11 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 allow 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.

11.1 Nullary and Binary Sums

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

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

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

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

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{abort}(e) : \tau} \tag{11.1a}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{l}\cdot e : \tau_1 + \tau_2} \tag{11.1b}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{r}\cdot e : \tau_1 + \tau_2} \tag{11.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\} : \tau} \tag{11.1d}$$

For the sake of readability, in rules (11.1b) and (11.1c) we have written $\mathtt{l}\cdot e$ and $\mathtt{r}\cdot e$ in place of the abstract syntax $\mathtt{in[l]}\{\tau_1;\tau_2\}(e)$ and $\mathtt{in[r]}\{\tau_1;\tau_2\}(e)$, which includes the types τ_1 and τ_2 explicitly. In rule (11.1d), both branches of the case analysis must have the same type. Because a type expresses a static "prediction" on the form of the value of an expression, and because an expression 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:

$$\frac{e \longmapsto e'}{\mathtt{abort}(e) \longmapsto \mathtt{abort}(e')} \tag{11.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{l}\cdot e\ \mathsf{val}} \tag{11.2b}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{r}\cdot e\ \mathsf{val}} \tag{11.2c}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{l}\cdot e \longmapsto \mathtt{l}\cdot e'}\right] \tag{11.2d}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{r}\cdot e \longmapsto \mathtt{r}\cdot e'}\right] \tag{11.2e}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\} \longmapsto \mathtt{case}\, e'\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\}} \tag{11.2f}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\,\mathtt{l}\cdot e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_1]e_1} \tag{11.2g}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\,\mathtt{r}\cdot e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_2]e_2} \tag{11.2h}$$

The bracketed premises and rules are 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 11.1 (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val or $e \longmapsto e'$ for some e'.

Proof The proof proceeds by induction on rules (11.2) for preservation, and by induction on rules (11.1) for progress. □

11.2 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:

Typ	τ	$::=$	$\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i\in I})$	$[\tau_i]_{i\in I}$	sum
Exp	e	$::=$	$\mathtt{in}[i]\{\vec{\tau}\}(e)$	$i \cdot e$	injection
			$\mathtt{case}(e;\{i \hookrightarrow x_i.e_i\}_{i\in I})$	$\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i\}_{i\in I}$	case analysis

The variable I stands for a finite index set over which sums are formed. The notation $\vec{\tau}$ stands for a finite function $\{i \hookrightarrow \tau_i\}_{i\in I}$ for some index set I. The type $\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i\in I})$, or $\sum_{i\in I} \tau_i$ for short, is the type of I-classified values of the form $\mathtt{in}[i]\{I\}(e_i)$, or $i \cdot e_i$ for short, where $i \in I$ and e_i is an expression of type τ_i. An I-classified value is analyzed by an I-way case analysis of the form $\mathtt{case}(e;\{i \hookrightarrow x_i.e_i\}_{i\in I})$.

When $I = \{ i_1, \ldots, i_n \}$, the type of I-classified values may be written

$$[i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n]$$

specifying the type associated with each class $l_i \in I$. Correspondingly, the I-way case analysis has the form

$$\mathtt{case}\, e\, \{i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n\}.$$

Finite sums generalize empty and binary sums 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 names for the classes so as to enhance readability.

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

$$\frac{\Gamma \vdash e : \tau_k \quad (1 \leq k \leq n)}{\Gamma \vdash i_k \cdot e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n]} \tag{11.3a}$$

$$\frac{\Gamma \vdash e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n] \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \mathtt{case}\, e\, \{i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n\} : \tau} \tag{11.3b}$$

These rules generalize the statics for nullary and binary sums given in Section 11.1.

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

$$\frac{[e\ \mathsf{val}]}{i \cdot e\ \mathsf{val}} \tag{11.4a}$$

$$\left[\frac{e \longmapsto e'}{i \cdot e \longmapsto i \cdot e'}\right] \tag{11.4b}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i\}_{i\in I} \longmapsto \mathtt{case}\, e'\, \{i \cdot x_i \hookrightarrow e_i\}_{i\in I}} \tag{11.4c}$$

$$\frac{i \cdot e\ \mathsf{val}}{\mathtt{case}\, i \cdot e\, \{i \cdot x_i \hookrightarrow e_i\}_{i\in I} \longmapsto [e/x_i]e_i} \tag{11.4d}$$

These again generalize the dynamics of binary sums given in Section 11.1.

Theorem 11.2 (Safety). *If $e : \tau$, then either e val or there exists $e' : \tau$ such that $e \longmapsto e'$.*

Proof The proof is like that for the binary case, as described in Section 11.1. □

11.3 Applications of Sum Types

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

11.3.1 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, $\langle\rangle$, whereas the type `void` has no elements at all. Consequently, if $e : \texttt{unit}$, then if e evaluates to a value, that value is $\langle\rangle$—in other words, e has *no interesting value*. On the other hand, if $e : \texttt{void}$, 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. Thus, 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!

11.3.2 Booleans

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

Typ	τ	::=	`bool`	`bool`	booleans
Exp	e	::=	`true`	`true`	truth
			`false`	`false`	falsity
			$\texttt{if}(e; e_1; e_2)$	$\texttt{if}\, e \,\texttt{then}\, e_1 \,\texttt{else}\, e_2$	conditional

The expression $\texttt{if}(e; e_1; e_2)$ branches on the value of $e : \texttt{bool}$.

The statics of Booleans is given by the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \tag{11.5a}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \tag{11.5b}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e \,\texttt{then}\, e_1 \,\texttt{else}\, e_2 : \tau} \tag{11.5c}$$

The dynamics is given by the following value and transition rules:

$$\frac{}{\texttt{true val}} \tag{11.6a}$$

$$\frac{}{\texttt{false val}} \tag{11.6b}$$

$$\frac{}{\texttt{if}\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \longmapsto e_1} \tag{11.6c}$$

$$\frac{}{\texttt{if}\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \longmapsto e_2} \tag{11.6d}$$

$$\frac{e \longmapsto e'}{\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \longmapsto \texttt{if}\,e'\,\texttt{then}\,e_1\,\texttt{else}\,e_2} \tag{11.6e}$$

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

$$\texttt{bool} = \texttt{unit} + \texttt{unit} \tag{11.7a}$$
$$\texttt{true} = \texttt{l} \cdot \langle\rangle \tag{11.7b}$$
$$\texttt{false} = \texttt{r} \cdot \langle\rangle \tag{11.7c}$$
$$\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2 = \texttt{case}\,e\,\{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.7d}$$

In Equation (11.7d), the variables x_1 and x_2 are chosen arbitrarily such that $x_1 \notin e_1$ and $x_2 \notin e_2$. It is a simple matter to check that the readily-defined statics and dynamics of the type `bool` are engendered by these definitions.

11.3.3 Enumerations

More generally, sum types can 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 $\clubsuit$, $\diamondsuit$, $\heartsuit$, and $\spadesuit$, has as elimination form the case analysis

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

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\texttt{suit} = [\texttt{unit}]_{\_\in I}$, where $I = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and the type family is constant over this set. The case analysis form for a labeled 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.

Other examples of enumeration types abound. For example, most languages have a type `char` of characters, which is a large enumeration type containing all possible `Unicode` (or other such standard classification) characters. Each character is assigned a *code* (such as `UTF-8`) used for interchange among programs. The type `char` is equipped with operations such as $\texttt{chcode}(n)$ that yield the `char` associated to the code n, and $\texttt{codech}(c)$ that yield the code of character c. Using the linear ordering on codes we may define a total ordering of characters, called the *collating sequence* determined by that code.

11.3.4 Options

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

Typ	τ	$::=$	$\mathtt{opt}(\tau)$	$\tau\ \mathtt{opt}$	option
Exp	e	$::=$	$\mathtt{null}$	$\mathtt{null}$	nothing
			$\mathtt{just}(e)$	$\mathtt{just}(e)$	something
			$\mathtt{ifnull}\{\tau\}\{e_1; x.e_2\}(e)$	$\mathtt{which}\, e\, \{\mathtt{null} \hookrightarrow e_1 \mid \mathtt{just}(x) \hookrightarrow e_2\}$	null test

The type $\mathtt{opt}(\tau)$ represents the type of "optional" values of type τ. The introduction forms are $\mathtt{null}$, corresponding to "no value," and $\mathtt{just}(e)$, 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:[1]

$$\tau\ \mathtt{opt} = \mathtt{unit} + \tau \tag{11.8a}$$

$$\mathtt{null} = \mathtt{l} \cdot \langle\rangle \tag{11.8b}$$

$$\mathtt{just}(e) = \mathtt{r} \cdot e \tag{11.8c}$$

$$\mathtt{which}\, e\, \{\mathtt{null} \hookrightarrow e_1 \mid \mathtt{just}(x_2) \hookrightarrow e_2\} = \mathtt{case}\, e\, \{\mathtt{l} \cdot \_ \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.8d}$$

We leave it to the reader to check 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 arises from two related errors. The first error is to deem values of certain types to be mysterious entities called *pointers*. This terminology arises from suppositions about how these values might be represented at run-time, rather than on their semantic role in the language. 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 stand for a value of that type at all, but rather rejects all attempts to use it.

To help avoid such failures, such languages usually include a function, say $\mathtt{null} : \tau \rightarrow \mathtt{bool}$, that yields $\mathtt{true}$ if its argument is null, and $\mathtt{false}$ otherwise. Such a test 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

$$\mathtt{if}\, \mathtt{null}(e)\, \mathtt{then} \ldots \mathit{error} \ldots\ \mathtt{else} \ldots \mathit{proceed} \ldots. \tag{11.9}$$

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 is the failure to distinguish the type τ from the type $\tau\ \mathtt{opt}$. Rather than think of the elements of type τ as pointers, and thereby have to worry about the null pointer, we instead distinguish 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,

$$\mathtt{which}\, e\, \{\mathtt{null} \hookrightarrow e_{error} \mid \mathtt{just}(x) \hookrightarrow e_{ok}\}, \tag{11.10}$$

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 necessary. Note that such a change of type is not achieved by the simple Boolean-valued test exemplified by expression (11.9); the advantage of option types is precisely that they do so.

11.4 Notes

Heterogeneous data structures are ubiquitous. Sums codify heterogeneity, yet few languages support them in the form given here. The best approximation in commercial languages is the concept of a class in object-oriented programming. A class is an injection into a sum type, and dispatch is case analysis on the class of the data object. (See Chapter 26 for more on this correspondence.) The absence of sums is the origin of C.A.R. Hoare's self-described "billion dollar mistake," the null pointer (Hoare, 2009). Bad language designs put the burden of managing "null" values entirely at run-time, instead of making the possibility or the impossibility of "null" apparent at compile time.

Exercises

11.1. Complete the definition of a finite enumeration type sketched in Section 11.3.3. Derive enumeration types from finite sum types.

11.2. The essence of Hoare's mistake is the misidentification of the type $\tau\,\mathtt{opt}$ with the type $\mathtt{bool} \times \tau$. Values of the latter type are pairs consisting of a boolean "flag" and a value of type τ. The idea is that the flag indicates whether the associated value is "present." When the flag is `true`, the second component is present, and, when the flag is `false`, the second component is absent.

Analyze Hoare's mistake by attempting to define $\tau\,\mathtt{opt}$ to be the type $\mathtt{bool} \times \tau$ by filling in the following chart:

$$\begin{aligned}
\mathtt{null} &\triangleq \mathbf{?}\\
\mathtt{just}(e) &\triangleq \mathbf{?}\\
\mathtt{which}\,e\,\{\mathtt{null} \hookrightarrow e_1 \mid \mathtt{just}(x) \hookrightarrow e_2\} &\triangleq \mathbf{?}
\end{aligned}$$

Argue that *even if* we adopt Hoare's convention of admitting a "null" value of every type, the chart cannot be properly filled.

11.3. Databases have a version of the "null pointer" problem that arises when not every tuple provides a value for every attribute (such as a person's middle name). More generally, many commercial databases are limited to a single atomic type for each attribute, presenting problems when the value of that attribute may have several types (for example, one may have different sorts of postal codes depending on the country). Consider how to address these problems using the methods discussed in Exercise **10.1**. Suggest how to handle null values and heterogeneous values that avoids some of the complications that arise in traditional formulations of databases.

11.4. A *combinational circuit* is an open expression of type

$$x_1 : \texttt{bool}, \ldots, x_n : \texttt{bool} \vdash e : \texttt{bool},$$

which computes a boolean value from n boolean inputs. Define a NOR and a NAND gate as boolean circuits with two inputs and one output. There is no reason to restrict to a single output. For example, define an HALF-ADDER that takes two boolean inputs, but produces two boolean outputs, the sum and the carry outputs of the HALF-ADDER. Then define a FULL-ADDER that takes three inputs, the addends and an incoming carry, and produces two outputs, the sum and the outgoing carry. Define the type NYBBLE to be the product $\texttt{bool} \times \texttt{bool} \times \texttt{bool} \times \texttt{bool}$. Define the combinational circuit NYBBLE-ADDER that takes two nybbles as input and produces a nybble and a carry-out bit as output.

11.5. A *signal* is a time-varying sequence of booleans, representing the status of the signal at each time instant. An RS latch is a fundamental digital circuit with two input signals and two output signals. Define the type `signal` of signals to be the function type $\texttt{nat} \to \texttt{bool}$ of infinite sequences of booleans. Define an RS latch as a function of type

$$(\texttt{signal} \times \texttt{signal}) \to (\texttt{signal} \times \texttt{signal}).$$

Note

1 We often write an underscore in place of a bound variable that is not used within its scope.