10 Product Types

The *binary product* of two types consists of *ordered pairs* of values, one from each type in the order specified. The associated elimination 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 elimination 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*, $\langle \tau_i \rangle_{i \in I}$, 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 *labeled tuples*, or *records*, indexed by finite sets of symbols. Similarly to binary products, finite products admit both an eager and a lazy interpretation.

10.1 Nullary and Binary Products

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

Typ	τ	$::=$	$\texttt{unit}$	$\texttt{unit}$	nullary product
			$\texttt{prod}(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	binary product
Exp	e	$::=$	$\texttt{triv}$	$\langle\rangle$	null tuple
			$\texttt{pair}(e_1;e_2)$	$\langle e_1, e_2 \rangle$	ordered pair
			$\texttt{pr[l]}(e)$	$e \cdot \texttt{l}$	left projection
			$\texttt{pr[r]}(e)$	$e \cdot \texttt{r}$	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.

$$\frac{}{\Gamma \vdash \langle\rangle : \texttt{unit}} \tag{10.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \tag{10.1b}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{l} : \tau_1} \tag{10.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{r} : \tau_2} \tag{10.1d}$$

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

$$\frac{}{\langle\rangle \ \mathsf{val}} \tag{10.2a}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2 \rangle \ \mathsf{val}} \tag{10.2b}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\langle e_1, e_2 \rangle \longmapsto \langle e_1', e_2 \rangle}\right] \tag{10.2c}$$

$$\left[\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto e_2'}{\langle e_1, e_2 \rangle \longmapsto \langle e_1, e_2' \rangle}\right] \tag{10.2d}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{l} \longmapsto e' \cdot \mathtt{l}} \tag{10.2e}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{r} \longmapsto e' \cdot \mathtt{r}} \tag{10.2f}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2 \rangle \cdot \mathtt{l} \longmapsto e_1} \tag{10.2g}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2 \rangle \cdot \mathtt{r} \longmapsto e_2} \tag{10.2h}$$

The bracketed rules and premises are 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 10.1 (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

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

Proof Preservation is proved by induction on transition defined by rules (10.2). Progress is proved by induction on typing defined by rules (10.1). □

10.2 Finite Products

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

Typ	τ	$::=$	$\mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i\in I})$	$\langle \tau_i\rangle_{i\in I}$	product
Exp	e	$::=$	$\mathtt{tpl}(\{i \hookrightarrow e_i\}_{i\in I})$	$\langle e_i\rangle_{i\in I}$	tuple
			$\mathtt{pr}[i](e)$	$e \cdot i$	projection

The variable I stands for a finite *index set* over which products are formed. The type $\mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i\in I})$, or $\prod_{i\in I} \tau_i$ for short, is the type of I*-tuples* of expressions e_i of type τ_i, one for each $i \in I$. An I-tuple has the form $\mathtt{tpl}(\{i \hookrightarrow e_i\}_{i\in I})$, or $\langle e_i\rangle_{i\in I}$ for short, and for each $i \in I$ the ith projection from an I-tuple e is written $\mathtt{pr}[i](e)$, or $e \cdot i$ for short.

When $I = \{ i_1, \ldots, i_n \}$, the I-tuple type may be written in the form

$$\langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n\rangle$$

where we make explicit the association of a type to each index $i \in I$. Similarly, we may write

$$\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle$$

for the I-tuple whose ith component is e_i.

Finite products generalize empty and binary products by choosing I to be empty or the two-element set $\{ \mathtt{l}, \mathtt{r} \}$, respectively. In practice, I is often chosen to be a finite set of symbols that serve as labels for the components of the tuple to enhance readability.

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle : \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n\rangle} \tag{10.3a}$$

$$\frac{\Gamma \vdash e : \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n\rangle \quad (1 \le k \le n)}{\Gamma \vdash e \cdot i_k : \tau_k} \tag{10.3b}$$

In rule (10.3b), the index $i_k \in I$ is a *particular* element of the index set I, whereas in rule (10.3a), the indices $i_1, \ldots, i_n$ range over the entire index set I.

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

$$\frac{[e_1 \ \mathsf{val} \quad \ldots \quad e_n \ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle \ \mathsf{val}} \tag{10.4a}$$

$$\left[\frac{\left\{\begin{matrix} e_1 \ \mathsf{val} \quad \ldots \quad e_{j-1} \ \mathsf{val} \quad e'_1 = e_1 \quad \ldots \quad e'_{j-1} = e_{j-1} \\ e_j \longmapsto e'_j \quad e'_{j+1} = e_{j+1} \quad \ldots \quad e'_n = e_n \end{matrix}\right\}}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle \longmapsto \langle i_1 \hookrightarrow e'_1, \ldots, i_n \hookrightarrow e'_n\rangle}\right] \tag{10.4b}$$

$$\frac{e \longmapsto e'}{e \cdot i \longmapsto e' \cdot i} \tag{10.4c}$$

$$\frac{[\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle \ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle \cdot i_k \longmapsto e_k} \tag{10.4d}$$

As formulated, rule (10.4b) specifies that the components of a tuple are evaluated in *some* sequential order, without specifying the order in which the components are considered. It is not hard, but a bit technically complicated, to impose an evaluation order by imposing a total ordering on the index set and evaluating components according to this ordering.

Theorem 10.2 (Safety)**.** *If $e : \tau$, then either e val or there exists e' such that $e' : \tau$ and $e \longmapsto e'$.*

Proof The safety theorem is decomposed into progress and preservation lemmas, which are proved as in Section 10.1. □

10.3 Primitive Mutual Recursion

Using products we may simplify the primitive recursion construct of **T** so that only the recursive result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\mathtt{iter}\{e_0; x.e_1\}(e)$, we may define $\mathtt{rec}\{e_0; x.y.e_1\}(e)$ to be $e' \cdot \mathtt{r}$, where e' is the expression

$$\mathtt{iter}\{\langle \mathtt{z}, e_0\rangle; x'.\langle \mathtt{s}(x' \cdot \mathtt{l}), [x' \cdot \mathtt{r}/x]e_1\rangle\}(e).$$

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 another 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 primitive recursion*, in which we define two functions simultaneously by primitive recursion. 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 if and only if n is even, and $o(n)$ is non-zero if and only if n is odd.

To define these functions in **T** enriched with products, we first define an auxiliary function $e_{\mathtt{eo}}$ of type

$$\mathtt{nat} \to (\mathtt{nat} \times \mathtt{nat})$$

that computes both results simultaneously by swapping back and forth on recursive calls:

$$\lambda\,(n : \mathtt{nat} \times \mathtt{nat})\,\mathtt{iter}\,n\,\{\mathtt{z} \hookrightarrow \langle 1, 0\rangle \mid \mathtt{s}(b) \hookrightarrow \langle b \cdot \mathtt{r}, b \cdot \mathtt{l}\rangle\}.$$

We may then define $e_{\mathtt{ev}}$ and $e_{\mathtt{od}}$ as follows:

$$e_{\mathtt{ev}} \triangleq \lambda\,(n : \mathtt{nat})\, e_{\mathtt{eo}}(n) \cdot \mathtt{l}$$
$$e_{\mathtt{od}} \triangleq \lambda\,(n : \mathtt{nat})\, e_{\mathtt{eo}}(n) \cdot \mathtt{r}.$$

10.4 Notes

Product types are the most basic form of structured data. All languages have some form of product type but often in a form that is combined with other, separable, concepts. Common manifestations of products include (1) functions with "multiple arguments" or "multiple results"; (2) "objects" represented as tuples of mutually recursive functions; (3) "structures," which are tuples with mutable components. There are many papers on finite product types, which include record types as a special case. Pierce (2002) provides a thorough account of record types and their subtyping properties (for which, see Chapter 24). Allen et al. (2006) analyze many of the key ideas in the framework of dependent type theory.

Exercises

10.1. A *database schema* may be thought of as a finite product type $\prod_{i \in I} \tau$, in which the *columns*, or *attributes*, are labeled by the indices I whose values are restricted to *atomic* types, such as `nat` and `str`. A value of a schema type is called a *tuple*, or *instance*, of that schema. A *database* may be thought of as a finite sequence of such tuples, called the *rows* of the database. Give a representation of a database using function, product, and natural numbers types, and define the *project* operation that sends a database with columns I to a database with columns $I' \subseteq I$ by restricting each row to the specified columns.

10.2. Rather than choose between a lazy and an eager dynamics for products, we can instead distinguish two forms of product type, called the *positive* and the *negative*. The statics of the negative product is given by rules (10.1), and the dynamics is lazy. The statics of the positive product, written $\tau_1 \otimes \tau_2$, is given by the following rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{fuse}(e_1; e_2) : \tau_1 \otimes \tau_2} \tag{10.5a}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 \otimes \tau_2 \quad \Gamma\, x_1 : \tau_1\, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \mathtt{split}(e_0; x_1, x_2.e) : \tau} \tag{10.5b}$$

The dynamics of `fuse`, the introduction form for the positive pair, is eager, essentially because the elimination form, `split`, extracts both components simultaneously.

Show that the negative product is definable in terms of the positive product using the unit and function types to express the lazy semantics of negative pairing. Show that the positive product is definable in terms of the negative product, provided that we have at our disposal a `let` expression with a by-value dynamics so that we may enforce eager evaluation of positive pairs.

10.3. Specializing Exercise **10.2** to nullary products, we obtain a positive and a negative unit type. The negative unit type is given by rules (10.1), with no elimination forms and one introduction form. Give the statics and dynamics for a positive unit type, and show that the positive and negative unit types are inter-definable without any further assumptions.