Chapter 10

Data

Conjunction Junction, what's their function?
I got "and"... and "or",
They'll get you pretty far.

"And": That's an additive, like "this and that". ...
And then there's "or":
O-R, when you have a choice like "This or that".
"And"... and "or",
Get you pretty far.

— *Conjunction Junction (Schoolhouse Rock), Bob Dorough*

Here's hoping we meet now and then
It was great fun
But it was just one of those things.

— *Jubilee, Cole Porter*

Well-designed data structures can make programs efficient, understandable, extensible, secure, and easy to debug. For this reason, programmers focus much of their energy on designing and using data structures. How successful they are depends in part on the tools provided by their programming language for declaring and manipulating data. This chapter explores some of the key data dimensions in programming languages.

10.1 Products

Products are compound values that result from gluing other values together. They are data structures that correspond to the product domains we have been using in our mathematical metalanguage (see Section A.3.2) to represent structured mathematical values with components. Standard examples of products are 2-dimensional points (consisting of x and y components), employee records (consisting of name, sex, age, identification number, hiring date, etc.), and the sequences of points in a polygon.

There are a wide variety of product data structures in programming languages that differ along a surprising number of dimensions:

- How are product values created and later decomposed into parts?
- Are the components of the product indexed by position or by name?
- When accessing a component, can its index be calculated or must an index be a manifest constant?
- Are the components values (as in call-by-value) or computations (as in call-by-name/call-by-need)?
- Are the components of the product immutable or mutable?
- Is the length of the product fixed or variable?
- Are all components of the product required to have the "same type," i.e., are products *homogeneous*?
- When products are nested, are the nested components all required to have the same size and/or "shape"?
- How are products passed as arguments, returned as results, and stored in assignments?
- Can the lifetime of a product exceed the lifetime of an invocation of a procedure in which it is created?

In this section, we will explore many of these dimensions, using our operational and denotational tools where appropriate to explain interesting points in the design space of products.

Products are known by a confusing variety of names — such as *array*, *vector*, *sequence*, *tuple*, *string*, *list*, *structure*, *record*, *environment*, *table*, *module*, and *association list* — that are used inconsistently between languages. We shall be using some of these names in our study of products, but it is important to keep in mind that our use of a name may denote a different kind of product than what you might be familiar with from your programming experience.

10.1.1 Positional Products

10.1.1.1 Simple Positional Products

The simplest kind of product is a pair, which glues two values together. In Chapter 6, we studied the semantics of pairs in call-by-name and call-by-value versions of FL.

Pairs are an example of a **positional product**, in which component values are indexed by their position in the product value. We can extend pairs into more general positional products by adding the following two constructs to call-by-value FL![1]:

$$\begin{array}{lll} E ::= \ldots & & \\ \quad \mid \texttt{(product } E^{*}\texttt{)} & \text{[Product Creation]} \\ \quad \mid \texttt{(proj } N\ E\texttt{)} & \text{[Product Projection]} \end{array}$$

The expression `(product` E_1 `...` E_n`)` constructs an immutable positional product value whose n components are the values of the subexpressions E_1 through E_n. Such a value is traditionally known as a **tuple**. `(proj` N E_{prod}`)` extracts the component of the tuple denoted by E_{prod} that is at literal index N, where the components of an n-component product are indexed from 1 to n. An attempt to extract a component outside this index range is an error. The name `proj` is short for "project," the verb traditionally used to extract the component of a product.

An operational semantics of immutable positional products in call-by-value FL! is presented in Figure 10.1. A `product` expression with value expression components is considered a new kind of value expression. The [*product-progress*] rule evaluates the subexpressions of a `product` expression, so that the expression

```
(product (= 0 1) (* 2 3) (+ 3 4))
```

evaluates to the value expression

```
(product false 6 7).
```

The [*product projection*] rule extracts the value component at index N. If N is not in the valid range of indices, the `proj` expression is stuck, modeling an error. Using these rules, it is straightforward to show that the following FL expression evaluates to *9*:

[1] We study products in the context of a stateful language to facilitate coverage of product dimensions that involve state.

$$V \in \text{ValueExp} = \ldots \cup \{(\texttt{product}\ V_1\ \ldots\ V_n)\}$$

$$\frac{\langle E_i, S\rangle \Rightarrow \langle E_i{}', S'\rangle}{\begin{array}{l}\ \ \langle(\texttt{product}\ V_1\ \ldots\ V_{i-1}\ E_i\ E_{i+1}\ \ldots\ E_n), S\rangle \\ \Rightarrow\langle(\texttt{product}\ V_1\ \ldots\ V_{i-1}\ E_i{}'\ E_{i+1}\ \ldots\ E_n), S'\rangle\end{array}} \quad [\textit{product progress}]$$

$$\begin{array}{c}\langle(\texttt{proj}\ N\ (\texttt{product}\ V_1\ \ldots\ V_n)), S\rangle \Rightarrow \langle V_N, S\rangle, \\ \text{where } 1 \le N \le \text{n}\end{array} \quad [\textit{product projection}]$$

Figure 10.1: Operational semantics of immutable positional products in CBV FL!.

```
(let ((p (product (= 0 1) (* 2 3) (+ 4 5))))
  (if (proj 1 p) (proj 2 p) (proj 3 p)))
```

The corresponding denotational semantics of positional products in call-by-value FL! is presented in Figure 10.2. The *Value* domain is extended with a new summand, *Prod*, whose elements — sequences of values — represent product values. The evaluation of the subexpressions of a `product` expression is handled by *with-values*, and *nth* is used to extract the component at a given index in a `proj` expression. The validity of the index N is determined by the predicate

$$1 \le (\mathcal{N}\ N)\ \textbf{and}\ (\mathcal{N}\ N) \le (\textit{length}\ v^*),$$

which is known as a **bounds check**. If the bounds check fails, the `proj` expression denotes an error.

In many programming languages, the size of all products is known by the implementation, and a bounds check for every projection can be performed either at compile time or at run time. Important exceptions are C and C++, in which arrays carry no size information and bounds checks are not performed when array components are accessed. Programmers in these languages must pass array size information separately from the array itself and are expected to perform their own bounds checks. The lack of automatic bounds checks in C/C++ is the root cause of a high percentage of security flaws in modern software applications, many of which are due to so-called **buffer overrun** exploits that take advantage of C's permissiveness to fill memory with malicious code that can then be executed by a privileged process.

Product values with n components are often drawn as a box with n slots, sometimes with explicit indices. For example, the three-component product from above would be drawn as

false	*6*	*7*
1	2	3

$$\begin{array}{ccccc} & & \mathit{Prod} & = & \mathit{Value}^* \\ v & \in & \mathit{Value} & = & \ldots + \mathit{Prod} \end{array}$$

$$\mathcal{E}[\![\texttt{(product } E^*\texttt{)}]\!] = \lambda e\,.\ (\mathit{with\text{-}values}\ (\mathcal{E}^*[\![E^*]\!]\ e)\ \mathit{Prod} \mapsto \mathit{Value})$$

$$\begin{array}{l} \mathcal{E}[\![\texttt{(proj } N\ E_{prod}\texttt{)}]\!] = \\ \quad \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_{prod}]\!]\ e) \\ \qquad (\lambda v\,.\ \textbf{matching}\ v \\ \qquad\quad \begin{array}{ll} \triangleright\, (\mathit{Prod} \mapsto \mathit{Value}\ v^*)\ [\!] & \textbf{if}\ 1 \leq (\mathcal{N}\ N)\ \textbf{and}\ (\mathcal{N}\ N) \leq (\mathit{length}\ v^*) \\ & \textbf{then}\ (\mathit{val\text{-}to\text{-}comp}\ \ (\mathit{nth}\ \ (\mathcal{N}\ N)\ \ v^*)) \\ & \textbf{else}\ \mathit{error\text{-}comp} \\ & \textbf{fi} \\ \triangleright\, \textbf{else}\ \mathit{error\text{-}comp} & \\ \textbf{endmatching}\ ) & \end{array} \end{array}$$

Figure 10.2: Denotational semantics of immutable positional products in CBV FL!.

Such a diagram suggests a low-level implementation in which the n components of a product are stored as the contents of n successive addresses in the memory of the computer, something we shall explore in more detail in Chapter 17.

We emphasize that tuples in FL! are immutable: there is no way to change the value stored in a slot. We will consider mutable products later. In the following subsections, we discuss many possible variants to the products presented above.

10.1.1.2 Sequences

In the projection expression considered above, the index is an integer literal, not an integer expression. This means that the projection index cannot be calculated. One benefit of this restriction is that the bounds check can always be performed at compile time and so need never be performed at run time. As we discuss later (page 423), literal indices also facilitate reasoning about programs written in statically typed languages.

The positional products studied above do not include any way to dynamically determine the size of the product, i.e., the number of components. It is assumed that the programmer knows the size of every tuple when writing the program. However, there are many situations where it is necessary or convenient to determine the size of a product and to extract a product component at an index calculated from an expression. For instance, given an arbitrary product containing numbers, determining the average of these numbers requires knowing the number of components and looping through all indices of the product to find the sum of the components. Such capabilities are normally associated with products called **arrays**. But since arrays usually imply mutable structures, we will instead use the name **sequence** for immutable products with calculated indices and dynamically determinable sizes. This terminology is consistent with our use of the term "sequence" in our mathematical metalanguage.

We can extend FL! with immutable sequences by adding the following constructs to the language:

$E ::= \ldots$
  | `(sequence` E*`)` [Sequence Creation]
  | `(seq-proj` E_{index} E_{seq}`)` [Sequence Projection]
  | `(seq-size` E_{seq}`)` [Sequence Size]

The expression `(sequence` E_1 `...` E_n`)` creates and returns a size-n sequence whose components are the values of the expressions E_1 through E_n. The expression `(seq-proj` E_{index} E_{seq}`)` returns the ith component of the sequence denoted by E_{seq}, where i is the integer denoted by the arbitrary expression E_{index} (which must be checked against the bounds of the sequence). The expression `(seq-size` E_{seq}`)` returns the number of components in the sequence. The formal semantics of sequences is left as an exercise.

As an example of sequence manipulation, here is a procedure that finds the average of a sequence of numbers:

```
(define (average s)
  (letrec ((n (seq-size s))
           (sum-loop
             (lambda (i sum)
               (if (= i 0)
                   (/ sum n)
                   (sum-loop (- i 1)
                         (+ sum (seq-proj i s)))))))
    (sum-loop n 0)))
```

10.1.1.3 Product indexing

The positional products discussed above use **1-based indexing**, in which the components of an n-component tuple are accessed via the indices $1 \ldots n$. Many languages instead have **0-based indexing**, in which the slots are accessed via the indices $0 \ldots n-1$. For example:

false	*6*	*7*
0	1	2

Why use the index 0 to access the first slot of a product? One reason is that it can simplify some addressing calculations in the compiled code, which results in the execution of fewer low-level instructions when projecting components from products. Another reason is that 0-based indexing simplifies certain addressing calculations for the programmer. For example, compare the following expressions for accessing the slot in row i and column j of a conceptually 2-dimensional matrix `m` with width w and height h that is actually represented as a one-dimensional sequence with $w \times h$ components stored in so-called row-major order[2]:

```
; 0-based indexing of matrices and sequences
(nth (+ (* w i) j) m)

; 1-based indexing of matrices and sequences
(nth (+ (* w (- i 1)) j) m)
```

The 0-based approach is simpler because it does not require the subtraction by 1 seen in the 1-based approach.

Using 0 or 1 as the index for the first component are not the only choices. Some languages allow using any integer range for product indices. Some, such as PASCAL, even allow using as index ranges any range of values that is isomorphic to an integer range. For instance, a PASCAL array can be indexed by the alphabetic characters from `'p'` to `'u'` or the days of the week from `monday` through `friday` (where an enumeration of days has been declared elsewhere).

10.1.1.4 Types

FL is a **dynamically typed language** in which each value is conceptually tagged with its type and type errors are not detected until the program is run. In contrast, many modern languages are **statically typed languages**, in which the type of every expression is known when the program is compiled. The goal of static typing affects the design of positional products in these languages. In particular, it must be possible for the compiler to determine the type of every value projected from a product value.

For instance, ML and HASKELL support so-called **heterogeneous tuples** in which each tuple component may have a different type. In order to determine the type of a projection, both tuple indices and sizes must be statically determinable. ML's vectors, HASKELL's arrays, and CLU's sequences are examples of updatable immutable products in typed languages. Since in general compilers cannot determine either the size of or the index of a projection from such products, these products are **homogeneous sequences** in which all components are

[2] **Row-major** order means the elements of each row are stored in consecutive locations in the sequence. This makes accessing elements along a row very inexpensive. Likewise, **Column-major order** means elements of each column are stored in consecutive sequence locations.

required to have the same type. Many languages treat homogeneous sequences of characters, known as **strings**, as a special kind of positional product. Immutable strings appear in languages such as JAVA, ML, HASKELL and CLU, while C and SCHEME provide mutable strings.

Product indices are usually restricted to the integer type, but, as mentioned above, some languages allow index types that are isomorphic to the integers or some finite range of the integers. For example, the HASKELL language allows arrays to be indexed by any type that provides the operations of an "indexable" type. In PASCAL, arrays can be indexed by any range type that is isomorphic to a finite integer range. Oddly, the index range (not merely the index type) is part of the array type in PASCAL, which means that the size of every PASCAL array is statically known, and it is not possible to write procedures that are parameterized over arrays of different lengths.

In later chapters, we will have much more to say about product types when we study types in more detail.

10.1.1.5 Specialized syntax

Many languages provide specialized syntax for product manipulation. For instance, ML tuples are constructed by comma separated expressions delimited by parentheses, and the ith tuple component is extracted via the syntax #i. Here is an ML version of our earlier s-expression example:

```
let val p = (0 = 1, 2 * 3, 4 + 5)
 in if #1(p) then #2(p) else #3(p)
end
```

For sequences (as well as for mutable arrays) a subscripting notation using square brackets is a standard way to project components, and `:=` might be used for an update operation. Below is a way to swap the components at indices `i` and `j` of an immutable vector `u` in a hypothetical version of ML extended with this specialized syntax:

```
let val v = u[i]
    val u2 = u[i] := u[j]
    val u3 = u2[j] := v
 in u3
end
```

▷ **Exercise 10.1** In the first three parts below, assume 1-based indexing.

a. Give an operational semantics for sequences in call-by-value FL.

b. Give a denotational semantics for sequences in call-by-value FL.

c. Explicitly enumerating the elements of a sequence in a `sequence` expression can be inconvenient. For example, a sequence of the squares of the integers from 1 to 5 would be written:

```
(sequence (* 1 1) (* 2 2) (* 3 3) (* 4 4) (* 5 5))
```

An alternative means of specifying such sequences is via a new construct

(`tabulate` E_{size} E_{proc})

where E_{size} denotes the size of the sequence and E_{proc} denotes a unary procedure f that maps the index i to the value $(f\ \ i)$. For instance, using `tabulate`, the above 5-element sequence could be written

```
(tabulate 5 (lambda (i) (* i i)))
```

Given an operational and denotational semantics of `tabulate` in call-by-value FL.

d. What changes would need to be made to the above three parts to specify 0-based indexing rather than one-based indexing?

e. What changes would need to be made to the syntax for sequences and the first three parts to specify an indexing scheme that starts at an arbitrary dynamically determinable value *lo* rather than 0 or 1? ◁

▷ **Exercise 10.2** Even with immutable products, it is still useful to provide a facility for updating elements in, inserting elements into, and removing elements from a product. Since the product is immutable, none of these operations actually change a given product value, but they return a new product value that shares most of its components with the given product value. We shall call sequences that support one or more of these operations **updatable sequences**, though this is by no means a standard term.

Consider the following constructs for one form of updatable sequence:

E ::= . . .
| (`usequence` E*) [Updatable Sequence Creation]
| (`useq-proj` E_{index} E_{useq}) [Updatable Sequence Projection]
| (`useq-size` E_{useq}) [Updatable Sequence Size]
| (`useq-update` E_{index} E_{val} E_{useq}) [Updatable Sequence Update]
| (`useq-insert` E_{index} E_{val} E_{useq}) [Updatable Sequence Insertion]
| (`useq-delete` E_{index} E_{useq}) [Updatable Sequence Deletion]

The `usequence`, `useq-proj`, and `useq-size` are the updatable sequence versions of the corresponding (non-updatable) sequence constructs. For `useq-update`, `useq-insert`, and `useq-delete`, suppose that E_{index} denotes an integer i, E_{val} denotes a value v_{new}, E_{useq} denotes a size-n updatable sequence v_{useq}, and `u` denotes the sequence with integer values `[7,5,8]`. If v is an updateable sequence, let $\#v$ denote the size of v and $v \downarrow j$ denote the jth component value of v, where $1 \leq j \leq \#v$. Then:

- if $1 \leq i \leq n$, (`useq-update` E_{index} E_{val} E_{useq}) returns a size-n updatable sequence v_{useq2} such that

$v_{useq2} \downarrow i = v_{new}$; and

$v_{useq2} \downarrow j = v_{useq} \downarrow j$ for all $1 \leq j \leq n$ where $j \neq i$.

For example, `(useq-update 2 6 u)` returns the updatable sequence `[7,6,8]`.

- if $1 \leq i \leq n{+}1$, (`useq-insert` E_{index} E_{val} E_{useq}) returns a size-$n{+}1$ updatable sequence v_{useq2} such that

  $v_{useq2} \downarrow j = v_{useq} \downarrow j$ for all $1 \leq j < i$;

  $v_{useq2} \downarrow i = v_{new}$; and

  $v_{useq2} \downarrow k = v_{useq} \downarrow k - 1$ for all $i < k \leq n + 1$;

  For example, `(useq-insert 2 6 u)` returns the updatable sequence `[7,6,5,8]`.

- if $1 \leq i \leq n$, (`useq-delete` E_{index} E_{useq}) returns a size-$n{-}1$ updatable sequence v_{useq2} such that

  $v_{useq2} \downarrow j = v_{useq} \downarrow j$ for all $1 \leq j < i$; and

  $v_{useq2} \downarrow k = v_{useq} \downarrow k + 1$ for all $i \leq k \leq n - 1$;

  For example, `(useq-delete 2 u)` returns the updatable sequence `[7,8]`

a. Give an operational semantics for updatable sequences in call-by-value FL.

b. Give a denotational semantics for updatable sequences in call-by-value FL.

c. Show that `useq-update` is not strictly necessary in a language with updatable sequences because it can be desugared into other constructs.

d. Consider a language with updatable sequences that also has a `(useq-empty)` construct that returns an empty updatable sequence. Show that `usequence` is not strictly necessary in such a language because it can be desugared into other constructs. What are the benefits and drawbacks of such a desguaring? ◁

10.1.2 Named Products

In a **named product**, components are indexed by names rather than by positions. In Section 7.2, we introduced the **record**, a classic form of named product, and studied its semantics. We saw that records were effectively reified environments. Here we discuss some of the dimensions of named products.

The simplest form of named product is a named version of positional products with a product creator (`record`) and a product projector (`select`):

E ::= ...
  | (`record` (I E)*) [Record Creation]
  | (`select` I E) [Record Projection]

As above, we assume that such constructs are embedded in a call-by-value language and denote immutable products.

As a simple example of records, consider the following expression, which evaluates to *9*:

```
(let ((r (record (test (= 0 1)) (yes (* 2 3)) (no (+ 4 5)))))
  (if (select test r) (select yes r) (select no r)))
```

In named products, the order of bindings in the record constructor is irrelevant, so the value of the above expression would not be changed if the `record` subexpression were changed to be

```
(record ((no (+ 4 5)) (test (= 0 1)) (yes (* 2 3))))
```

Many languages with named products have special syntax for record creation and projection. For instance, here is our running example expressed in ML record syntax:

```
let val r = {test = (0=1), yes=2*3, no=4+5}
 in if #test(r) then #yes(r) else #no(r)
end
```

A more common syntax for record selection is the "dot notation" used with Pascal records, C structures, and Java objects, as in:

```
if r.test then r.yes else r.no
```

In a language like ML that permits numeric record labels, positional products can be viewed as syntactic sugar for named products. E.g., the ML tuple `(true, 17)` is syntactic sugar for `{1=true, 2=17}`.

Simple records can be augmented with operations that parallel many of the extensions for positional products:

- `(record-size` E_{rcd}`)`: Returns the number of components in a record.

- `(record-insert` I E_{val} E_{rcd}`)`: Let v_{rcd} be the record denoted by E_{rcd}. Then the `record-insert` expression returns a new record that has a binding of I to the value of E_{val} in addition to all the bindings of v_{rcd}. If v_{rcd} already has a binding for I, the new binding overrides it. With named products, `record-insert` corresponds to both `seq-insert` and `seq-update` for positional products.

- `(record-delete` I E_{rcd}`)`: Let v_{rcd} be the record denoted by E_{rcd}. Then the `record-delete` expression returns a new record that has all the bindings of v_{rcd} except for any with the name I.

The `override` construct from Section 7.2 is a generalization of `record-insert` that combines two environments, while the `conceal` construct presented there is a generalization of `record-delete`. Other forms of record combination and name manipulation are also possible. For instance, it is possible to take the "intersection" or "difference" of two environments, or to specify the names that should be kept in a record rather than those that should be concealed.

It is even possible, but rare, to have a named index that can be calculated. In FL, such a construct might have the form (`select-sym` E_{sym} E_{rcd}), where E_{sym} is an expression denoting a symbol value v_{sym} and `select-sym` selects from the record denoted by E_{rcd} the value associated with the label that is the underlying identifier of I_{sym}. It would be hard to imagine such a construct in a statically typed language. However, this idiom is often used in dynamically typed languages (such as Lisp dialects) in the form of **association lists**, which are list of bindings between explicit symbols and values.

10.1.3 Non-strict Products

Our discussion so far has focused on **strict products**, in which the expressions specifying the product components are fully evaluated into values that are stored within the resulting product value. Another option is to have **non-strict products**, in which the component computations themselves are stored within the product value and are only run when their values are "demanded." Such products are the default in non-strict languages like Haskell, but we will see that there are considerable benefits to integrating non-strict products into a call-by-value language, which is the focus of this section.

A simple approach to non-strict products is to adapt the call-by-name parameter passing mechanism to product formation. We will call the resulting data **call-by-name (CBN) products** in contrast to the **call-by-value (CBV) products** we have studied so far. An operational and denotational semantics for immutable positional CBN products in a call-by-value version of FL! is presented in Figure 10.3. We use the names `nproduct`/ `nproj` instead of `product`/`proj` to syntactically distinguish CBN products from CBV products. In the operational semantics, the delayed computation of product components is modeled by *not* having any progress rules for evaluating the component expressions of an `nproduct` expression. In the denotational semantics, a product value is represented as a sequence of computations rather than as a sequence of values. Intuitively, these computations are only "forced" into values upon projection from the CBN product by the occurrences of *with-value* that are sprinkled throughout the rest of the denotational semantics for CBV FL!.

As a simple example of how CBN products differ from CBV products, con-

Operational semantics for CBN products

$$V \in \text{ValueExp} = \ldots \cup \{(\texttt{nproduct}\ E_1\ \ldots\ E_n)\}$$

$$\langle(\texttt{nproj}\ N\ (\texttt{nproduct}\ E_1\ \ldots\ E_n)), S\rangle \Rightarrow \langle E_N, S\rangle, \quad \text{where } 1 \le N \le \text{n} \qquad [\textit{nproduct projection}]$$

Denotational semantics for CBN products

$$\begin{array}{rrcl} & \textit{NProd} & = & \textit{Computation}^* \\ v \in & \textit{Value} & = & \ldots + \textit{NProd} \end{array}$$

$$\mathcal{E}[\![(\texttt{nproduct}\ E^*)]\!] = \lambda e\,.\,(\textit{NProd} \mapsto \textit{Value}\ (\mathcal{E}^*[\![E^*]\!]\ e))$$

```
$\mathcal{E}[\![(\texttt{nproj}\ N\ E_{prod})]\!]$ =
  $\lambda e\,.$ $\textit{with-value}$ $(\mathcal{E}[\![E_{prod}]\!]\ e)$
      $(\lambda v\,.$ matching $v$
          $\triangleright (\textit{NProd} \mapsto \textit{Value}\ c^*)$ $]\!]$ if $1 \le (\mathcal{N}\ N)$ and $(\mathcal{N}\ N) \le (\textit{length}\ c^*)$
                                    then $(\textit{nth}\ (\mathcal{N}\ N)\ c^*)$
                                    else $\textit{error-comp}$
                                    fi
          $\triangleright$ else $\textit{error-comp}$
          endmatching )
```

Figure 10.3: Operational and denotational semantics for CBN positional products in call-by-name FL!.

sider the following expression:

```
(let ((c (cell 5)))
  (let ((p (nproduct (begin (:= c (+ (^ c) 1)) (^ c))
                     (begin (:= c (* (^ c) 2)) (^ c)))))
    (list (nproj 2 p) (nproj 1 p) (nproj 1 p) (nproj 2 p))))
```

The value of this expression is $[10, 11, 12, 24]$, indicating that the increments and doublings of the argument expressions are performed at every projection rather than when the CBN product is formed. If we had instead used CBV products, the above expression would yield $[6, 12, 12, 6]$, indicating that the side effects of the argument expressions are performed exactly once when the product is created.

In CBN products, the component computation is re-evaluated at every projection. Another option, inspired by the call-by-need parameter passing mechanism, is to evaluate the component computation at the very first projection and memoize the resulting value for later projections. We shall call this form of non-strict product a **lazy (CBL) product**. Using a lazy product in the above example would yield the list $[10, 11, 11, 10]$, which indicates that the side effects are performed on the first projections but not on subsequent projections.

The operational semantics of lazy products is presented in Figure 10.4. We use the names `lproduct` and `lproj` to distinguish lazy products from CBV and CBN products. A lazy product value is a sequence of locations in the store that may contain non-value expressions. At the first projection of a lazy product component, the [*lproj progress*] rule forces the evaluation of an unevaluated component expression to a value that is returned by the [*lproduct projection*] rule. Because the resulting value is "remembered" in the component location, subsequent projections of the component will return the value directly.

In the denotational semantics for lazy products (Figure 10.5), this memoizing behavior is modeled by extending *Storable* to be *Memo*,[3] which includes both values and computations. For a CBV language, we modify the *allocating* function to inject the initial value for a location in *Memo*, and introduce *allocatingComp* and *allocatingComps* for storing computations in freshly allocated locations. We modify *fetching* so that whenever the contents of a location is fetched, any computation stored at that location is evaluated to a value that is memoized at that location. A lazy product itself is modeled as a sequence of locations holding elements of *Memo*.

[3] This domain implies that computations could be stored at any location (such as cell locations), but in fact they can only be stored in lazy product locations. A practical implementation of lazy products would localize the overhead of memoization to lazy product component locations so that the efficiency of manipulating cell locations was not affected.

$$
\begin{array}{lcll}
S & \in & \text{Store} & = \text{Assignment*} \\
Z & \in & \text{Assignment} & = \text{Location} \times \text{Exp} \\
V & \in & \text{ValueExp} & = \ldots \cup \{\texttt{(lproduct } L_1 \ \texttt{...} \ L_n\texttt{)}\}
\end{array}
$$

$get : \text{Location} \rightarrow \text{Store} \rightharpoonup \text{Exp}$

$$
\begin{array}{c}
\langle \texttt{(lproduct } E_1 \ \texttt{...} \ E_n\texttt{)}, S\rangle \\
\Rightarrow \langle \texttt{(lproduct } L_1 \ \texttt{...} \ L_n\texttt{)}, [\langle L_1, E_1\rangle, \ldots, \langle L_n, E_n\rangle] \ @ \ S\rangle, \\
\text{where } L_1 \ \ldots \ L_n \text{ are fresh locations not appearing in } S.
\end{array}
\qquad [\textit{lproduct creation}]
$$

$$
\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\begin{array}{c}
\langle \texttt{(lproj } N \ \texttt{(lproduct } L_1 \ \texttt{...} \ L_n\texttt{))}, S\rangle \\
\Rightarrow \langle \texttt{(lproj } N \ \texttt{(lproduct } L_1 \ \texttt{...} \ L_n\texttt{))}, (\langle L_N, E'\rangle \ \texttt{.} \ S')\rangle, \\
\text{where } 1 \leq N \leq \text{n and } (get \ L_N \ S) = E
\end{array}}
\qquad [\textit{lproj progress}]
$$

$$
\begin{array}{c}
\langle \texttt{(lproj } N \ \texttt{(lproduct } L_1 \ \texttt{...} \ L_n\texttt{))}, S\rangle \Rightarrow \langle V, S\rangle, \\
\text{where } 1 \leq N \leq \text{n and } (get \ L_N \ S) = V
\end{array}
\qquad [\textit{lproduct projection}]
$$

Figure 10.4: Operational semantics for CBL products.

Non-strict products may be added to stateless languages like FL. We have chosen to focus on the stateful language FL! for two reasons:

1. It is easier to demonstrate the differences between the three forms of products in a language with state. In FL, only termination and errors could be used to distinguish strict and non-strict products, and CBN and CBL products are observationally indistinguishable.

2. Explaining the memoization of CBL products requires some form of state, so for presentational purposes it is easier to add these to a language like FL! that already has state.

The main benefit of non-strict products is that they enable the creation of conceptually infinite data structures that improve program modularity. For instance, we can introduce infinite lists, sometimes called **streams**, into a CBV language with the following sugar for `scons` (stream `cons`):

$$\mathcal{D}_{\text{exp}}[\![\texttt{(scons } E_1 \ E_2\texttt{)}]\!] = \texttt{(lproduct } E_1 \ E_2\texttt{)}$$

along with the following procedures:

```
(define (scar x) (lproj 1 x))
(define (scdr x) (lproj 2 x))
```

$$
\begin{array}{rclcl}
mm & \in & \mathit{Memo} & = & \mathit{Computation} + \mathit{Value} \\
\sigma & \in & \mathit{Storable} & = & \mathit{Memo} \\
 & & \mathit{LProd} & = & \mathit{Location}^* \\
v & \in & \mathit{Value} & = & \ldots + \mathit{LProd}
\end{array}
$$

```
$allocating : Value \rightarrow (Location \rightarrow Computation) \rightarrow Computation$
$= \lambda vf\,.\ \lambda s\,.\ (f\ \ (fresh\text{-}loc\ \ s)\ \ (assign\ \ (fresh\text{-}loc\ \ s)\ \ (Value \mapsto Memo\ \ v)\ \ s))$

$allocatingComp : Computation \rightarrow (Location \rightarrow Computation) \rightarrow Computation$
$= \lambda cf\,.\ \lambda s\,.\ (f\ \ (fresh\text{-}loc\ \ s)\ \ (assign\ \ (fresh\text{-}loc\ \ s)\ \ (Computation \mapsto Memo\ \ c)\ \ s))$

$allocatingComps : Computation^* \rightarrow (Location^* \rightarrow Computation) \rightarrow Computation$
$= \lambda c^* f\,.$ ($\mathbf{matching}$ $c^*$
              $\triangleright [\,]_{Computation}\ \rrbracket\ (f\ \ [\,]_{Location})$
              $\triangleright (c\ .\ c^*)\ \rrbracket\ allocatingComp\ \ c\ \ (\lambda l\,.\,allocatingComps\ \ c^*\ \ (\lambda l^*\,.\,f\ \ (l\ .\ l^*)))$
              $\mathbf{endmatching}$ )

$fetching : Location \rightarrow (Value \rightarrow Computation) \rightarrow Computation$
$= \lambda lf\,.\ \lambda s\,.$ $\mathbf{matching}$ $(fetch\ \ l\ \ s)$
                 $\triangleright (Storable \mapsto Assignment\ \ mm)\ \rrbracket$
                   $\mathbf{matching}$ $mm$
                   $\triangleright (Value \mapsto Memo\ \ v)\ \rrbracket\ f\ \ v\ \ s$
                   $\triangleright (Computation \mapsto Memo\ \ c)\ \rrbracket$
                     $with\text{-}value\ \ c\ (\lambda vs'\,.\ f\ \ v\ \ (assign\ l\ (Value \mapsto Memo\ \ v)\ \ s'))$
                   $\mathbf{endmatching}$
                 $\triangleright$ $\mathbf{else}$ $(error\text{-}comp\ \ s)$
                 $\mathbf{endmatching}$

$\mathcal{E}[\![\texttt{(lproduct}\ E^*\texttt{)}]\!] = \lambda e\,.\ (allocatingComps\ \ (\mathcal{E}^*[\![E^*]\!]\ \ e)\ \ (\lambda l^*\,.\,(LProd \mapsto Value\ \ l^*)))$

$\mathcal{E}[\![\texttt{(lproj}\ N\ E_{prod}\texttt{)}]\!] =$
  $\lambda e\,.\ (with\text{-}value\ (\mathcal{E}[\![E_{prod}]\!]\ \ e)$
          $(\lambda v\,.$ ($\mathbf{matching}$ $v$
                 $\triangleright (LProd \mapsto Value\ \ l^*)\ \rrbracket$ $\mathbf{if}$ $1 \le (\mathcal{N}\ \ N)$ $\mathbf{and}$ $(\mathcal{N}\ \ N) \le (length\ \ l^*)$
                                     $\mathbf{then}$ $(fetching\ \ (nth\ \ (\mathcal{N}\ \ N)\ \ l^*)\ \ val\text{-}to\text{-}comp)$
                                     $\mathbf{else}$ $error\text{-}comp$
                                     $\mathbf{fi}$
                 $\triangleright$ $\mathbf{else}$ $error\text{-}comp$
                 $\mathbf{endmatching}$ )))
```

Figure 10.5: Denotational semantics for CBL products in FL!.

The stream of all natural numbers can be created via `(ints-from 0)`, where the `ints-from` procedure is defined as

```
(define (ints-from n) (scons n (ints-from (+ n 1))))
```

The fact that the evaluation of the component expression `(ints-from (+ n 1))` is delayed until it is accessed prevents what would otherwise be an infinite recursion if `cons` were used instead of `scons`.

To view a prefix of a stream as a regular list, we will use the following procedure:

```
(define (prefix n str)
  (if (= n 0)
      (list)
      (cons (scar str) (prefix (- n 1) (scdr str)))))
```

For example:

`(prefix 5 (ints-from 3))` $\xrightarrow[FL]{}$ $[\mathit{3},\ \mathit{4},\ \mathit{5},\ \mathit{6},\ \mathit{7}]$

The stream mapping and filtering procedures in Figure 10.6 are handy for creating streams, such as the examples in Figure 10.7. Note how laziness enables the streams `nats`, `twos`, and `fibs` to all be defined directly in terms of themselves, without the need for an explicit recursive generating function like `ints-from`. The stream of prime numbers, `primes`, is calculated using the sieve of Eratosthenes method, which begins at 2 and keeps as primes only those following integers that are not multiples of previous primes. It is worth emphasizing that all of these examples could be implemented using regular lists (manipulated via `cons`, `car`, and `cdr`) in a call-by-name language or a call-by-need language; special lazy products are only necessary in a call-by-value language.

As an example of the modularity benefits of the conceptually infinite data structures enabled by non-strict products, consider the `first-bigger-than` procedure, which returns the first value in a numeric stream that is strictly bigger than a given threshhold `n`.

```
(define (first-bigger-than n str)
  (if (> (scar str) n)
      (scar str)
      (first-bigger-than n (scdr str))))
```

`(first-bigger-than 1000 nats)` $\xrightarrow[FL!]{}$ $\mathit{1001}$
`(first-bigger-than 1000 evens)` $\xrightarrow[FL!]{}$ $\mathit{1002}$
`(first-bigger-than 1000 twos)` $\xrightarrow[FL!]{}$ $\mathit{1024}$
`(first-bigger-than 1000 fibs)` $\xrightarrow[FL!]{}$ $\mathit{1597}$
`(first-bigger-than 1000 primes)` $\xrightarrow[FL!]{}$ $\mathit{1009}$

```
; Applies a unary function F elementwise to stream STR.
(define (smap f str)
  (scons (f (scar str))
         (smap f (scdr str))))

; Applies a binary function G elementwise to corresponding
; elements of STR1 and STR2.
(define (smap2 g str1 str2)
  (scons (g (scar str) (scar str2))
         (smap2 g (scdr str1) (scdr str2))))

; Returns a stream with only those elements of STR
; satisfying the predicate PRED.
(define (sfilter pred str)
  (if (pred (scar str))
      (scons (scar str) (sfilter pred (scdr str)))
      (sfilter pred (scdr str))
```

Figure 10.6: Mapping and filtering procedures for streams.

Infinite lists allow a list processing termination condition to be specified in the consumer of a list rather than in the producer of a list. With strict lists, all lists must be finite, so the termination condition must be specified when the list is produced. To get the behavior of `first-bigger-than` with strict lists, it would be necessary to intertwine the details of generating the next element with checking it against the threshhold – a strategy that would compromise the modularity of having a separate `first-bigger-than` procedure.

▷ **Exercise 10.3** The **Hamming numbers** are all positive integers whose non-trivial factors are 2, 3, and 5 exclusively. Define a stream of the Hamming numbers. What is the first Hamming number strictly larger than 1000? ◁

▷ **Exercise 10.4** Many SCHEME implementations support a form of stream created out of pairs where the second component is lazy but the first is not:

$\mathcal{D}_{\text{exp}}[\![$`(cons-stream` E_1 E_2`)`$]\!]$ = `(cons` E_1 `(delay` E_2`))`

```
(define (head str) (car str))
(define (tail str) (force (cdr str)))
```

Here, `delay` and `force` implement a memoized delayed value, like `lazy` and `touch` did in Exercise 7.1.

a. Show that it is possible to define all lazy lists illustrated in this section as SCHEME streams.

```
; All natural numbers
(define nats (scons 0 (smap (+ 1) nats))
```

`(prefix 5 nats)` $\xrightarrow[FL]{} [\mathit{0},\ \mathit{1},\ \mathit{2},\ \mathit{3},\ \mathit{4}]$

```
; All even natural numbers
(define evens (sfilter (lambda (x) (= (rem x 2) 0)) nats))
```

`(prefix 5 evens)` $\xrightarrow[FL]{} [\mathit{0},\ \mathit{2},\ \mathit{4},\ \mathit{6},\ \mathit{8}]$

```
; All powers of two
(define twos (scons 1 (smap (* 2) twos)))
```

`(prefix 5 twos)` $\xrightarrow[FL]{} [\mathit{1},\ \mathit{2},\ \mathit{4},\ \mathit{8},\ \mathit{16}]$

```
; All Fibonacci numbers
(define fibs (scons 0 (scons 1 (smap2 + fibs (scdr fibs)))))
```

`(prefix 10 fibs)` $\xrightarrow[FL]{} [\mathit{0},\ \mathit{1},\ \mathit{1},\ \mathit{2},\ \mathit{3},\ \mathit{5},\ \mathit{8},\ \mathit{13},\ \mathit{21},\ \mathit{34}]$

```
; All prime numbers
(define primes
  (letrec
    ((sieve
       (lambda (str)
         (scons (scar str)
                (sieve (sfilter (lambda (x)
                                  (not (= (rem x (scar str)) 0)))
                                (scdr str)))))))
    (sieve (ints-from 2))))
```

`(prefix 10 primes)` $\xrightarrow[FL]{} [\mathit{2},\ \mathit{3},\ \mathit{5},\ \mathit{7},\ \mathit{11},\ \mathit{13},\ \mathit{17},\ \mathit{19},\ \mathit{23}]$

Figure 10.7: Some sample streams of numbers.

b. Design a stream in which laziness in the first component is essential – that is, which can be defined via `scons/scar/scdr` but not via `cons-stream/head/tail`. ◁

▷ **Exercise 10.5**

a. Use `lproduct/lproj` to define constructors and selectors for infinite binary trees in which each node holds a value in addition to its left and right subtrees.

b. Use your constructs to define an infinite binary tree whose left-to-right inorder traversal yields the positive integers in order of magnitude.

c. Define an `inorder-stream` procedure that returns a stream of the elements of an infinite binary tree as they would be encountered in a left-to-right inorder traversal. ◁

10.1.4 Mutable Products

Thus far we have discussed only immutable products — those whose components do not change over time. But in popular imperative languages, the vast majority of built-in data structures are mutable products. Here we explore some design dimensions of mutable products and some examples of mutable products in real languages.

All of the dimensions we explored above for immutable products are relevant to mutable products. For example, mutable product components are either named or positional. Examples of mutable products with named components include C's structures and PASCAL's records. A canonical example of a fixed-size mutable product with positional components is SCHEME's pairs, whose two components may be altered via `set-car!` and `set-cdr!`. Mutable sequences are typically called arrays (as in C/C++, JAVA, PASCAL, FORTRAN, and CLU) or vectors (as in SCHEME and JAVA). All of these support the ability to update the component at any index, often via a special subscripting notation, such as `a[i] = 2*a[i];` in C/C++/JAVA. Only some of these — CLU's arrays and JAVA's vectors (but not JAVA's arrays) — support the ability to expand or contract the size of the mutable sequence by inserting or removing elements. All of these examples of mutable products have 0-based indexing except for FORTRAN (which has 1-based arrays), CLU (whose arrays can have any lower bound but are 1-based by default), and PASCAL (whose arrays support arbitrary enumerations as indices). In all of these examples, all components are required to be of the same type, except for SCHEME's vectors (where any slot may contain any value) and JAVA's vectors (where any slot may contain any object).

Although the mutable products mentioned above seem similar on the surface, their semantics differ in fundamental ways. Below we explore some of the dimensions along which mutable products can differ. For simplicity, we consider only mutable fixed-length positional products of heterogeneous values, which we shall call **mutable tuples**. It is easy to generalize these to other kinds of mutable products. We will study the addition of mutable tuples to FL!. We assume a CBV parameter passing mechanism unless otherwise stated. Here are the constructs we will consider:

E ::= . . .
  | `(mprod` E^*`)` [Mutable Tuple Creation]
  | `(mget` N_{index} E_{mt}`)` [Mutable Tuple Projection]
  | `(mset!` N_{index} E_{mt} E_{new}`)` [Mutable Tuple Assignment]

Informally, these constructs have the following semantics:

- `(mprod` E_1 `...` E_n`)` creates a new mutable tuple with n mutable slots indexed from 1 to n where slot i is initially filled with the value of E_i.

- In `(mget` N_{index} E_{mt}`)`, assume that E_{mt} evaluates to a mutable tuple mt with n slots, where $1 \leq N_{index} \leq n$. Then `mget` returns the value in the ith slot of mt. Otherwise, `mget` signals an error.

- In `(mset!` N_{index} E_{mt} E_{new}`)`, assume that E_{mt} evaluates to a mutable tuple mt with n slots, where $1 \leq N \leq n$, and E_{new} evaluates to v. Then `mset!` changes the value in the ith slot of mt to be v. Otherwise, `mset!` signals an error.

For example, here is an expression involving a mutable tuple:

```
(let ((m (mprod 3 4)))
  (begin
    (mset! 1 m (+ (mget 1 m) (mget 2 m))) ; 1st slot is now 7.
    (mset! 2 m (+ (mget 1 m) (mget 2 m))) ; 2nd slot is now 11.
    (* (mget 1 m) (mget 2 m)))) $\xrightarrow[FL!]{}$ 77
```

A very simple way to include mutable products in a language is to have a single kind of mutable entity — such as a mutable cell — and allow this entity to be a component of otherwise immutable structures. This is the approach taken in ML, where immutable tuples, vectors, and user-defined datatypes may have mutable cells as components. We can model this approach in FL! via the following desugarings for `mprod`, `mget`, and `mset!`:

$$\mathcal{D}[\![\texttt{(mprod } E_l \ \ldots E_n\texttt{)}]\!] = \texttt{(product (cell } \mathcal{D}[\![E_l]\!]\texttt{) ... (cell } \mathcal{D}[\![E_n]\!]\texttt{))}$$
$$\mathcal{D}[\![\texttt{(mget } N \ E_{mp}\texttt{)}]\!] = \texttt{(cell-ref (proj } N \ \mathcal{D}[\![E_{mp}]\!]\texttt{))}$$
$$\mathcal{D}[\![\texttt{(mset! } N \ E_{mp} \ E_{new}\texttt{)}]\!] = \texttt{(cell-set! (proj } N \ \mathcal{D}[\![E_{mp}]\!]\texttt{) } \mathcal{D}[\![E_{new}]\!]\texttt{)}$$

In typical imperative languages, a more common design is to directly support various kinds of mutable products, perhaps along with some immutable ones. The CLU language, for example, supports a variety of different built-in datatypes, each of which comes in both mutable and immutable flavors.

In most imperative languages, mutable products would be modeled as a sequence of locations, as shown in the denotational semantics presented in Figure 10.8, which is a straightforward generalization to the semantics of mutable cells. Mutable tuple values are represented as sequences of locations. This is similar to the representation of lazy products, except that in `mprod`, the computed values of the subexpressions (rather than the computatations for these subexpressions) are stored in the locations.

A key issue in the semantics of mutable products is how they are passed as parameters. When mutable products are added as values to the CBV version of FL! we have studied, we shall say that the they are passed via a **call-by-value-sharing (CBVS)** mechanism because both the caller and the callee share access to the same locations in the mutable product. For example, in the following expression, references to `t` and `m` in the body of the procedure `f` refer to the same mutable product, so that changes to the components of one are visible in the other:

```
(let ((t (mprod 5 6)))
  (let ((f (lambda (m)
             (begin
               (mset! 1 t (* 10 (mget 1 t)))
               (mset! 2 m (* 100 (mget 2 m)))
               (mget 1 m)))))
    (+ (f t) (mget 2 t))))
```

$$\xrightarrow[CBVS\ \ FL!]{} \mathit{650}$$

This is the behavior expected for mutable products in languages such as Java, Scheme and CLU. Conceptually, when a mutable product is assigned to a variable, passed as a parameter, returned as a result, or stored in a data structure, no new product locations are created; the existing product locations are simply shared in all parts of the program to which the given product value has "flowed."

An alternative strategy for passing mutable products in a CBV language is to create a new product with new locations whenever a product is passed from one part of a program to another. This approach, which we shall term **call-by-value-copy (CBVC)** is explained by the denotational semantics for a variant of FL! with mutable products (Figure 10.9). Whenever a value is passed, a copy of the value is made. Primitive values, procedures, and locations (i.e., cells) are not copied, but a mutable tuple with n slots is copied by allocating n new locations and filling these with copies of the contents of the existing locations.

```
  $v \in \mathit{Value} = \ldots + \mathit{MProd}$
$\mathit{mt} \in \mathit{MProd} = \mathit{Location}*$

$\mathit{allocatingVals} : \mathit{Value}* \to (\mathit{Location}* \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda v{*}f\ .$ matching $v*$
            $\triangleright\ [\,]_{\mathit{Value}}\ [\![\ (f\ \ [\,]_{\mathit{Location}})$
            $\triangleright\ (v\ .\ v*)\ [\![\ (\mathit{allocating}\ v\ (\lambda l\ .\ \mathit{allocatingVals}\ v*\ (\lambda l*\ .\ f\ (l\ .\ l*))))$
            endmatching

$\mathcal{E}[\![$(mprod $E*$)$]\!]$
$= \lambda e\ .\ (\mathit{with\text{-}values}\ \ (\mathcal{E}*[\![E_l]\!]\ e)$
             $(\ \lambda v*\ .\ \ (\mathit{allocatingVals}\ v*\ (\lambda l*\ .\ (\mathit{MProd} \mapsto \mathit{Value}\ \ l*))))$

$\mathcal{E}[\![$(mget $N\ E_{mp}$)$]\!]$
$= \lambda e\ .\ (\mathit{with\text{-}value}\ (\mathcal{E}[\![E_{mp}]\!]\ e)$
             $(\lambda v_{mp}\ .$ matching $v_{mp}$
                        $\triangleright\ (\mathit{MProd} \mapsto \mathit{Value}\ \ l*)\ [\![$
                          if $1 \leq (\mathcal{N}\ \ N)$ and $(\mathcal{N}\ \ N) \leq (\mathit{length}\ \ l*)$
                           then $(\mathit{fetching}\ \ (\mathit{nth}\ \ (\mathcal{N}\ \ N)\ \ l*)\ \ (\lambda v\ .\ (\mathit{val\text{-}to\text{-}comp}\ \ v)))$
                           else $\mathit{error\text{-}comp}$
                           fi
                        $\triangleright$ else $\mathit{error\text{-}comp}$
                        endmatching ))

$\mathcal{E}[\![$(mset! $N\ E_{mp}\ \ E_{new}$)$]\!]$
$= \lambda e\ .\ (\mathit{with\text{-}value}\ (\mathcal{E}[\![E_{mp}]\!]\ e)$
             $(\lambda v_{mp}\ .\ \ (\mathit{with\text{-}value}\ (\mathcal{E}[\![E_{new}]\!]\ e)$
                          $(\lambda v_{new}\ .$ matching $v_{mp}$
                                      $\triangleright\ (\mathit{MProd} \mapsto \mathit{Value}\ \ l*)\ [\![$
                                        if $1 \leq (\mathcal{N}\ \ N)$ and $(\mathcal{N}\ \ N) \leq (\mathit{length}\ \ l*)$
                                         then $(\mathit{update}\ \ (\mathit{nth}\ \ (\mathcal{N}\ \ N)\ \ l*)\ \ v_{new})$
                                         else $\mathit{error\text{-}comp}$
                                         fi
                                      $\triangleright$ else $\mathit{error\text{-}comp}$
                                      endmatching ))))
```

Figure 10.8: Denotational semantics of mutable tuples with CBVS parameter passing.

In a CBVC interpretation of the example expression considered for CBVS, the names t and m refer to two distinct mutable tuples, so that changes to one pair are not visible in the other:

```
(let ((t (mprod 5 6)))
  (let ((f (lambda (m)
             (begin
               (mset! 1 t (* 10 (mget 1 t)))
               (mset! 2 m (* 100 (mget 2 m)))
               (mget 1 m)))))
    (+ (f t) (mget 2 t))))
```

$$\xrightarrow[\mathit{CBVC\ \ FL!}]{} \mathit{11}$$

$$v \in \mathit{Value} = \mathit{Unit} + \mathit{Bool} + \mathit{Int} + \mathit{Sym} + \mathit{Procedure} + \mathit{Location} + \mathit{MProd}$$

$$\begin{array}{l}
\mathit{allocatingCopies} : \mathit{Location}^* \to (\mathit{Value} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}) \\
\qquad\qquad\qquad \to (\mathit{Location}^* \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda l^* g f \,.\ \textbf{matching}\ l^* \\
\qquad\quad \triangleright [\,]_{\mathit{Location}} \ \| \ f\ [\,]_{\mathit{Location}} \\
\qquad\quad \triangleright (l_{old}\ .\ l_{old}{}^*) \ \| \ \mathit{fetching}\ l_{old} \\
\qquad\qquad\qquad\qquad (\lambda v \,.\ g\ v\ (\lambda v' \,.\ \mathit{allocating}\ v' \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda l_{new} \,.\ \mathit{allocatingCopies}\ l_{old}{}^* \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda l_{new}{}^* \,.\ f\ (l_{new}\ .\ l_{new}{}^*)))))
\end{array}$$

$$\begin{array}{l}
\mathit{deepCopying} : \mathit{Value} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda v f \,.\ \textbf{matching}\ v \\
\qquad \triangleright (\mathit{MProd} \mapsto \mathit{Value}\ l_{old}{}^*) \\
\qquad\ \ \| \ (\mathit{allocatingCopies}\ l_{old}{}^*\ \mathit{deepCopying}\ (\lambda l_{new}{}^* \,.\, f\ (\mathit{MProd} \mapsto \mathit{Value}\ l_{new}{}^*))) \\
\qquad \triangleright \textbf{else}\, f\ v \\
\qquad \textbf{endmatching}
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e \,.\ \mathit{with\text{-}procedure\text{-}comp}\ (\mathcal{E}[\![E_1]\!]\ e) \\
\qquad\qquad\qquad\qquad (\lambda p \,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_2]\!]\ e) \\
\qquad\qquad\qquad\qquad\qquad (\lambda v \,.\ (\mathit{deepCopying}\ v\ \mathit{val\text{-}to\text{-}comp})))
\end{array}$$

Figure 10.9: Call-by-value-copy (CBVC) semantics for passing mutable tuples.

The CBVC strategy for passing mutable products is used for passing arrays and records by value in Pascal and for passing structures by value in C. On the other hand, arrays in C are passed via CBVS. Passing arrays in C via CBVC can be achieved by wrapping an array in a one-component struct! The inconsistency between the mechanisms for passing named vs. positional products in C is perplexing from the viewpoint of semantics but was apparently motivated by pragmatic issues.

The kind of data copying performed in Figure 10.9 is known as a **deep copy** because the copying process is recursively applied at all levels of the data. An alternative strategy, known as a **shallow copy**, is to copy only the first level of a data structure and share the contents of the other levels. Although it would be possible to use shallow copying in the call-by-copy strategy, we do not know of a real programming language that uses this strategy.

In languages supporting the call-by-reference (CBR) mechanism presented in Section 8.3.3.4, mutable products introduce new ways to alias locations between the caller and callee. When an `mget` construct is used in a parameter position, its L-value (the location of the product slot, as determined by $\mathcal{LV}$ in Figure 10.10) is passed rather than its R-value (the contents of the L-value). In the following CBR example, the L-values of `(mget 2 u)` and `r` denote the same location:

```
(let ((u (mprod 7 8)))
  (let ((g (lambda (p r)
             (begin
               (set! r (+ 20 r))
               (mset! 2 p (+ 100 (mget 2 p)))))))
    (begin (g u (mget 2 u))
           (mget 2 u))))
```

$\xrightarrow[CBR\ FLAVAR!]{}$ *128*

In contrast, under a CBV interpretation, changes to `r` would not affect `u` and `p`. The above expression would evaluate to *108* under CBVS and *8* under CBVC.

```
$\mathcal{LV}[\![$(mget $N$ $E_{mp}$)$]\!]$
$= \lambda e\,.\ (\textit{with-value}\ (\mathcal{E}[\![E_{mp}]\!]\ e)$
           $(\lambda v_{mp}\,.$ matching $v_{mp}$
                    $\triangleright (\textit{MProd} \mapsto \textit{Value}\ l^*)\ [\!]$
                      if $1 \leq (\mathcal{N}\ N)$ and $(\mathcal{N}\ N) \leq (\textit{length}\ l^*)$
                      then $(\textit{val-to-comp}\ (\textit{Location} \mapsto \textit{Value}\ \ (\mathcal{N}\ N))l^*)$
                      else $\textit{error-comp}$
                      fi
                    $\triangleright$ else $\textit{error-comp}$
                    endmatching ))
```

Figure 10.10: Extension to the CBR FLAVAR! semantics to handle mutable tuples.

▷ **Exercise 10.6** Write a single expression that returns the symbol `sharing` under CBVS, `deep` under CBVC with deep copying, and `shallow` under CBVC with shallow copying. Your expression should only use symbols, mutable tuples, and procedures. ◁

▷ **Exercise 10.7**

a. Modify the CBVC denotational semantics in Figure 10.9 to use shallow rather than deep copying.

b. Write three different versions of an operational semantics for FL! with mutable tuples that differ in their parameter passing mechanism: (1) CBVS (2) CBVC with deep copying (3) CBVC with shallow copying. ◁

10.2 Sums

Sums are entities that can be one of several different kinds of values. They are data structures that correspond to the sum domains that we have been using in our mathematical metalanguage (see Section A.3.3) to represent mathematical values that can come from several different component domains. Intuitively, a sum value augments an underlying component value with a **tag** that indicates which kind of value it is. Whenever a sum value is processed, this tag is dynamically examined to determine how to handle the underlying value. Sums are used in situations where programmers use the terms "either" or "one of" to informally describe a data structure. For example:

- A linked list is either a list node (with head and tail components) or the empty list.

- A graphics system might support shapes that are either circles, rectangles, or triangles.

- In a banking system, transactions might be one of deposit, withdrawal, transfer, or balance query.

Sums are known by such names as *tagged sums*, *unions*, *tagged unions*, *discriminated unions*, *oneofs*, and *variants*.

Just as sum domains are duals of product domains, sum data structures are dual to product data structures: for any given product structure, there is a dual sum data structure. Sums therefore vary along the same dimensions as products: positional vs. named, immutable vs. mutable, and dynamically typed vs. statically typed. Our discussion will focus on the first of these dimensions: **positional sums**, in which the different cases are distinguished only by their position in the sum specification (i.e., their tags are natural numbers) vs. **named sums**, in which the different cases are distinguished by specified names.

10.2.1 Positional Sums

Positional product data structures use integer indices to distinguish product components. Similarly, positional sums use integer tags to distinguish summands. To add positional sums to FL, we extend the syntax of expressions as follows:

E ::= ...
  | `(inj` N E`)` [Sum Introduction]
  | `(sumcase` E_{disc} I_{val} E_{body}`*)` [Sum Elimination]

`(inj` N E`)` creates a sum value whose tag is the integer N, where N is a manifest constant and not a computed value. Sum values are taken apart with `(sumcase` E_{disc} I_{val} E_{body}`*)`, which evaluates the **discriminant** E_{disc} to what should be a sum value, examines the numeric tag N of this sum value, and evaluates the Nth body expression with I_{val} bound to the untagged sum component.

Figure 10.11 shows a simple bank transaction system implemented with positional sums. An account is a pair of a savings balance and a checking balance. There are four kinds of transactions distinguished by an integer tag:

1. Deposit of an integer amount to savings.
2. Withdrawal of an integer amount from checking.
3. Transfer of an integer amount from savings to checking.
4. Transfer of an integer amount from checking to savings.

Given a transaction and account, the `process` procedure returns an updated account that reflects the actions of the translation.

The operational semantics for call-by-value sums is presented in Figure 10.12. Stuck states arise when any of the subexpressions get stuck, when the discriminant does not evaluate to a sum value, or when the integer tag does not correspond to an appropriate body (e.g., the integer is negative, zero, or larger than the number of bodies supplied). Call-by-name sums are similar (Figure 10.12), except that no attempt is made to evaluate the expression being injected.

The denotational semantics for call-by-value sums is presented in Figure 10.14. It might seem odd that the domain *Sum* of sum values is modeled via a product that pairs an integer tag and the injected value. But such a product is isomorphic to an infinite sum of injected values, so it does indeed represent a sum. The clause for `sumcase` calculates the denotations of all body expressions and chooses one based on the integer tag of the sum value. However, as with the denotational semantics of `if` expressions, only the chosen body is "evaluated." The denotational semantics of call-by-name sums is left as an exercise.

Positional sums are awkward to use in practice for several reasons:

```
(define (make-account checking savings) (pair checking savings))
(define (checking account) (left account))
(define (savings account) (right account))

(define (process transaction account)
  (sumcase transaction amount
    ; Deposit to savings
    (make-account (checking account)
                  (+ (savings account) amount))
    ; Withdrawal from checking
    (if (<= amount (checking account))
        (make-account (- (checking account) amount)
                      (savings account))
        (error 'insufficient-checking))
    ; Transfer from savings to checking
    (if (<= amount (savings account))
        (make-account (+ (checking account) amount)
                      (- (savings account) amount))
        (error 'insufficient-savings))
    ; Transfer from checking to savings
    (if (<= amount (checking account))
        (make-account (- (checking account) amount)
                      (+ (savings account) amount))
        (error 'insufficient-checking))
    ))
```

`(process (inj 1 10) (make-account 25 40))` $\xrightarrow[FL]{}$ $\langle 25, 50\rangle$

`(process (inj 2 10) (make-account 25 40))` $\xrightarrow[FL]{}$ $\langle 15, 40\rangle$

`(process (inj 3 10) (make-account 25 40))` $\xrightarrow[FL]{}$ $\langle 35, 30\rangle$

`(process (inj 4 10) (make-account 25 40))` $\xrightarrow[FL]{}$ $\langle 15, 50\rangle$

Figure 10.11: Bank transactions with positional sums.

$$V \;\in\; \text{ValueExp} \;=\; \ldots \cup \{\texttt{(inj}\ N\ V\texttt{)}\}$$

$$\frac{E \Rightarrow E'}{\texttt{(inj}\ N\ E\texttt{)} \Rightarrow \texttt{(inj}\ N\ E'\texttt{)}} \qquad [\textit{inj-progress}]$$

$$\frac{E \Rightarrow E'}{\texttt{(sumcase}\ E\ I\ E_1\ \texttt{...)} \Rightarrow \texttt{(sumcase}\ E'\ I\ E_1\ \texttt{...)}} \qquad [\textit{sumcase-progress}]$$

$$\begin{array}{c}\texttt{(sumcase (inj}\ N\ V\texttt{)}\ I\ E_1\ \texttt{...}\ \ E_m\texttt{)} \Rightarrow [V/I]E_N,\\ \text{where}\ \ 1 \le N \le m\end{array} \qquad [\textit{sumcase}]$$

Figure 10.12: CBV operational semantics for positional sums

$$V \;\in\; \text{ValueExp} \;=\; \ldots \cup \{\texttt{(inj}\ N\ E\texttt{)}\}$$

$$\frac{E \Rightarrow E'}{\texttt{(sumcase}\ E\ I\ E_1\ \texttt{...)} \Rightarrow \texttt{(sumcase}\ E'\ I\ E_1\ \texttt{...)}} \qquad [\textit{sumcase-progress}]$$

$$\begin{array}{c}\texttt{(sumcase (inj}\ N\ E_{val}\texttt{)}\ I\ E_1\ \texttt{...}\ \ E_m\texttt{)} \Rightarrow [E_{val}/I]E_N,\\ \text{where}\ \ 1 \le N \le m\end{array} \qquad [\textit{sumcase}]$$

Figure 10.13: CBN operational semantics for positional sums

$$\begin{array}{rcrcl} su & \in & \textit{Sum} & = & \textit{Int} \times \textit{Value}\\ v & \in & \textit{Value} & = & \ldots + \textit{Sum}\end{array}$$

$$\begin{aligned}\mathcal{E}[\![\texttt{(inj}\ N\ E\texttt{)}]\!] = \lambda e\,.\ & \textit{with-value}\ (\mathcal{E}[\![E]\!]\ e)\\ & \quad (\lambda v\,.\ (\textit{val-to-comp}\ (\textit{Sum} \mapsto \textit{Value}\ \langle(\mathcal{N}\ N),\ v\rangle)))\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{(sumcase}\ E_{disc}\ \ I_{val}\ \ E^{*}\texttt{)}]\!] =\\ &\quad \lambda e\,.\ \textit{with-value}\ (\mathcal{E}[\![E_{disc}]\!]\ e)\\ &\qquad\quad (\lambda v_{disc}\,.\ \ \textbf{matching}\ v_{disc}\\ &\qquad\qquad\qquad \triangleright (\textit{Sum} \mapsto \textit{Value}\ \langle i,\ v\rangle)\ \|\ (\textit{nth}_{\textit{Computation}}\ \ i\ \ (\mathcal{E}^{*}[\![E^{*}]\!]\ \ [I_{val} : v]e))\\ &\qquad\qquad\qquad \triangleright\, \textbf{else}\ \textit{error-comp}\\ &\qquad\qquad\qquad \textbf{endmatching}\,)\end{aligned}$$

Figure 10.14: CBV denotational semantics for sums

1. The programmer must remember the arbitrary association between each integer tag and its intended meaning;

2. In a `sumcase` expression, the body expressions must be carefully ordered to have the correct (implicit) index;

3. Since all sum values use integer tags, a sum value intended for one purpose may accidentally be used for another without any error being reported.

▷ **Exercise 10.8** In call-by-name positional sums, the expression (`inj` N E) does not tag the *value* denoted by E but rather tags the *computation* denoted by E. Modify the denotational semantics for positional sums in Figure 10.14 to be call-by-name rather than call-by-value. ◁

▷ **Exercise 10.9** The simplest kind of positional product is a pair, which glues together two component values. Dually, the simplest kind of positional sum chooses between two component values. Such a sum value is called an an **either**. It has two two possible tags: *left* or *right*.

Here we consider an extension to FL that supports eithers rather than general positional sums. Suppose we extend the syntax of FL as follows:

E ::= . . .
  | (`inleft` E) [Either Left Injection]
  | (`inright` E) [Either Right Injection]
  | (`ecase` E_{disc} I_{val} E_{left} E_{right}) [Either Case Analysis]

(`inleft` E) creates an either whose tag is *left* and whose value is the value of E.

(`inright` E) creates an either with whose tag is *right* and whose value is the value of E.

(`ecase` E_{disc} I_{val} E_{left} E_{right}) examines the discriminant value represented by E_{disc}, binds the untagged value to the identifier I_{val}, and then evaluates E_{left}, if the tag is *left*, or E_{right}, if the tag is *right*. It is an error if E_{disc} is not an either.

For example, we can use eithers in an extended version of FL to encode whether a geometric shape is a square (in which case the value of the either is the length of a side) or a circle (in which case the value of the either is the radius). We can then write a procedure for computing the area of a shape:

```
(define (square side) (inleft side))
(define (circle radius) (inright radius))
(define pi 3.14159)
(define (area shape)
  (ecase shape v
    (f* v v)  ; square case (f* multiplies floating point numbers)
    (f* pi (f* v v)) ; circle case
    ))
```

`(area (square 10.0))` $\xrightarrow[FL]{}$ *100.0*
`(area (circle 10.0))` $\xrightarrow[FL]{}$ *314.159*

a. Write an operational semantics for CBV eithers. What causes stuck states in your semantics?

b. Write a denotational semantics for CBV eithers. You may find it convenient to have a new domain for eithers as well as new *Left* and *Right* domains.

c. Write an operational semantics for CBN eithers. What causes stuck states in your semantics?

d. Write a denotational semantics for CBN eithers. You may find it convenient to have a new domain for eithers as well as new *Left* and *Right* domains. ◁

10.2.2 Named Sums

Named sums address the problems of positional sums by using programmer-supplied names to distinguish the various cases in a sum value. Named sums involve two new constructs:

E ::= . . .
| `(one` I_{tag} E`)` [Oneof Intro]
| `(tagcase` E_{disc} I_{val} `(`I_{tag} E_{body}`)*` `[(else` E_{else}`)])` [Oneof Elim]

A named sum value, which we shall call a **oneof**, is created by the evaluation of the expression `(one` I_{tag} E`)`, which conceptually pairs the tag I_{tag} with the component value given by E. Oneofs are decomposed via the expression `(tagcase` E_{disc} I_{val} `(`I_{tag} E_{body}`)*)`, which dispatches to a clause based on the tag of the oneof value of the discriminant expression E_{disc}. The value of the `tagcase` is the result of evaluating the body of the clause with the matching tag in a scope where I_{val} is bound to the untagged oneof component. A `tagcase` expression may have an optional `else` clause whose body E_{else} is evaluated and returned when no clause tag matches the discriminant tag. It is an error if E_{disc} does not evaluate to a oneof or if there is no clause in an `else`-less `tagcase` whose tag matches the discriminant tag.

Figure 10.15 shows how the bank transaction example can be expressed with named sums. Using symbolic tags instead of integers makes such programs easier to read and write; the tags serve as comments and allow the `tagcase` clauses to be written in any order. Although it is still possible for the same symbolic tag to be used for conceptually different oneofs, the likelihood that a oneof will be used in a incorrect context without generating a dynamic error is greatly reduced.

```
(define (process transaction account)
  (tagcase transaction amount
    (savings-deposit
      (make-account (checking account)
                    (+ (savings account) amount)))
    (checking-withdrawal
      (if (<= amount (checking account))
          (make-account (- (checking account) amount)
                        (savings account))
         (error 'insufficient-checking)))
    (savings->checking
      (if (<= amount (savings account))
          (make-account (+ (checking account) amount)
                        (- (savings account) amount))
          (error 'insufficient-savings)))
    (checking->savings
      (if (<= amount (checking account))
          (make-account (- (checking account) amount)
                        (+ (savings account) amount))
          (error 'insufficient-checking)))
    ))
```

`(process (one savings-deposit 10) (make-account 25 40))` $\xrightarrow[FL]{} \langle 25, 50 \rangle$

`(process (one checking-withdrawal 10) (make-account 25 40))` $\xrightarrow[FL]{} \langle 15, 40 \rangle$

`(process (one savings->checking 10) (make-account 25 40))` $\xrightarrow[FL]{} \langle 35, 30 \rangle$

`(process (one checking->savings 10) (make-account 25 40))` $\xrightarrow[FL]{} \langle 15, 50 \rangle$

Figure 10.15: Bank transactions with named sums.

Oneofs have semantics similar to that for positional sums, except that identifiers are used as tags rather than integers. Figure 10.16 gives the call-by-value operational semantics for oneofs. As before, stuck states arise when a value that is not a oneof appears as the discriminant of a `tagcase` or when a `tagcase` does not specify a clause appropriate for the dynamic tag of the oneof value.

The denotational semantics for call-by-value oneofs (Figures 10.17–10.18) shows their duality with records quite clearly. Records use an environment to

$$V \in \text{ValueExp} = \ldots \cup \{\texttt{(one } I\ V\texttt{)}\}$$

$$\frac{E \Rightarrow E'}{\texttt{(one } I\ E\texttt{)} \Rightarrow \texttt{(one } I\ E'\texttt{)}} \qquad [\textit{one-progress}]$$

$$\frac{E \Rightarrow E'}{\texttt{(tagcase } E\ I\ \texttt{...)} \Rightarrow \texttt{(tagcase } E'\ I\ \texttt{...)}} \qquad [\textit{tagcase-progress}]$$

$$\texttt{(tagcase (one } I_i\ V\texttt{) } I\ \texttt{(}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{))} \Rightarrow [V/I]E_i \qquad [\textit{tagcase}]$$

$$\begin{array}{l}\texttt{(tagcase (one } I_{tag}\ V\texttt{) } I \\ \quad \texttt{(}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{) (else } E_{else}\texttt{))} \Rightarrow [V/I]E_{else}, \\ \qquad\qquad \text{where } I_{tag} \not\in \{I_1, \ldots I_n\}\end{array} \qquad [\textit{tagcase-else}]$$

Figure 10.16: CBV operational semantics for named sums

glue together named values, one of which is later chosen at each `select` site. Dually, `one` creates a sum that is later processed in the context of a `tagcase` that uses an environment to glue together named clause bodies. In a continuation-based semantics, the environment associated with the `tagcase` would map names to continuations, suggesting a duality between values and continuations.

▷ **Exercise 10.10**

a. Modify the operational semantics for named sums in Figure 10.16 to be call-by-name rather than call-by-value.

b. Modify the denotational semantics for named sums in Figure 10.18 to be call-by-name rather than call-by-value.

c. Write a denotational semantics for call-by-name and call-by-value named sums in a continuation-based semantics. ◁

10.3 Sum-of-Products

In practice, sum and product data are often used together in idiomatic ways. Many common data structures can be viewed as a tree constructed from different kinds of nodes, each of which has multiple components. Here are some examples:

- A shape in a simple geometry system is either:

$$t \quad \in \quad \mathit{Tag\text{-}environment} \quad = \quad \text{Identifier} \to \mathit{Denotable} \to \mathit{Computation}$$

```
$\mathit{empty\text{-}tenv} : \mathit{Tag} - \mathit{environment} \ = \lambda I \delta\ .\ \mathit{error\text{-}comp}$

$\mathit{extend\text{-}tenv} :\ \mathit{Environment} \to \text{Identifier} \to \text{Identifier} \to \text{Exp} \to \mathit{Tag} - \mathit{environment}$
                  $\to \mathit{Tag} - \mathit{environment}$
  $= \lambda e\ I_{val}\ I\ E\ t\,.\,\lambda I'\ \delta\ .\ \textbf{if}\ (\mathit{same\text{-}identifier?}\ I'\ \ I)\ \textbf{then}\ (\mathcal{E}[\![E]\!]\ [I_{val} : \delta]e)\ \textbf{else}\ (t\ I')$

$\mathit{extend\text{-}tenv}^* :\ \mathit{Environment} \to \text{Identifier} \to \text{Identifier}^* \to \text{Exp}^* \to \mathit{Tag} - \mathit{environment}$
                  $\to \mathit{Tag} - \mathit{environment}$
$= \lambda e\ I_{val}\ I^*\ E^*\ t\ .$ $\textbf{matching}\ \langle I^*, E^* \rangle$
                  $\triangleright \langle [\,]_{\text{Identifier}}, [\,]_{\text{Exp}} \rangle\ [\!]\ \ t$
                  $\triangleright \langle I\ .\ I_{rest}{}^*, E\ .\ E_{rest}{}^* \rangle$
                    $[\!]\ (\mathit{extend\text{-}tenv}^*\ e\ \ I_{val}\ \ I_{rest}\ \ E_{rest}{}^*$
                                $(\mathit{extend\text{-}tenv}\ \ e\ \ I_{val}\ \ I_{rest}\ \ E_{rest}\ \ t))$
                  $\triangleright$ **else** $\mathit{empty\text{-}tenv}$
                  **endmatching**
```

Figure 10.17: Auxiliary domains and functions for denotational semantics of named sums (oneofs)

  - a circle with a radius;
  - a rectangle with a width and a height;
  - a triangle with three side lengths.

- A list of integers is either:
  - an empty list;
  - a list node with an integer head and an integer list tail.

- An ELM expression is either:
  - an integer literal;
  - an argument expression with an index;
  - an arithmetic operation with an operator symbol, a left operand expression, and a right operand expression.

In each of the above examples, the variety of possible nodes for a data structure can be modeled as a sum, and each individual kind of node can be modeled as a product. For this reason, such data structures are known as **sum-of-product** structures.

As a simple example, consider the following list of geometric shapes:

$$\begin{array}{rcrcl} su & \in & \mathit{Sum} & = & \text{Identifier} \times \mathit{Value} \\ v & \in & \mathit{Value} & = & \ldots + \mathit{Sum} \end{array}$$

$$\begin{array}{l} \mathcal{E}[\![\texttt{(one}\ I\ E\texttt{)}]\!] = \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E]\!]\ e) \\ \qquad\qquad\qquad\qquad (\lambda v\,.\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Sum} \mapsto \mathit{Value}\ \langle I,\ v\rangle))) \end{array}$$

$$\begin{array}{l} \mathcal{E}[\![\texttt{(tagcase}\ E_{disc}\ I_{val}\ \texttt{(}I_1\ E_1\texttt{)}\ \texttt{...}\ \texttt{(}I_n\ E_n\texttt{))}]\!] = \\ \quad \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_{disc}]\!]\ e) \\ \qquad\qquad (\lambda v_{disc}\,.\ \textbf{matching}\ v_{disc} \\ \qquad\qquad\qquad\qquad \triangleright (\mathit{Sum} \mapsto \mathit{Value}\ \langle I_{tag},\ v\rangle) \\ \qquad\qquad\qquad\qquad\quad [\!]\ ((\mathit{extend\text{-}tenv}^*\ e\ I_{val}\ [I_1 \ldots I_n]\ [E_1 \ldots E_n]\ \mathit{empty\text{-}tenv})\ I_{tag}\ v) \\ \qquad\qquad\qquad\qquad \triangleright\,\textbf{else}\ \mathit{error\text{-}comp} \\ \qquad\qquad\qquad\qquad \textbf{endmatching}\,) \end{array}$$

$$\begin{array}{l} \mathcal{E}[\![\texttt{(tagcase}\ E_{disc}\ I_{val}\ \texttt{(}I_1\ E_1\texttt{)}\ \texttt{...}\ \texttt{(}I_n\ E_n\texttt{)}\ \texttt{(else}\ E_{else}\texttt{))}]\!] = \\ \quad \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_{disc}]\!]\ e) \\ \qquad\qquad \textbf{let}\ \mathit{elsetenv}\ \textbf{be}\ (\lambda I\delta\,.\ (\mathcal{E}[\![E_{else}]\!]\ [I_{val} : \delta]e))\ \textbf{in} \\ \qquad\qquad\quad (\lambda v_{disc}\,.\ \textbf{matching}\ v_{disc} \\ \qquad\qquad\qquad\qquad\quad \triangleright (\mathit{Sum} \mapsto \mathit{Value}\ \langle I_{tag},\ v\rangle) \\ \qquad\qquad\qquad\qquad\qquad [\!]\ ((\mathit{extend\text{-}tenv}^*\ e\ I_{val}\ [I_1 \ldots I_n]\ [E_1 \ldots E_n]\ \mathit{elsetenv})\ I_{tag}\ v) \\ \qquad\qquad\qquad\qquad\quad \triangleright\,\textbf{else}\ \mathit{error\text{-}comp} \\ \qquad\qquad\qquad\qquad\quad \textbf{endmatching}\,) \\ \qquad\qquad \textbf{letend} \end{array}$$

Figure 10.18: CBV denotational semantics for oneofs

```
(list (one rectangle (record (width 3) (height 4)))
      (one triangle (record (side1 5) (side2 6) (side3 7)))
      (one square (record (side 2))))
```

In this encoding, oneof tags are used to distinguish squares, rectangles, and triangles. The two sides of a rectangle (width and height) and three sides of a triangle (side1, side2, and side3) are named as fields in a record. Even though a square has only a single side length (side), it too is encapsulated in a record for uniformity. Of course, we could have used positional rather than named products, in which case the meaning of each position would need to be specified.

Manipulating a sum-of-product datum typically involves performing a case analysis on its tag and extracting the components of the associated record. For example, here is a procedure that calculates the perimeter of a shape:

```
(define (perim shape)
  (tagcase shape r
    (square (* 4 (select side r)))
    (rectangle (* 2 (+ (select width r) (select height r))))
    (triangle (+ (select side1 r)
                 (+ (select side2 r) (select side3 r))))))
```

As another example, consider the sum-of-product encoding of the ELM temperature conversion expression (/ (* 5 (- (arg 1) 32)) 9) shown in Figure 10.19. In this encoding, oneof tags distinguish arithmetic operations (arithop), integer literals (lit), and argument references (arg). The three components of an arithmetic operation — the operation (op) (a symbol) and two operands (rand1 and rand2) are represented as a record. As with square shapes, the single number component of a literal expression and index component of an argument expression are boxed up into records for uniformity.

To handle this representation for ELM expressions, the elm-eval procedure from Chapter 6 would be rewritten:

```
(define (elm-eval exp args)
  (tagcase exp r
    (lit rcd (select num r))
    (arg rcd (arg-get (select index r) args))
    (arithop rcd ((primop->proc (select op r))
                  (elm-eval (select rand1 r) args)
                  (elm-eval (select rand2 r) args)))))
```

The rigidity of the above sum-of-product encodings is sometimes relaxed in practice. For instance, the case where a product has a single component can be optimized by replacing the product by the component value. If a product

```
(one arithop
  (record
    (op '/)
    (rand1 (one arithop
             (record
               (op '*)
               (rand1 (one lit (record (num 5))))
               (rand2 (one arithop
                        (record
                          (op '-)
                          (rand1 (one arg (record (index 1))))
                          (rand2 (one lit (record (num 32))))))))))
    (rand2 (one lit (record (num 9))))))
```

Figure 10.19: An ELM expression for converting temperatures from degrees Fahrenheit to degrees Celsius.

has zero components, it can be replaced by the unit value. In several popular data structures (including linked lists and binary trees), there are only two summands, one of which has no components. This situation is often handled by representing the non-trivial summand (e.g., list or tree node) directly as a product and representing the nullary summand (e.g., empty list or tree leaf) as a distinguished **null pointer** value. Conceptually, there is still a sum in this case: a value is *either* a null pointer or a node. But in terms of pragmatics, it is not necessary to associate a tag with a node because it is assumed that there is a cheap test that determines whether or not a node is the null pointer. For example, some runtime systems represent a null pointer with a value that contains all zeros to take advantage of efficient machine instructions for testing for zero.

Programming languages differ widely in terms of their support for sum-of-product data. For example:

- The ML and HASKELL programming languages have powerful facilities for declaring and manipulating sum-of-product data. We shall see similar facilities in the following sections.

- In object-oriented languages, such as JAVA, SMALLTALK, and C++, the dynamic dispatch performed when invoking a method on an object effectively performs a case analysis on the class (think tag) of the object, whose instance variables can be viewed as a record.

- In LISP dialects, it is common to represent a sum-of-product datum as a list s-expression whose first element is a symbolic tag indicating the summand and whose remaining elements are the components of the product. For instance, the Fahrenheit-to-Centigrade conversion expression given above can be represented as the following Lisp s-expression:

```
(arithop /
         (arithop *
                  (lit 5)
                  (arithop - (arg 1) (lit 32)))
         (lit 9))
```

This, in turn, can be optimized without ambiguity into an s-expression identical to the ELM concrete s-expression syntax:

```
(/ (* 5
      (- (arg 1)
         32))
   9)
```

Indeed, syntax trees are without a doubt the most important sum-of-product data structure used in the study of programming languages. The ease with which they can be represented as s-expressions is the reason we have adopted s-expression grammars for the toy languages in this book.

- In document description languages like HTML and XML, summand tags appear in begin/end markups and product components are encoded both in the association lists of markups as well as in components nested within the begin/end markups. For instance, Figure 10.20 shows how the Fahrenheit-to-Centigrade expression might be encoded in XML. The reader is left to ponder why XML, which at one level is a verbose encoding of s-expressions, is a far more popular standard for expressing structured data than s-expressions. In fact, the Water language [Plu02] goes the distance, using XML as a representation for s-expressions in a language with Scheme-like semantics.

- In the C programming language, programmers must "roll their own" sum-of-product data structures using `union` and `struct`. For instance, Figure 10.21 shows how the geometric shape example from above can be expressed in C. In C, `union` is used to declare storage that can contain one of several different kinds of values. However, there is no built-in support for tagging such values. Instead, an explicit `struct` is typically used to associate a tag (`shapetag` in the example) with the value (`sum` in the example). Values with multiple components (e.g., `rect` and `tri`) are

```
<arithop>
  <op name="/"/>
  <rand1>
    <arithop>
      <op name="*"/>
      <rand1>
        <lit num=5/>
      </rand1>
      <rand2>
        <arithop>
          <op name="-"/>
          <rand1>
            <arg index=1/>
          </rand1>
          <rand2>
            <lit num=32/>
          </rand2>
        </arithop>
      </rand2>
    </arithop>
  </rand1>
  <rand2>
    <lit num=9/>
  </rand2>
</arithop>
```

Figure 10.20: The ELM Fahrenheit-to-Centigrade expression in XML notation.

themselves encoded via additional `struct` declarations.

As is apparent from the example in Figure 10.21, encoding sum-of-product data in C is awkward. Nesting `struct` declarations to provide explicit tags is cumbersome and leads to unwieldy name paths like `s.sum.rect.width`. But much worse is the fact that the language enforces no connection between the tag and the sum. For instance, consider the following sequence of C statements:

```
shape s4;
s4.tag = square;
s4.sum.rect.width = 8;
s4.sum.rect.height = 9;
printf("The perimeter of s4 is \%d\n", perim(s4));
```

Although conceptually it makes no sense to manipulate a rectangle's components in a square, in many C implementations, the above code compiles and runs without error, yielding 32 as the perimeter of `s4`. Why? Because the storage set aside for a `union` type is that required for the largest summand (in this case, the three integers of a triangle) and `s4.sum.side`, `s4.sum.rect.width`, and `s4.sum.tri.side1` are all just synonyms that reference the first slot of this storage.

This is a classic example of a **type loophole** in C. Pascal's variant records, which encode sum-of-product datatypes in a way reminiscent of C, exhibit a similar type loophole. The same sort of undesirable behavior can be exhibited with the Lisp s-expression `(square 8 9)`, for which a perimeter procedure would return 32 if the means of extracting the side of a square was returning the second element of an s-expression list. But the difference between Lisp and C/Pascal on this score is that C and Pascal, unlike Lisp, sport a static type system that might be expected to catch such type-related bugs at compile time. We will have much more to say about static typing in Chapter **??**.

10.4 Data Declarations

Programming with "raw" sums and products is cumbersome and error-prone. Here we study a high-level data declaration facility that simplifies the creation and manipulation of sum-of-product data. We extend our FL family of languages with a `define-data` declaration that specifies a new kind of sum-of-product data. We introduce this construct via a declaration for geometric shapes:

```
(define-data shape
  (square side)
  (rectangle width height)
  (triangle side1 side2 side3))
```

```
typedef enum {square, rectangle, triangle} shapetag;

typedef struct {
  shapetag tag;
  union {
    int side;
    struct {int width; int height;} rect;
    struct {int side1; int side2; int side3;} tri;
  } sum;
} shape;

int perim (shape s) {
  switch (s.tag) {
  case square:
    return 4*(s.sum.side);
  case rectangle:
    return 2*(s.sum.rect.width + s.sum.rect.height);
  case triangle:
    return (s.sum.tri.side1 + s.sum.tri.side2 + s.sum.tri.side3);
  }
}

int main () {
  shape s1, s2, s3;
  s1.tag = square;
  s1.sum.side = 2;
  s2.tag = rectangle;
  s2.sum.rect.width = 3;
  s2.sum.rect.height = 4;
  s3.tag = triangle;
  s3.sum.tri.side1 = 5;
  s3.sum.tri.side2 = 6;
  s3.sum.tri.side3 = 7;
  printf("The perimeter of s1 is \bs\%d\bs{}n", perim(s1));
  printf("The perimeter of s2 is \bs\%d\bs{}n", perim(s2));
  printf("The perimeter of s3 is \bs\%d\bs{}n", perim(s3));
}
```

Figure 10.21: The shape example encoded using `struct` and `union` in C.

This declaration specifies that a shape is either a square with one component, a rectangle with two components, or a triangle with three components. Each of the names `square`, `rectangle`, and `triangle` is a **value constructor procedure** (or just **constructor** for short) that takes the specified number of components and returns a sum-of-product datum with those components. For example, the list of shapes

```
(list (square 2) (rectangle 3 4) (triangle 7 8 9))
```

is equivalent to the list

```
(list (one square (product 2)))
      (one rectangle (product 3 4))
      (one triangle (product 5 6 7))
```

In contrast with the previous section, the sum-of-product data created by `define-data` constructors uses positional rather than named products.

In the example, the data name `shape` and the component names `side`, `width`, `height`, etc. are just comments. Only the *number* of components specified for a constructor is relevant. For instance, we could emphasize that all components are integers by writing

```
(define-data shape
  (square int)
  (rectangle int int)
  (triangle int int int)),
```

or we could use nonsense words to specify an equivalent declaration, as in

```
(define-data frob
  (square foo)
  (rectangle bar baz)
  (triangle quux quuux quuuux)).
```

The reason for requiring such comments is that the comment positions will assume a non-trivial meaning when we study a typed version of `define-data` in Chapter 15.

For every constructor procedure C that takes n arguments, `define-data` also declares an associated **deconstructor procedure** that takes three arguments:

1. the value v to be deconstructed;

2. a **success continuation**, an n-argument procedure that is applied to the n components of v in the case where v is constructed by C;

3. a **failure continuation**, a nullary procedure that is invoked in the case where v is not constructed by C.

We assume a convention in which the deconstructor has a name that is the name of the constructor followed by the tilde character, ~, which is pronounced "twiddle." For instance, the square~, rectangle~, and triangle~ deconstructors introduced by the shape declaration can be used to calculate the perimeter of a shape:

```
(define (perim s)
  (square~ s (lambda (s) (* 4 s))
    (lambda ()
      (rectangle~ s (lambda (w h) (* 2 (+ w h)))
        (lambda ()
          (triangle~ s (lambda (s1 s2 s3) (+ s1 s2 s3))
            (lambda ()
              (error not-a-shape))))))))
```

Deconstructors are somewhat awkward to use directly. In the next section we will study a pattern-matching facility based on deconstructors that significantly simplifies the deconstruction of sum-of-product data.

As another example of constructors and deconstructors, consider the elm-exp declaration in Figure 10.22. The lit, arg, and arithop constructors introduced by this declaration are illustrated in the Fahrenheit-to-Centigrade expression f2c, and the deconstructors lit~, arg~, and arithop~ are used to define elm-eval.

We can even use define-data to define list constructors and deconstructors (Figure 10.23), replacing the desugaring given in Chapter 6.

A formal definition of define-data is presented in Figure 10.24. The syntax of FL programs is extended to include define-data clauses along with the usual definitions. The meaning of a define-data declaration can be explained by desugaring the declaration into a sequence of procedure definitions via $\mathcal{D}_{\text{def}}$, which has signature $D \rightarrow D^*$. The resulting sequence of definitions is spliced into the program construct, and all program definitions are further desugared as shown in Chapter 6. Each summand clause (I_{tag} I_1 ... I_n) desugars into 2 definitions:

- An n-argument constructor procedure named I_{tag} that constructs a oneof with tag I_{tag} of a product whose components are $I_1 \ldots I_n$.

- A three-argument deconstructor procedure that applies the second argument (an n-argument success continuation) to the n components of the product if the oneof has the right tag and otherwise invokes the third argument (a nullary failure continuation). The name of this deconstructor is created from the name I_{tag} by adding ~ as a suffix. We shall use the no-

```
(define-data elm-exp
  (lit num)
  (arg index)
  (arithop op rand1 rand2))

(define f2c (arithop ’/
              (arithop ’*
                      (lit 5)
                      (arithop ’-
                               (arg 1)
                               (lit 32)))
              (lit 9)))

(define (elm-eval exp args)
  (lit˜ exp (lambda (n) n)
    (lambda ()
      (arg˜ exp (lambda (i) (get-arg i args))
        (lambda ()
          (arithop˜ exp
            (lambda (op r1 r2)
              ((primop->proc op) (elm-eval r1 args) (elm-eval r2 args)))
            (lambda () (error not-an-elm-exp)))))))
```

Figure 10.22: ELM examples.

```
(define-data list
  (null)
  (cons head tail))

(define (null? xs)
  (null˜ xs (lambda () true) (lambda () false)))

(define (car xs)
  (cons˜ xs (lambda (hd tl) hd)
    (lambda () (error car-of-nonlist-or-empty-list))))

(define (cdr xs)
  (cons˜ xs (lambda (hd tl) tl)
    (lambda () (error cdr-of-nonlist-or-empty-list))))
```

Figure 10.23: Defining lists via `define-data`.

tation $I_1 \bowtie I_2$ to concatenate identifiers. For example, square⋈~ denotes the identifier square~.

For example, Figure 10.25 shows the constructors and deconstructors introduced by the shape declaration.

Syntax

P ::= (program $D_{definitions}{}^{*}$ E_{body}) [Program]

D ::= (define I_{name} E_{value}) [Definition]
  | (define-data I_{data} (I_{tag} I*)*)

Sugar

If $D =$ (define-data I_{data} (I_{tag_1} $I_{1,1}$ ... I_{1,k_1}) ... (I_{tag_n} $I_{n,1}$ ... I_{n,k_n})),

$\mathcal{D}_{\text{def}}[\![D]\!] = \mathcal{D}_{\text{cl}}[\![$($I_{tag_1}$ $I_{1,1}$ ... I_{1,k_1})$]\!]$ @ ⋯ @ $\mathcal{D}_{\text{cl}}[\![$($I_{tag_n}$ $I_{n,1}$ ... I_{1,k_n})$]\!]$

and $\mathcal{D}_{\text{cl}}[\![$($I_{tag_i}$ $I_{i,1}$ ... I_{i,k_i})$]\!] =$

Constructor	*Deconstructor*
[(define (I_{tag_i} x1...x⋈k$_i$) (one I_{tag_i} (product x1...x⋈k$_i$)))] ,	(define (I_{tag_i}⋈~ val succ fail) (tagcase val x (I_{tag_i} (succ (proj 1 x) ⋮ (proj k$_i$ x))) (else (fail))))]

Figure 10.24: Syntax and desugaring of define-data.

▷ **Exercise 10.11** Extend the declaration of elm-exp and the definition of elm-eval to handle the full EL language. ◁

▷ **Exercise 10.12** It is possible to tweak the desugaring of define-data to use more efficient representations than those given in Figure 10.24.

a. Modify the define-data desugaring to avoid creating products for constructors that take zero or one argument.

b. Modify the define-data desugaring to represent a sum-of-products datum with tag I_{tag} and components $v_1 \dots v_n$ as the heterogeneous sequence

   (sequence (symbol I_{tag}) v_1 ... v_n)

   (This desugaring makes sense for a dynamically typed language but not a statically typed one.) ◁

```
(define square
  (lambda (x1)
    (one square (product x1))))

(define square~
  (lambda (val succ fail)
    (tagcase val x
      (square (succ (proj 1 x)))
      (else (fail)))))

(define rectangle
  (lambda (x1 x2)
    (one rectangle (product x1 x2))))

(define rectangle~
  (lambda (val succ fail)
    (tagcase val x
      (rectangle (succ (proj 1 x) (proj 2 x)))
      (else (fail)))))

(define triangle
  (lambda (x1 x2 x3)
    (one triangle (product x1 x2 x3))))

(define triangle~
  (lambda (val succ fail)
    (tagcase val x
      (triangle (succ (proj 1 x) (proj 2 x) (proj 3 x)))
      (else (fail)))))
```

Figure 10.25: Value constructors and deconstructors introduced by the `shape` declaration.

▷ **Exercise 10.13** SML and Haskell support user-defined datatype declarations. Below are the geometric shape declarations expressed in SML and Haskell:

SML	Haskell
datatype Shape =	data Shape =
Square of int	Square Int
| Rectangle of int * int	| Rectangle Int Int
| Triangle of int * int * int	| Triangle Int Int Int

In SML, passing multiple arguments to a data constructor is modeled by collecting the arguments into a tuple, as in `Triangle(5,6,7)`, where the tuple `(5,6,7)` has type `int * int * int`. It is a type error to supply the constructor with the wrong number of arguments, as in `Triangle(5,6)`.

In contrast, Haskell data declarations allow curried constructors that can take multiple arguments one at a time. For instance, the invocation `Triangle 5 6` denotes a unary function that "expects" the third side of the triangle.

Is FL extended with `define-data` more like ML or Haskell in this respect? For example, does `(triangle 5 6)` denote an error or a unary function? How would you change the desugaring of `define-data` to model the other language? ◁

▷ **Exercise 10.14** The desugaring for `define-data` in Figure 10.24 introduces two procedures (a constructor I_{tag} and a deconstructor I_{tag}⋈~) for each summand clause (I_{tag} I_1 ... I_n). An alternative approach is to introduce $n+2$ procedures:

- An n-argument constructor procedure named I_{tag}.
- A unary predicate named I_{tag}⋈~ that returns true for a oneof value with tag I_{tag} and false for any other oneof value. It is an error to apply this predicate to a value that is not a oneof value.
- n unary selector procedures named $I_1 \ldots I_n$, where I_i extracts the ith component of a product tagged with I_{tag}. It is an error to apply a selector procedure to a value that is not a oneof value or a oneof value with a tag that is not I_{tag}.

In this approach, the component names matter, since they are names of selectors, not just comments. For example, here is the `perim` procedure in this approach:

```
(define (perim s)
 (cond
   ((square? s) (* 4 (side s)))
   ((rectangle? s) (* 2 (+ (width s) (height s))))
   ((triangle? s) (+ (side1 s) (+ (side2 s) (side3 s))))
   ))
```

a. Give a desugaring for `define-data` that implements the new approach.

b. In your new desugaring, compare the evaluation of the conditional clause

```
((triangle? s) (+ (side1 s) (+ (side2 s) (side3 s))))
```

with the following deconstructor application in the original desugaring

```
(triangle~ s (lambda (s1 s2 s3) (+ s1 s2 s3))
  (lambda ()
    (error not-a-shape)))
```

Which evaluation is more efficient?

c. One drawback of having `define-data` desugar into so many procedures is that it increases the possibility of name conflicts. For instance, the `shape` declaration introduces procedures with names like `square`, `rectangle?`, and `width` that very well might be useful in other contexts. One way to address this problem is for programmers use more specific names within data declarations, as in:

```
(define-data shape
  (shape-square shape-side)
  (shape-rectangle shape-width shape-height)
  (shape-triangle shape-side1 shape-side2 shape-side3))
```

Another approach is to modify the desugaring for `define-data` to automatically concatenate the data type name with the name of every constructor, predicate, and selector procedure. For instance, something like this is done in Common Lisp's `defstruct` facility. Discuss the benefits and drawbacks of these two ways to address potential name conflicts in a program with data declarations.

d. Yet another way to address name conflicts is to treat constructor, predicate, and selector applications as special forms that refer to a different namespace than the usual value namespace. Design an extension to FL that handles datat declarations based on this idea. Do you think it is a good way to handle name conflicts? ◁

10.5 Pattern Matching

10.5.1 Introduction to Pattern Matching

Deconstructors are a sufficient mechanism for dispatching on and extracting the components of sum-of-product data, but they are awkward to use in practice. It is more convenient to manipulate sum-of-product data using a **pattern matching** facility that simultaneously tests for a summand and names the components of the associated product when the test succeeds. We have made extensive use of a form of pattern matching (via the **matching** construct) in the mathematical metalanguage of this book. Pattern matching is also an important feature of some real-world programming languages, such as Prolog, ML and Haskell.

We will study pattern matching in the context of an extension to FL that includes `define-data` from the previous section along with a new `match` construct. First, we will give an informal introduction to `match` via a series of examples. Then we will describe the semantics of `match` in detail by desugaring it into deconstructor applications.

The `match` construct has the form `(match` E_{disc} `(`P_{pat} E_{body}`)*)`, where E_{disc} is the **discriminant** and each **match clause** of the form `(`P_{pat} E_{body}`)` has a **pattern** P_{pat} and a **body** E_{body}. A pattern P consists of either an FL literal value, an identifier, a wild card (“_”), or a tagged list of patterns:

$$\begin{array}{rll}
P ::= & L & \text{[Literal]} \\
\mid & I & \text{[Pattern Variable]} \\
\mid & \_ & \text{[Wild Card]} \\
\mid & \texttt{(}I\ P^*\texttt{)} & \text{[Tagged List]}
\end{array}$$

Informally, a `match` expression is evaluated by first evaluating E_{disc} into a value v_{disc}, then finding the first clause whose pattern P_i matches v_{disc}, and finally evaluating the associated body E_i of this clause relative to any bindings introduced by the successful match of v_{disc} to P_i. If no clause has a pattern matching v_{disc}, the `match` expression signals an error.

We begin with a few examples of `match` involving patterns that are just literals, identifiers, or wild cards. Here is a procedure that converts a boolean to an integer (and signals an error for a non-boolean input).

```
(define (bool->int b)
  (match b
    (#f 0)
    (#t 1)))
```

The `negate` procedure below returns a symbol that negates the sense of a `yes` or `no` input but returns `unknown` for any other input. The underscore pattern is a wildcard pattern that matches any discriminant.

```
(define (negate s)
  (match s
    (’yes ’no)
    (’no ’yes)
    (_ ’unknown)))
```

The following procedure returns one more than the square of a given number, except at the inputs -1 and 1, where it returns 0:

```
(define (squarish n)
  (match (* n n)
    (1 0)
    (x (+ 1 x))))
```

A pattern variable like `x` always successfully matches any discriminant value, and the name may be used to denote this value in the associated body expression.

To introduce tagged list patterns, we consider pattern matching involving lists of integers. Consider the following two procedures:

```
(define (match-ints-1 ints)
  (match ints
    ((cons x (null)) (* x x))
    (_ 17)
    ))

(define (match-ints-2 ints)
  (match ints
    ((cons x (null)) (* x x))
    ((cons 3 (cons y ns)) (+ y (length ns)))
    (_ 17)
    ))
```

- The pattern `(cons x (null))` matches a list that contains exactly one element, and names that element `x` in the scope of the body. So both procedures return the square of the first (only) element of the list when given a singleton list.

- The pattern `(cons 3 (cons y ns))` matches a list that has at least two elements, the first of which is the integer `3`. In the case of a match, the body is evaluated in a scope where the second element is named `y` and the list of all but the first two elements is named `ns`. So when this pattern matches, the second procedure returns the sum of the second element and the length of the rest of the list.

- The final wild card pattern in both procedures matches any value not matched by the first two patterns, in which case a `17` is returned.

The following table shows the results returned by these two procedures when supplied with various integer lists as an argument:

	`(list)`	`(list 3)`	`(list 3 4)`	`(list 6 8)`	`(list 3 6 8)`
`match-ints-1`	17	9	17	17	17
`match-ints-2`	17	9	4	17	7

The most important use of `match` is to perform pattern matching on user-defined sum-of-product data. For instance, here is a succinct version of the perimeter procedure based on pattern matching:

```
(define (perim shape)
  (match shape
    ((square s) (* 4 s))
    ((rectangle w h) (* 2 (+ w h)))
    ((triangle s1 s2 s3) (+ s1 (+ s2 s3)))))
```

```
(define (elm-eval exp args)
  (match exp
    ((lit n) n)
    ((arg i) (get-arg i args))
    ((arithop op r1 r2)
     ((primop->proc op) (elm-eval r1 args) (elm-eval r2 args)))))

(define (get-arg index nums)
  (match (list index nums)
    ((list 1 (cons n _)) n)
    ((list i (cons _ ns)) (get-arg i ns))))

(define (primop->proc sym)
  (match sym ('+ +) ('- -) ('* *) ('/ /)))
```

Figure 10.26: A complete ELM evaluator based on pattern matching.

The pattern `(square s)` matches a sum-of-product value constructed by the constructor application `(square` v_{side}`)`, in which case `s` names v_{side} in the body of the match clause. Similarly, the pattern `(rectangle w h)` matches a value constructed by `(rectangle` v_{width} v_{height}`)`, where `w` names v_{width} and `h` names v_{height}. The `triangle` pattern is handled similarly.

Some other nice illustrations of the conciseness of pattern matching involve the ELM language. Figure 10.26 presents a complete ELM evaluator based on pattern matching. The twelve lines of code are easy to understand and analyze. A compelling use of nested patterns is in the crude algebraic simplifier for ELM expressions in Figure 10.27. The second `match` clause in the `simp` procedure expresses that literals and argument references are self-evaluating (i.e., they simplify to themselves). The first clause simplifies an `arithop` by simplifying the arguments and then attempting to further simplify the resulting `arithop`. `simp-arithop` handles six special cases. The first four clauses express that zero is an identity for addition and one is an identity for multiplication. The next two clauses capture that multiplication by zero yields zero.[4] In order to appreciate the succinctness of pattern matching, the reader is encouraged to re-express the `simp` procedure in a version of FL that does not support pattern matching.

All the examples seen so far are “well-typed” in the sense that the discriminant of the `match` is “expected” to be a particular type (e.g., a list of integers, a shape, an ELM expression) and the results of all the clause bodies in a given `match` have the same type. But in a dynamically typed language, `match` is not

[4] This is not a safe transformation when the other subexpression contains a dynamic error!

```
(define (simp exp)
  (match exp
    ((arithop p r1 r2) (simp-arithop (arithop p (simp r1) (simp r2))))
    (x x)))

(define (simp-arithop exp)
  (match exp
    ((arithop '+ (lit 0) x) x)
    ((arithop '+ x (lit 0)) x)
    ((arithop '* (lit 1) x) x)
    ((arithop '* x (lit 1)) x)
    ((arithop '* (lit 0) _) (lit 0))
    ((arithop '* _ (lit 0)) (lit 0))
    (_ exp)))
```

Figure 10.27: An algebraic simplifier for ELM expressions.

required to have this behavior, as indicated by the following example:

```
(define (dynamic x)
  (match x
    ((0 #f)
     (#t 'zero)
     ('one 17))))
```

In Chapter **??**, we will study a statically typed version of FL in which `dynamic` will not be a legal procedure. However, all the other `match` examples above will still be legal.

10.5.2 A Desugaring-based Semantics of `match`

In order to motivate the structure of the desugaring of `match`, which is rather complex, we will incrementally develop the desugaring in the context of some concrete `match` examples rather than simply presenting the final desugaring. We begin with the `bool->int` procedure from the previous subsection:

```
(define (bool->int b)
  (match b
    (#f 0)
    (#t 1)))
```

It would be natural to desugar the `match` in `bool->int` into a series of `if` expressions:

```
(define (bool->int b)
  (if (equal? b #f)
      0
      (if (equal? b #t)
          1
          (error no-match))))
```

The case where `b` is not a boolean is handled by an explicit `error` expression indicating that the value of the discriminant did not match the pattern of any `match` clause.

In general, the discriminant of a `match` will be an arbitrary expression whose value should be calculated only once. To avoid recalculation of the discriminant, our `match` desugaring first names the discriminant (using `let`) and then performs a case analysis on the name. As shown in Exercise 10.19, this name can be eliminated when it is not necessary. For example, in `bool->int`, the discriminant is already bound to the variable `b`. Here is a revised desugaring for `bool->int` that names the discriminant:[5]

```
(define (bool->int b)
  (let ((disc b))
    (if (equal? disc #f)
        0
        (if (equal? disc #t)
            1
            (error no-match)))))
```

Whenever a mismatch between a pattern and a value is discovered, the matching process should stop processing the pattern in the current `match` clause and begin processing the pattern in the next `match` clause. When we study the desugaring of tagged patterns later, we will see that such a mismatch may be discovered at many different points in the processing of a given pattern. To avoid replicating the code that begins processing the pattern in the next `match` clause, our desugaring will wrap this code into a **failure thunk** that may potentially be invoked from several different points in the desugared code. Here is a version of the desugaring for `bool->int` that includes failure thunks named `fail1` and `fail2`:

[5] In the examples, all new identifiers introduced by the desugaring are assumed to be fresh so they do not clash with any program variables.

```
(define (bool->int b)
  (let ((disc b))
    (let ((fail1 (lambda ()
                   (let ((fail2 (lambda () (error no-match))))
                     (if (equal? disc #t)
                         1
                         (fail2))))))
      (if (equal? disc #f)
          0
          (fail1)))))
```

In the simple `match` within `bool->int`, each failure thunk is invoked exactly once. But soon we will see examples in which the failure thunk is invoked multiple times. In the case where the failure thunk is invoked zero or one times, it is possible for the desugarer to avoid introducing a named failure thunk. We leave this as an exercise.

The discussion so far leads to a first cut for the `match` desugaring shown in Figure 10.28. The desugaring of `match` is performed by $\mathcal{D}_{\text{match}}$. For simplicity, we assume that all `match` constructs are first eliminated by $\mathcal{D}_{\text{match}}$ in a separate pass over the program before other FL desugarings are performed. It is possible to merge all desugarings into a single pass, but that would make the description of the `match` desugaring more complex.

$$
\begin{array}{l}
\mathcal{D}_{\text{match}}[\![\texttt{(match}\ E_{disc}\ \texttt{(}P_1\ \ E_1\texttt{)}\ \ \texttt{...}\ \ \texttt{(}P_n\ \ E_n\texttt{))}]\!] = \\
\quad \texttt{(let ((}I_{disc}\ \ E_{disc}\texttt{))}\ \texttt{;}\ I_{disc}\ \texttt{fresh} \\
\quad\quad \texttt{(}\mathcal{D}_{\text{clauses}}\ [P_1, \ldots, P_n]\ [E_1, \ldots, E_n]\ I_{disc}\texttt{))} \\
\\
\mathcal{D}_{\text{clauses}}\ [\,]\ [\,]\ I_{disc}\ = \texttt{(error no-match)} \\
\mathcal{D}_{\text{clauses}}\ (P_1\ \ .\ \ P_{rest}{*})\ (E_1\ \ .\ \ E_{rest}{*})\ I_{disc}\ = \\
\quad \texttt{(let ((}I_{fail}\ \texttt{(lambda ()}\ \texttt{;}\ \text{Failure thunk: if } P_1 \text{ doesn't match, try the other clauses} \\
\quad\quad\quad\quad\quad\quad\quad \texttt{(}\mathcal{D}_{\text{clauses}}\ P_{rest}{*}\quad E_{rest}{*}\quad I_{disc}\texttt{))))} \\
\quad\quad \texttt{(}\mathcal{D}_{\text{pat}}[\![P_1]\!]\ I_{disc}\ E_1\ I_{fail}\texttt{))} \\
\\
\mathcal{D}_{\text{pat}}[\![L]\!]\ I_{disc}\ E_{succ}\ I_{fail}\ =\ \texttt{(if (equal}_L\ I_{disc}\ L\texttt{)}\ E_{succ}\ \texttt{(}I_{fail}\texttt{))} \\
\mathcal{D}_{\text{pat}}[\![\_]\!]\ I_{disc}\ E_{succ}\ I_{fail}\ =\ \text{To be added} \\
\mathcal{D}_{\text{pat}}[\![I]\!]\ I_{disc}\ E_{succ}\ I_{fail}\ =\ \text{To be added} \\
\mathcal{D}_{\text{pat}}[\![\texttt{(}I\ \ P_1\ \ \texttt{...}\ \ P_n\texttt{)}]\!]\ I_{disc}\ E_{succ}\ I_{fail}\ =\ \text{To be added}
\end{array}
$$

Figure 10.28: A first cut of the `match` desugaring.

$\mathcal{D}_{\text{match}}$ first introduces the fresh name I_{disc} for the value of the discriminant expression E_{disc} and then processes the `match` clauses via $\mathcal{D}_{\text{clauses}}$. The

$\mathcal{D}_{\text{clauses}}$ function takes three arguments: (1) a list of clause patterns, (2) a list of clause body expressions, and (3) the identifier naming the discriminant. The third argument allows the desugarer to refer to the discriminant by its identifier when processing the clauses. The $\mathcal{D}_{\text{clauses}}$ function uses $\mathcal{D}_{\text{pat}}$ to process the first pattern and body expression in a context where the fresh identifier I_{fail} names the failure thunk that processes the rest of the clauses. When no clauses remain, the desugarer yields an `error` expression that will be reached only when the desugared code for processing the clauses finds no pattern that matches the discriminant.

The core of the `match` desugaring is the $\mathcal{D}_{\text{pat}}$ function. This takes four arguments: (1) the pattern being matched, (2) the identifier naming the discriminant, (3) the **success expression** that is evaluated when the pattern matches the discriminant, and (4) the name of the failure thunk that is invoked when the pattern does not match the discriminant. A literal pattern is an easy case. The desugared code first compares the literal and discriminant via the equality operator $\texttt{equal}_L$. In a dynamically typed language, $\texttt{equal}_L$ is just the generic equality-testing procedure `equal?`, but when desugaring `match` in a statically typed language (as in Section 15.5), the equality operator $\texttt{equal}_L$ depends on the domain of the literal L. If the literal and discriminant are the same, the success expression is evaluated; otherwise, the failure thunk is invoked, which will either process the next `match` clause (if there is one) or signal a `no-match` error (if there is no next clause).

The literal case is the only $\mathcal{D}_{\text{pat}}$ case that is needed to explain the `bool->int` desugaring. The desugarings for the other three types of patterns (wildcards, identifiers, and tagged lists) are not shown in Figure 10.28 but will be fleshed out in the following discussion.

We first consider the wildcard pattern, as used in the `negate` procedure:

```
(define (negate s)
  (match s
    ('yes 'no)
    ('no 'yes)
    (_ 'unknown)))
```

The wildcard pattern always matches the discriminant, so the desugarer can simply emit the success expression for this case:

$$\mathcal{D}_{\text{pat}}[\![\_]\!]\ I_{disc}\ E_{succ}\ I_{fail} = E_{succ}$$

The result of desugaring the `match` expression within the `negate` procedure is:

```
(define (negate s)
  (let ((disc s))
    (let ((fail1
            (lambda ()
              (let ((fail2
                      (lambda ()
                        (let ((fail3 (lambda ()
                                       (error no-match))))
                          'unknown))))
                (if (equal? disc 'no) 'yes (fail2))))))
      (if (equal? disc 'yes) 'no (fail1)))))
```

It turns out that `fail3` can never be referenced, so the subexpression:

```
(let ((fail3 (lambda () (error no-match))))
  'unknown)
```

could simply be replaced by `'unknown`. This optimization could be performed by the desugarer itself or by a post-desugaring optimization pass (see Exercise 10.19.)

The case of patterns that are identifiers is similar to the wildcard case, except that the success expression must be evaluated in an environment where the identifier name is bound to the value of the discriminant:

$$\mathcal{D}_{\text{pat}}[\![I]\!]\ I_{disc}\ E_{succ}\ I_{fail} = \texttt{(let ((}I\ I_{disc}\texttt{))}\ E_{succ}\texttt{)}$$

As an example, consider the `squarish` procedure introduced above:

```
(define (squarish n)
  (match (* n n)
    (1 0)
    (x (+ 1 x))))
```

After desugaring the `match` expression within `squarish`, the procedure becomes:

```
(define (squarish n)
  (let ((disc (* n n)))
    (let ((fail1
            (lambda ()
              (let ((fail2 (lambda () (error no-match))))
                (let ((x disc))
                  (+ x 1))))))
      (if (equal? disc 1)
          0
          (fail1)))))
```

As in the `negate` example, the creation of the innermost failure thunk can be eliminated by an optimization (see Exercise 10.19). Note that the binding of the discriminant to an identifier is significant here: `(* n n)` would otherwise be evaluated twice. If the discriminant expression performed any side effects, this would be a semantic issue as well as an efficiency concern.

The last case for $\mathcal{D}_{\text{pat}}$ is a tagged list pattern of the form `(`$I\ P_{1}\ \ldots\ P_{n}$`)`. Recall that I in this case is some sort of constructor procedure, such as `cons` or `triangle` in the pattern matching examples. Handling this case is tricky because it requires decomposing a constructed value into parts and recursively matching the subpatterns $P_{1}\ \ldots\ P_{n}$ against these parts. It turns out that deconstructor procedures are an excellent way to deal with tagged list patterns:

```
$\mathcal{D}_{\text{pat}}[\![$($I\ P_{1}\ \ldots\ P_{n}$)$]\!]\ I_{disc}\ E_{succ}\ I_{fail}\ =$
  ($I$⋈~ $I_{disc}$
    (lambda ($I_{1}\ \ldots I_{n}$) ; Fresh identifiers for components.
      ; Match the component parts of the constructed value.
      $(\mathcal{D}_{\text{pats}}\ [P_{1}, \ldots, P_{n}]\ [I_{1}, \ldots, I_{n}]\ E_{succ}\ I_{fail}))$
    $I_{fail}$)

$\mathcal{D}_{\text{pats}}\ [\,]\ [\,]\ E_{succ}\ I_{fail}\ =\ E_{succ}$
$\mathcal{D}_{\text{pats}}\ (P_{1}\ .\ P_{rest}{*})\ (I_{1}\ .\ I_{rest}{*})\ E_{succ}\ I_{fail}\ =$
  $\mathcal{D}_{\text{pat}}[\![P_{1}]\!]\ I_{1}\ (\mathcal{D}_{\text{pats}}\ P_{rest}{*}\quad I_{rest}{*}\quad E_{succ}\ I_{fail})\ I_{fail}$
```

The $\mathcal{D}_{\text{pat}}$ function processes a tagged list pattern `(`$I\ P_{1}\ \ldots\ P_{n}$`)` by emitting code that invokes the deconstructor associated with I on the discriminant value denoted by I_{disc}, a success expression (call it E_{pats}) constructed by $\mathcal{D}_{\text{pats}}$, and the current failure thunk, denoted by I_{fail}. The E_{pats} expression is constructed by recursively matching the patterns $P_{1} \ldots P_{n}$ against the components denoted by $I_{1} \ldots I_{n}$ relative to the initial success expression E_{succ} and the failure thunk I_{fail}. Observe that I_{fail} is the same for all invocations of $\mathcal{D}_{\text{pat}}$ and $\mathcal{D}_{\text{pats}}$ in the processing of a single `match` clause, and that this I_{fail} denotes the failure thunk that processes the rest of the `match` clauses. This means that should there be any mismatch between the patterns and component values when E_{pats} is evaluated at run-time, I_{fail} will be invoked, terminating the attempt to match the current `match` clause against the discriminant, and starting to match the next `match` clause against the discriminant. On the other hand, if no mismatch is found when E_{pats} is evaluated, then the initial success expression E_{succ} will be evaluated in a context where all pattern variables are bound to the appropriate component values.

As concrete examples of desugaring tagged list patterns, we will study the desugarings of `match` within the `match-ints-1`, and `match-ints-2` procedures presented earlier. Recall that `match-ints-1` was defined as follows:

```
(define (match-ints-1 ints)
  (match ints
    ((cons x (null)) (* x x))
    (_ 17)
    ))
```

Here is a version of match-ints-1 in which the match expression has been desugared:

```
(define (match-ints-1 ints)
  (let ((disc ints))
    (let ((fail1 (lambda ()
                   (let ((fail2 (lambda ()
                                  (error no-match))))
                     17))))
      (cons~ disc
        (lambda (v1 v2)
          (let ((x v1))
            (null~ v2
              (lambda () (* x x))
              fail1)))
        fail1))))
```

If the value denoted by ints and disc is a singleton list, then the cons~ and null~ deconstructors will both succeed, and (* x x) will be evaluated in an environment where x is bound to the single element (denoted by x and v1). If the discriminant is not a singleton list, then one of cons~ or null~ will invoke the failure continuation fail1, which returns the 17 specified in the second clause.

The code generated by the desugarer for match-ints-1 is inefficient in many respects. By making the desugarer cleverer and/or transforming the result of the desugarer by a simple optimizer, it is possible to generate the following more compact and efficient code:

```
(define (match-ints-1 ints)
  (let ((fail1 (lambda () 17)))
    (cons~ ints
      (lambda (v1 v2)
        (null~ v2
          (lambda () (* v1 v1))
          fail1))
      fail1)))
```

As a second example of desugaring tagged list patterns, reconsider match-ints-2:

```
(define (match-ints-2 ints)
  (match ints
    ((cons x (null)) (* x x))
    ((cons 3 (cons y ns)) (+ y (length ns)))
    (_ 17)
    ))
```

The match desugaring functions yield the desugared definition in Figure 10.29. Everything is the same as the desugaring for match-ints-1 except that the failure thunk fail1 now corresponds to matching the second and third clauses of the match within match-ints-2 and the failure thunk fail2 now corresponds to matching the third clause. Note how the desugaring guarantees that the expression (+ y (length ns)) is evaluated in an environment that contains correct bindings for the two names y and ns. Also observe that the second clause pattern (cons 3 (cons y ns)) can fail to match the discriminant for three distinct reasons, all of which cause the invocation of the failure thunk fail2:

1. the discriminant disc is not a pair;

2. the discriminant disc is a pair whose first element v3 is not 3;

3. the discriminant disc is a pair whose first element v3 is 3 but whose second element v4 is not a pair.

In general, a failure thunk is only invoked in two situations: (1) a literal is not equal to the value it is matched against or (2) a deconstructor invokes the failure thunk as its failure continuation when the discriminant does not match the associated constructor.

With the handling of tagged lists, we have completed the presentation of the desugaring of match. Whew! The complete desugaring rules for match are presented in Figure 10.30. Recall that we assume the usual FL desugaring is performed on the expression resulting from the match desugaring.

We have presented an approach to pattern matching based on desugaring and deconstructors. But this is by no means the only way to specify or implement pattern matching. For instance, the dynamic semantics for the core language of SML [MTHM97] treats pattern matching as a fundamental language feature that is explained via operational semantics rules. Whereas the deconstructor-based desugaring requires linearly testing the match clauses one-by-one in order, the SML definition does not imply a particular implementation. Indeed, there are clever implementations of ML pattern matching that can greatly reduce the number of tests that need to be performed [JM88].

```
(define (match-ints-2 ints)
  (let ((disc ints))
    (let ((fail1
            (lambda ()
              (let ((fail2
                      (lambda ()
                        (let ((fail3 (lambda ()
                                       (error no-match))))
                          17))))
                (cons~ disc
                  (lambda (v3 v4)
                    (if (equal? v3 3)
                        (cons~ v4
                          (lambda (v5 v6)
                            (let ((y v5))
                              (let ((ns v6))
                                (+ y (length ns)))))
                          fail2)
                        (fail2))))
                  fail2))))
      (cons~ disc
        (lambda (v1 v2)
          (let ((x v1))
            (null~ v2
              (lambda () (* x x))
              fail1)))
        fail1))))
```

Figure 10.29: The result of desugaring `match` in `match-ints-2`.

```
$\mathcal{D}_{\text{match}}[\![$(match $E_{disc}$ ($P_1$ $E_1$) ... ($P_n$ $E_n$))$]\!]$ =
  (let (($I_{disc}$ $E_{disc}$)) ; $I_{disc}$ fresh
    ($\mathcal{D}_{\text{clauses}}$ $[P_1,\ldots,P_n]$ $[E_1,\ldots,E_n]$ $I_{disc}$))

$\mathcal{D}_{\text{clauses}}$ [] [] $I_{disc}$ =(error no-match)
$\mathcal{D}_{\text{clauses}}$ ($P_1$ . $P_{rest}$*) ($E_1$ . $E_{rest}$*) $I_{disc}$ =
  (let (($I_{fail}$ ; $I_{fail}$ fresh
            (lambda () ; Failure thunk: if $P_1$ doesn't match, try the other clauses
              ($\mathcal{D}_{\text{clauses}}$ $P_{rest}$*   $E_{rest}$*   $I_{disc}$))))
    ($\mathcal{D}_{\text{pat}}[\![P_1]\!]$ $I_{disc}$ $E_1$ $I_{fail}$))

$\mathcal{D}_{\text{pat}}[\![L]\!]$ $I_{disc}$ $E_{succ}$ $I_{fail}$ = (if (equal? $I_{disc}$ $L$) $E_{succ}$ ($I_{fail}$))
$\mathcal{D}_{\text{pat}}[\![\_]\!]$ $I_{disc}$ $E_{succ}$ $I_{fail}$ = $E_{succ}$
$\mathcal{D}_{\text{pat}}[\![I]\!]$ $I_{disc}$ $E_{succ}$ $I_{fail}$ = (let (($I$ $I_{disc}$)) $E_{succ}$)
$\mathcal{D}_{\text{pat}}[\![$($I$ $P_1$ ... $P_n$)$]\!]$ $I_{disc}$ $E_{succ}$ $I_{fail}$ =
  ($I$⋈~ $I_{disc}$
    (lambda ($I_1$ ...$I_n$) ; Fresh identifiers for components.
      ; Match the component parts of the constructed value.
      ($\mathcal{D}_{\text{pats}}$ $[P_1,\ldots,P_n]$ $[I_1,\ldots,I_n]$ $E_{succ}$ $I_{fail}$))
    $I_{fail}$)

$\mathcal{D}_{\text{pats}}$ [] [] $E_{succ}$ $I_{fail}$ = $E_{succ}$
$\mathcal{D}_{\text{pats}}$ ($P_1$ . $P_{rest}$*) ($I_1$ . $I_{rest}$*) $E_{succ}$ $I_{fail}$ =
  $\mathcal{D}_{\text{pat}}[\![P_1]\!]$ $I_1$ ($\mathcal{D}_{\text{pats}}$ $P_{rest}$* $I_{rest}$* $E_{succ}$ $I_{fail}$) $I_{fail}$
```

Figure 10.30: The final version of the match desugaring

▷ **Exercise 10.15** Define the free identifiers of a `match` expression directly (i.e., without desugaring it). ◁

▷ **Exercise 10.16** Extend the `match` desugaring to directly handle record and oneof patterns. As an example of such patterns, consider the following alternative definition of the perimeter procedure.

```
(define (perim shape)
  (match perim
    ((one square (record (side s)))
     (* 4 s))
    ((one rectangle (record (width w) (height h)))
     (* 2 (+ w h)))
    ((one triangle (record (side1 s1) (side2 s2) (side3 s3)))
     (+ s1 (+ s2 s3)))
    ))
```

◁

▷ **Exercise 10.17** Extend the `match` desugaring to handle `list` patterns like those in the following procedure:

```
(define (match-list ints)
  (match ints
    ((list x) (+ x 1))
    ((list _ y) (* 2 y)
    ((list x y 3) (* x y))
    (_ 0)
    ))
```

For example:

`(match-list (list))` $\xrightarrow[FL]{}$ `0`
`(match-list (list 4))` $\xrightarrow[FL]{}$ `5`
`(match-list (list 7 8))` $\xrightarrow[FL]{}$ `16`
`(match-list (list 5 4 3))` $\xrightarrow[FL]{}$ `20`
`(match-list (list 3 4 5))` $\xrightarrow[FL]{}$ `0`
`(match-list (list 1 2 3 4))` $\xrightarrow[FL]{}$ `0` ◁

▷ **Exercise 10.18** Consider the following procedure for removing duplicates from a sorted list of integers:

```
(define (remove-dups sorted-list)
  (match sorted-list
    ((cons x (cons y zs))
     (if (= x y)
         (remove-dups (cons y zs))
         (cons x (remove-dups (cons y zs)))))
    (_ sorted-list)
    ))
```

Matching with nested tagged list patterns helps to extract the first two elements (`x` and `y`) of a list with at least two elements. But it is inelegant to name the remainder of such a list (`zs`) and to rebuild the tail of `sorted-list` via `(cons y zs)`.

One way to avoid these problems is to use nested `match` constructs:

```
(define (remove-dups-2 sorted-list)
  (match sorted-list
    ((cons x ys)
     (match ys
       ((cons y _)
        (if (= x y)
            (remove-dups ys)
            (cons x (remove-dups ys))))
       (_ sorted-list)))
    (_ sorted-list)
    ))
```

But this is verbose and requires duplication of the last `match` clause.

A more elegant approach is to introduce **named patterns** of the form `(<->` *I* *P*`)`. When such a pattern is matched against a value *v*:

- if *P* matches *v*, then `(<->` *I* *P*`)` also matches *v*, and the environment is extended with a binding between *I* and *v* as well as with any bindings implied by the match of *P* against *v*;

- if *P* does not match *v*, then `(<->` *I* *P*`)` does not match *v*.

For example, with named patterns, `remove-dups` can be elegantly expressed as:

```
(define (remove-dups-3 sorted-list)
  (match sorted-list
    ((cons x (<-> ys (cons y _)))
     (if (= x y)
         (remove-dups ys)
         (cons x (remove-dups ys))))
    (_ sorted-list)
    ))
```

Extend the `match` desugaring to handle named patterns and show the result of your extended `match` desugaring for `remove-dups-3`. ◁

▷ **Exercise 10.19** Modify the `match` desugaring functions and/or define a post-desugaring optimizer to make the desugared code more compact and efficient. You should handle at least the following optimizations:

- Optimize unnecessary renamings of the form `(let ((`I_1 I_2`)) ...)`. E.g., the expression `(let ((x v1)) (* x x))` should be replaced by `(* v1 v1)`.
- Eliminate the creation of failure thunks that are never used. E.g., the expression `(let ((fail (lambda ()` E_1`)))` E_2`)` should be replaced by E_2 if `fail` is not free within E_2.
- Eliminate the naming of failure thunks that are referenced only once. The single reference should be replaced by the `lambda` expression itself. E.g., the expression

  `(let ((fail` E_1`)) (cons~` E_2 E_3 `fail))`

  should be replaced by `(cons~` E_2 E_3 E_1`)`.
- Optimize the invocation of a manifest thunk. E.g., `((lambda ()` E`))` should be replaced by E. ◁

10.5.3 Views

While the deconstructor-based desugaring of pattern matching may be inherently inefficient compared to other approaches, it provides an important advantage in expressiveness for the programmer. In languages like ML and Haskell, sum-of-product datatypes can only be deconstructed by referencing the constructor in a pattern context. But using `match`, programmers can define arbitrary deconstructors from scratch and use them in patterns.

As an example, consider the `snoc`[6] procedure, which postpends an element to the back of a list:

```
(define (snoc xs x)
  (if (null? xs)
      (list x)
      (cons (car xs) (snoc (cdr xs) x))))
```

It is often handy to have a deconstructor corresponding to `snoc` that decomposes a non-empty list L into two values: the list of all elements in L excluding the last, and the last element L. This can be expressed with the following deconstructor[7]:

[6] So-called because it is a "backwards `cons`."

[7] An alternative approach to defining `snoc~` would be to express it in terms of two auxiliary procedures, one of which returns all but the last element of a non-empty list and the other of which returns the last element of a non-empty list. In such a definition, `snoc~` would walk over the given list twice. The definition given above effectively uses the success continuation to return multiple values and only walks over the given list once.

```
(define (snoc~ xs succ fail)
  (if (null? xs)
      (fail)
      (if (null? (cdr xs))
          (succ nil (car xs))
          (snoc~ (cdr xs)
                 (lambda (but-last last)
                   (succ (cons (car xs) but-last) last))
                 (lambda () (error cant-fail))))))
```

For example:

```
(snoc~ (list 1 2 3)
  (lambda (ns n) (cons n ns))
  (lambda () nil)) $\xrightarrow[FL]{}$ $[\mathit{3},\ \mathit{1},\ \mathit{2}]$
```

Because of the way the `match` desugaring is defined, it is possible to invoke `snoc~` by referencing `snoc` in a pattern context. For example, here is a compact definition of a quadratic time list reversal procedure using `snoc~` via pattern matching:

```
(define (reverse xs)
  (match xs
    ((null) xs)
    ((cons _ (null)) xs)
    ((snoc ys y) (cons y (reverse ys)))))
```

The ability to choose from multiple deconstructors when decomposing a data structure characterizes what is known as a **views facility**, so-called because it allows a compound data value to be viewed from different perspectives depending on the context [Wad87]. For example, among the many possible views of a non-empty length-n list are

- the `cons` view: the list is the first element prepended onto a list containing elements 2 through n.

- the `snoc` view: the list is the nth element postpended onto the sublist containing the elements 1 through $(n - 1)$.

- the `split` view: a list is the result of appending a left sublist (elements 1 through $\lceil n/2 \rceil$) and a right sublist (elements $\lceil n/2 \rceil + 1$ through n).

- the `interleave` view: a list is the result of interleaving a list containing all the odd-indexed elements with a list containing all the even-indexed elements.

- the `join` view: the list is the result of sandwiching element $\lceil n/2 \rceil$ between a left sublist (elements 1 through $\lceil n/2 \rceil - 1$) and a right sublist (elements $\lceil n/2 \rceil + 1$ through n).

These views show up in many standard list algorithms. For instance, the `interleave` (or `split`) view is at the heart of a mergesort algorithm for sorting lists:

```
(define (mergesort nums)
  (match nums
    ((null) nums)
    ((cons _ (null)) nums)
    ((interleave ms ns) ; Could decompose with split as well
     (merge (mergesort ms) ; merge left as an exercise
            (mergesort ns)))))
```

In addition to allowing compound data to be decomposed via pattern matching in different ways in different contexts, the views facility provided by user-defined deconstructor procedures helps to overcome a key drawback of ML and Haskell style pattern matching: the lack of abstraction in patterns. While such patterns are wonderful for concisely specifying algorithms that manipulate sum-of-product data, the fact that they expose concrete implementation details hinders program development by making it difficult to change the implementation of data abstractions.

As an example of the sort of flexibility lost with ML-style patterns, consider a simple implementation of binary trees with integers stored in the nodes:

```
(define (node num left right) (product num left right))
(define (leaf) unit)
(define (leaf? t) (unit? t))
(define (val t) (proj 1 t))
(define (left t) (proj 2 t))
(define (right t) (proj 3 t))
```

Given these basic tree manipulation primitives, we can define many other tree procedures. For example:

```
(define (sum t)
  (if (leaf? t)
      0
      (+ (val t)
         (+ (sum (left t))
            (sum (right t))))))

(define (height t)
  (if (leaf? t)
      0
      (+ 1 (max (height (left t))
                (height (right t))))))
```

Suppose we wish to modify this implementation so that each node additionally keeps track of its height. This can be accomplished with only minor changes:

```
(define (node num left right)
  (product num left right
           (+ 1 (max (height left)
                     (height right)))))

(define (height t) (proj 4 t))
```

No other changes need to be made. In particular, procedures like `sum` that do not use the height remain unchanged.

Now instead suppose that we used sum-of-products data and pattern matching to implement the initial version of trees, where nodes did not maintain their height (Figure 10.31).

Let's now modify the nodes so that they maintain a height component. If we want `node` to remain a three-argument procedure, in an ML-style system, we must give a different name (say, `hnode`) to the constructor that takes a fourth argument, the height. In every pattern that uses `node`, we must change the constructor name to `hnode` and add an extra pattern to account for the height component (Figure 10.32).

It might seem easy to make these changes. But suppose we have hundreds of tree procedures in our program that needed to be changed in this manner. It would be tedious and error-prone to make the change everywhere — so much so that we might avoid making such representation changes. The concrete nature of ML-style patterns thus stands in the way of a software engineering principle that dictates that programming languages should be designed in such a way to facilitate changing representations.

A view mechanism like explicit deconstructors addresses this issue. When

```
(define-data int-tree
  (leaf)
  (node val left right))

(define (sum t)
  (match t
    ((leaf) 0)
    ((node v l r) (+ v (sum l) (sum r)))))

(define (height t)
  (match t
    ((leaf) 0)
    ((node v l r)
     (+ 1 (max (height l) (height r))))))
```

Figure 10.31: Integer binary trees expressed via `define-data` and `match`.

we introduce `hnode`, in addition to defining a new `node` procedure that has the same meaning as the old `node` constructor, we can also define a new `node~` deconstructor:

```
(define (node~ val succ fail)
  (match val
    ((leaf) (fail))
    ((hnode v l r h) (succ v l r))))
```

With this deconstructor, the original definition of `sum` that used `node` in its `match` clause need not be modified even though the representation of nodes has changed. In this way, user-defined deconstructors (and view facilities in general) facilitate representation changes to programs.

▷ **Exercise 10.20** Define the list deconstructors `split~`, `interleave~`, and `join~` described in the discussion on views. Give examples of algorithms where such views are helpful. ◁

▷ **Exercise 10.21** Define a `partition~` deconstructor for a non-empty list of integers L that decomposes it into three parts:

a. the first element of L (known as the **pivot**);

b. a list of all elements in the tail of L less than or equal to the pivot (with the same relative order as in L);

c. a list of all elements in the tail of L that are strictly greater than the pivot (with the same relative order as in L).

```
(define-data int-tree
  (leaf)
  (hnode val left right height))

(define (node v l r)
  (+ 1 (max (height l) (height r))))

(define (sum t)
  (match t
    ((leaf) 0)
    ((hnode v l r _) (+ v (sum l) (sum r)))))

(define (height t)
  (match t
    ((leaf) 0)
    ((hnode v l r h) h)))
```

Figure 10.32: Adding a height component requires changing all `node` patterns.

Using your `partition˜`, it should be possible to define the quicksort algorithm for sorting lists:

```
(define (quicksort nums)
  (match nums
    ((null) nums)
    ((cons _ (null)) nums)
    ((partition pivot lesses greaters)
     (append (quicksort lesses)
             (cons pivot (quicksort greaters))))))
```

◁

▷ **Exercise 10.22** The convention of naming deconstructors by extending the constructor name with the suffix "~" is really just a crude but simple way of associating a deconstructor with a constructor. Here we consider an alternative way to specify this association.

Suppose that FL is extended with a declaration construct, `define-constructor`, that associates a constructor name with *two* procedures: a constructor and its associated deconstructor. Using this construct, new list constructors `kons` and `knull` could be specified as follows:

```
(define-constructor kons
  (lambda (elt lst) (pair elt lst)) ; Constructor
  (lambda (val succ fail)           ; Deconstructor
    (if (pair? val)
        (succ (left pair) (right pair))
        (fail)))
  )

(define-constructor knull
  (lambda () #u)              ; Constructor
  (lambda (val succ fail)     ; Deconstructor
    (if (unit? val) (succ) (fail)))
  )
```

The intention is that the name declared by `define-constructor` can be used within expressions to denote the constructor procedure and within patterns to denote the deconstructor procedure. Sometimes it is necessary to access to the deconstructor procedure within an expression; for this case, FL is also extended with a new expression (`decon` *I*) that accesses the "deconstructor part" of *I*. For example:

```
(kons 1 (kons 2 (knull))) $\xrightarrow[FL]{}$ $[1,\ 2]$

(match (kons 1 (kons 2 (knull)))
  ((kons x (kons y (knull))) (+ x y))) $\xrightarrow[FL]{}$ $3$

((decon kons) (kons 1 (kons 2 (knull)))
   (lambda (hd tl) (kons hd (kons hd tl)))
   (lambda () (kons 5 (knull)))) $\xrightarrow[FL]{}$ $[1,\ 1,\ 2]$

((decon kons) (knull)
   (lambda (hd tl) (kons hd (kons hd tl)))
   (lambda () (kons 5 (knull)))) $\xrightarrow[FL]{}$ $[5]$
```

The `match` desugaring for this extended version of FL is the same as before except that within $\mathcal{D}_{\text{pat}}$, the occurrence of $I\bowtie^{\sim}$ is replaced by (`decon` *I*).

a. One way to model the semantics of (`define-constructor` I E_1 E_2) is to say that it binds the name I to the *pair* of values that result from evaluating E_1 and E_2. Extend the denotational semantics of FL to reflect this model, and explain (1) the meaning of `define-constructor`, (2) the invocation of constructors, and (3) the semantics of `decon`.

b. Another way to model the semantics of (`define-constructor` I E_1 E_2) is to say that the extended version of FL has *two* namespaces: one for "normal" values (including constructors) and one for deconstructors. Extend the denotational semantics of FL to reflect this model, and explain (1) the meaning of `define-constructor`, (2) the invocation of constructors, and (3) the semantics of `decon`.

c. What are the benefits and drawbacks of using `define-constructor` and `decon` vs. the convention of naming deconstructors with a `~`? ◁