	`sum`$(\tau_1;\tau_2)$	$\tau_1 + \tau_2$
Expr	e	::=	`abort[`τ`](`e`)`	`abort`$_\tau\, e$
		$\mid$	`in[l][`τ`](`e`)`	`in[l](`e`)`
		$\mid$	`in[r][`τ`](`e`)`	`in[r](`e`)`
		$\mid$	`case(`$e; x_1.e_1; x_2.e_2$`)`	`case` $e\,\{$`in[l](`x_1`)` $\Rightarrow e_1 \mid$ `in[r](`x_2`)` $\Rightarrow e_2\}$

The type `void` is the *nullary sum* type, whose values are selected from a choice of zero alternatives — there are no values of this type, and so no introductory forms. The eliminatory form, `abort[`τ`](`e`)`, aborts the computation in the event that e evaluates to a value, which it cannot do. The type

$\tau = \texttt{sum}(\tau_1;\tau_2)$ is the *binary sum*. The elements of the sum type are *labelled* to indicate whether they are drawn from the left or the right summand, either $\texttt{in[l]}[\tau](e)$ or $\texttt{in[r]}[\tau](e)$. A value of the sum type is eliminated by case analysis on the label of the value.

The static semantics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}[\tau](e) : \tau} \tag{17.1a}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \texttt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{in[l]}[\tau](e) : \tau} \tag{17.1b}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \texttt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{in[r]}[\tau](e) : \tau} \tag{17.1c}$$

$$\frac{\Gamma \vdash e : \texttt{sum}(\tau_1;\tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e; x_1.e_1; x_2.e_2) : \tau} \tag{17.1d}$$

Both branches of the case analysis must have the same type. Since a type expresses a static "prediction" on the form of the value of an expression, and since a value of sum type could evaluate to either form at run-time, we must insist that both branches yield the same type.

The dynamic semantics of sums is given by the following rules:

$$\frac{e \mapsto e'}{\texttt{abort}[\tau](e) \mapsto \texttt{abort}[\tau](e')} \tag{17.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[l]}[\tau](e)\ \mathsf{val}} \tag{17.2b}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[r]}[\tau](e)\ \mathsf{val}} \tag{17.2c}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[l]}[\tau](e) \mapsto \texttt{in[l]}[\tau](e')}\right\} \tag{17.2d}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[r]}[\tau](e) \mapsto \texttt{in[r]}[\tau](e')}\right\} \tag{17.2e}$$

$$\frac{e \mapsto e'}{\texttt{case}(e; x_1.e_1; x_2.e_2) \mapsto \texttt{case}(e'; x_1.e_1; x_2.e_2)} \tag{17.2f}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case}(\texttt{in[l]}[\tau](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1} \tag{17.2g}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{case(in[r]\,[\tau]\,(e);x_1\,.\,e_1;x_2\,.\,e_2)} \mapsto [e/x_2]e_2} \tag{17.2h}$$

The bracketed premises and rules are to be included for an eager semantics, and excluded for a lazy semantics.

The coherence of the static and dynamic semantics is stated and proved as usual.

Theorem 17.1 (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'.

Proof. The proof proceeds along standard lines, by induction on Rules (17.2) for preservation, and by induction on Rules (17.1) for progress. □

17.2 Finite Sums

Just as we may generalize nullary and binary products to finite products, so may we also generalize nullary and binary sums to finite sums. The syntax for finite sums is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\mathtt{sum}[I]\,(i \mapsto \tau_i)$	$\sum_{i \in I} \tau_i$
Expr	e	::=	$\mathtt{in}[I]\,[j]\,(e)$	$\mathtt{in}[j]\,(e)$
		|	$\mathtt{case}[I]\,(e; i \mapsto x_i\,.\,e_i)$	$\mathtt{case}\,e\,\{\mathtt{in}[i]\,(x_i) \Rightarrow e_i\}_{i \in I}$

The abstract binding tree representation of the finite `case` expression involves an I-indexed family of abstractors $x_i\,.\,e_i$, but is otherwise similar to the binary form. We write $\sum \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ for $\sum_{i \in I} \tau_i$, where $I = \{ i_0, \ldots, i_{n-1} \}$.

The static semantics of finite sums is defined by the following rules:

$$\frac{\Gamma \vdash e : \tau_j \quad j \in I}{\Gamma \vdash \mathtt{in}[I]\,[j]\,(e) : \mathtt{sum}[I]\,(i \mapsto \tau_i)} \tag{17.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{sum}[I]\,(i \mapsto \tau_i) \quad (\forall i \in I)\ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \mathtt{case}[I]\,(e; i \mapsto x_i\,.\,e_i) : \tau} \tag{17.3b}$$

These rules generalize to the finite case the static semantics for nullary and binary sums given in Section 17.1 on page 139.

The dynamic semantics of finite sums is defined by the following rules:

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{in}[I]\,[j]\,(e)\ \mathsf{val}} \tag{17.4a}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{in}[I]\,[j]\,(e) \mapsto \mathtt{in}[I]\,[j]\,(e')}\right\} \tag{17.4b}$$

$$\frac{e \mapsto e'}{\mathtt{case}[I]\,(e; i \mapsto x_i\,.\,e_i) \mapsto \mathtt{case}[I]\,(e'; i \mapsto x_i\,.\,e_i)} \tag{17.4c}$$

$$\frac{\mathtt{in}[I]\,[j]\,(e)\ \mathsf{val}}{\mathtt{case}[I]\,(\mathtt{in}[I]\,[j]\,(e); i \mapsto x_i\,.\,e_i) \mapsto [e/x_j]e_j} \tag{17.4d}$$

These again generalize the dynamic semantics of binary sums given in Section 17.1 on page 139.

Theorem 17.2 (Safety). *If $e : \tau$, then either e val or there exists $e' : \tau$ such that $e \mapsto e'$.*

Proof. The proof is similar to that for the binary case, as described in Section 17.1 on page 139. □

As with products, nullary and binary sums are special cases of the finite form. The type `void` may be defined to be the sum type $\sum_{\_\in\emptyset}\emptyset$ of the empty family of types. The expression `abort(`e`)` may corresponding be defined as the empty case analysis, `case`$\,e\,\{\emptyset\}$. Similarly, the binary sum type $\tau_1 + \tau_2$ may be defined as the sum $\sum_{i\in I}\tau_i$, where $I = \{\,\mathtt{l},\mathtt{r}\,\}$ is the two-element index set. The binary sum injections `in[l](`e`)` and `in[r](`e`)` are defined to be their counterparts, `in[l](`e`)` and `in[r](`e`)`, respectively. Finally, the binary case analysis,

$$\mathtt{case}\,e\,\{\mathtt{in}[\mathtt{l}]\,(x_\mathtt{l}) \Rightarrow e_\mathtt{l} \mid \mathtt{in}[\mathtt{r}]\,(x_\mathtt{r}) \Rightarrow e_\mathtt{r}\},$$

is defined to be the case analysis, $\mathtt{case}\,e\,\{\mathtt{in}[i]\,(x_i) \Rightarrow \tau_i\}_{i\in I}$. It is easy to check that the static and dynamic semantics of sums given in Section 17.1 on page 139 is preserved by these definitions.

Two special cases of finite sums arise quite commonly. The *n-ary sum* corresponds to the finite sum over an index set of the form $\{\,0,\ldots,n-1\,\}$ for some $n \geq 0$. The *labelled sum* corresponds to the case of the index set being a finite set of symbols serving as symbolic indices for the injections.

17.3 Uses for Sum Types

Sum types have numerous uses, several of which we outline here. More interesting examples arise once we also have recursive types, which are introduced in Part VI.

17.3.1 Void and Unit

It is instructive to compare the types `unit` and `void`, which are often confused with one another. The type `unit` has exactly one element, `triv`, whereas the type `void` has no elements at all. Consequently, if e : `unit`, then if e evaluates to a value, it must be `unit` — in other words, e has *no interesting value* (but it could diverge). On the other hand, if e : `void`, then e *must not yield a value*; if it were to have a value, it would have to be a value of type `void`, of which there are none. This shows that what is called the `void` type in many languages is really the type `unit` because it indicates that an expression has no interesting value, not that it has no value at all!

17.3.2 Booleans

Perhaps the simplest example of a sum type is the familiar type of Booleans, whose syntax is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`bool`	`bool`
Expr	e	::=	`tt`	`tt`
		|	`ff`	`ff`
		|	`if(`e`;`e_1`;`e_2`)`	`if` e `then` e_1 `else` e_2

The values of type `bool` are `tt` and `ff`. The expression `if(`e`;`e_1`;`e_2`)` branches on the value of e : `bool`. We leave a precise formulation of the static and dynamic semantics of this type as an exercise for the reader.

The type `bool` is definable in terms of binary sums and nullary products:

$$\texttt{bool} = \texttt{sum(unit;unit)} \tag{17.5a}$$
$$\texttt{tt} = \texttt{in[l][bool](triv)} \tag{17.5b}$$
$$\texttt{ff} = \texttt{in[r][bool](triv)} \tag{17.5c}$$
$$\texttt{if(}e\texttt{;}e_1\texttt{;}e_2\texttt{)} = \texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \tag{17.5d}$$

In the last equation above the variables x_1 and x_2 are chosen arbitrarily such that $x_1 \# e_1$ and $x_2 \# e_2$. (We often write an underscore in place of a variable to stand for a variable that does not occur within its scope.) It is a simple matter to check that the evident static and dynamic semantics of the type `bool` is engendered by these definitions.

17.3.3 Enumerations

More generally, sum types may be used to define *finite enumeration* types, those whose values are one of an explicitly given finite set, and whose elimination form is a case analysis on the elements of that set. For example, the type `suit`, whose elements are ♣, ♢, ♡, and ♠, has as elimination form the case analysis

$$\texttt{case}\, e\, \{\clubsuit \Rightarrow e_0 \mid \diamondsuit \Rightarrow e_1 \mid \heartsuit \Rightarrow e_2 \mid \spadesuit \Rightarrow e_3\},$$

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\texttt{suit} = \sum_{\_ \in I} \texttt{unit}$, where $I = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and the type family is constant over this set. The case analysis form for a labelled sum is almost literally the desired case analysis for the given enumeration, the only difference being the binding for the uninteresting value associated with each summand, which we may ignore.

17.3.4 Options

Another use of sums is to define the *option* types, which have the following syntax:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`opt(`τ`)`	τ `opt`
Expr	e	::=	`null`	`null`
		|	`just(`e`)`	`just(`e`)`
		|	`ifnull[`τ`](`e`;`e_1`;`x`.`e_2`)`	`check` e{`null` $\Rightarrow e_1$ | `just(`x`)` $\Rightarrow e_2$}

The type `opt(`τ`)` represents the type of "optional" values of type τ. The introductory forms are `null`, corresponding to "no value", and `just(`e`)`, corresponding to a specified value of type τ. The elimination form discriminates between the two possibilities.

The option type is definable from sums and nullary products according to the following equations:

$$\begin{aligned}
\texttt{opt}(\tau) &= \texttt{sum}(\texttt{unit}; \tau) && \text{(17.6a)}\\
\texttt{null} &= \texttt{in}[\texttt{l}][\texttt{opt}(\tau)](\texttt{triv}) && \text{(17.6b)}\\
\texttt{just}(e) &= \texttt{in}[\texttt{r}][\texttt{opt}(\tau)](e) && \text{(17.6c)}\\
\texttt{ifnull}[\tau](e; e_1; x_2.e_2) &= \texttt{case}(e; _.e_1; x_2.e_2) && \text{(17.6d)}
\end{aligned}$$

We leave it to the reader to examine the static and dynamic semantics implied by these definitions.

The option type is the key to understanding a common misconception, the *null pointer fallacy*. This fallacy, which is particularly common in object-oriented languages, is based on two related errors. The first error is to deem the values of certain types to be mysterious entities called *pointers*, based on suppositions about how these values might be represented at run-time, rather than on the semantics of the type itself. The second error compounds the first. A particular value of a pointer type is distinguished as *the null pointer*, which, unlike the other elements of that type, does not designate a value of that type at all, but rather rejects all attempts to use it as such.

To help avoid such failures, such languages usually include a function, say $\texttt{null} : \tau \to \texttt{bool}$, that yields $\texttt{tt}$ if its argument is null, and $\texttt{ff}$ otherwise. This allows the programmer to take steps to avoid using null as a value of the type it purports to inhabit. Consequently, programs are riddled with conditionals of the form

$$\texttt{if}\,\texttt{null}(e)\,\texttt{then}\ldots \textit{error}\ldots\,\texttt{else}\ldots \textit{proceed}\ldots. \tag{17.7}$$

Despite this, "null pointer" exceptions at run-time are rampant, in part because it is quite easy to overlook the need for such a test, and in part because detection of a null pointer leaves little recourse other than abortion of the program.

The underlying problem may be traced to the failure to distinguish the type τ from the type $\texttt{opt}(\tau)$. Rather than think of the elements of type τ as pointers, and thereby have to worry about the null pointer, one instead distinguishes between a *genuine* value of type τ and an *optional* value of type τ. An optional value of type τ may or may not be present, but, if it is, the underlying value is truly a value of type τ (and cannot be null). The elimination form for the option type,

$$\texttt{ifnull}[\tau](e; e_{error}; x.e_{ok}) \tag{17.8}$$

propagates the information that e is present into the non-null branch by binding a genuine value of type τ to the variable x. The case analysis effects a change of type from "optional value of type τ" to "genuine value of type τ", so that within the non-null branch no further null checks, explicit or implicit, are required. Observe that such a change of type is not achieved by the simple Boolean-valued test exemplified by expression (17.7); the advantage of option types is precisely that it does so.

17.4 Exercises

1. Formulate general n-ary sums in terms of nullary and binary sums.
2. Explain why is makes little sense to consider self-referential sum types.

Chapter 18

Pattern Matching

Pattern matching is a natural and convenient generalization of the elimination forms for product and sum types. For example, rather than write

$$\texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ \texttt{pr}_\texttt{l}(x) + \texttt{pr}_\texttt{r}(x)$$

to add the components of a pair, e, of natural numbers, we may instead write

$$\texttt{match}\ e\ \{x, y \,.\, \langle x, y\rangle \Rightarrow x + y\},$$

using pattern matching to name the components of the pair and refer to them directly. The first argument to the `match` expression is called the *match value* and the second argument consist of a finite sequence of *rules*, separated by vertical bars. In this example there is only one rule, but as we shall see shortly there is, in general, more than one rule in a given `match` expression. Each rule consists of a *pattern*, possibly involving variables, and an *expression* that may involve those variables (as well as any others currently in scope). The value of the `match` is determined by considering each rule in the order given to determine the first rule whose pattern matches the match value. If such a rule is found, the value of the `match` is the value of the expression part of the matching rule, with the variables of the pattern replaced by the corresponding components of the match value.

Pattern matching becomes more interesting, and useful, when combined with sums. The patterns `in[l](`x`)` and `in[r](`x`)` match the corresponding values of sum type. These may be used in combination with other patterns to express complex decisions about the structure of a value. For example, the following `match` expresses the computation that, when given a pair of type `(unit + unit)` $\times$ `nat`, either doubles or squares its sec-

ond component depending on the form of its first component:

$$\texttt{match}\,e\,\{x.\langle\texttt{in[l]}(\langle\rangle), x\rangle \Rightarrow x + x \mid y.\langle\texttt{in[r]}(\langle\rangle), y\rangle \Rightarrow y * y\}. \quad (18.1)$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 16 and 17.

In this chapter we study a simple language, $\mathcal{L}\{pat\}$, of pattern matching over eager product and sum types.

18.1 A Pattern Language

The main challenge in formalizing $\mathcal{L}\{pat\}$ is to manage properly the binding and scope of variables. The key observation is that a rule, $p \Rightarrow e$, binds variables in *both* the pattern, p, and the expression, e, simultaneously. Each rule in a sequence of rules may bind a different number of variables, independently of the preceding or succeeding rules. This gives rise to a somewhat unusual abstract syntax for sequences of rules that permits each rule to have a different valence. For example, the abstract syntax for expression (18.1) is given by

$$\texttt{match}\,e\,\{r_1; r_2\},$$

where r_1 is the rule

$$x.\langle\texttt{in[l]}(\langle\rangle), x\rangle \Rightarrow x + x$$

and r_2 is the rule

$$y.\langle\texttt{in[r]}(\langle\rangle), y\rangle \Rightarrow y * y.$$

The salient point is that each rule binds its own variables, in both the pattern and the expression.

The abstract syntax of $\mathcal{L}\{pat\}$ is defined by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	$\texttt{match}(e; rs)$	$\texttt{match}\,e\,\{rs\}$
Rules	rs	::=	$\texttt{rules[}n\texttt{]}(r_1; \ldots; r_n)$	$r_1 \mid \ldots \mid r_n$
Rule	r	::=	$x_1, \ldots, x_k.\texttt{rule}(p; e)$	$x_1, \ldots, x_n.\, p \Rightarrow e$
Pattern	p	::=	$\texttt{wild}$	$\_$
		|	x	x
		|	$\texttt{triv}$	$\langle\rangle$
		|	$\texttt{pair}(p_1; p_2)$	$\langle p_1, p_2\rangle$
		|	$\texttt{in[l]}(p)$	$\texttt{in[l]}(p)$
		|	$\texttt{in[r]}(p)$	$\texttt{in[r]}(p)$

The operator `rules[`n`]` has arity $(k_1, \ldots, k_n)$, where $n \geq 0$ and, for each $1 \leq i \leq n$, the ith rule has valence $k_i \geq 0$. Correspondingly, the ith rule consists of an abstractor binding k_i variables in the pattern and expression. A pattern is either a variable, a *wild card* pattern, a *unit pattern* matching only the trivial element of the `unit` type, a *pair pattern*, or a *choice pattern*.

18.2 Statics

The static semantics of $\mathcal{L}\{pat\}$ makes use of a *linear* hypothetical judgement of the form

$$x_1 : \tau_1, \ldots, x_k : \tau_k \Vdash p : \tau.$$

The meaning of this judgement is almost the same as that of the ordinary judgement

$$x_1 : \tau_1, \ldots, x_k : \tau_k \vdash p : \tau,$$

except that the hypotheses are treated specially so as to ensure that each variable is used exactly once in the pattern. This is achieved by dropping the usual structural rules of weakening and contraction, and limiting the combination of assumptions $\Lambda_1\ \Lambda_2$ to disjoint sets of assumptions, which is written $\Lambda_1 \# \Lambda_2$.

The pattern typing judgement $\Lambda \Vdash p : \tau$ is inductively defined by the following rules:

$$\frac{}{x : \tau \Vdash x : \tau} \tag{18.2a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau} \tag{18.2b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}} \tag{18.2c}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \quad \Lambda_2 \Vdash p_2 : \tau_2 \quad \Lambda_1 \# \Lambda_2}{\Lambda_1\ \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2} \tag{18.2d}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1}{\Lambda_1 \Vdash \mathtt{in[l]}(p) : \tau_1 + \tau_2} \tag{18.2e}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2}{\Lambda_2 \Vdash \mathtt{in[r]}(p) : \tau_1 + \tau_2} \tag{18.2f}$$

Rule (18.2a) states that a variable is a pattern of type τ provided that $x : \tau$ is the *only* assumption of the judgement. Rule (18.2d) expresses the formation of a pair pattern from patterns for its components, and imposes the

requirement that the variables used in the two sub-patterns must be disjoint, ensuring thereby that no variable may be used more than once in a pattern.

The judgment $x_1, \ldots, x_k \,.\, p \Rightarrow e : \tau > \tau'$ states that the rule $x_1, \ldots, x_k \,.\, p \Rightarrow e$ matches a value of type τ against the pattern p, binding the variables $x_1, \ldots, x_k$, and yields a value of type τ'.

$$\frac{\Lambda \Vdash p : \tau \quad \Gamma\,\Lambda \vdash e : \tau' \quad \Lambda = x_1 : \tau_1, \ldots, x_k : \tau_k \quad \Gamma \# \Lambda}{\Gamma \vdash x_1, \ldots, x_k \,.\, p \Rightarrow e : \tau > \tau'} \tag{18.3a}$$

$$\frac{\Gamma \vdash r_1 : \tau > \tau' \quad \ldots \quad \Gamma \vdash r_n : \tau > \tau'}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'} \tag{18.3b}$$

Rule (18.3a) makes use of the pattern typing judgement to determine both the type of the pattern, p, and also the types of its variables, Λ.[1] These variables are available for use within e, along with any other variables that may be in scope, without restriction. In Rule (18.3b) if the parameter, n, is zero, then the rule states that the empty sequence has an arbitrary domain and range, since it matches no value and yields no result.

Finally, the typing rule for the `match` expression is given as follows:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau'}{\Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'} \tag{18.4}$$

The `match` expression has type τ' if the rules transform any value of type τ, the type of the match expression, to a value of type τ'.

18.3 Dynamics

The dynamics of pattern matching is defined using substitution to "guess" the bindings of the pattern variables. The dynamics is given by the judgements $e \mapsto e'$, representing a step of computation, and e err, representing the checked condition of pattern matching failure.

$$\frac{e \mapsto e'}{\texttt{match}\, e\, \{rs\} \mapsto \texttt{match}\, e'\, \{rs\}} \tag{18.5a}$$

$$\frac{}{\texttt{match}\, e\, \{\}\ \mathsf{err}} \tag{18.5b}$$

[1] It may help to read the hypotheses, Λ, as an "output," rather than as an "input," of the judgement, in contrast to the usual reading of a hypothetical judgement.

$$\frac{e_1 \text{ val} \quad \ldots \quad e_k \text{ val} \quad [e_1,\ldots,e_k/x_1,\ldots,x_k]p_0 = e}{\texttt{match}\, e\, \{x_1,\ldots,x_k \,.\, p_0 \Rightarrow e_0; rs\} \mapsto [e_1,\ldots,e_k/x_1,\ldots,x_k]e_0} \tag{18.5c}$$

$$\frac{\neg\exists e_1,\ldots,e_k.[e_1,\ldots,e_k/x_1,\ldots,x_k]p_0 = e \qquad e \text{ val} \qquad \texttt{match}\, e\, \{rs\} \mapsto e'}{\texttt{match}\, e\, \{x_1,\ldots,x_k \,.\, p_0 \Rightarrow e_0; rs\} \mapsto e'} \tag{18.5d}$$

Rule (18.5b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (18.5c) specifies that the rules are to be considered in order. If the match value, e, matches the pattern, p_0, of the initial rule in the sequence, then the result is the corresponding instance of e_0; otherwise, matching continues by considering the remaining rules.

Theorem 18.1 (Preservation)**.** *If $e \mapsto e'$ and $e : \tau$, then $e' : \tau$.*

Proof. By a straightforward induction on the derivation of $e \mapsto e'$, making use of the evident substitution lemma for the statics. □

The formulation of pattern matching given in Rules (18.5) does not define *how* pattern matching is to be accomplished, rather it simply checks *whether* there is substitution for the variables in the pattern that results in the candidate value. This streamlines the presentation of the dynamics and the proof of preservation, but could be considered "too slick" in that it does not show how to find such a substitution or to determine that none exists. This gap may be filled by introducing two judgements. The first,

$$e_1 \lhd x_1, \ldots, e_k \lhd x_k \Vdash p \lhd e,$$

where e val and e_i val for each $1 \leq i \leq k$, is a linear hypothetical judgement stating that $[e_1,\ldots,e_k/x_1,\ldots,x_k]p = e$. The second, $e \perp p$, where e val, states that e fails to match the pattern p.

The pattern matching judgement is defined by the following rules, writing Θ for the assumptions governing variables:

$$\frac{}{x \lhd e \Vdash x \lhd e} \tag{18.6a}$$

$$\frac{}{\varnothing \Vdash \_ \lhd e} \tag{18.6b}$$

$$\frac{}{\varnothing \Vdash \langle\rangle \lhd \langle\rangle} \tag{18.6c}$$

$$\frac{\Theta_1 \Vdash p_1 \lhd e_1 \quad \Theta_2 \Vdash p_2 \lhd e_2 \quad \Theta_1 \# \Theta_2}{\Theta_1\,\Theta_2 \Vdash \langle p_1, p_2\rangle \lhd \langle e_1, e_2\rangle} \tag{18.6d}$$

$$\frac{\Theta \Vdash p \lhd e}{\Theta \Vdash \mathtt{in[l]}(p) \lhd \mathtt{in[l]}(e)} \tag{18.6e}$$

$$\frac{\Theta \Vdash p \lhd e}{\Theta \Vdash \mathtt{in[r]}(p) \lhd \mathtt{in[r]}(e)} \tag{18.6f}$$

The rules for a pattern mismatch are as follows:

$$\frac{e_1 \perp p_1}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{18.7a}$$

$$\frac{e_2 \perp p_2}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{18.7b}$$

$$\frac{}{\mathtt{in[l]}(e) \perp \mathtt{in[r]}(p)} \tag{18.7c}$$

$$\frac{e \perp p}{\mathtt{in[l]}(e) \perp \mathtt{in[l]}(p)} \tag{18.7d}$$

$$\frac{}{\mathtt{in[r]}(e) \perp \mathtt{in[l]}(p)} \tag{18.7e}$$

$$\frac{e \perp p}{\mathtt{in[r]}(e) \perp \mathtt{in[r]}(p)} \tag{18.7f}$$

Neither a variable nor a wildcard nor a null-tuple can mismatch any value of appropriate type. A pair can only mismatch a pair pattern due to a mismatch in one of its components. An injection into a sum type can mismatch the opposite injection, or it can mismatch the same injection by having its argument mismatch the argument pattern.

The salient property of these judgements is that they are complementary.

Theorem 18.2. *Suppose that $e : \tau$, $x_1 : \tau_1, \ldots, x_k : \tau_k \Vdash p : \tau$, and e val. Then either there exists $e_1, \ldots, e_k$ such that $x_1 \lhd e_1, \ldots, x_k \lhd e_k \Vdash p \lhd e$, or $e \perp p$.*

Proof. By rule induction on Rules (18.2), making use of the canonical forms lemma to characterize the shape of e based on its type. □

18.4 Exhaustiveness and Redundancy

While it is possible to state and prove a progress theorem for $\mathcal{L}\{pat\}$ as defined in Section 18.1 on page 148, it would not have much force, because the statics does not rule out pattern matching failure. What is missing is enforcement of the *exhaustiveness* of a sequence of rules, which ensures that every value of the domain type of a sequence of rules must match some rule in the sequence. In addition it would be useful to rule out *redundancy* of rules, which arises when a rule can only match values that are also matched by a preceding rule. Since pattern matching considers rules in the order in which they are written, such a rule can never be executed, and hence can be safely eliminated.

The statics of rules given in Section 18.1 on page 148 does not ensure exhaustiveness or irredundancy of rules. To do so we introduce a language of *match conditions* that identify a subset of the closed values of a type. With each rule we associate a match condition that classifies the values that are matched by that rule. A sequence of rules is *exhaustive* if every value of the domain type of the rule satisfies the match condition of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match condition also satisfies the match condition of some preceding rule.

The language of match conditions is defined by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Cond	ξ	$::=$	$\mathtt{any}[\tau]$	$\top_\tau$
		$\mid$	$\mathtt{in}[\mathtt{l}][\mathtt{sum}(\tau_1;\tau_2)](\xi_1)$	$\mathtt{in}[\mathtt{l}](\xi_1)$
		$\mid$	$\mathtt{in}[\mathtt{r}][\mathtt{sum}(\tau_1;\tau_2)](\xi_2)$	$\mathtt{in}[\mathtt{r}](\xi_2)$
		$\mid$	$\mathtt{triv}$	$\langle\rangle$
		$\mid$	$\mathtt{pair}(\xi_1;\xi_2)$	$\langle\xi_1,\xi_2\rangle$
		$\mid$	$\mathtt{nil}[\tau]$	$\bot_\tau$
		$\mid$	$\mathtt{alt}(\xi_1;\xi_2)$	$\xi_1 \vee \xi_2$

The judgement $\xi : \tau$ is defined by the following rules:

$$\frac{}{\top_\tau : \tau} \tag{18.8a}$$

$$\frac{\xi_1 : \tau_1}{\mathtt{in}[\mathtt{l}](\xi_1) : \tau_1 + \tau_2} \tag{18.8b}$$

$$\frac{\xi_1 : \tau_2}{\mathtt{in}[\mathtt{r}](\xi_1) : \tau_1 + \tau_2} \tag{18.8c}$$

$$\frac{}{\langle\rangle : \mathtt{unit}} \tag{18.8d}$$

$$\frac{\xi_1 : \tau_1 \quad \xi_2 : \tau_2}{\langle \xi_1, \xi_2 \rangle : \tau_1 \times \tau_2} \tag{18.8e}$$

$$\frac{}{\bot_\tau : \tau} \tag{18.8f}$$

$$\frac{\xi_1 : \tau \quad \xi_2 : \tau}{\xi_1 \vee \xi_2 : \tau} \tag{18.8g}$$

Informally, $\xi : \tau$ means that ξ constrains values of type τ.

For $\xi : \tau$, $e : \tau$, and e val, we define the *satisfaction* judgement $e \models \xi$ as follows:

$$\frac{}{e \models \top_\tau} \tag{18.9a}$$

$$\frac{e_1 \models \xi_1}{\mathtt{in[l]}(e_1) \models \mathtt{in[l]}(\xi_1)} \tag{18.9b}$$

$$\frac{e_2 \models \xi_2}{\mathtt{in[r]}(e_2) \models \mathtt{in[r]}(\xi_2)} \tag{18.9c}$$

$$\frac{}{\langle\rangle \models \langle\rangle} \tag{18.9d}$$

$$\frac{e_1 \models \xi_1 \quad e_2 \models \xi_2}{\langle e_1, e_2 \rangle \models \langle \xi_1, \xi_2 \rangle} \tag{18.9e}$$

$$\frac{e \models \xi_1}{e \models \xi_1 \vee \xi_2} \tag{18.9f}$$

$$\frac{e \models \xi_2}{e \models \xi_1 \vee \xi_2} \tag{18.9g}$$

The *entailment* judgement $\xi_1 \models \xi_2$, where $\xi_1 : \tau$ and $\xi_2 : \tau$, is defined to hold iff $e \models \xi_1$ implies $e \models \xi_2$.

Finally, we instrument the statics of patterns and rules to associate a match condition that specifies the values that may be matched by that pattern or rule. This allows us to ensure that rules are both exhaustive and irredundant.

The judgement $\Lambda \Vdash p : \tau\ [\xi]$ augments the judgement $\Lambda \Vdash p : \tau$ with a match constraint characterizing the set of values of type τ matched by the pattern p. It is inductively defined by the following rules:

$$\frac{}{x : \tau \Vdash x : \tau\ [\top_\tau]} \tag{18.10a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau\ [\top_\tau]} \tag{18.10b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}\ [\langle\rangle]} \tag{18.10c}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1\ [\xi_1]}{\Lambda_1 \Vdash \mathtt{in[l]}\,(p) : \tau_1 + \tau_2\ [\mathtt{in[l]}\,(\xi_1)]} \tag{18.10d}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2\ [\xi_2]}{\Lambda_2 \Vdash \mathtt{in[r]}\,(p) : \tau_1 + \tau_2\ [\mathtt{in[r]}\,(\xi_2)]} \tag{18.10e}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1\ [\xi_1] \quad \Lambda_2 \Vdash p_2 : \tau_2\ [\xi_2] \quad \Lambda_1 \# \Lambda_2}{\Lambda_1\,\Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2\ [\langle \xi_1, \xi_2\rangle]} \tag{18.10f}$$

Rules (18.10a) to (18.10b) specify that all values of the pattern type are matched. Rule (18.10c) specifies that the only value of type `unit` is matched by the pattern. Rules (18.10d) to (18.10e) specify that the pattern matches only those values with the specified injection tag and whose argument is matched by the specified pattern. Rule (18.10f) specifies that the pattern matches only pairs whose components match the specified patterns.

The judgement $\Gamma \vdash r : \tau > \tau'\ [\xi]$ augments the formation judgement for a rule with a match constraint characterizing the pattern component of the rule. The judgement $\Gamma \vdash rs : \tau > \tau'\ [\xi]$ augments the formation judgement for a sequence of rules with a match constraint characterizing the values matched by some rule in the given rule sequence.

$$\frac{\Lambda \Vdash p : \tau\ [\xi] \quad \Gamma\,\Lambda \vdash e : \tau'}{\Gamma \vdash x_1, \ldots, x_k \,.\, p \Rightarrow e : \tau > \tau'\ [\xi]} \tag{18.11a}$$

$$\frac{\Gamma \vdash r_1 : \tau > \tau'\ [\xi_1] \qquad \ldots \qquad \Gamma \vdash r_n : \tau > \tau'\ [\xi_n] \qquad (\forall 1 \leq i \leq n)\ \xi_i \not\models \xi_1 \vee \ldots \vee \xi_{i-1}}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'\ [\xi_1 \vee \ldots \vee \xi_n]} \tag{18.11b}$$

Rule (18.11b) ensures that each successive rule is irredundant relative to the preceding rules in that it demands that it *not* be the case that *every* value

satisfying ξ_i satisfies some preceding ξ_j. That is, it requires that there be *some* value satisfying ξ_i that does *not* satisfy some preceding ξ_j.

Finally, the typing rule for `match` expressions requires exhaustiveness:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau' \; [\xi] \quad \top_\tau \models \xi}{\Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'} \tag{18.12}$$

The third premise ensures that *every* value of type τ satisfies the constraint ξ representing the values matched by *some* rule in the given rule sequence.

The additional constraints on the statics are sufficient to ensure progress, because no well-formed `match` expression can fail to match a value of the specified type. If a given sequence of rules is inexhaustive, this can always be rectified by including a "default" rule of the form $x \,.\, x \Rightarrow e_x$, where e_x handles the unmatched value x gracefully, perhaps by raising an exception (see Chapter 28 for a discussion of exceptions).

Theorem 18.3. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

18.5 Exercises