Chapter 13

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.

13.1 Nullary and Binary Products

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

Type	τ	::=	`unit`	`unit`	nullary product
			`prod(`τ_1`;`τ_2`)`	$\tau_1 \times \tau_2$	binary product
Expr	e	::=	`triv`	$\langle\rangle$	null tuple
			`pair(`e_1`;`e_2`)`	$\langle e_1, e_2\rangle$	ordered pair
			`proj[l](`e`)`	$e \cdot \mathtt{l}$	left projection
			`proj[r](`e`)`	$e \cdot \mathtt{r}$	right projection

There is no elimination form for the unit type, there being nothing to extract from the null tuple.

The statics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \mathtt{triv} : \mathtt{unit}} \tag{13.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(e_1;e_2) : \mathtt{prod}(\tau_1;\tau_2)} \tag{13.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[l]}(e) : \tau_1} \tag{13.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[r]}(e) : \tau_2} \tag{13.1d}$$

The dynamics of product types is specified by the following rules:

$$\frac{}{\mathtt{triv}\ \mathsf{val}} \tag{13.2a}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{pair}(e_1;e_2)\ \mathsf{val}} \tag{13.2b}$$

$$\left\{\frac{e_1 \mapsto e_1'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1';e_2)}\right\} \tag{13.2c}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1;e_2')}\right\} \tag{13.2d}$$

$$\frac{e \mapsto e'}{\mathtt{proj[l]}(e) \mapsto \mathtt{proj[l]}(e')} \tag{13.2e}$$

$$\frac{e \mapsto e'}{\mathtt{proj[r]}(e) \mapsto \mathtt{proj[r]}(e')} \tag{13.2f}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{proj[l](pair(}e_1\texttt{;}e_2\texttt{))} \mapsto e_1} \tag{13.2g}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{proj[r](pair(}e_1\texttt{;}e_2\texttt{))} \mapsto e_2} \tag{13.2h}$$

The bracketed rules and premises are to be omitted for a lazy dynamics, and included for an eager dynamics 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 13.1 (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 (13.2). Progress is proved by induction on typing defined by Rules (13.1). □

13.2 Finite Products

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

Type	τ	::=	$\texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)}$	$\prod_{i\in I} \tau_i$	product
Expr	e	::=	$\texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)}$	$\langle e_i\rangle_{i\in I}$	tuple
			$\texttt{proj[}I\texttt{][}i\texttt{](}e\texttt{)}$	$e \cdot i$	projection

For I a finite index set of size $n \geq 0$, the syntactic form $\texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)}$ 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 $\texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)}$ 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 statics of finite products is given by the following rules:

$$\frac{(\forall i \in I)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)} : \texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)}} \tag{13.3a}$$

$$\frac{\Gamma \vdash e : \texttt{prod[}I\texttt{](}i \mapsto e_i\texttt{)} \quad j \in I}{\Gamma \vdash \texttt{proj[}I\texttt{][}j\texttt{](}e\texttt{)} : \tau_j} \tag{13.3b}$$

In Rule (13.3b) the index $j \in I$ is a *particular* element of the index set I, whereas in Rule (13.3a), the index i ranges over the index set I.

The dynamics of finite products is given by the following rules:

$$\frac{\{(\forall i \in I)\ e_i\ \mathsf{val}\}}{\texttt{tuple[}I\texttt{](}i \hookrightarrow e_i\texttt{)}\ \mathsf{val}} \tag{13.4a}$$

$$\left\{\frac{e_j \mapsto e'_j \quad (\forall i \neq j)\ e'_i = e_i}{\texttt{tuple[}I\texttt{](}i \hookrightarrow e_i\texttt{)} \mapsto \texttt{tuple[}I\texttt{](}i \hookrightarrow e'_i\texttt{)}}\right\} \tag{13.4b}$$

$$\frac{e \mapsto e'}{\texttt{proj[}I\texttt{][}j\texttt{](}e\texttt{)} \mapsto \texttt{proj[}I\texttt{][}j\texttt{](}e'\texttt{)}} \tag{13.4c}$$

$$\frac{\texttt{tuple[}I\texttt{](}i \hookrightarrow e_i\texttt{)}\ \mathsf{val}}{\texttt{proj[}I\texttt{][}j\texttt{](tuple[}I\texttt{](}i \hookrightarrow e_i\texttt{))} \mapsto e_j} \tag{13.4d}$$

Rule (13.4b) 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 13.2 (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 13.1 on page 118. □

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 $e \cdot \mathtt{l}$ and $e \cdot \mathtt{r}$ 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 $l \in L$.

13.3 Primitive and Mutual Recursion

In the presence of products we may simplify the primitive recursion construct defined in Chapter 11 so that only the result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\texttt{natiter}\, e\, \{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$, we may define primitive recursion in the sense of Chapter 11 to be the expression $e' \cdot \texttt{r}$, where e' is the expression

$$\texttt{natiter}\, e\, \{\texttt{z} \Rightarrow \langle \texttt{z}, e_0 \rangle \mid \texttt{s}(x) \Rightarrow \langle \texttt{s}(x \cdot \texttt{l}), [x \cdot \texttt{l}, x \cdot \texttt{r} / x_0, x_1] e_1 \rangle\}.$$

The idea is to compute inductively both the number, n, and the result of the recursive call on n, from which we can compute both $n + 1$ and the result of an additional recursion using e_1. The base case is computed directly as the pair of zero and e_0. It is easy to check that the statics and dynamics of the recursor are preserved by this definition.

We may also use product types to implement *mutual recursion*, which allows several mutually recursive computations to be defined simultaneously. For example, 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 $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, 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 $\texttt{eo}$ of type $\texttt{nat} \to \texttt{nat} \to \texttt{nat}$ so that $\texttt{eo}(\overline{0})$ represents E and $\texttt{eo}(\overline{1})$ 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

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

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

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

where e_E is the expression

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

and e_O is the expression

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

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.

13.4 Notes

Product types are the abstract essence of structured data [68]. Structures are tuples whose components are accessible using projections. Most languages have some form of product type, but frequently in a form that is mixed up with representation commitments and restrictions on how they may be used. Rather than introduce *ad hoc* mechanisms for passing multiple arguments or returning multiple results, one may instead systematize the concepts of tupling and pattern matching, and decouple them from function call and return. Similarly, "objects" are just tuples of mutually recursive functions; it seems preferable to break out the building blocks, namely functions, products, and recursion, rather than to amalgamate them into a monolith.

Chapter 14

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.

14.1 Binary and Nullary Sums

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

Type	τ	$::=$	`void`	`void`	nullary sum
			$\mathtt{sum}(\tau_1;\tau_2)$	$\tau_1 + \tau_2$	binary sum
Expr	e	$::=$	$\mathtt{abort}[\tau](e)$	$\mathtt{abort}_\tau\, e$	abort
			$\mathtt{in}[\mathtt{l}][\tau](e)$	$\mathtt{l}\cdot e$	left injection
			$\mathtt{in}[\mathtt{r}][\tau](e)$	$\mathtt{r}\cdot e$	right injection
			$\mathtt{case}(e;x_1.e_1;x_2.e_2)$	$\mathtt{case}\, e\, \{\mathtt{l}\cdot x_1 \Rightarrow e_1 \mid \mathtt{r}\cdot x_2 \Rightarrow e_2\}$	case analysis

The nullary sum represents a choice of zero alternatives, and hence admits no introductory form. The eliminatory form, $\mathtt{abort}[\tau](e)$, aborts the computation in the event that e evaluates to a value, which it cannot do. The elements of the binary sum type are labelled to indicate whether they are drawn from the left or the right summand, either $\mathtt{in[l][}\tau\mathtt{](}e\mathtt{)}$ or $\mathtt{in[r][}\tau\mathtt{](}e\mathtt{)}$. A value of the sum type is eliminated by case analysis.

The statics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{abort[}\tau\mathtt{](}e\mathtt{)} : \tau} \tag{14.1a}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \mathtt{sum(}\tau_1\mathtt{;}\tau_2\mathtt{)}}{\Gamma \vdash \mathtt{in[l][}\tau\mathtt{](}e\mathtt{)} : \tau} \tag{14.1b}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \mathtt{sum(}\tau_1\mathtt{;}\tau_2\mathtt{)}}{\Gamma \vdash \mathtt{in[r][}\tau\mathtt{](}e\mathtt{)} : \tau} \tag{14.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{sum(}\tau_1\mathtt{;}\tau_2\mathtt{)} \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case(}e\mathtt{;}x_1\mathtt{.}e_1\mathtt{;}x_2\mathtt{.}e_2\mathtt{)} : \tau} \tag{14.1d}$$

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 dynamics of sums is given by the following rules:

$$\frac{e \mapsto e'}{\mathtt{abort[}\tau\mathtt{](}e\mathtt{)} \mapsto \mathtt{abort[}\tau\mathtt{](}e'\mathtt{)}} \tag{14.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{in[l][}\tau\mathtt{](}e\mathtt{)}\ \mathsf{val}} \tag{14.2b}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{in[r][}\tau\mathtt{](}e\mathtt{)}\ \mathsf{val}} \tag{14.2c}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{in[l][}\tau\mathtt{](}e\mathtt{)} \mapsto \mathtt{in[l][}\tau\mathtt{](}e'\mathtt{)}}\right\} \tag{14.2d}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{in[r][}\tau\mathtt{](}e\mathtt{)} \mapsto \mathtt{in[r][}\tau\mathtt{](}e'\mathtt{)}}\right\} \tag{14.2e}$$

$$\frac{e \mapsto e'}{\mathtt{case(}e\mathtt{;}x_1\mathtt{.}e_1\mathtt{;}x_2\mathtt{.}e_2\mathtt{)} \mapsto \mathtt{case(}e'\mathtt{;}x_1\mathtt{.}e_1\mathtt{;}x_2\mathtt{.}e_2\mathtt{)}} \tag{14.2f}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{case(in[l][}\tau\mathtt{](}e\mathtt{);}x_1\mathtt{.}e_1\mathtt{;}x_2\mathtt{.}e_2\mathtt{)} \mapsto [e/x_1]e_1} \tag{14.2g}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{case(in[r][}\tau\mathtt{](}e\mathtt{);}x_1\mathtt{.}e_1\mathtt{;}x_2\mathtt{.}e_2\mathtt{)} \mapsto [e/x_2]e_2} \tag{14.2h}$$

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

The coherence of the statics and dynamics is stated and proved as usual.

Theorem 14.1 (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 by induction on Rules (14.2) for preservation, and by induction on Rules (14.1) for progress. □

14.2 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:

Type	τ	::=	$\mathtt{sum}(\langle i : \tau_i \rangle_{i \in I})$	$\sum_{i \in I} \tau_i$	sum
Expr	e	::=	$\mathtt{in}[\langle i : \tau_i \rangle_{i \in I}][i](e)$	$i \cdot e$	injection
			$\mathtt{case}[I](e; \langle i : x_i \, . \, e_i \rangle_{i \in I})$	$\mathtt{case}\, e \, \{ i \cdot x_i \Rightarrow e_i \}_{i \in I}$	case analysis

The general sum $\sum_{i \in I} \tau_i$ is sometimes written in the form $\sum \langle i : \tau_i \rangle_{i \in I}$. The finite family of types $\langle i : \tau_i \rangle_{i \in I}$ is often abbreviated to $\vec{\tau}$ when the finite index set, I, is clear from context.

The statics of finite sums is defined by the following rules:

$$\frac{\Gamma \vdash e : \tau_i \quad i \in I}{\Gamma \vdash \mathtt{in}[\langle i : \tau_i \rangle_{i \in I}][i](e) : \mathtt{sum}(\langle i : \tau_i \rangle_{i \in I})} \tag{14.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{sum}(\langle i : \tau_i \rangle_{i \in I}) \quad (\forall i \in I)\ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \mathtt{case}[I](e; \langle i : x_i \, . \, e_i \rangle_{i \in I}) : \tau} \tag{14.3b}$$

These rules generalize to the finite case the statics for nullary and binary sums given in Section 14.1 on page 123.

The dynamics of finite sums is defined by the following rules:

$$\frac{\{e \ \mathsf{val}\}}{\mathtt{in}[\vec{\tau}][i](e) \ \mathsf{val}} \tag{14.4a}$$

$$\left\{ \frac{e \mapsto e'}{\mathtt{in}[\vec{\tau}][i](e) \mapsto \mathtt{in}[\vec{\tau}][i](e')} \right\} \tag{14.4b}$$

$$\frac{e \mapsto e'}{\mathtt{case}[I](e; \langle i : x_i \, . \, e_i \rangle_{i \in I}) \mapsto \mathtt{case}[I](e'; \langle i : x_i \, . \, e_i \rangle_{i \in I})} \tag{14.4c}$$

$$\frac{\mathtt{in}[\vec{\tau}][i](e) \ \mathsf{val}}{\mathtt{case}[I](\mathtt{in}[\vec{\tau}][i](e); \langle i : x_i \, . \, e_i \rangle_{i \in I}) \mapsto [e/x_i]e_i} \tag{14.4d}$$

These again generalize the dynamics of binary sums given in Section 14.1 on page 123.

Theorem 14.2 (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 14.1 on page 123. □

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 $\mathtt{l}\cdot e$ and $\mathtt{r}\cdot e$ are defined to be their counterparts, $\mathtt{l}\cdot e$ and $\mathtt{r}\cdot e$, respectively. Finally, the binary case analysis,

$$\mathtt{case}\,e\,\{\mathtt{l}\cdot x_{\mathtt{l}} \Rightarrow e_{\mathtt{l}} \mid \mathtt{r}\cdot x_{\mathtt{r}} \Rightarrow e_{\mathtt{r}}\},$$

is defined to be the case analysis, `case` $e\,\{i\cdot x_i \Rightarrow \tau_i\}_{i\in I}$. It is easy to check that the static and dynamics of sums given in Section 14.1 on page 123 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 names for the injections.

14.3 Applications of 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.

14.3.1 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!

14.3.2 Booleans

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

Type	τ	::=	`bool`	`bool`	booleans
Expr	e	::=	`tt`	`tt`	truth
			`ff`	`ff`	falsity
			`if`$(e;e_1;e_2)$	`if` e `then` e_1 `else` e_2	conditional

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

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

$$\texttt{bool} = \texttt{sum(unit;unit)} \tag{14.5a}$$

$$\texttt{tt} = \texttt{in[l][bool](triv)} \tag{14.5b}$$

$$\texttt{ff} = \texttt{in[r][bool](triv)} \tag{14.5c}$$

$$\texttt{if}(e;e_1;e_2) = \texttt{case}(e;x_1.e_1;x_2.e_2) \tag{14.5d}$$

In the last equation above the variables x_1 and x_2 are chosen arbitrarily such that $x_1 \notin e_1$ and $x_2 \notin e_2$. (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 dynamics of the type `bool` is engendered by these definitions.

14.3.3 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.

14.3.4 Options

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

Type	τ	::=	`opt(`τ`)`	τ `opt`	option
Expr	e	::=	`null`	`null`	nothing
			`just(`e`)`	`just(`e`)`	something
			`ifnull[`τ`](`$e;e_1;x.e_2$`)`	`check` e {`null` $\Rightarrow e_1$ | `just(`x`)` $\Rightarrow e_2$}	null test

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:

$$\texttt{opt}(\tau) = \texttt{sum}(\texttt{unit};\tau) \tag{14.6a}$$

$$\texttt{null} = \texttt{in[l][opt}(\tau)\texttt{](triv)} \tag{14.6b}$$

$$\texttt{just}(e) = \texttt{in[r][opt}(\tau)\texttt{]}(e) \tag{14.6c}$$

$$\texttt{ifnull}[\tau](e;e_1;x_2.e_2) = \texttt{case}(e;\_.e_1;x_2.e_2) \tag{14.6d}$$

We leave it to the reader to examine the statics and dynamics 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 `null` : $\tau \rightarrow$ `bool`, that yields `tt` if its argument is null, and `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

$$\texttt{if}\,\texttt{null}(e)\,\texttt{then}\ldots \mathit{error}\ldots\texttt{else}\ldots\mathit{proceed}\ldots. \tag{14.7}$$

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,

$$\texttt{ifnull}[\tau](e; e_{error}; x\,.\,e_{ok}) \tag{14.8}$$

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 (14.7); the advantage of option types is precisely that it does so.

14.4 Notes

Heterogeneous data structures are ubiquitous. Sums codify heterogeneity, yet few languages properly support them. Much of object-oriented programming is concerned with heterogeneity. Although often confused with types, classes are, as here, tags; dispatch is case analysis, but without the benefit of ensuring that all cases are properly covered. And most such languages succumb to what Tony Hoare calls his "billion dollar mistake," the cursed "null pointer."

Chapter 15

Pattern Matching

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

$$\texttt{let}\, x \,\texttt{be}\, e \,\texttt{in}\, x \cdot \texttt{l} + x \cdot \texttt{r}$$

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

$$\texttt{match}\, e \,\{\langle x_1, x_2 \rangle \Rightarrow x_1 + x_2\},$$

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 $\texttt{l} \cdot x$ and $\texttt{r} \cdot x$ 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 $(\texttt{unit} + \texttt{unit}) \times \texttt{nat}$, either doubles or squares its second component depending on the form of its first component:

$$\texttt{match}\, e\, \{\langle \texttt{l} \cdot \langle\rangle, x\rangle \Rightarrow x + x \mid \langle \texttt{r} \cdot \langle\rangle, y\rangle \Rightarrow y * y\}. \qquad (15.1)$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 13 and 14.

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

15.1 A Pattern Language

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

Expr	e	$::=$	$\texttt{match}(e;rs)$	$\texttt{match}\, e\, \{rs\}$	case analysis
Rules	rs	$::=$	$\texttt{rules}[n](r_1;\ldots;r_n)$	$r_1 \mid \ldots \mid r_n$	$(n \geq 0)$
Rule	r	$::=$	$\texttt{rule}[k](p;x_1,\ldots,x_k.e)$	$p \Rightarrow e$	$(k \geq 0)$
Pat	p	$::=$	$\texttt{wild}$	$\_$	wild card
			x	x	variable
			$\texttt{triv}$	$\langle\rangle$	unit
			$\texttt{pair}(p_1;p_2)$	$\langle p_1, p_2\rangle$	pair
			$\texttt{in[l]}(p)$	$\texttt{l} \cdot p$	left injection
			$\texttt{in[r]}(p)$	$\texttt{r} \cdot p$	right injection

The operator `match` has arity $(0,0)$, specifying that it takes two operands, the expression to match and a series of rules. A sequence of rules is constructed using the operatator $\texttt{rules}[n]$, which has arity $(0,\ldots,0)$ specifying that it has $n \geq 0$ operands. Each rule is constructed by the operator $\texttt{rule}[k]$ of arity $(0,k)$ which specifies that it has two operands, binding k variables in the second.

15.2 Statics

The statics of $\mathcal{L}\{pat\}$ makes use of a special form of hypothetical judgement, written

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

with almost the same meaning as

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

except that each variable is required to be used *at most once* in p. When reading the judgement $\Lambda \Vdash p : \tau$ it is helpful to think of Λ as an *output*, and p and τ as *inputs*. Given p and τ, the rules determine the hypotheses Λ such that $\Lambda \Vdash p : \tau$.

$$\frac{}{x : \tau \Vdash x : \tau} \tag{15.2a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau} \tag{15.2b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \texttt{unit}} \tag{15.2c}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \quad \Lambda_2 \Vdash p_2 : \tau_2 \quad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\ \Lambda_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2} \tag{15.2d}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1}{\Lambda_1 \Vdash \texttt{l} \cdot p : \tau_1 + \tau_2} \tag{15.2e}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2}{\Lambda_2 \Vdash \texttt{r} \cdot p : \tau_1 + \tau_2} \tag{15.2f}$$

Rule (15.2a) states that a variable is a pattern of type τ. Rule (15.2d) states that a pair pattern consists of two patterns with disjoint variables.

The typing judgments for a rule,

$$p \Rightarrow e : \tau > \tau',$$

and for a sequence of rules,

$$r_1 \mid \ldots \mid r_n : \tau > \tau',$$

specify that rules transform a value of type τ into a value of type τ'. These judgements are inductively defined as follows:

$$\frac{\Lambda \Vdash p : \tau \quad \Gamma\ \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau > \tau'} \tag{15.3a}$$

$$\frac{\Gamma \vdash r_1 : \tau > \tau' \quad \ldots \quad \Gamma \vdash r_n : \tau > \tau'}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'} \tag{15.3b}$$

Using the typing judgements for rules, the typing rule for a `match` expression may be stated quite easily:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau'}{\Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'} \tag{15.4}$$

15.3 Dynamics

A *substitution*, θ, is a finite mapping from variables to values. If θ is the substitution $\langle x_1 : e_1\rangle \otimes \cdots \otimes \langle x_k : e_k\rangle$, we write $\hat{\theta}(e)$ for $[e_1, \ldots, e_k / x_1, \ldots, x_k]e$. The judgement $\theta : \Lambda$ is inductively defined by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{15.5a}$$

$$\frac{\theta : \Lambda \quad \theta(x) = e \quad e : \tau}{\theta : \Lambda, x : \tau} \tag{15.5b}$$

The judgement $\theta \Vdash p \lhd e$ states that the pattern, p, matches the value, e, as witnessed by the substitution, θ, defined on the variables of p. This judgement is inductively defined by the following rules:

$$\frac{}{\langle x : e\rangle \Vdash x \lhd e} \tag{15.6a}$$

$$\frac{}{\emptyset \Vdash \_ \lhd e} \tag{15.6b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle \lhd \langle\rangle} \tag{15.6c}$$

$$\frac{\theta_1 \Vdash p_1 \lhd e_1 \quad \theta_2 \Vdash p_2 \lhd e_2 \quad dom(\theta_1) \cap dom(\theta_2) = \emptyset}{\theta_1 \otimes \theta_2 \Vdash \langle p_1, p_2\rangle \lhd \langle e_1, e_2\rangle} \tag{15.6d}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{l} \cdot p \lhd \mathtt{l} \cdot e} \tag{15.6e}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{r} \cdot p \lhd \mathtt{r} \cdot e} \tag{15.6f}$$

These rules simply collect the bindings for the pattern variables required to form a substitution witnessing the success of the matching process.

The judgement $e \perp p$ states that e does not match the pattern p. It is inductively defined by the following rules:

$$\frac{e_1 \perp p_1}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{15.7a}$$

$$\frac{e_2 \perp p_2}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{15.7b}$$

$$\frac{}{\mathtt{l} \cdot e \perp \mathtt{r} \cdot p} \tag{15.7c}$$

$$\frac{e \perp p}{\mathtt{l} \cdot e \perp \mathtt{l} \cdot p} \tag{15.7d}$$

$$\frac{}{\mathtt{r} \cdot e \perp \mathtt{l} \cdot p} \tag{15.7e}$$

$$\frac{e \perp p}{\mathtt{r} \cdot e \perp \mathtt{r} \cdot p} \tag{15.7f}$$

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.

Theorem 15.1. *Suppose that $e : \tau$, e val, and $\Lambda \Vdash p : \tau$. Then either there exists θ such that $\theta : \Lambda$ and $\theta \Vdash p \lhd e$, or $e \perp p$.*

Proof. By rule induction on Rules (15.2), making use of the canonical forms lemma to characterize the shape of e based on its type. □

The dynamics of the `match` expression is given in terms of the pattern match and mismatch judgements as follows:

$$\frac{e \mapsto e'}{\mathtt{match}\, e\, \{rs\} \mapsto \mathtt{match}\, e'\, \{rs\}} \tag{15.8a}$$

$$\frac{e \;\mathsf{val}}{\mathtt{match}\, e\, \{\} \;\mathsf{err}} \tag{15.8b}$$

$$\frac{e \;\mathsf{val} \quad \theta \Vdash p_0 \lhd e}{\mathtt{match}\, e\, \{p_0 \Rightarrow e_0 \,|\, rs\} \mapsto \hat{\theta}(e_0)} \tag{15.8c}$$

$$\frac{e \;\mathsf{val} \quad e \perp p_0 \quad \mathtt{match}\, e\, \{rs\} \mapsto e'}{\mathtt{match}\, e\, \{p_0 \Rightarrow e_0 \,|\, rs\} \mapsto e'} \tag{15.8d}$$

Rule (15.8b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (15.8c) 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 15.2 (Preservation). *If $e \mapsto e'$ and $e : \tau$, then $e' : \tau$.*

Proof. By a straightforward induction on the derivation of $e \mapsto e'$. □

15.4 Exhaustiveness and Redundancy

While it is possible to state and prove a progress theorem for $\mathcal{L}\{pat\}$ as defined in Section 15.1 on page 132, 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.

15.4.1 Match Constraints

To express exhaustiveness and irredundancy, we introduce a language of *match constraints* that identify a subset of the closed values of a type. With each rule we associate a constraint 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 constraint of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match constraint also satisfies the match constraint of some preceding rule.

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

Constr	ξ	::=	`all[`τ`]`	$\top$	truth
			`and(`ξ_1`;`ξ_2`)`	$\xi_1 \wedge \xi_2$	conjunction
			`nothing[`τ`]`	$\bot$	falsity
			`or(`ξ_1`;`ξ_2`)`	$\xi_1 \vee \xi_2$	disjunction
			`in[l](`ξ_1`)`	$\mathtt{l} \cdot \xi_1$	left injection
			`in[r](`ξ_2`)`	$\mathtt{r} \cdot \xi_2$	right injection
			`triv`	$\langle\rangle$	unit
			`pair(`ξ_1`;`ξ_2`)`	$\langle \xi_1, \xi_2 \rangle$	pair

It is easy to define the judgement $\xi : \tau$ specifying that the constraint ξ constrains values of type τ.

The *De Morgan Dual*, $\overline{\xi}$, of a match constraint, ξ, is defined by the following rules:

$$
\begin{aligned}
\overline{\top} &= \bot \\
\overline{\xi_1 \wedge \xi_2} &= \overline{\xi_1} \vee \overline{\xi_2} \\
\overline{\bot} &= \top \\
\overline{\xi_1 \vee \xi_2} &= \overline{\xi_1} \wedge \overline{\xi_2} \\
\overline{\mathtt{l} \cdot \xi_1} &= \mathtt{l} \cdot \overline{\xi_1} \vee \mathtt{r} \cdot \top \\
\overline{\mathtt{r} \cdot \xi_1} &= \mathtt{r} \cdot \overline{\xi_1} \vee \mathtt{l} \cdot \top \\
\overline{\langle\rangle} &= \bot \\
\overline{\langle \xi_1, \xi_2 \rangle} &= \langle \overline{\xi_1}, \xi_2 \rangle \vee \langle \xi_1, \overline{\xi_2} \rangle \vee \langle \overline{\xi_1}, \overline{\xi_2} \rangle
\end{aligned}
$$

Intuitively, the dual of a match constraint expresses the negation of that constraint. In the case of the last four rules it is important to keep in mind that these constraints apply only to specific types.

The *satisfaction* judgement, $e \models \xi$, is defined for values e and constraints ξ of the same type by the following rules:

$$\frac{}{e \models \top} \tag{15.9a}$$

$$\frac{e \models \xi_1 \quad e \models \xi_2}{e \models \xi_1 \wedge \xi_2} \tag{15.9b}$$

$$\frac{e \models \xi_1}{e \models \xi_1 \vee \xi_2} \tag{15.9c}$$

$$\frac{e \models \xi_2}{e \models \xi_1 \vee \xi_2} \tag{15.9d}$$

$$\frac{e_1 \models \xi_1}{\mathtt{l} \cdot e_1 \models \mathtt{l} \cdot \xi_1} \tag{15.9e}$$

$$\frac{e_2 \models \xi_2}{\mathtt{r} \cdot e_2 \models \mathtt{r} \cdot \xi_2} \tag{15.9f}$$

$$\frac{}{\langle\rangle \models \langle\rangle} \tag{15.9g}$$

$$\frac{e_1 \models \xi_1 \quad e_2 \models \xi_2}{\langle e_1, e_2 \rangle \models \langle \xi_1, \xi_2 \rangle} \tag{15.9h}$$

The De Morgan dual construction negates a constraint.

Lemma 15.3. *If ξ is a constraint on values of type τ, then $e \models \overline{\xi}$ if, and only if, $e \not\models \xi$.*

We define the *entailment* of two constraints, $\xi_1 \models \xi_2$ to mean that $e \models \xi_2$ whenever $e \models \xi_1$. By Lemma 15.3 we have that $\xi_1 \models \xi_2$ iff $\models \overline{\xi_1} \vee \xi_2$. We often write $\xi_1, \ldots, \xi_n \models \xi$ for $\xi_1 \wedge \ldots \wedge \xi_n \models \xi$ so that in particular $\models \xi$ means $e \models \xi$ for every value $e : \tau$.

15.4.2 Enforcing Exhaustiveness and Redundancy

To enforce exhaustiveness and irredundancy the statics of pattern matching is augmented with constraints that express the set of values matched by a given set of rules. A sequence of rules is *exhaustive* if every value of suitable type satisfies the associated constraint. A rule is *redundant* relative to the preceding rules if every value satisfying its constraint satisfies one of the preceding constraints. A sequence of rules is *irredundant* iff no rule is redundant relative to the rules that precede it in the sequence.

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:

$$\frac{}{x : \tau \Vdash x : \tau\ [\top]} \tag{15.10a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau\ [\top]} \tag{15.10b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}\ [\langle\rangle]} \tag{15.10c}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1\ [\xi_1]}{\Lambda_1 \Vdash \mathtt{l} \cdot p : \tau_1 + \tau_2\ [\mathtt{l} \cdot \xi_1]} \tag{15.10d}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2\ [\xi_2]}{\Lambda_2 \Vdash \mathtt{r} \cdot p : \tau_1 + \tau_2\ [\mathtt{r} \cdot \xi_2]} \tag{15.10e}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1\ [\xi_1] \quad \Lambda_2 \Vdash p_2 : \tau_2\ [\xi_2] \quad \Lambda_1 \# \Lambda_2}{\Lambda_1\, \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2\ [\langle \xi_1, \xi_2\rangle]} \tag{15.10f}$$

Lemma 15.4. *Suppose that $\Lambda \Vdash p : \tau\ [\xi]$. For every $e : \tau$ such that e val, $e \models \xi$ iff $\theta \Vdash p \lhd e$ for some θ, and $e \not\models \xi$ iff $e \perp p$.*

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.

$$\frac{\Lambda \Vdash p : \tau \; [\xi] \quad \Gamma \, \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau > \tau' \; [\xi]} \tag{15.11a}$$

$$\frac{\begin{array}{c}(\forall 1 \leq i \leq n) \; \xi_i \not\models \xi_1 \vee \ldots \vee \xi_{i-1} \\ \Gamma \vdash r_1 : \tau > \tau' \; [\xi_1] \qquad \ldots \qquad \Gamma \vdash r_n : \tau > \tau' \; [\xi_n]\end{array}}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau' \; [\xi_1 \vee \ldots \vee \xi_n]} \tag{15.11b}$$

Rule (15.11b) requires that each successive rule not be redundant relative to the preceding rules. The overall constraint associated to the rule sequence specifies that every value of type τ satisfy the constraint associated with some rule.

The typing rule for `match` expressions demands that the rules that comprise it be exhaustive:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau' \; [\xi] \quad \models \xi}{\Gamma \vdash \mathtt{match}\, e \, \{rs\} : \tau'} \tag{15.12}$$

Rule (15.11b) ensures that ξ is a disjunction of the match constraints associated to the constituent rules of the match expression. The requirement that ξ be valid amounts to requiring that every value of type τ satisfies the constraint of at least one rule of the match.

Theorem 15.5. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. The exhaustiveness check in Rule (15.12) ensures that if e val and $e : \tau$, then $e \models \xi$. The form of ξ given by Rule (15.11b) ensures that $e \models \xi_i$ for some constraint ξ_i corresponding to the ith rule. By Lemma 15.4 on the facing page the value e must match the ith rule, which is enough to ensure progress. □

15.4.3 Checking Exhaustiveness and Redundancy

Checking exhaustiveness and redundacy reduces to showing that the constraint validity judgement $\models \xi$ is decidable. We will prove this by defining a judgement Ξ incon, where Ξ is a finite set of constraints of the same type, with the meaning that no value of this type satisfies all of the constraints in Ξ. We will then show that either Ξ incon or not.

The rules defining inconsistency of a finite set, Ξ, of constraints of the same type are as follows:

$$\frac{\Xi \text{ incon}}{\Xi, \top \text{ incon}} \tag{15.13a}$$

$$\frac{\Xi, \xi_1, \xi_2 \ \mathsf{incon}}{\Xi, \xi_1 \wedge \xi_2 \ \mathsf{incon}} \tag{15.13b}$$

$$\frac{}{\Xi, \perp \ \mathsf{incon}} \tag{15.13c}$$

$$\frac{\Xi, \xi_1 \ \mathsf{incon} \quad \Xi, \xi_2 \ \mathsf{incon}}{\Xi, \xi_1 \vee \xi_2 \ \mathsf{incon}} \tag{15.13d}$$

$$\frac{}{\Xi, \mathtt{l} \cdot \xi_1, \mathtt{r} \cdot \xi_2 \ \mathsf{incon}} \tag{15.13e}$$

$$\frac{\Xi \ \mathsf{incon}}{\mathtt{l} \cdot \Xi \ \mathsf{incon}} \tag{15.13f}$$

$$\frac{\Xi \ \mathsf{incon}}{\mathtt{r} \cdot \Xi \ \mathsf{incon}} \tag{15.13g}$$

$$\frac{\Xi_1 \ \mathsf{incon}}{\langle \Xi_1, \Xi_2 \rangle \ \mathsf{incon}} \tag{15.13h}$$

$$\frac{\Xi_2 \ \mathsf{incon}}{\langle \Xi_1, \Xi_2 \rangle \ \mathsf{incon}} \tag{15.13i}$$

In Rule (15.13f) we write $\mathtt{l} \cdot \Xi$ for the finite set of constraints $\mathtt{l} \cdot \xi_1, \ldots, \mathtt{l} \cdot \xi_n$, where $\Xi = \xi_1, \ldots, \xi_n$, and similarly in Rules (15.13g), (15.13h), and (15.13i).

Lemma 15.6. *It is decidable whether or not Ξ incon.*

Proof. The premises of each rule involves only constraints that are proper components of the constraints in the conclusion. Consequently, we can simplify Ξ by inverting each of the applicable rules until no rule applies, then determine whether or not the resulting set, Ξ', is contradictory in the sense that it contains $\perp$ or both $\mathtt{l} \cdot \xi$ and $\mathtt{r} \cdot \xi'$ for some ξ and ξ'. □

Lemma 15.7. *Ξ incon iff $\Xi \models \perp$.*

Proof. From left to right we proceed by induction on Rules (15.13). From right to left we may show that if Ξ incon is not derivable, then there exists a value e such that $e \models \Xi$, and hence $\Xi \not\models \perp$. □

15.5 Notes

Pattern-matching against heterogeneous structured data was first explored in the context of logic programming languages, such as Prolog [48, 21], but with an execution model based on proof search. Pattern matching in the form described here is present in the functional languages Miranda [93], Hope [17], Haskell [44], Standard ML [60], and Caml [24].

Chapter 16

Generic Programming

16.1 Introduction

Many programs can be seen as instances of a general pattern applied to a particular situation. Very often the pattern is determined by the types of the data involved. For example, in Chapter 11 the pattern of computing by recursion over a natural number is isolated as the defining characteristic of the type of natural numbers. This concept will itself emerge as an instance of the concept of *type-generic*, or just *generic*, programming.

Suppose that we have a function, f, of type $\sigma \to \sigma'$ that transforms values of type σ into values of type σ'. For example, f might be the doubling function on natural numbers. We wish to extend f to a transformation from type $[\sigma/t]\tau$ to type $[\sigma'/t]\tau$ by applying f to various spots in the input where a value of type σ occurs to obtain a value of type σ', leaving the rest of the data structure alone. For example, τ might be $\texttt{bool} \times \sigma$, in which case f could be extended to a function of type $\texttt{bool} \times \sigma \to \texttt{bool} \times \sigma'$ that sends the pairs $\langle a, b\rangle$ to the pair $\langle a, f(b)\rangle$.

This example glosses over a significant problem of ambiguity of the extension. Given a function f of type $\sigma \to \sigma'$, it is not obvious in general how to extend it to a function mapping $[\sigma/t]\tau$ to $[\sigma'/t]\tau$. The problem is that it is not clear which of many occurrences of σ in $[\sigma/t]\tau$ are to be transformed by f, even if there is only one occurrence of σ. To avoid ambiguity we need a way to mark which occurrences of σ in $[\sigma/t]\tau$ are to be transformed, and which are to be left fixed. This can be achieved by isolating the *type operator*, $t\,.\,\tau$, which is a type expression in which a designated variable, t, marks the spots at which we wish the transformation to occur. Given $t\,.\,\tau$ and $f : \sigma \to \sigma'$, we can extend f unambiguously to a function of type $[\sigma/t]\tau \rightarrow [\sigma'/t]\tau$.

The technique of using a type operator to determine the behavior of a piece of code is called *generic programming*. The power of generic programming depends on which forms of type operator are considered. The simplest case is that of a *polynomial* type operator, one constructed from sum and product of types, including their nullary forms. These may be extended to *positive* type operators, which also permit restricted forms of function types.

16.2 Type Operators

A *type operator* is a type equipped with a designated variable whose occurrences mark the positions in the type where a transformation is to be applied. A type operator is represented by an abstractor $t\,.\,\tau$ such that $t\ \mathsf{type} \vdash \tau\ \mathsf{type}$. An example of a type operator is the abstractor

$$t\,.\,\mathtt{unit} + (\mathtt{bool} \times t)$$

in which occurrences of t mark the spots in which a transformation is to be applied. An *instance* of the type operator $t\,.\,\tau$ is obtained by substituting a type, σ, for the variable, t, within the type τ. We sometimes write $\mathtt{Map}[t\,.\,\tau](\sigma)$ for the substitution instance $[\sigma/t]\tau$.

The *polynomial* type operators are those constructed from the type variable, t, the types `void` and `unit`, and the product and sum type constructors, $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$. It is a straightforward exercise to give inductive definitions of the judgement $t\,.\,\tau\ \mathsf{poly}$ stating that the operator $t\,.\,\tau$ is a polynomial type operator.

16.3 Generic Extension

The *generic extension* primitive has the form

$$\mathtt{map}[t\,.\,\tau](x\,.\,e';e)$$

with statics given by the following rule:

$$\frac{t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Gamma, x : \sigma \vdash e' : \sigma' \quad \Gamma \vdash e : [\sigma/t]\tau}{\Gamma \vdash \mathtt{map}[t\,.\,\tau](x\,.\,e';e) : [\sigma'/t]\tau} \tag{16.1}$$

The abstractor $x\,.\,e'$ specifies a transformation from type σ, the type of x, to type σ', the type of e'. The expression e of type $[\sigma/t]\tau$ determines the value to be transformed to obtain a value of type $[\sigma'/t]\tau$. The occurrences of t in τ determine the spots at which the transformation given by $x.e$ is to be performed.

The dynamics of generic extension is specified by the following rules. We consider here only polynomial type operators, leaving the extension to positive type operators to be considered later.

$$\frac{}{\mathtt{map}[t.t](x.e';e) \mapsto [e/x]e'} \tag{16.2a}$$

$$\frac{}{\mathtt{map}[t.\mathtt{unit}](x.e';e) \mapsto \langle\rangle} \tag{16.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{map}[t.\tau_1 \times \tau_2](x.e';e) \\ \mapsto \\ \langle \mathtt{map}[t.\tau_1](x.e';e\cdot\mathtt{l}),\mathtt{map}[t.\tau_2](x.e';e\cdot\mathtt{r})\rangle\end{array}} \tag{16.2c}$$

$$\frac{}{\mathtt{map}[t.\mathtt{void}](x.e';e) \mapsto \mathtt{abort}(e)} \tag{16.2d}$$

$$\frac{}{\begin{array}{c}\mathtt{map}[t.\tau_1 + \tau_2](x.e';e) \\ \mapsto \\ \mathtt{case}\, e\, \{\mathtt{l}\cdot x_1 \Rightarrow \mathtt{l}\cdot\mathtt{map}[t.\tau_1](x.e';x_1) \mid \mathtt{r}\cdot x_2 \Rightarrow \mathtt{r}\cdot\mathtt{map}[t.\tau_2](x.e';x_2)\}\end{array}} \tag{16.2e}$$

Rule (16.2a) applies the transformation $x.e'$ to e itself, since the operator $t.t$ specifies that the transformation is to be perfomed directly. Rule (16.2b) states that the empty tuple is transformed to itself. Rule (16.2c) states that to transform e according to the operator $t.\tau_1 \times \tau_2$, the first component of e is transformed according to $t.\tau_1$ and the second component of e is transformed according to $t.\tau_2$. Rule (16.2d) states that the transformation of a value of type `void` aborts, since there can be no such values. Rule (16.2e) states that to transform e according to $t.\tau_1 + \tau_2$, case analyze e and reconstruct it after transforming the injected value according to $t.\tau_1$ or $t.\tau_2$.

Consider the type operator $t.\tau$ given by $t.\mathtt{unit} + (\mathtt{bool} \times t)$. Let $x.e$ be the abstractor $x.\mathtt{s}(x)$, which increments a natural number. Using Rules (16.2) we may derive that

$$\mathtt{map}[t.\tau](x.e;\mathtt{r}\cdot\langle\mathtt{tt},n\rangle) \mapsto^* \mathtt{r}\cdot\langle\mathtt{tt},n+1\rangle.$$

The natural number in the second component of the pair is incremented, since the type variable, t, occurs in that position in the type operator $t\,.\,\tau$.

Theorem 16.1 (Preservation). *If* $\texttt{map}[t\,.\,\tau](x\,.\,e';e) : \rho$ *and* $\texttt{map}[t\,.\,\tau](x\,.\,e';e) \mapsto e''$*, then* $e'' : \rho$*.*

Proof. By inversion of Rule (16.1) we have

1. t type $\vdash \tau$ type;
2. $x : \sigma \vdash e' : \sigma'$ for some σ and σ';
3. $e : [\sigma/t]\tau$;
4. ρ is $[\sigma'/t]\tau$.

We proceed by cases on Rules (16.2). For example, consider Rule (16.2c). It follows from inversion that $\texttt{map}[t\,.\,\tau_1](x\,.\,e';e\cdot\texttt{l}) : [\sigma'/t]\tau_1$, and similarly that $\texttt{map}[t\,.\,\tau_2](x\,.\,e';e\cdot\texttt{r}) : [\sigma'/t]\tau_2$. It is easy to check that

$$\langle \texttt{map}[t\,.\,\tau_1](x\,.\,e';e\cdot\texttt{l}),\texttt{map}[t\,.\,\tau_2](x\,.\,e';e\cdot\texttt{r})\rangle$$

has type $[\sigma'/t]\tau_1 \times \tau_2$, as required. □

The *positive* type operators extend the polynomial type operators to admit restricted forms of function type. Specifically, $t\,.\,\tau_1 \to \tau_2$ is a positive type operator, provided that (1) *t does not occur* in τ_1, and (2) $t\,.\,\tau_2$ is a positive type operator. In general, any occurrences of a type variable t in the domain a function type are said to be *negative occurrences*, whereas any occurrences of t within the range of a function type, or within a product or sum type, are said to be *positive occurrences*.[1] A positive type operator is one for which only positive occurrences of the parameter, t, are permitted.

The generic extension according to a positive type operator is defined similarly to the case of a polynomial type operator, with the following additional rule:

$$\frac{}{\texttt{map}[t\,.\,\tau_1 \to \tau_2](x\,.\,e';e) \mapsto \lambda\,(x_1{:}\tau_1.\,\texttt{map}[t\,.\,\tau_2](x\,.\,e';e(x_1)))} \quad (16.3)$$

[1]The origin of this terminology seems to be that a function type $\tau_1 \to \tau_2$ is analogous to the implication $\phi_1 \supset \phi_2$, which is classically equivalent to $\neg\phi_1 \vee \phi_2$, so that occurrences in the domain are under the negation.

Since t is not permitted to occur within the domain type, the type of the result is $\tau_1 \to [\sigma'/t]\tau_2$, assuming that e is of type $\tau_1 \to [\sigma/t]\tau_2$. It is easy to verify preservation for the generic extension of a positive type operator.

It is interesting to consider what goes wrong if we relax the restriction on positive type operators to admit negative, as well as positive, occurrences of the parameter of a type operator. Consider the type operator $t\,.\,\tau_1 \to \tau_2$, without restriction on t, and suppose that $x : \sigma \vdash e' : \sigma'$. The generic extension $\texttt{map}[t\,.\,\tau_1 \to \tau_2](x\,.\,e';e)$ should have type $[\sigma'/t]\tau_1 \to [\sigma'/t]\tau_2$, given that e has type $[\sigma/t]\tau_1 \to [\sigma/t]\tau_2$. The extension should yield a function of the form

$$\lambda\,(x_1 : [\sigma'/t]\tau_1 . \ldots (e(\ldots(x_1))))$$

in which we apply e to a transformation of x_1 and then transform the result. The trouble is that we are given, inductively, that $\texttt{map}[t\,.\,\tau_1](x\,.\,e';-)$ transforms values of type $[\sigma/t]\tau_1$ into values of type $[\sigma'/t]\tau_1$, but *we need to go the other way around* in order to make x_1 suitable as an argument for e. But there is no obvious way to obtain the required transformation.

One solution to this is to assume that the fundamental transformation $x\,.\,e'$ is *invertible* so that we may apply the inverse transformation on x_1 to get an argument of type suitable for e, then apply the forward transformation on the result, just as in the positive case. Since we cannot invert an arbitrary transformation, we must instead pass both the transformation and its inverse to the generic extension operation so that it can "go backwards" as necessary to cover negative occurrences of the type parameter. So in the general case the generic extension applies only when we are given a *type isomorphism* (a pair of mutually inverse mappings between two types), and then results in another isomorphism pair. We leave the formulation of this as an exercise for the reader.

16.4 Notes

The concept of the functorial action of a type constructor has its roots in category theory [52]. Generic programming is essentially the application of this idea to computation [42].