Chapter 15

Abstract Types

The human heart has hidden treasures,
In secret kept, in silence sealed.

— Evening Solace, Charlotte Bronte

15.1 Data Abstraction

A cornerstone of modern programming methodology is the principle of data abstraction, which states that programmers should be able to use data structures without understanding the details of how they are implemented. Data abstraction is based on establishing a contract, also known as an application programming interface (API), or just interface, that specifies the abstract behavior of all operations that manipulate a data structure without describing the representation of the data structure or the algorithms used in the operations.

The contract serves as an abstraction barrier that separates the concerns of the two parties that participate in a data abstraction. On one side of the barrier is the implementer, who is responsible for implementing the operations so that they satisfy the contract. On the other side of the barrier is the client, who is blissfully unaware of the hidden implementation details and uses the operations based purely on their advertised specifications in the contract. This arrangement gives the implementer the flexibility to change the implementation at any time as long as the contract is still satisfied. Such changes should not require the client to modify any code.\footnote{However, the client may need to recompile existing code in order to use a modified implementation.} This separation of concerns is especially
useful when large programs are being developed by multiple programmers, many of whom may never communicate except via contracts. But it is even helpful in programs written by a single person who plays the roles of implementer and client at different times in the programming process.

15.1.1 A Point Abstraction

As an extremely simple example of data abstraction, consider an abstraction for points on a two-dimensional grid. The point abstraction is defined by the following contract, which specifies an operation for creating a point from its two coordinates and operations for extracting each coordinate:

- \((\text{make-pt } x \ y)\): Create a point whose \(x\) coordinate is the integer \(x\) and whose \(y\) coordinate is the integer \(y\).
- \((\text{pt-x } p)\): Return the \(x\) coordinate of the given point \(p\).
- \((\text{pt-y } p)\): Return the \(y\) coordinate of the given point \(p\).

An implementation of the point abstraction should satisfy the following axioms:

1. For any integers \(n_1\) and \(n_2\), \((\text{pt-x } (\text{make-pt } n_1 \ n_2))\) evaluates to \(n_1\).
2. For any integers \(n_1\) and \(n_2\), \((\text{pt-y } (\text{make-pt } n_1 \ n_2))\) evaluates to \(n_2\).

Even for this simple abstraction, there are a surprising number of possible implementations. For concreteness, below we give two point implementations in the dynamically typed FL language. Our convention will be to package up the operations of a data abstraction into a record, but that is not essential.

```scheme
(define pair-point-impl
 (record
  (make-pt (lambda (x y) (pair x y)))
  (pt-x (lambda (p) (left p)))
  (pt-y (lambda (p) (right p)))))

(define proc-point-impl
 (record
  (make-pt (lambda (x y) (lambda (b) (if b x y))))
  (pt-x (lambda (p) (p #t)))
  (pt-y (lambda (p) (p #f)))))
```

In \texttt{pair-point-impl}, the two coordinates are stored in a pair. Alternatively, we could have stored them in the opposite order or glued them together in a different kind of product (e.g., array, record, or list). In \texttt{proc-point-impl}, a
point is represented as a first-class procedure that “remembers” the coordinates in its environment and uses a boolean argument to determine which coordinate to return when called. Alternatively, some other key (such as a symbol or string message) could be used to select the coordinate.

As a sample client of the point abstraction, consider the following procedure, which, for a given point implementation, defines a coordinate-swapping transpose procedure and a point->pair procedure that converts a point to a concrete pair (regardless of its underlying representation) and uses these on the point (1,2).

\[
\text{(define test-point-impl}
\begin{align*}
&\text{ (lambda (point-impl)}
&\text{ (with-fields (make-pt pt-x pt-y) point-impl}
&\text{ (let ((transpose (lambda (p) (make-pt (pt-y p) (pt-x p))))}
&\text{ (point->pair (lambda (p) (pair (pt-x p) (pt-y p)))))}
&\text{ (point->pair (transpose (make-pt 1 2)))))}
\end{align*}
\text{)}
\]

The result of invoking test-point-impl on a valid point implementation should be the pair value (2, 1).

In this example, there is little reason to prefer one of the implementations over the other. The pair implementation might be viewed as being more straightforward, requiring less memory space, or being more efficient because it requires fewer procedure calls. However, judgments about efficiency are often tricky and require a deep understanding of low-level implementation details. In more realistic examples, such as abstractions for data structures like stacks, queues, priority queues, sets, tables, databases, etc., one implementation might be preferred over another because of asymptotically better running times or memory usage for certain operations.

### 15.1.2 Procedural Abstraction is not Enough

Any language with procedural abstraction can be used to implement data abstraction in the way illustrated in the point example. However, in order for the full benefits of data abstraction to be realized, this approach requires that the client never commit abstraction violations. An abstraction violation is the inappropriate use of abstract values or their operations.

In our implementation of points that uses pairs, the client can inspect the representation of an abstract value and use this knowledge to manipulate abstract values concretely. For instance, if points are represented as pairs, then the client might write \texttt{(left p)} rather than \texttt{(pt-x p)} to extract the x coordinate of a point \texttt{p}, or might create a point “forgery” using \texttt{(pair 1 2)} in place of \texttt{(make-pt 1 2)}. Although these concrete manipulations will not cause errors,
such abuses of the exposed representation are dangerous because they are not
guaranteed to work if the implementation is changed. For example, (left p)
would lead to a runtime type error if the implementation were changed to use
a procedural representation for points, and would give the incorrect value if the
implementation was changed to put the y coordinate before the x coordinate in
a pair.

Furthermore, many representations involve **representation invariants** that
are maintained by the abstract operations but which concrete manipulations may
violate. A representation invariant is a set of conceptual or actual predicates
that a representation must satisfy to be legal. For instance, a string collection
implementation might store the strings in a sorted array. Thus a sorted predicate
would be true for this representation. If the client creates a forgery with an
unsorted array, all bets are off concerning the behavior of the abstract operations
on this forgery.

Without an enforcement of the relationship between abstract values and their
operations, it is even possible to interchange values of different abstractions that
happen to have the same concrete representation. For instance, if an implemen-
tation of a rational number abstraction represents a rational number as a pair
of two integers, then a rational number could be dissected with \(\text{pt-x}\) and \(\text{pt-y}\),
assuming that points are also represented as pairs of integers.

Although our examples have been for a dynamically typed language, the
same problems occur in a statically typed language with structural type equality.
Clearly, attempting to achieve data abstraction using procedural abstraction
alone is fraught with peril. There must additionally be some sort of mechanism
to guarantee that abstract data is **secure**. We will call a language secure when
barriers associated with a data abstraction cannot be violated. Such a security
mechanism must effectively hide the representation of abstract data by making
it illegal to create or operate on abstract values with anything other than the
appropriate abstract operations.

In the remainder of this chapter, we first consider how secure data abstrac-
tions can be achieved dynamically using a lock and key mechanism. Then we
study various ways to achieve such security statically using types.

---

**Exercise 15.1** In languages with first-class procedures, one approach to hiding the
representations of data structures is to encapsulate them in message-passing objects. For
example, the following two point-making procedures encapsulate the pair representation
and procedural representation, respectively:
15.2. Dynamic Locks and Keys

One approach for securely encapsulating a data abstraction representation is to make it inaccessible by “locking” abstract values with a “key” in such a way that only the very same key can unlock a locked value to access the representation. We explore a dynamic lock and key mechanism by extending FL! with the following primitives:

- **(new-key)** generates a unique unforgeable key value.

- **(lock key value)** creates a new kind of “locked value” that pairs key with value in such a way that key cannot be extracted and value can only be extracted by supplying key.

- **(unlock key locked)** returns the value stored in locked if key matches the key used to create locked. Otherwise, signals an error.

We extend FL! rather than FL because the presence of cells and a single-threaded store simplify specifying the semantics of these constructs. Indeed,
new-key, lock, and unlock can all be implemented as user-defined procedures in FL! (Figure 15.1). The new-key procedure creates a new cell whose location is a unique and unforgeable key; the value in the cell is arbitrary and can be ignored. The lock procedure represents a locked value as a procedure that "remembers" the given key and value and only returns the value if it is invoked on the original key (as done in unlock). The procedural representation of locked values prevents direct access to the key, and the value can only be extracted by supplying the key, as desired.

```
(define new-key (lambda () (cell 0)))
(define lock (lambda (key val) (lambda (key1) (if (cell=? key key1) val (error wrong-key)))))
(define unlock (lambda (key locked) (locked key)))
```

Figure 15.1: Implementation of a dynamic lock and key mechanism in FL!.

Figure 15.2 shows how the lock and key mechanism can be used to securely encapsulate two pair representations of points that differ only in the order of the coordinates. The procedures up and down use lock and unlock to mediate between the concrete pair values and the abstract point values. Because all operators for a single implementation use the same key, the operators for pt-impl1 work together, as do those for pt-impl2. For example:

```
((select pt-x pt-impl1) ((select make-pt pt-impl1) 1 2))  \(FL!^1\)
((select pt-y pt-impl2) ((select make-pt pt-impl2) 1 2))  \(FL!^2\)
```

However, because different implementations use different keys, point values created by one of the implementations cannot be dissected by operations of the other. Furthermore, because the operators create and use locked values, neither point implementation can be used with concrete pair operations. For example, all of the following four expressions generate dynamic errors when evaluated:

```
((select pt-x pt-impl1) ((select make-pt pt-impl2) 1 2))
((select pt-y pt-impl2) ((select make-pt pt-impl1) 1 2))
(left ((select make-pt pt-impl1) 1 2))
((select pt-y pt-impl2) (pair 1 2))
```
15.2. DYNAMIC LOCKS AND KEYS

\begin{verbatim}
(define pt-impl1
  (let ((key (new-key)))
    (let ((up (lambda (x) (lock key x)))
          (down (lambda (x) (unlock key x))))
      (record
       (make-pt (lambda (x y) (up (pair x y))))
       (pt-x (lambda (p) (left (down p))))
       (pt-y (lambda (p) (right (down p)))))))

(define pt-impl2
  (let ((key (new-key)))
    (let ((up (lambda (x) (lock key x)))
          (down (lambda (x) (unlock key x))))
      (record
       (make-pt (lambda (x y) (up (pair y x))))
       (pt-x (lambda (p) (right (down p))))
       (pt-y (lambda (p) (left (down p)))))))
\end{verbatim}

Figure 15.2: Using the lock and key mechanism to hide point representations.

Some syntactic sugar can facilitate the definition of implementation records. We introduce a \texttt{cluster} macro that abstracts over the pattern used in the point implementations:

\[
D_{\text{exp}}[(\text{cluster } (I\ E)^*)] = \\
(\text{let } ((I\ key\ (\text{new-key}))); I_{key} \text{ fresh} \\
  (\text{let } ((I\ up\ (\lambda\ (x)\ (\text{lock } I_{key}\ x))))\ \\
   (\text{let } ((I\ down\ (\lambda\ (x)\ (\text{unlock } I_{key}\ x))))\ \\
    (\text{recordrec } (I\ E)^*)))))
\]

The \texttt{up} and \texttt{down} procedures implicitly introduced by the desugaring may be used in any of the cluster bindings. Using \texttt{recordrec} in place of \texttt{record} allows for mutually recursive operations. Here is the definition of \texttt{pt-impl1} re-expressed using the \texttt{cluster} notation:

\begin{verbatim}
(define pt-impl1
  (cluster
    (make-pt (lambda (x y) (up (pair x y))))
    (pt-x (lambda (p) (left (down p))))
    (pt-y (lambda (p) (right (down p)))))))
\end{verbatim}

Note that \texttt{cluster} creates a new data abstraction every time it is evaluated. For instance, consider:
(define make-wrapper
  (lambda ()
    (cluster
      (wrap (lambda (x) (up x)))
      (unwrap (lambda (x) (down x)))))

(define wrapper1 (make-wrapper))
(define wrapper2 (make-wrapper))

Evaluating ((select unwrap wrapper2) ((select wrap wrapper1) 17)) signals a dynamic error because the `wrap` procedure from `wrapper1` and the `unwrap` procedure from `wrapper2` use different keys.

Exercise 15.2 Consider an integer set abstraction that supports the following operations:

- (empty) creates an empty set of integers.
- (insert int intset) returns the set that results from inserting `int` into the integer set `intset`.
- (member? int intset) returns `true` if `int` is a member of the integer set `intset` and `false` otherwise.

a. Define a cluster `list-intset-impl` that represents an integer set as a list of integers without duplicates sorted from low to high.

b. Define a cluster `pred-intset-impl` that represents an integer set as a predicate - a procedure that takes an integer and returns `true` if that integer is in the set represented by the predicate and `false` otherwise.

c. Extend both `list-intset-impl` and `pred-intset-impl` to handle union, intersection, and difference operations on two integer sets.

d. Some representations have advantages over other for implementing particular operations. Show that `size` (which returns the number of elements in an integer set) is easy to implement for `list-intset-impl` but but impossible to implement for `pred-intset-impl` (without changing the representation). Similarly, show that `complement` (which returns the set of all integers not in the given set) is easy to implement for `pred-intset-impl` but impossible to implement for `list-intset-impl`.

Exercise 15.3

a. Extend the SOS for FLK! to directly handle the primitives `new-key`, `lock`, and `unlock`. Assume that the syntactic domains MixedExp and ValueExp are extended with expressions of the form `(*key* L)` to represent keys and `(*locked* L V)` to represent locked values.
15.2. DYNAMIC LOCKS AND KEYS

b. It is helpful to have the following additional primitives as well:

- \( \text{(key? thing)} \) determines if \( \text{key} \) is a key value.
- \( \text{(key=? key1 key2)} \) returns true if \( \text{key1} \) is the same key value as \( \text{key2} \) and returns false otherwise. Signals an error if either \( \text{key1} \) or \( \text{key2} \) is not a key value.
- \( \text{(locked? thing)} \) determines if \( \text{thing} \) is a locked value.

Extend your SOS to handle these primitives.

c. Can you extend the implementation in Figure 15.1 to handle the additional primitives? Explain.

Exercise 15.4 It is not always desirable to export every binding of a cluster in the resulting record. For example, in the following implementation of a rational number cluster, the \( \text{gcd} \) function (which calculates the greatest common divisor of two numbers) is intended to be an unexported local recursive function used by \( \text{make-rat} \).

\[
\text{(define rat-impl}
\text{\hspace{1em} (cluster}
\text{\hspace{2em} (make-rat (lambda (x y))}
\text{\hspace{3em} (let ((g (gcd x y)))}
\text{\hspace{4em} (up (pair (div x g) (div y g))))))}
\text{\hspace{2em} (numer (lambda (r) (left (down r))))}
\text{\hspace{2em} (denom (lambda (r) (right (down r))))}
\text{\hspace{2em} (gcd (lambda (a b))}
\text{\hspace{3em} (if (= b 0) a (gcd b (rem a b))))}
\text{\hspace{1em})}
\]

In this case, we could make the definition of \( \text{gcd} \) local to \( \text{make-rat} \), but this strategy does not work if the local value is used in multiple bindings. Alternatively, we can extend the \( \text{cluster} \) syntax to be:

\[
(\text{cluster} \; (I_{exp} \ast) \; (I \ast))
\]

where \( (I_{exp} \ast) \) is an explicit list of \textit{exports} – those bindings we wish to be included in the resulting record. For instance, if we use \( \text{(make-rat numer denom)} \) as the export list in \text{rat-impl}, then \( \text{gcd} \) would not appear in the resulting record. Modify the desugaring of \text{cluster} to support explicit export lists.

Exercise 15.5 A dynamic lock and key mechanism can be added to a statically typed language like FL/X.

a. Extend the type syntax and typing rules of FL/X to handle \textit{new-key, lock,} and \textit{unlock}. 
b. We can add a cluster form to FL/X using the syntax

\[
(\text{cluster } T_{\text{rep}} \ (I_1 \ T_1 \ E_1) \ldots \ (I_n \ T_n \ E_n)),
\]

where \( T_{\text{rep}} \) is the concrete representation type of the data abstraction and \( T_n \) is the type of \( E_n \). Give a typing rule for this explicitly typed \text{cluster} form.

c. Why is it necessary to include \( T_{\text{rep}} \) and the \( T_i \) in the explicitly typed \text{cluster} form? Would these be necessary in a cluster form for FL/R?

booksectionExistential Types

The dynamic lock and key mechanism enforces data abstraction by signaling a run-time error whenever an abstraction violation is encountered. The main drawback of this approach is its dynamic nature. It would be desirable to have a static mechanism that reports abstraction violations when the program is type checked. As usual, the constraints of computability prevent a static system from detecting exactly those violations that would be caught by a dynamic lock and key mechanism. Nevertheless, by relinquishing some expressive power, it is possible to design type systems that prevent abstraction violations via a static lock and key mechanism known as an abstract type. In the next three sections, we shall study three designs for abstract types.

Our first abstract type system is based on extending the explicitly typed language FL/XSP with existential types. To motivate existential types, consider the types of the pair-point-impl and proc-point-impl implementations introduced in Section 15.1:

\[
\begin{align*}
(\text{define-type pair-point-impl-type} & \\
& \text{recordof} \\
& (\text{make-pt} \ (-\to \ (\text{int} \ \text{int}) \ (\text{pairof} \ \text{int} \ \text{int}))) \\
& (\text{pt-x} \ (-\to \ ((\text{pairof} \ \text{int} \ \text{int})) \ \text{int})) \\
& (\text{pt-y} \ (-\to \ ((\text{pairof} \ \text{int} \ \text{int})) \ \text{int}))
\end{align*}
\]

\[
\begin{align*}
(\text{define-type proc-point-impl-type} & \\
& \text{recordof} \\
& (\text{make-pt} \ (-\to \ (\text{int} \ \text{int}) \ (-\to \ (\text{bool} \ \text{int})))) \\
& (\text{pt-x} \ (-\to \ ((-\to \ (\text{bool} \ \text{int})) \ \text{int})) \\
& (\text{pt-y} \ (-\to \ ((-\to \ (\text{bool} \ \text{int})) \ \text{int}))
\end{align*}
\]

These two types are the same except for the concrete type used to represent an abstract point value: \((\text{pairof} \ \text{int} \ \text{int})\) in the first case and \((-\to \ (\text{bool} \ \text{int})\) in the second. We would like to be able to say that both implementations have the same abstract type. We call values that implement an abstract type a package. To represent the type of a package, we use a new type construct, \text{packof}_{\text{exist}}, to introduce an abstract type name, point, that stands for the concrete type used in a particular implementation:
15.2. DYNAMIC LOCKS AND KEYS

\begin{verbatim}
(define-type pt-eface
 (packof_exist point
 (recordof
   (make-pt (-> (int int) point))
   (pt-x (-> (point) int))
   (pt-y (-> (point) int)))))
\end{verbatim}

We informally read the above \texttt{(packof\_exist \ldots)} type as “there exists a concrete point representation (call it \texttt{point}) such that there are \texttt{make-pt}, \texttt{pt-x}, and \texttt{pt-y} procedures with the specified types that manipulate this representation.” Such a type is called an \textbf{existential type} because it posits the existence of an abstract type and indicates how it is used without saying anything about its concrete representation.\textsuperscript{2} In the following discussion, we will often refer to this particular existential type, so we have given it the name \texttt{pt-eface}, where \texttt{eface} is short for existential interface.

A summary of existential types is presented in Figure 15.3. The form of an existential type is \texttt{(packof\_exist \textit{I} \textit{T})}. The existential variable \textit{I} is a binding occurrence of a type variable whose scope is \textit{T}. The particular name of this variable is irrelevant; as is indicated by the \texttt{[exists=]} type equality rule, it can be consistently renamed without changing the essence of the type. So the type

\begin{verbatim}
(packof_exist q
 (recordof
   (make-pt (-> (int int) q))
   (pt-x (-> (q) int))
   (pt-y (-> (q) int)))))
\end{verbatim}

is equivalent to the existential type using \texttt{point} above.

Values of existential type, which we shall call \textbf{existential packages}, are created by the form \texttt{(pack\_exist \textit{I}_{abs} \textit{T}_{rep} \textit{E}_{impl})}. The type identifier \textit{I}_{abs} is a type name that is used to hide the concrete representation type \textit{T}_{rep} within the type of the implementation expression \textit{E}_{impl}. For example, Figure 15.4 shows two existential packages that implement the type contract specified by \texttt{pt-eface}. In the first package, the abstract name \texttt{point} stands for the type of pair of integers, while in the second package, it stands for the type of a procedure that maps a boolean to an integer. As in the dynamic cluster form studied in Section 15.2, the \texttt{pack\_exist} form implicitly introduces \texttt{up} and \texttt{down} procedures that convert between the concrete and abstract values.

\textsuperscript{2}In the literature, such types are often written with \texttt{\exists} or \texttt{exists} just as \texttt{\forall} and \texttt{forall} are used for universal polymorphism. For example, a more standard syntax for the \texttt{pt-eface} type is: \texttt{\exists point . (make-pt: int\*int → point, pt-x: point → int, pt-y: point → int).}
CHAPTER 15. ABSTRACT TYPES

Syntax

\[
E ::= \ldots \mid \text{pack}_{\text{exist}} I_{\text{abs}} T_{\text{rep}} E_{\text{impl}} \quad \text{[Existential Introduction]}
\]

\[
| \text{unpack}_{\text{exist}} E_{\text{pkg}} I_{\text{ty}} I_{\text{impl}} E_{\text{body}} \quad \text{[Existential Elimination]}
\]

\[
T ::= \ldots \mid \text{packof}_{\text{exist}} I_{\text{abs}} T_{\text{impl}} \quad \text{[Existential Type]}
\]

Type Rules

\[
A \uparrow\downarrow (\rightarrow (T_{\text{rep}}) I_{\text{abs}}),\downarrow \downarrow (\rightarrow (I_{\text{abs}}) T_{\text{rep}})] E_{\text{impl}} : T_{\text{impl}} \quad \text{epack}
\]

\[
A \vdash (\text{pack}_{\text{exist}} I_{\text{abs}} T_{\text{rep}} E_{\text{impl}}) : (\text{packof}_{\text{exist}} I_{\text{abs}} T_{\text{impl}})
\]

where \( I_{\text{abs}} \notin \{(\text{FTV} A(I)) I \in \text{FreeIds}[E_{\text{impl}}]\} \quad \text{[import restriction]}

\[
A \vdash E_{\text{pkg}} : (\text{packof}_{\text{exist}} I_{\text{abs}} T_{\text{impl}})
\]

\[
A[I_{\text{impl}}\downarrow I_{\text{ty}}/I_{\text{abs}}T_{\text{impl}}] E_{\text{body}} : T_{\text{body}} \quad \text{epunpack}
\]

where \( I_{\text{ty}} \notin \{(\text{FTV} A(I)) I \in \text{FreeIds}[E_{\text{body}}]\} \quad \text{[import restriction]}

\[
I_{\text{ty}} \notin \{(\text{FTV} T_{\text{body}}) \quad \text{[export restriction]}
\]

Type Equality

\[
(\text{packof}_{\text{exist}} I T) = (\text{packof}_{\text{exist}} I' [I'/I] T) \quad \text{exists=}
\]

Type Erasure

\[
[(\text{pack}_{\text{exist}} I_{\text{abs}} T_{\text{rep}} E_{\text{impl}})]
\]

\[
= (\text{let } ((\text{up } (\text{lambda } (x) x))
\]

\[
(\text{down } (\text{lambda } (x) x))))
\]

\[
[E_{\text{impl}})]
\]

\[
[(\text{unpack}_{\text{exist}} E_{\text{pkg}} I_{\text{ty}} I_{\text{impl}} E_{\text{body}})] = (\text{let } ((I_{\text{impl}} [E_{\text{pkg}}]) [E_{\text{body}}])
\]

Figure 15.3: The essence of existential types in FL/XSP.
15.2. DYNAMIC LOCKS AND KEYS

\begin{verbatim}
(define pair-point-epkg pt-eface
  (pack exist point (pairof int int)
    (record
      (make-pt (lambda ((x int) (y int)) (up (pair x y))))
      (pt-x (lambda ((p point)) (left (down p))))
      (pt-y (lambda ((p point) (right (down p)))))))))
(define proc-point-epkg pt-eface
  (pack exist point (-> (bool) int)
    (record
      (make-pt (lambda ((x int) (y int))
        (up (lambda ((b bool)) (if b x y))))
      (pt-x (lambda ((p point)) ((down p) true)))
      (pt-y (lambda ((p point) ((down p) false))))))))
\end{verbatim}

Figure 15.4: Two existential packages that implement \texttt{pt-eface}.

The \texttt{[epack]} type rule in Figure 15.3 specifies how different implementations can have exactly the same existential type. The implementation expression \( E_{\text{impl}} \) is checked in a type environment where \texttt{up} converts from the concrete representation type \( T_{\text{rep}} \) to the abstract type name \( I_{\text{abs}} \) and \texttt{down} converts from \( I_{\text{abs}} \) to \( T_{\text{rep}} \). These conversions allow the implementer to hide the concrete representation type with an \texttt{opaque} type name, so called because the concrete type cannot be “seen” through the name, even though the name is an abbreviation for the concrete type.\footnote{\texttt{up} and \texttt{down} are just one way to distinguish concrete and abstract types in an existential type. Some alternative approaches are explored in Exercise 15.9.} If type checking of a \texttt{pack exist} expression succeeds, we have a proof that there is at least one representation for \( I_{\text{abs}} \) (namely \( T_{\text{rep}} \)) and one implementation using this representation (namely \( E_{\text{impl}} \)) that satisfies the implementation type \( T_{\text{impl}} \). This knowledge is recorded with the type \( \texttt{packof exist I_{abs} T_{impl}} \), in which any implementation details related to \( T_{\text{rep}} \) and \( E_{\text{impl}} \) have been purposely omitted.

The \texttt{pack exist} expression can be viewed as a way to package up an implementation in such a way that representation details are hidden. As indicated by the type erasure for \texttt{pack exist} in Figure 15.3, the dynamic meaning of a \texttt{pack exist} expression is just the implementation expression in a context where \texttt{up} and \texttt{down} are identity operations. The remaining parts of the expression (\( I_{\text{abs}} \) and \( T_{\text{rep}} \)) are just type annotations whose purpose is to specify the existential type.

The existential elimination form, \( \texttt{unpack exist E_{pkg} I_{ty} I_{impl} E_{body}} \), is the means of using the underlying implementation hidden by an existential package.
The type erasure of this expression — \((\text{let} \ ((\text{I impl} \ [E_{pkg}])) \ [E_{body}])\) — indicates that the dynamic meaning of this expression is simply to give the name \(I_{impl}\) to the implementation in the scope of the body. The type name \(I_{ty}\) serves as a local name for the abstract type of the existential package that can be used within \(E_{body}\). The abstract name within the existential type itself is unsuitable for this purpose because (1) it is not lexically apparent to \(E_{body}\) and (2) it is a bound name that is subject to renaming.

As an example of \(\text{unpack}_{\text{exist}}\), consider the following procedure, which is a typed version of the \(\text{test-point-impl}\) procedure presented in Section 15.1.

\[
(\text{define} \ \text{test-point-epkg} \ (\rightarrow \ (\text{pt-eface}) \ (\text{pairof} \ \text{int} \ \text{int}))) \\
(\lambda (\text{point-epkg} \ \text{pt-eface})) \\
(\text{unpack}_{\text{exist}} \ \text{point-epkg} \ \text{pt} \ \text{point-ops} \\
(\text{with} \ \text{point-ops} \\
(\text{let} \ ((\text{transpose} \ (\lambda (\text{p pt})) \\
(\text{make-pt} \ (\text{pt}-y \ \text{p}) \ (\text{pt}-x \ \text{p})))) \\
(\text{point}->\text{pair} \ (\lambda (\text{p pt})) \\
(\text{pair} \ (\text{pt}-x \ \text{p}) \ (\text{pt}-y \ \text{p})))) \\
(\text{point}->\text{pair} \ (\text{transpose} \ (\text{make-pt} \ 1 \ 2))))))))
\]

The \(\text{point-epkg}\) argument to \(\text{test-point-epkg}\) is any existential package with type \(\text{pt-eface}\). The \(\text{unpack}_{\text{exist}}\) form gives the local name \(\text{pt}\) to the abstract point type and the local name \(\text{point-ops}\) to the implementation record containing the \(\text{make-pt}\), \(\text{pt}-x\), and \(\text{pt}-y\) procedures. In the context of local bindings for these procedures (made available by \(\text{with}\)), the local \(\text{transpose}\) and \(\text{point}->\text{pair}\) procedures are created. Each of these takes a point as an argument and so must refer to the local abstract type name \(\text{pt}\) for the abstract point type. Finally, \(\text{test-point-epkg}\) returns a pair of the swapped coordinates for the point \((1,2)\).

In the \(\text{[unpack]}\) type rule, it is assumed that the package expression \(E_{pkg}\) has type \((\text{packof}_{\text{exist}} \ I_{abs} \ T_{impl})\). The body expression \(E_{body}\) is type checked under the assumption that \(I_{impl}\) has as its type a version of \(T_{impl}\) in which the bound name \(I_{abs}\) has been replaced by the local abstract type name \(I_{ty}\). For instance, in the above \(\text{unpack}_{\text{exist}}\) example, where \(\text{pt}\) is the local abstract type name, the \(\text{make-pt}\) procedure has type \((\rightarrow \ (\text{int} \ \text{int}) \ \text{pt})\). The fact that the result type is \(\text{pt}\) rather than \(\text{point}\) is essential for matching up the return type of \(\text{transpose}\) and the declared argument type of \(\text{point}->\text{pair}\).

In the \(\text{[epack]}\) rule, there is an \textbf{import restriction} on the abstract type name \(I_{abs}\) that prevents it from accidentally capturing a type identifier mentioned in the type of a free variable in \(E_{impl}\). Here is an expression that would unsoundly be declared well-typed without this restriction:
15.2. DYNAMIC LOCKS AND KEYS

\[
(\text{plambda } (t) \\
(\text{lambda } ((z t)) \\
(\text{pack}_{\text{exist}} t \text{ int } (\text{down } z))))
\]

The application \((\text{down } z)\) applies the \text{down} procedure to a value \(z\) of arbitrary type \(t\). But since \text{down} has type \((\to (t) \text{ int})\), where \(t\) abstracts over the concrete type \text{int}, this application would unsoundly be declared well-typed without the import restriction. A similar import restriction is also needed in the \([\text{eunpack}]\) rule. The import restriction is not a serious issue for programmers because it can be satisfied by automatically \(\alpha\)-renaming a program to give distinct names to logically distinct type identifiers.

In contrast, the \textbf{export restriction} \(I_y \notin (FTV \ T_{\text{body}})\) in the \([\text{eunpack}]\) rule can be a serious impediment. This restriction says that the local abstract type name \(I_y\) is not allowed to escape the scope of the \text{unpack}_{\text{exist}} expression by appearing in the type \(T_{\text{body}}\) of the body expression \(E_{\text{body}}\). A consequence is that no value of the abstract type can escape from \text{unpack}_{\text{exist}} in any way.

Without the export restriction, the \([\text{eunpack}]\) rule would be unsound. Consider the following example of what would go wrong if the restriction were removed:

\[
\text{(let ((p (unpack}_{\text{exist}} \text{proc-point-epkg } t \text{ point-ops1} \\
\quad (\text{with point-ops1 (make-pt 1 2)))))} \\
\text{(f (unpack}_{\text{exist}} \text{pair-point-epkg } t \text{ point-ops2} \\
\quad (\text{with point-ops2 pt-x}))))} \\
\text{(f p)).}
\]

The first \text{unpack}_{\text{exist}} makes a procedural point whose type within the \text{unpack}_{\text{exist}} is the local abstract type \(t\). This point escapes from the \text{unpack}_{\text{exist}} and is \text{let-bound} to the name \(p\). The type of the point at this time is still \(t\), which is an unbound type variable in this context. The second \text{unpack}_{\text{exist}} unpackages a pair point implementation and returns its \text{pt-x} operation, which is renamed \(f\). Since \(t\) is also used as the local abstract type in the second \text{unpack}_{\text{exist}}, the type of \(f\) is \((\to (t) \text{ int})\), where \(t\) again is actually an unbound type variable. Since \(f\) has type \((\to (t) \text{ int})\) and \(p\) has type \(t\), the application \((f \ p)\) would be well-typed. But dynamically an attempt is being made to take the left component of a procedural point, which should be a type error! This example makes clear that while it is powerful to be able to locally name the abstract type within \text{unpack}_{\text{exist}}, the local type name has no meaning outside the scope of the \text{unpack}_{\text{exist}} and so cannot be allowed to escape.

The export restriction fundamentally limits the usefulness of existential types in practice. For instance, in the \texttt{test-point-epkg} procedure studied above, it would be more natural to return the transposed point directly, but then
the type of the `unpack` expression would be the abstract type `pt`, which is forbidden by the export restriction. Instead we must first convert the abstract point to a concrete pair in order to satisfy the export restriction. The restriction also prevents us from writing a `make-transpose` procedure that takes a point package and returns a `transpose` procedure appropriate for that package. The type of `make-transpose` would presumably be something like `\((\rightarrow \text{pt-eface}) \rightarrow (I_{\text{abs}} I_{\text{abs}}))\)`, where `I_{\text{abs}}` is the name of the abstract type used by the given point package. But there is no way to refer to that type except within `unpack` expressions inside the body of `make-transpose`, and that type cannot escape any such expressions to end up in the result type of `make-transpose`.

In practice, there are a few ways to finesse the export type restriction. One approach is to organize programs in such a way that large regions of the program are within the body of `unpack` expressions that open up commonly used data abstractions. Within these large regions, it is possible to freely manipulate values of the abstract type. The problem with this approach is that it can make it more difficult to take advantage of one of the key benefits of existential types: the ability to abstract code over different implementations of the same abstract type and choose implementations at run-time based on dynamic conditions.

In cases where we really want to pass values that mention the abstract type outside the scope of an `unpack`, we can program around the restriction by packaging up such values together with their abstract type into a new existential type. For example, Figure 15.5 shows how to define an `extend-point-epkg` procedure that can take any package with type `pt-eface` and return a new package that has new operations and values in addition to the old ones. While this technique addresses the problem, it can be cumbersome, especially since all values mentioning the same abstract type must always be put together into the same package (or else later they could not be used with each other). Furthermore, the components of the original package need to be repackaged to get the right abstract type (and satisfy the import restriction).\(^4\)

One paradigm in which the packaging overhead is not too onerous is a simple form of object-oriented programming. Figure 15.6 shows how the pair and procedural point representations can be encapsulated as existential packages whose implementations combine the state and methods of an object. As shown in the figure, in this paradigm, it is possible to express a generic top-level `transpose` method that operates on any value with type `point-object`. For example, the following expression is well-typed:

\(^4\)This is an artifact of using `up/down` to convert between abstract and concrete types. Such repackaging is not necessary in some other approaches; see Exercise 15.9.
### 15.2. Dynamic Locks and Keys

<table>
<thead>
<tr>
<th><code>define-type new-pt-eface</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(packof\textit{exist} point)</code></td>
</tr>
<tr>
<td><code>(recordof)</code></td>
</tr>
<tr>
<td><code>(make-pt (\to (int \to int) \textit{point}))</code></td>
</tr>
<tr>
<td><code>(pt-x (\to \textit{point} \to int))</code></td>
</tr>
<tr>
<td><code>(pt-y (\to \textit{point} \to int))</code></td>
</tr>
<tr>
<td><code>(transpose (\to \textit{point} \to \textit{point}))</code></td>
</tr>
<tr>
<td><code>(point-&gt;pair (\to \textit{point} \to \text{pairof int int}))</code></td>
</tr>
<tr>
<td><code>(origin point)</code></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><code>(define extend-point-epkg (\to (\textit{pt-eface}) new-pt-eface) (lambda ((point-epkg pt-eface))) (unpack\textit{exist} point-epkg pt point-ops (with point-ops (pack\textit{exist} newpt pt (record (make-pt (lambda ((x int) (y int)) (up (make-pt x y)))) (pt-x (lambda ((p newpt)) (pt-x (down p)))) (pt-y (lambda ((p newpt)) (pt-y (down p)))) (transpose (lambda ((p newpt)) (up (make-pt (pt-y (down p)) (pt-x (down p))))) (point-&gt;pair (lambda ((p newpt)) (pair (pt-x (down p)) (pt-y (down p)))) (origin (up (make-pt 0 0)))))))</code></td>
</tr>
</tbody>
</table>

Figure 15.5: The `extend-point-epkg` procedure shows how values mentioning an abstract type can be passed outside `unpack\textit{exist}` as long as they are first packaged together with their abstract type.
(let ((points (list point-object
  (make-pair-point 1 2)
  (make-proc-point 3 4))))
  ((pcall append point-object)
   points
  ((pcall map point-object point-object) transpose points)))

For simplicity, the existential type system considered here does not permit parameterized abstract types, but it can be extended to do so. For instance, here is an interface type for immutable stacks that is parameterized over the stack component type $t$:

(define-type stack-eface
  (poly (t)
    (packof_exist (stackof t)
      (recordof
        (empty (-> () (stackof t)))
        (empty? (-> ((stackof t)) bool))
        (push (-> (t (stackof t)) (stackof t)))
        (pop (-> ((stackof t)) (stackof t)))
        (top (-> ((stackof t)) t)))))

Parameterized existential types are explored in Exercise 15.8.

Exercise 15.6  This exercise revisits the integer set abstraction introduced in Exercise 15.2.

a. Define an interface type intset-eface for integer sets supporting the operations empty, insert, and member?.

b. Define an existential package list-intset-epkg implementing intset-eface that represents integer sets as integer lists.

c. Define an existential package pred-intset-epkg implementing intset-eface that represents integer sets as integer predicates.

d. Define a testing procedure test-intset that takes any implementation of type intset-eface, creates a set $s$ containing the integers 1 and 3, and returns a three-element boolean list whose $i$th element (1-indexed) indicates whether $s$ contains the integer $i$.

Exercise 15.7

a. Illustrate the necessity of the import restriction for the [eunpack] rule by giving an expression that would unsoundly be well-typed without the restriction.

b. Alf Aaron Ames claims that the import restriction in the [epack] rule and the import and export restrictions in the [eunpack] rule are all unnecessary if before
15.2. DYNAMIC LOCKS AND KEYS

Figure 15.6: Encoding two pair object representations using existential types.
type checking the program is α-renamed to make all logically distinct type identifiers unique. Is Alf correct? Use suitably modified versions of the unsoundness examples in this section to support your answer.

Exercise 15.8

a. Extend the syntax and typing rules of FL/XSP to handle parameterized existential types like (stackof t), which appears in the stack-eface example above.

b. Define an implementation stack-list-epkg of immutable stacks that has type stack-eface and represents a stack as a list of elements ordered from the top down.

c. Define a procedure int-stack-test that tests a stack package by (1) defining a swap procedure that swaps the top to elements of an integer stack; (2) defining a stack->list procedure that converts an integer stack to an integer list; and (3) returning the result of invoking stack->list on the result of calling swap on a stack that contains the elements 1 and 2.

d. Define an interface mstack-eface for mutable stacks and repeat parts b and c for mutable stacks.

Exercise 15.9

The pack_exist form uses up and down procedures to explicitly convert between a concrete representation type and an opaque type name. Here we explore alternative ways to specify abstract vs. concrete types in pack_exist. These alternatives also work for the other forms of pack that we shall study.

a. One alternative to using up and down is to extend pack_exist to have the form (pack_exist I_abs T_rep T_impl E_impl), in which the implementation type $T_{impl}$ is explicitly supplied. For example, here is one way to express a pair implementation of points using the modified form of pack_exist:

\[
\text{pack_exist point (pairof int int)} \\
\text{recordof (make-pt (-> (int int) point))} \\
\text{\hspace{1cm} (pt-x (-> (point) int))} \\
\text{\hspace{1cm} (pt-y (-> (point) int))}
\]

\[
\text{record} \\
\text{\hspace{1cm} (make-pt (lambda ((x int) (y int)) (pair x y)))} \\
\text{\hspace{1cm} (pt-x (lambda ((p point)) (left p)))} \\
\text{\hspace{1cm} (pt-y (lambda ((p (pairof int int))) (right p))))}
\]

Within $E_{impl}$, the abstract type point and the concrete type (pairof int int) are interconvertible.

Give a typing rule for this form of pack_exist. Your rule should not introduce up and down procedures. Use examples to justify the design of your rule.

b. An alternative to specifying $T_{impl}$ in pack_exist is to require the programmer to use explicit type ascriptions (via FL/XSP’s the) to cast concrete to abstract types or vice versa. Explain, using examples.
15.3. NONCE TYPES

We have seen that the export restriction makes existential types an impractical way to express data abstraction in a typed language. The export restriction is a consequence of the fact that the abstract type name in an existential type and the local abstract type names introduced by unpack \textit{exist} forms are not connected to each other or to the concrete type in any way. One way to address this problem is by replacing the abstract type names by globally unique type symbols that we call nonce types. We shall see that nonce types are in many ways a more flexible approach to abstract types than existential types, but suffer from problems of their own.

As an example, the type $T_{\text{point}}$ of one implementation of a point abstraction might be the nonce package type

\[
\text{(packof}_{\text{nonce}} \ #1729 \\
\text{recordof} \\
\text{(make-pt} (\lambda \text{int int}) \ #1729)) \\
\text{(pt-x} (\lambda \text{int}) \ #1729)) \\
\text{(pt-y} (\lambda \text{int}) \ #1729))
\]

where \#1729 is the concrete notation for the globally unique nonce type for this particular implementation. Another point abstraction implementation would have the same packof\textit{nonce} type, except that a different unique nonce type (say \#6821) for that implementation would be substituted for each occurrence of
#1729. Nonce types are automatically introduced by the type checker and cannot be written down directly by the programmer.

Whereas (packof exist Iabs Timpl) is a binding construct declaring that the name Iabs may be used in the scope of Timpl, (packof nonce \( \nu \)abs Timpl) is not a binding construct. Rather, it effectively pairs the nonce type \( \nu \)abs with an implementation type in such a way that the two components can be unbundled by an elimination form (unpacknonce). Like Iabs, \( \nu \)abs is an opaque name that hides a concrete representation type. But unlike Iabs, which has no meaning outside the scope of the packof exist, \( \nu \)abs names a particular concrete representation throughout the entire program. It serves as a globally unique tag for guaranteeing that the operations of a data abstraction are performed only on the appropriate abstract values, regardless of how the operations and values are packaged and unpackaged. For example, a value of type #1729 is necessarily created by the make-pt operation with type (-> (int int) #1729), and it is safe to operate on this value with pt-x and pt-y operations having type (pt-x (-> (#1729) int)). In contrast, these operations are incompatible with abstract values having nonce type #6821.

The essence of the nonce type approach to abstract data types in FL/XSP is presented in Figure 15.7. The syntax for creating and eliminating nonce packages (using packnonce and unpacknonce) and typing nonce packages (packofnonce) parallels the syntax for existential packages in order to facilitate comparisons.

For example, here is an expression \( E_{\text{pair-point-npkg}} \) that describes a pair implementation of a point abstraction as a nonce package:

\[
\begin{align*}
\text{packnonce point (pairof int int)} \\
&\text{(record)} \\
&\quad \text{(make-pt (lambda ((x int) (y int)) (up (pair x y))))} \\
&\quad \text{(pt-x (lambda ((p point)) (left (down p))))} \\
&\quad \text{(pt-y (lambda ((p point) (right (down p)))))).}
\end{align*}
\]

According to the [unpack] typing rule, \( E_{\text{pair-point-npkg}} \) could have the packofnonce type \( T_{\text{point-npkg}} \) given earlier. Each application of the [unpack] rule introduces a fresh nonce type \( \nu \) (in this case #1729) that is not used in any other application of the [unpack] rule. This nonce type replaces all occurrences of the programmer-specified abstract type name Iabs (in this case point) in Eimpl. As in existential types, up and down procedures are used to mediate between the concrete and abstract types. Note that the Type domain must be extended to include nonce types (Nonce-Type), which are distinct from type identifiers and type reconstruction variables. They are instead a sort of newly generated type constant, similar to Skolem constants used in logic.

The following expression \( E_{\text{pair-point-test}} \) is a use of the example package that
15.3. NONCE TYPES

Syntax

\[ E ::= \ldots \mid (\text{pack}_{\text{nonce}} \ I_{\text{abs}} \ T_{\text{rep}} \ E_{\text{impl}}) \quad \text{[Nonce Package Introduction]} \]
\[ \mid (\text{unpack}_{\text{nonce}} \ E_{\text{pkg}} \ I_{\text{ty}} \ I_{\text{impl}} \ E_{\text{body}}) \quad \text{[Nonce Package Elimination]} \]

\[ \nu \in \text{Nonce-Type} \]

\[ T ::= \ldots \mid \nu \quad \text{[Nonce Type]} \]
\[ \mid (\text{packof}_{\text{nonce}} \ \nu \ T_{\text{impl}}) \quad \text{[Nonce Package Type]} \]

Type Rules

\[
\frac{A[\text{up}: (\rightarrow (T_{\text{rep}}) \ \nu), \text{down}: (\rightarrow (\nu) \ T_{\text{rep}})] \vdash [\nu/I_{\text{abs}}]E_{\text{impl}} : T_{\text{impl}}}{A \vdash (\text{pack}_{\text{nonce}} \ I_{\text{abs}} \ T_{\text{rep}} \ E_{\text{impl}}) : (\text{packof}_{\text{nonce}} \ \nu \ T_{\text{impl}})} \quad \text{[npack]}\]
\[
\text{where } \nu \text{ is a fresh nonce type} \quad \text{[freshness condition]}
\]
\[
T_{\text{rep}} \text{ does not contain any } \text{plambda-bound identifiers} \quad \text{[rep restriction]} \]

\[
\frac{A \vdash E_{\text{pkg}} : (\text{packof}_{\text{nonce}} \ \nu \ T_{\text{impl}})}{A[I_{\text{impl}} : T_{\text{impl}}] \vdash [\nu/I_{\text{ty}}]E_{\text{body}} : T_{\text{body}}} \quad \text{[nunpack]}\]

Type Equality

No new type equality rules.

Type Erasure

Same as for pack\text{exist}/unpack\text{exist}.

Figure 15.7: The essence of nonce types in FL/XSP.
is possible with nonce packages but not with existential packages:

\[
\begin{align*}
\text{(let ((pair-point-npkg } & E_{\text{pair-point-npkg}}) \\
\text{(let ((transpose (unpack\_nonce pair-point-npkg t pair-point-ops}
\text{(with pair-point-ops}
\text{(lambda ((p t))
\text{(make-pt (pt-y p) (pt-x p)))))))}
\text{pt (unpack\_nonce pair-point-npkg t pair-point-ops}
\text{(with pair-point-ops
\text{(make-pt 1 2))))))))
\end{align*}
\]

The $\text{unpack\_nonce}$ typing rule can be used to show that $E_{\text{pair-point-test}}$ is well-typed. Since the same nonce type $\#1729$ is used within both occurrences of $\text{unpack\_nonce}$, $\text{transpose}$ has type $(\rightarrow (#1729) #1729)$ and $\text{pt}$ has type $\#1729$, so $(\text{transpose pt})$ (as well as $E_{\text{pair-point-test}}$) has type $\#1729$. As shown by $\text{unpack\_nonce}$, the type identifier $I_{ty}$ in $\text{unpack\_nonce}$ allows the programmer to locally name the nonce type of $E_{\text{pkg}}$, which cannot be written down directly. There are no import or export restrictions in $\text{unpack\_nonce}$. The substitution $[\nu/I_{ty}]E_{body}$ converts all local type identifiers into nonce types that may safely enter and escape from $\text{unpack\_nonce}$ because they are globally unique type symbols that denote the same implementation in all contexts.

Although $\text{unpack\_nonce}$ has no restrictions, there are two restrictions in $\text{unpack\_nonce}$. The freshness condition requires that a different nonce type be used for each occurrence of $\text{pack\_nonce}$ encountered in the type checking process. The restriction requires careful attention in practice. One way to formalize it in the type rules would be to modify the type rules to pass a nonce type counter through the type checking process in a single-threaded fashion and increment the counter whenever $\text{unpack\_nonce}$ is used. In languages that allow separate analysis and compilation of modular units, nonce types could include a unique identifier of the computer on which type-checking was performed along with a timestamp of the time when type-checking took place.

The $\text{unpack\_nonce}$ rule also has a rep restriction that prohibits the concrete representation type $T_{\text{rep}}$ from containing any plambda-bound type identifiers. In the simple form of nonce packages that we are studying, this restriction prevents a single nonce type from being implicitly parameterized over any types that are not known when type checking is performed on the $\text{pack\_nonce}$ expression. For example, consider the following expression, which would unsoundly be well-typed without the restriction:
15.3. NONCE TYPES

(let ((make-wrapper
       (plambda (t)
       (packnonce abs t
       (record
           (wrap (lambda ((x t)) (up x)))
           (unwrap (lambda ((y abs)) (down y)))))))
      (let ((wrap-int (unpacknonce (pcall make-wrapper int) wint ir
                                  (select ir wrap)))
            (unwrap-bool (unpacknonce (pcall make-wrapper bool) wbool br
                                  (select br unwrap))))
            (unwrap-bool (wrap-int 3))).

If #251 is used as the nonce type in the packnonce expression, then wrap-int has type (-> (int) #251), unwrap-bool has type (-> (#251) bool), and (unwrap-bool (wrap-int 3)) has type bool even though it dynamically evaluates to the integer 3! The problem is that #251 should not be a single nonce type but some sort of type constructor that is parameterized over t.

The key advantage of nonce packages over existential packages for expressing abstract types is that they have no export restriction. As illustrated by $E_{pair-point-test}$, values of and operations on the abstract type may escape from unpacknonce expressions. Programmers do not have to rearrange their programs or adopt an awkward programming style to prevent these from happening.

Despite their advantages over existential packages, nonce packages suffer from two drawbacks as a mechanism for abstract types:

1. **Difficulties with expressing nonce types.** The fact that nonce types cannot conveniently be written down directly by the programmer is problematic, especially in an explicitly typed language. For example, in \textit{FL/XSP}, the programmer cannot write a top level definition of the form

   (define pair-point-npkg T E_{pair-point-npkg})

because there is no way to write down the concrete nonce type needed in \textit{T}. This is not just an issue of type syntax; the programmer does not know which nonce type the type checker will choose when checking $E_{pair-point-npkg}$.

One way to address this problem is to embed nonce packages in a language with implicit types, where type reconstruction can infer nonce package types that the programmer cannot express (see Exercise 15.14). This is the approach taken in \textit{SML}, where the nonce-based \texttt{abstype} mechanism allows the local declaration of abstract data types. But in reconstructible languages, it is still sometimes necessary to write down explicit types, and
CHAPTER 15. ABSTRACT TYPES

\begin{verbatim}
(let ((make-rat-impl
  (lambda ((b bool))
    (pack\_nonce rat (pairof int int))
    (record
      (make-rat (lambda ((n int) (d int))
        (up (if b (pair n d) (pair d n)))))
      (numer (lambda ((r rat))
        (if b (left (down r)) (right (down r)))))
      (denom (lambda ((r rat))
        (if b (right (down r)) (left (down r)))))))
  (let ((leftist-rat (make-rat-impl true))
    (rightist-rat (make-rat-impl false)))
    ((unpack\_nonce rightist-rat rty lty lops (select numer lops))
      ((unpack\_nonce leftist-rat leftist-rat lty rty (select make-rat rty)) 1 2)))

Figure 15.8: A form of abstraction violation that can occur with nonce types.
\end{verbatim}

the inability to express nonce package types reduces expressivity in these cases.

Another alternative is to require that the abstract type name $I_{abs}$ in $(pack_{nonce} I_{abs} T_{rep} E_{impl})$ is a globally unique name that serves as a concrete nonce type. This lets the programmer rather than the type checker choose the abstract type name. In this case, the programmer can write down the abstract type name and there is no need for the local abstract type name in $unpack_{nonce}$. There are serious modularity problems with this approach, but it makes sense in restricted systems where all nonce packages are created at top level; see Exercise 15.15.

2. \textit{Insufficient abstraction}. Nonce packages are sound in the sense that there are no \textbf{representation violations} — a well-typed program cannot encounter a run time type error. However, there is still a form of \textbf{abstraction violation} that can occur with nonce packages. An example of this is shown in Figure 15.8. The \texttt{make-rat-impl} procedure makes a rational number implementation, which in all cases represents a rational number as a pair of integers. However, it is abstracted over a boolean argument $b$ that chooses one of two representations. When $b$ is \texttt{true}, the numerator is the left element of the pair and the denominator is the right element; we will call this the “leftist representation.” When $b$ is \texttt{false}, a “rightist representation” is used, in which the numerator is on the right and the denominator is on the left.

In the example, the \texttt{numer} procedure of the rightist representation is ap-
plied to a leftist rational with numerator 1 and denominator 2. Since nonce
types are determined by static occurrences of \texttt{pack}_{\text{nonce}} and there is only
one of these in the example, the two dynamic invocations of \texttt{make-rat-impl}
yield implementations that use the same nonce type for \texttt{rat}. Thus, the
application is well-typed and at run time will return the value 2.

Thus, the nonce package system allows different implementations of a data
abstraction to intermingle as long as they are represented via the same
concrete type. Although this might seem reasonable in some cases, we
normally expect an abstract type system to enforce the contract chosen by
the designer of an abstraction. Enforcing the contract (and not just ensur-
ing compatible representations) enables the abstraction and the clients to
rely on important invariants that, among other things, ensure correctness
of programs.

\begin{itemize}
\item \textbf{Exercise 15.10}
\begin{itemize}
\item a. Would the expression in Figure 15.8. be well-typed if all occurrences of \texttt{pack}_{\text{nonce}}
and \texttt{unpack}_{\text{nonce}} were replaced by \texttt{pack}_{\text{exist}} and \texttt{unpack}_{\text{exist}}? Explain.
\item b. Below are three replacements for the body of the inner \texttt{let} in the example in
Figure 15.8. For each replacement, indicate whether the whole example expression
would be well-typed using (1) nonce packages and (2) existential packages\textsuperscript{5}. For
each case, discuss whether you think the type system does the “right thing” in
that case.
\begin{enumerate}
\item ((\texttt{unpack leftist-rat lty1 lops1 (select numer lops2)})
\hspace{1cm}
((\texttt{unpack leftist-rat lty2 lops2 (select make-rat lops1)}) 1 2))
\item ((\texttt{unpack leftist-rat lty lops})
\hspace{1cm}
(\texttt{unpack rightist-rat rty rops})
\hspace{1cm}
((\texttt{select numer rops}) ((\texttt{select make-rat lops}) 1 2))))))
\item ((\texttt{unpack leftist-rat lty1 lops1})
\hspace{1cm}
(\texttt{unpack leftist-rat lty2 lops2})
\hspace{1cm}
((\texttt{select numer lops2}) ((\texttt{select make-rat lops1}) 1 2))))
\end{enumerate}
\end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{Exercise 15.11} As noted above, the inability to write down nonce types is in-
compatible with top level \texttt{define} declarations in FL/XSP, which require an explicit
type to handle potentially recursive definitions. Design a top level definition mecha-
nism for FL/XSP that enables the declaration of non-recursive global values. Illus-
trate how your mechanism can be used to give the global name \texttt{pair-point-npkg} to
\textit{E}_{\texttt{pair-point-npkg}}.
\end{itemize}
\textsuperscript{5}Assume that the occurrence of \texttt{pack}_{\text{nonce}} in the figure is changed to \texttt{pack}_{\text{exist}} for the existential case.
Exercise 15.12  If the abstract type name \( I_{\text{abs}} \) in \((\text{pack}_{\text{nonce}} I_{\text{abs}} T_{\text{rep}} E_{\text{impl}})\) were required to be a globally unique name, then it could serve as a programmer-specified nonce type.

a. Give typing rules for versions of \( \text{pack}_{\text{nonce}} \) and \( \text{unpack}_{\text{nonce}} \) that are consistent with this interpretation. In this interpretation, there is no need for the identifier \( I_{\text{ty}} \) in \( \text{unpack}_{\text{nonce}} \), since the unique name \( I_{\text{abs}} \) would be used instead.

b. Describe how to modify the type checking rules to verify the global uniqueness requirement.

c. Discuss the advantages and disadvantages of this approach to nonce types. Is it a good idea?

Exercise 15.13  By design, the nonce package types of two syntactically distinct occurrences of \( \text{pack}_{\text{nonce}} \) are necessarily different. For example, two nonce packages implementing a point abstraction would both have types of the form

\[
(\text{packof}_{\text{nonce}} \nu_{\text{point}} \\
(\text{recordof} \\
(\text{make-pt} (\rightarrow (\text{int int}) \nu_{\text{point}})) \\
(\text{pt-x} (\rightarrow (\nu_{\text{point}}) \text{int}))) \\
(\text{pt-y} (\rightarrow (\nu_{\text{point}}) \text{int}))))
\]

but the nonce type \( \nu_{\text{point}} \) would be different for the two packages. Nevertheless, the similarity in form suggests that it should be possible to abstract over different implementations of the same abstract type.

a. Suppose that we modify the syntax of \( \text{packof}_{\text{nonce}} \) to be \((\text{packof}_{\text{nonce}} T T_{\text{impl}})\) but do not change the typing rules \([\text{packof}_{\text{nonce}} T]\) and \([\text{unpack}]\) in any way. Given this change, write a \( \text{make-transpose} \) procedure that takes any nonce package implementing a point abstraction and returns a coordinate swapping procedure for that implementation. (Hint: use \( \text{plambda} \) to abstract over the nonce type.)

b. Explain why it is necessary to modify the syntax of \( \text{packof}_{\text{nonce}} \) in order to write \( \text{make-transpose} \).

Exercise 15.14  In this exercise, we consider existential and nonce types in the context of type reconstruction by adding them to FL/R.

a. Nonce packages can be added to FL/R by extending it with the expression \((\text{pack}_{\text{nonce}} E_{\text{impl}})\) and nonce types.

i. Give an FL/R typing rule for the modified \( \text{pack}_{\text{nonce}} \) form.

ii. Describe how to extend the FL/R reconstruction algorithm to reconstruct the modified \( \text{pack}_{\text{nonce}} \) form.

iii. The \( \text{unpack}_{\text{nonce}} \) expression and \( \text{packof}_{\text{nonce}} \) type are not necessary in FL/R for most programs. Explain why.
15.3. **NONCE TYPES**

iv. In versions of FL/XSP and FL/R supporting records (without row variables), write an FL/XSP program that cannot be re-expressed in FL/R due to the inability to write down an explicit nonce type.

b. Existential packages can be added to FL/R by extending it with the expressions \( \text{pack}_{\text{exist}} E_{\text{impl}} \) and \( \text{unpack}_{\text{exist}} E_{\text{pkg}} T_{\text{impl}} I_{\text{impl}} \) and the type \( \text{pack}_o{\text{f}}_{\text{exist}} I_{\text{abs}} T_{\text{impl}} \). In the modified \( \text{unpack}_{\text{exist}} \) form, the type \( T_{\text{pkg}} \) is the type of the expression \( E_{\text{pkg}} \).

i. Give FL/R typing rules for the modified \( \text{pack}_{\text{exist}} \) and \( \text{unpack}_{\text{exist}} \) forms.

ii. Describe how to extend the FL/R reconstruction algorithm to reconstruct the modified \( \text{pack}_{\text{exist}} \) and \( \text{unpack}_{\text{exist}} \) forms.

iii. Explain why it is necessary for the reconstruction system to be explicitly given the type \( T_{\text{pkg}} \) of the existential package expression \( E_{\text{pkg}} \). Why can’t it reconstruct this package type?

c. What changes would need to be made above to handle existential and nonce packages with parameterized types?

\[ \triangleright \textbf{Exercise 15.15} \] Many languages support abstract types that can only be declared globally. Here we explore an abstract type mechanism introduced by a top level \textbf{define-cluster} form. For simplicity, we assume that programs have the form:

\[
\text{(program}
\begin{align*}
\text{(define-cluster } I_{\text{impl}} & \text{ } I_{\text{abs}} & T_{\text{rep}} & E_{\text{impl}}) \\
\text{;}
\text{(define-cluster } I_{\text{impl}} & \text{ } I_{\text{abs}} & T_{\text{rep}} & E_{\text{impl}}) \\
\text{E_{body}}
\end{align*}
\text{)}
\]

a. One interpretation of \textbf{define-cluster} is given by the following desugaring for the above program form:

\[
\text{(let}
\begin{align*}
\text{((} I_{\text{impl}} & \text{ (pack}_{\text{exist}} I_{\text{abs}} & T_{\text{rep}} & E_{\text{impl}}))) \\
\text{;}
\text{((} I_{\text{impl}} & \text{ (pack}_{\text{exist}} I_{\text{abs}} & T_{\text{rep}} & E_{\text{impl}}))) \\
\text{(unpack}_{\text{exist}} I_{\text{impl}} & I_{\text{abs}} & I_{\text{impl}} & E_{\text{body}})))
\end{align*}
\text{)}
\]

Would the interpretation be any different if all occurrences of \( \text{pack}_{\text{exist}} \) and \( \text{unpack}_{\text{exist}} \) were replaced by \( \text{pack}_{\text{nonce}} \) and \( \text{unpack}_{\text{nonce}} \), respectively?

b. Give a direct typing rule for the program form with \textbf{define-cluster} declarations that gives the same static semantics as the above desugaring.

c. An alternative desugaring for the program form is:
(let ((I Impl\_t (pack\_exist I\_abs\_t T\_rep\_t E\_impl\_t)))
  (unpack\_exist I\_impl\_t I\_abs\_t I\_impl\_t)
  ...
  (let ((I\_impl\_k (pack\_exist I\_abs\_k T\_rep\_k E\_impl\_k)))
    (unpack\_exist I\_impl\_k I\_abs\_k I\_impl\_k E\_body)))

What advantage does this desugaring have over the previous one? Give an example where this desugaring would be preferred.

d. Give a direct typing rule for mutually recursive top level define-cluster declarations.

e. Give a simple example program where mutually recursive clusters are useful.

f. Suppose that we want to be able to locally define a collection of clusters anywhere in a program via the following form:

(\textbf{let-clusters} ((I\_clust I\_abs T\_rep E\_impl)*) E\_body)

Discuss the design issues involved in specifying the semantics of \textbf{let-clusters}.

15.4 Dependent Types

As we saw with existential packages, the inability to express “the type exported by \textit{this} package” makes many programs awkward to write.Nonce types provide a way to express this idea but suffer from two key drawbacks: (1) nonce types are thorny to express: either they are chosen by the system, and are inconvenient or impossible for the programmer to write down explicitly; or they are chosen by the programmer, in which case their global uniqueness requirement is at odds with modularity; and (2) they allow abstract types from different instances of the same syntactic package expression to be confused.

There is another option: use a structured name to select a type out of a package just as components are selected out of a product. In particular, we introduce a new type form \textbf{dtype} \textit{E\_pkg} to mean “the type exported by the package denoted by \textit{E\_pkg}.” It is an error if \textit{E\_pkg} does not denote a package. As with nonce packages, such a package may be viewed as a pair containing a type and a value. The difference is that the programmer has a convenient way to express the type component outside the scope of an unpack expression. The type defined by a package is sometimes referred to as the package’s \textit{carrier}.

6In a practical system in which packages can export multiple abstractions, we could write (dselect I E) just as we select named values from records.
15.4. DEPENDENT TYPES

A type that contains a value expression is called a **dependent type** because the type it represents depends in some sense on the value. The reader may be justifiably concerned about this unholy commingling of types and values, especially with respect to static type checking. As we shall see, dependent types raise some nettlesome issues that the language designer must address in order to reap the expressiveness benefits.

### 15.4.1 A Dependent Package System

This section explores a simple package system that uses dependent types to express abstract types (see Figure 15.9). We shall use the term **dependent package** to refer to a package based on dependent types. The `pack-depend` and `packof-depend` forms work in a way similar to the previous package systems. For example, we can define a pair point implementation (`pair-point-dpkg`) and a procedural point implementation (`proc-point-dpkg`) exactly as in Section 15.2 except that we replace all occurrences of `pack-exist` by `pack-depend`. Both of the new packages will have the following interface type:

```
(define-type point-dface
  (packof-depend point
    (recordof
      (make-pt (-> ((x int) (y int)) point))
      (pt-x (-> ((p point)) int))
      (pt-y (-> ((p point)) int))))
```

Notice that procedure types have been extended to include the names of the formal parameters: the arrow type constructor `->` is now a binding form in which the formal names are available in the return type. We shall see below how this is used.

As with existential packages, `pack-depend` and `unpack-depend` have an import restriction that prevents the local name of the abstraction from capturing an existing type name. As before, this restriction can be eliminated by automatically α-renaming programs to make all logically distinct type identifiers unique. The `unpack-depend` form has an additional restriction that we will discuss in more detail later.

As with nonce packages (but not existential packages), dependent packages have no export restriction, so abstract values may exit the scope of an `unpack-depend` expression. But unlike the nonce type system, free references to the abstract type name exported by `unpack-depend` are replaced by a user-writable type: a dependent type that records the program code that generated the package exporting the type. For example, what is the type $T_{pair\rightarrow point}$ in the following definition?
Syntax
\[ E ::= \ldots \mid (\text{pack\_depend } I_{abs} \ T_{rep} \ E_{impl}) \quad \text{[Dependent Package Introduction]} \]
\[ \mid (\text{unpack\_depend } E_{pkg} \ I_{ty} \ E_{impl} \ E_{body}) \quad \text{[Dependent Package Elimination]} \]
\[ T ::= \ldots \mid (\rightarrow ((I \ T)*) \ T) \quad \text{[Dependent Arrow Type]} \]
\[ \mid (\text{packof\_depend } I_{abs} \ T_{impl}) \quad \text{[Dependent Package Type]} \]
\[ \mid (\text{dtype } E_{pkg}) \quad \text{[Dependent Type Selection]} \]

Type Rules
\[ A \vdash (\text{pack\_depend } I_{abs} \ T_{rep} \ E) : (\text{packof\_depend } I_{abs} \ T) \quad \text{[dpack]} \]
\[ A \vdash (\text{packof\_depend } I_{abs} \ T_{impl}) : (\text{dtype } E_{pkg} \ I_{ty} \ T_{impl}) \quad \text{[unpack]} \]
\[ A \vdash (\text{dtype } E_{pkg} \ I_{ty} \ T_{impl}) : T_{body} \quad \text{[pure restriction]} \]
\[ A \vdash (\text{lambda } ((I_1 \ T_1) \ldots (I_n \ T_n)) \ E : (\rightarrow ((I_1 \ T_1) \ldots (I_n \ T_n)) \ T) \quad \text{[\lambda]} \]
\[ A \vdash (\text{E\_rator } : (\rightarrow ((I_1 \ T_1) \ldots (I_n \ T_n)) \ T_{body}) \quad \text{[apply]} \]
\[ \forall i=1, \ldots, n . \ A \vdash E_i : T_i \quad \text{[purity restriction]} \]
\[ A \vdash (\text{let } ((I_1 \ E_1) \ldots (I_n \ E_n)) \ E_{body} : T_{body}) \quad \text{[let]} \]

Rules for \texttt{letrec} and \texttt{with} are similar and left as exercises.

Type Erasure
Same as for \texttt{pack\_exist/unpack\_exist} and \texttt{pack\_nonce/unpack\_nonce}.

Type Equality
\[ \forall i=1, \ldots, n . \ I_i \not\in \text{FreeIds}[T_{body}] \]
\[ \quad \Rightarrow (((I_1 \ T_1) \ldots (I_n \ T_n)) \ T_{body}) \quad \text{[\rightarrow=} \]
\[ \equiv (((I_1' \ T_1) \ldots (I_n' \ T_n)) [I_{i\rightarrow} I_i]' T_{body}) \quad \text{[\equiv=]} \]
\[ (\text{packof\_depend } I \ T) = (\text{packof\_depend } I' [I'\rightarrow I] T) \quad \text{[dpackof=]} \]
\[ E_1 = \text{depends } E_2 \quad \text{[dtype=} \]
\[ (\text{dtype } E_1) = (\text{dtype } E_2) \quad \text{[dtype=} \]

where \texttt{=depends} is discussed in the text.

Figure 15.9: The essence of dependent types in FL/XSP.
15.4. DEPENDENT TYPES

(define pair-point \texttt{T}_{\textit{pair-point}}
  (\texttt{unpack}_{\textit{depend}} \texttt{pair-point-dpkg} \texttt{point} \texttt{point-ops}
  (\texttt{with} \texttt{point-ops} (\texttt{make-pt} 1 2))))

The return type of the \texttt{make-pt} procedure in \texttt{pair-point-dpkg} is \texttt{point}. According to the \texttt{[dunpack]} rule, \((\texttt{dtype} \texttt{pair-point-dpkg})\) is substituted for this type. So the invocation of \texttt{make-pt}, the \texttt{with} expression, and the \texttt{unpack}_{\textit{depend}} expression all have the type \(T_{\textit{pair-point}} = (\texttt{dtype} \texttt{pair-point-dpkg})\).

The expressive power of dependent types is illustrated by the \texttt{make-transpose} procedure:

\begin{verbatim}
(define make-transpose \texttt{T}_{\textit{make-transpose}}
  (\texttt{lambda} ((\texttt{point-dpkg} \texttt{point-dface}))
    (\texttt{unpack}_{\textit{depend}} \texttt{point-dpkg} \texttt{pt} \texttt{point-ops}
      (\texttt{with} \texttt{point-ops}
        (\texttt{lambda} ((\texttt{p} \texttt{pt}))
          (\texttt{make-pt} (\texttt{pt-y} \texttt{p}) (\texttt{pt-x} \texttt{p})))))
  )
)
\end{verbatim}

What is the type \(T_{\textit{make-transpose}}\) of this procedure? It is a procedure that takes a package that implements the \texttt{point-dface} interface and returns a procedure that takes a point implemented by the given package and returns a point from the same package with swapped coordinates:

\[(\rightarrow ((\texttt{point-dpkg} \texttt{point-dface})))\]

\[(\rightarrow ((\texttt{p} (\texttt{dtype} \texttt{point-dpkg}))) (\texttt{dtype} \texttt{point-dpkg})))\]

It should now be clear why the \(\rightarrow\) type constructor is a binding form in a dependent type system: the return type can depend on the value of a parameter (such as \texttt{point-dpkg} above), so we need a way to refer to the parameter. Not all parameter names are actually used in this fashion. For instance, the parameter name \texttt{p} is ignored. We shall refer to values of the dependent arrow type as dependent procedures.

What happens when we apply a dependent procedure? Consider the application \(E_{\textit{transpose-test}}:\)

\(((\texttt{make-transpose} \texttt{pair-point-dpkg}) \texttt{pair-point-dpkg})\)

By the \texttt{[apply]} rule, when we apply \texttt{make-transpose} to a package satisfying \texttt{point-dface}, we substitute the actual argument expression for the formal parameter \texttt{point-dpkg} in the type

\[(\rightarrow ((\texttt{p} (\texttt{dtype} \texttt{point-dpkg}))) (\texttt{dtype} \texttt{point-dpkg})))\]

to give the type

\[(\rightarrow ((\texttt{p} (\texttt{dtype} \texttt{pair-point-dpkg}))) (\texttt{dtype} \texttt{pair-point-dpkg})))\]
Since \texttt{pair-point} has type \((\texttt{dtype} \texttt{pair-point-dpkg})\), \(E_{\text{transpose-test}}\) is a well-typed expression denoting a value with type \((\texttt{dtype} \texttt{pair-point-dpkg})\).

Similarly, free references to \texttt{let}-bound variables are substituted away in the result type of a \texttt{let}. Now that types refer to values, anytime a value escapes the scope of a variable binding, that variable is replaced with its definition expression if it occurs free in the value’s type. The rules for \texttt{letrec} and \texttt{with} are similar to the [\texttt{apply}] and [\texttt{let}] rules and are left as exercises.

It is instructive to revisit the rational number example from Figure 15.8 in the context of dependent types. Consider the following two expressions:

\begin{verbatim}
((unpack depend rightist-rat rty rops (select numer rops))
 ((unpack depend leftist-rat lty lops (select make-rat lops))
  1 2))

((unpack depend leftist-rat lty2 lops2 (select numer lops2))
 ((unpack depend leftist-rat lty1 lops1 (select make-rat lops1))
  1 2))
\end{verbatim}

With dependent types, the first expression is ill-typed because an attempt is made to apply a procedure of type \((\texttt{->} ((r (\texttt{dtype} \texttt{rightist-rat}))) \texttt{int})) to a value of type \((\texttt{dtype} \texttt{leftist-rat})\). However, the second expression is well-typed since the procedure parameter and the argument point both have type \((\texttt{dtype} \texttt{leftist-rat})\). So dependent types are able to catch the abstraction violation in the first expression while permitting operations and values of the same abstract type to interoperate outside of \texttt{unpack} depend in the second expression. In contrast, neither expression is well-typed with existential packages (due to the export restriction) and both expressions are well-typed with nonce types (which cannot distinguish different instantiations of a \texttt{pack} nonce expression).

15.4.2 Design Issues with Dependent Types

Dependent types are clearly very powerful. However, care must be taken to ensure that a dependent type system is sound. Moreover, programmers typically expect that a statically typed language will respect the \textbf{phase distinction}: the well-typedness of their programs will be verified in a first (terminating) type-checking phase that runs to completion before the second (possibly non-terminating) run time computation phase begins. We shall see that in some designs for dependent types these phases are interleaved and type checking may not terminate.

There are several design dimensions in systems with dependent types. One dimension involves how types are bundled up into and extracted from packages.
15.4. DEPENDENT TYPES

In the system we have studied so far, dependent packages have a single type that is extracted via \texttt{dtype}, but more generally, a dependent package can have several type components. These are typically named and extracted in a record-like fashion. We will see an example of this in the module system in Section 15.5.

Another design dimension involves the details of performing substitutions in the typing rules. Some alternatives to the rules presented in Figure 15.9 are explored in the exercises for this section.

Perhaps the most important design dimension is type equality on dependent types: when is the type \((\texttt{dtype} E_1)\) considered equal to \((\texttt{dtype} E_2)\)? There are a range of choices here that have significant impact on the properties of the language. We spend the rest of this section discussing some of the options.

One option is to treat types as first-class run time entities and dependent packages as pairs of a type and a value. Such pairs are known as strong sums\(^7\) because the type component serves as a tag that can be used for dynamic dispatch. In this interpretation, \((\texttt{dtype} E)\) extracts the type component of the pair, which is convertible with the package’s representation type. Since type checking and evaluation are inextricably intertwined in this design, there is no phase distinction. Furthermore, abstraction is surrendered by making representation types transparent. The PEbble language [BL84] took this approach and used a lock and key mechanism (similar to that described in Section 15.2) to support data abstraction.

Another option is to consider \((\texttt{dtype} E_1)\) to be the same as \((\texttt{dtype} E_2)\) if the expressions \(E_1\) and \(E_2\) are “equal” for a suitable notion of equality. This is the approach taken in the \([\texttt{dtype}=]\) rule of Figure 15.9, which is parameterized over a notion of equality (=\_\texttt{depends}) that is not defined in the figure. There are two broad approaches to defining =\_\texttt{depends}:

- \textbf{Value equality}: At one end of the spectrum, we can interpret two expressions to be the same under =\_\texttt{depends} if they denote the same package in the usual dynamic semantics of expressions. In the general case, this implies that type checking may require expression evaluation. As with strong sums, type checking in this approach may not terminate or may need to be done at run time. Even worse, determining if two package values are the same in general requires comparing procedures for equality, which is uncomputable! In practice, some computable conservative approximation for procedure equality must be used. Such an approximation must necessarily distinguish procedures that are denotationally equivalent. A common technique is to associate a unique identifier with each procedure value and to say that two procedures are equal only if they have the same identifier.

\(^7\)In contrast, existential packages are sometimes called weak sums.
• *Static equality:* In order to preserve static type checking (with no run time requirements and a guarantee that type checking terminates), we desire a definition of \( \equiv_{\text{depends}} \) that is statically computable. The easiest solution is to say that two dtypes are equal if their expressions are textually the same (more precisely, if they have equal abstract syntax trees). This is obviously statically computable, and this simple solution is the one that we adopt here.

There are other choices for \( \equiv_{\text{depends}} \) besides textual equality. We could, for example, allow the expressions in equivalent dtypes to admit \( \alpha \)-renaming. We could allow certain substitutions to take place (say if expressions are equivalent after their \texttt{let}s are substituted away). As long as the equivalence is statically computable and ensures that expressions that denote different values are not equal, the system is sound. We refer to any such system as a static dependent type (SDT) system.

For our system to be sound, we need to guarantee that a value that uses a type exported from one package cannot masquerade as a value of some other type, e.g., a point from the \texttt{proc-point-dpkg} cannot be passed to an operation from \texttt{pair-point-dpkg}.

One requirement is that programs must be \( \alpha \)-renamed on input. Otherwise, it would be possible for textually identical expressions to mean different things in different contexts. Our substitutions when a value exits a binding construct are not enough. Consider the following code:

\[
\begin{align*}
\text{(let ((trans (make-transpose pair-point-dpkg))} \\
\quad (pair-point-dpkg proc-point-dpkg)) \\
\text{(trans (unpack depend pair-point-dpkg point pt-ops}} \\
\quad \text{(with pt-ops} \\
\text{\quad (make-pt 1 2))))}
\end{align*}
\]

If the new binding of \texttt{pair-point-dpkg} were not \( \alpha \)-renamed, then it would be confused with the \texttt{pair-point-dpkg} that occurs free in the type of \texttt{trans}, which would be unsound.

A second requirement is that in a dependent type (\texttt{dtype} \( E_{\text{pkg}} \)), the expression \( E_{\text{pkg}} \) be pure — i.e., it must not vary with state. This is true whether a language uses value or static equality. In a language with mutation, the same syntactic expression might have different meanings at different times. For example, consider the following expression:
15.4. DEPENDENT TYPES

(let ((c (cell pair-point-pkg)))
 (let ((p (unpack (^ c) pt ops
 ((select make-pt ops) 1 2))))
 (begin
  (= c proc-point-pkg)
  ((unpack (^ c) pt ops
   (select pt-x ops)) p))))

When cell c contains pair-point-pkg, the pair point p is created and has type (dtype (^ c)). Then c is modified to contain proc-point-pkg, at which time the procedural point operation pt-x (with type (-> ((dtype (^c))) int) is applied to p. It should be an abstraction violation to apply the procedural point operation pt-x to a pair point, but the type system encounters no error, because the argument type of pt-x and the type of p are both (dtype (^c)). The type system does not track the fact that (^c) refers to different packages at different times.

We address this problem by instituting a purity restriction on E_{pkg} in the [dunpack] rule, which introduces all dependent types. Of course, it is undecidable to know when an expression is pure. A simple conservative approximation is to require that E_{pkg} be a “syntactic value,” a notion introduced in Section 8.2.5 and used in polymorphic types (Section 13.2) and in the type reconstruction system of FL/R (Section 14.2). However, we will see in Section 15.5 that this approximation prohibits many expressions we would like to write. A better alternative is to use an effect system (see Chapter 16) to conservatively approximate pure expressions.

> Exercise 15.16

a. Write typing rules for letrec and with in a language with dependent types.

b. Dependent types permit code to be abstracted over particular implementations of a data abstraction. The typing rules of this section require that such abstractions be curried by the programmer because of the scoping of parameter names in procedure types. The make-transpose procedure studied above is an example of such currying. In its type,

  (-> ((point-dpkg point-dface))
  (-> ((p (dtype point-dpkg)))
   (dtype point-dpkg))),

the argument type of the transposition procedure refers to point-dpkg.

Suppose that we want to modify the typing rules for a dependently typed language to implicitly curry multiple parameters — i.e., to allow the types of later parameters to refer to the names of earlier parameters. For example, in the modified system, an uncurried form of make-transpose could have the type
(define-type uncurried-make-transpose-type
  (-> ((point-dpkg point-dface) (p (dtype point-dpkg)))
      (dtype point-dpkg)))

Curiously, the \(\lambda\) rule does not need to change to support implicitly curried parameters. The \([\text{apply}]\) rule, however, must change. Write a new \([\text{apply}]\) rule that supports procedure formal parameters that refer to previous parameters. E.g., a procedure of type \text{uncurried-make-transpose-type} must be applied to two arguments where the type of the second argument depends on the value of the first.

\[\text{Exercise 15.17}\]
Del Sharkmon suggests the following alternative \([\text{dunpack'}]\) type rule that does dependent type substitutions on the way out of \text{unpack\_depend} rather than on the way in.

\[
\begin{align*}
A \vdash E_{pkg} : (\text{packof\_depend} \ I_{abs} \ T_{impl}) \\
A[I_{impl} : [I_{ty}/I_{abs}]T_{impl}] \vdash E_{bdy} : T_{bdy} \\
A \vdash (\text{unpack\_depend} E_{pkg} I_{ty} I_{impl} E_{bdy}) : ([\text{dtype} E_{pkg}]I_{ty}]T_{bdy}
\end{align*}
\]

\[\text{dunpack'}\]

a. Using dependent packages, redo Exercise 15.10(b) using (1) the original \([\text{dunpack}]\) rule and (2) Del’s \([\text{dunpack'}]\) rule. Which rule do you think is better and why?

b. Del claims that \([\text{dunpack'}]\) is better than \([\text{dunpack}]\) in some situations where \(E_{pkg}\) contains side effects. Write an expression that is well-typed with \([\text{dunpack'}]\) but not \([\text{dunpack}]\).

c. For any expression that is well-typed with \([\text{dunpack'}]\) but not \([\text{dunpack}]\), it is possible to make the expression well-typed by naming \(E_{pkg}\) with a \text{let}. Show this in the context of your expression from the previous part.

\[\text{Exercise 15.18}\]
Ben Bitdiddle looks at the typing rules in Figure 15.9 and your solution to Exercises 15.16 and 15.17 and complains that all the substitutions make him dizzy. He suggests leaving them all out except for those in the \([\text{apply}]\) rule. Under what assumptions is his idea sound? Write a type safe program that type checks under the given rules but does not type check under Ben’s.

15.5 Modules

15.5.1 An Overview of Modules and Linking

It is desirable to decompose a program, especially a large one, into modular components that can be separately written, compiled, tested, and debugged.

\[^{8}\text{In a system with kinds, the \([\lambda]\) rule \textit{would} change because we would need the scope to be manifest to verify that dependent types are well-formed.}\]
Such components are typically called modules but are also known as packages, structures, units, and classes.\textsuperscript{9} Ideally, each individual module is described by an interface that specifies the components required by the module from the rest of the program (the imports) and the components supplied by the module to the rest of the program (the exports). Interfaces often list the names and types of imported and exported values along with informal English descriptions of these values. Such interfaces make it possible for programmers to implement a module without having to know the implementation details of other modules. They also make it possible for a compiler to check for type consistency within a single module.

In most module systems, modules are record-like entities that have both type and value components. In this respect, they are more elaborate versions of the packages we have studied. Indeed, modules are often used as a means of expressing abstract types in addition to being a mechanism for decomposing a program into parts, which is why we study them here. The key difference between the modules discussed here and the packages we studied earlier is that there is an expectation that modules can be separately written and compiled and later combined to form a whole program.

The process of combining modules to form a whole program is called linking. The specification for how to combine the modules to form a program is written in a linking language. Linking is typically performed in a distinct link time phase that is performed after all the individual modules are compiled (compile time) but before the entire program is executed (run time).

A crude form of linking involves hard-wiring the file names for imported modules within the source code for a given module. In more flexible approaches, a module is parameterized over names for the imported modules and the linking language specifies the actual modules to be used for the parameters. Ideally, the linking language should check that the interface types of the actual module parameters are consistent with those of the formal module parameters. In this case, the linking language is effectively a simple typed programming language.

Often, a linking language simply lists the modules to be combined. For example, the object files of a C program are linked by supplying a list of file names to the compiler. A linking language can be made more powerful by adding other programming language features that allow more computation to be performed during the linking process. But the desire to make linking languages more expressive is often in tension with the desire to guarantee that (1) the linking process terminates and (2) mere mortals can reliably understand and use it.

\textsuperscript{9}In many languages, such as C, files serve as de facto modules, but in general the relationship between source files and program modules can be more complex.
the sophisticated types that often accompany more expressive linking languages.

An extreme design point is to make the linking language the same as the base language used to express the modules themselves. Such first-class module systems are powerful because arbitrary modules can be created at run time and the decision of which module to import can be based on dynamic conditions. These systems blur the distinction between link time and run time, and for any Turing-complete base language, the linking process may not terminate. For these reasons, linking is usually specified in a different language from the base language that is suitably restricted to guarantee termination and designed to link programs in a separate phase. In such second-class module systems, modules are not first-class values that can be manipulated in the base language. The module systems of the CLU, SML and OCAML languages are examples of expressive second-class module systems.

### 15.5.2 A First-Class Module System

We conclude this chapter by presenting a first-class module system based on static dependent types and incorporating extensions for sum-of-product data type definitions, pattern matching, higher-order abstractions (type constructors), and multiple abstract type and value definitions in a single module. Our module system illustrates how the simple ideas presented earlier can be combined into a more realistic system, and also how delicate a balance must be struck to make the system both useful and correct.

We add the module features to FL/R to yield the language FL/RM. A language where types are reconstructed is far more convenient for programming than an explicitly typed language, where the explicit types can be tedious and challenging to write. However, as we shall see below, certain types (the types of modules) must be declared because they cannot be reconstructed. This is not a big drawback since explicit module types are important in software engineering for documenting module interfaces.

#### 15.5.2.1 The Structure of FL/RM

The syntax and semantics of FL/RM are presented in Figures 15.10–15.13. Figure 15.10 presents the new expression and type syntax that FL/RM adds to FL/R. A module is a record-like entity whose abstract type components are declared via `define-datatype` (discussed below) and whose value components are declared via `define`. The type of a module is a moduleof type that records the abstract types and the types of each of the named value components. The named type and value components of the module denoted by $E_{mod}$ are made
available to a client expression \( E_{\text{body}} \) via the \((\text{open } E_{\text{mod}} \ E_{\text{body}})\) form, which is the module analog of the \textit{with} form for typed records. Programmers load separately compiled modules from an external storage system via the \textit{load} construct. Loading is described in more detail in Section 15.5.2.5.

Dependent types have the form \((\text{dselect } I_{tc} \ E_{\text{mod}})\), which selects the abstract type constructor named \( I_{tc} \) from the module denoted by \( E_{\text{mod}} \). As in Section 15.4, dependent procedure types of the form \((\to (\cdot t) T)\) must be supported. Parameter names may be omitted from non-dependent procedure types (i.e., where the parameter names don’t appear in the return type); it is assumed that the system treats these as dependent procedure types with fresh parameter names. The type syntax also includes type constructor applications of the form \((\text{TC } T^*)\), where a type constructor \( \text{TC} \) is either an identifier or the result of extracting a type constructor from a module. In both cases, the type constructor is presumed to be either a name the programmer declares via \texttt{define-datatype} or a predefined type constructor name like \texttt{listof}.

Conventional type reconstruction cannot, in general, infer module types. For this reason, optional declarations have been added to the syntax for \texttt{lambda} and \texttt{letrec} expressions and \texttt{define} declarations. (We use the convention that syntax enclosed by square brackets is optional.) Whenever an identifier introduced by \texttt{lambda}, \texttt{letrec}, or \texttt{define} denotes a module (or a value whose type includes a module type), that identifier \texttt{must} have its type supplied. If the type is omitted, the program will not type check.

### 15.5.2.2 Datatypes and Pattern Matching

The \texttt{define-datatype} form is a typed version of the \texttt{define-data} sum-of-products declaration introduced in Section 10.4. It declares a parameterized abstract type constructor along with a collection of constructor procedures and their associated deconstructors. For example,

\[
\begin{align*}
\text{(define-datatype (treeof t)} & \\
\text{ (leaf)} & \\
\text{ (node t (treeof t) (treeof t))))}
\end{align*}
\]

declares a binary tree type constructor, \texttt{treeof}, that is parameterized over the node value type \( t \). It also declares two constructor/deconstructor pairs with the types shown in Figure 15.14. The constructor procedure types are quantified over the type constructor parameter \( t \). The deconstructor procedure types are additionally quantified over the return type \( r \) of the deconstructor. The types of the constructor and deconstructor procedures associated with a \texttt{define-datatype} declaration are formalized by the \( \oplus \) operator (see Figure 15.12), which extends
### Syntax

\[ E ::= \ldots \mid \text{lambda} (LF^*) \ E \]
\[ \mid \text{letrec} ((I \ [T] \ E)^*) \ E \]
\[ \mid \text{match} \ E_{\text{disc}} \ (P \ E)^* \]
\[ \mid \text{module} \ DD^* \ D^* \]  
\[ \mid \text{open} \ E_{\text{mod}} \ E_{\text{body}} \]
\[ \mid \text{load} \ S \]

\[ LF ::= I \mid (I \ T) \]  
\[ P ::= L \mid I \mid I \ P^* \]  
\[ DD ::= \text{define-datatype} \ AB \ (I_{\text{constr}} \ T_{\text{comp}}^*)^* \]  
\[ D ::= \text{define} \ I \ [T] \ E \]  
\[ AB ::= (I_{tc} \ I_{\text{param}}^*) \]  
\[ UID ::= \text{System dependent} \]

\[ T ::= \ldots \mid \text{moduleof} \ AB^* \ (I \ TS)^* \]  
\[ \mid (TC \ T^*) \]  
\[ \mid (-\to \ ((I \ T)^*) \ T) \]  
\[ \mid (-\to \ (T^*) \ T) \]  
\[ \mid \text{dselect} \ I_{tc} \ E_{\text{mod}} \]

\[ TC ::= I_{tc} \mid \text{dselect} \ I_{tc} \ E_{\text{mod}} \]
\[ TS ::= T \mid \text{generic} \ (I^*) \ T \]

### Sugar

The usual FL/R desugaring function \( D \) is extended as follows:

\[ D(\text{match} \ E_{\text{disc}} \ (P \ E)^*) = D_{\text{match}}(\text{match} \ E_{\text{disc}} \ (P \ E)^*) \]

where \( D_{\text{match}} \) is the pattern matching desugarer presented in Figure 10.30 with the modification to \( \text{equal}_L \) described in the text.

Figure 15.10: Syntax for the module system of FL/RM.
15.5. MODULES

**Type Rules**

\[ \forall m \geq 1 : (A \oplus [DD_1, \ldots, DD_n]) [I_1 : T_1, \ldots, I_m : T_m] \vdash E : T \]

\[ A \vdash (\text{module } DD_1 \ldots DD_n \text{ [module]}) \]

\[ (\text{define } I_1 [T_1] E_1) \ldots (\text{define } I_m [T_m] E_m) \]

\[ : (\text{moduleof } ([I_{tc_1} I_{p_{1,1}} \ldots I_{p_{1,m}}, \ldots I_{tc_n} I_{p_{n,1}} \ldots I_{p_{n,m}}]) (I_1 (\text{mgen } T_1)) \ldots (I_m (\text{mgen } T_m))) \]

where \[ DD_i = (\text{define-datatype } (I_{tc_i} I_{p_{i,1}} \ldots I_{p_{i,m}}) \ldots) \]

\[ (\text{ngen } T) = (\text{generic } (J^*) T) \]

\[ J^* = \text{FTV}(T) - \text{FTE}(A) - \{I_{tc_1}, \ldots, I_{tc_n} \} \]

\[ A \vdash E_m : (\text{moduleof } ([I_{tc_1} \ldots (I_{tc_n})]) (I_1 \text{TS}_1) \ldots (I_m \text{TS}_m)) \]

\[ A[I_1 : \text{sub} (\text{TS}_1), \ldots, I_m : \text{sub} (\text{TS}_m)] \vdash \text{sub}(E_b) : T_b \] [open]

\[ A \vdash (\text{open } E_m \ E_b) : T_b \]

where \[ \text{sub}(X) = \sum_{i=1}^n (\text{dselect } I_{tc_i} E_m) / I_{tc_i} X \] [Dependent type introduction]

\[ E_m \text{ is pure} \]

flowchart {
  node [style=rectangle, align=left, font=\footnotesize]
  \node [style=rectangle, align=left, font=\footnotesize] {\vdash \text{contents } [S] : T} [load]

\node [style=rectangle, align=left, font=\footnotesize] {A \vdash (\text{load } S) : T} [load]

The \{proc\}, \{apply\}, \{let\}, and \{letrec\} rules are similar to those in Section 15.4 and are left as exercises.

**Type Equality**

\[ \forall i \geq 1 : T_i = T_i' \]

\[ (\text{moduleof } (AB_1 \ldots AB_k) (((I_{i_1} T_1) \ldots (I_{n} T_n)))) = (\text{moduleof } (AB_1 \ldots AB_k) (((I_{i_1} T_1') \ldots (I_{n} T_n')))) \] [moduleof=]

(This is more restrictive than necessary; see discussion in text.)

\[ E_{m_1} = \text{depend } E_{m_2} \]

\[ (\text{dselect } I E_{m_1}) = (\text{dselect } I E_{m_2}) \] [dselect=]

where \( = \text{depend} \) is textual equality and all programs are appropriately \( \alpha \)-renamed.

\[ \{I_1, \ldots, I_n\} = \{I_1', \ldots, I_n'\} \]

\[ T = T' \] [generic=]

\[ (\text{generic } (I_1 \ldots I_n) T) = (\text{generic } (I_1' \ldots I_n') T') \]

\[ T = (\text{generic } () T) \] [type-generic=]

The \( \Rightarrow \) rule from Figure 15.9 is used for dependent arrow types.

Figure 15.11: Static semantics for the module system of FL/RM.
The notation $A \oplus DD$ denotes a type environment such that

$$(A \oplus DD)(I) = \begin{cases} \\
(generic (I_{p_1} \ldots I_{p_n})) \\
(-\rightarrow (T_1 \ldots T_m) (I_{tc} I_{p_1} \ldots I_{p_n})), \\
if I = I_{con} and DD = (define-datatype (I_{tc} I_{p_1} \ldots I_{p_n}) \\
\quad \ldots (I_{con} T_1 \ldots T_m) \ldots) \\
\quad (generic (I_{ret} I_{p_1} \ldots I_{p_n}) ; I_{ret} is fresh \\
\quad (-\rightarrow (I_{tc} I_{p_1} \ldots I_{p_n})) \\
\quad ; datum \\
\quad (-\rightarrow (T_1 \ldots T_m) I_{ret}) ; success continuation \\
\quad (-\rightarrow () I_{ret})) ; failure continuation \\
\quad I_{ret}), \\
\quad if I = I_{con*\Rightarrow} and DD = (define-datatype (I_{tc} I_{p_1} \ldots I_{p_n}) \\
\quad \ldots (I_{con} T_1 \ldots T_m) \ldots) \\
\quad A(I), otherwise \\
\end{cases}$$

$A \oplus [DD_1, \ldots, DD_n]$ is an abbreviation for $(A \oplus DD_1) \oplus \ldots \oplus DD_n$.

Figure 15.12: Notation for extending type environments with datatypes.

Type Erasure

$$\begin{align*}
&[(define-datatype (I_{tc} I_{p_1} \ldots I_{p_n}) \\
\quad (I_{cstr_1} T_{1,1} \ldots T_{1,m_1}) \ldots (I_{cstr_n} T_{n,1} \ldots T_{n,m_n}))]
= (define-data I_{tc} (I_{cstr_1} x_1 \ldots x_{m_1}) \ldots (I_{cstr_n} x_1 \ldots x_{m_n})) \\
&[(module DD_1 \ldots DD_k (define I_1 [T_1] E_1) \ldots (define I_n [T_n] E_n))]
= (let ((I_{dd_1} E_{dd_1}) \ldots (I_{dd_n} E_{dd_n})) \\
\quad (recordrec (I_1 [E_1]) \ldots (I_n [E_n])))
\quad where \ D_{def}[[DD_1]] @ \ldots @ D_{def}[[DD_k]]
\quad = [(define i_{dd_1} E_{dd_1}) \ldots (define i_{dd_n} E_{dd_n})]
\quad using the D_{def} function from Figure 10.24. \\
&[(open E_{mod} E_{body})] = (with-fields (I_1 \ldots I_n) [E_{mod}] [E_{body}])
\quad where E_{mod} : (moduleof (AB^*) (I_1 T_1) \ldots (I_n T_n))
\end{align*}$$

Operational Semantics

$$\begin{align*}
&(load S) \Rightarrow contents[S] \quad [load]
\end{align*}$$

Figure 15.13: Dynamic semantics for the module system of FL/RM.
15.5. MODULES

leaf : (generic (t) (-> () (treeof t)))
leaf~ : (generic (r t)
  (-> ((treeof t) (-> () r) (-> () r)) r)
node : (generic (t) (-> (t (treeof t) (treeof t)) (treeof t)))
node~ : (generic (r t)
  (-> ((treeof t) (-> (t (treeof t) (treeof t)) r)
       (-> () r))
       r)

Figure 15.14: The types of the constructor and deconstructor procedures introduced by the treeof datatype declaration.

Although deconstructors may be used explicitly in programs, they are usually used implicitly via the match construct introduced in Section 10.5. In the module language, the match construct can be desugared exactly as in Figure 10.30, except that equal_L must be an equality operation appropriate for the type of the literal L rather than the generic equal?. For instance, equal_17 is =, equal_true is bool=? and equal_foo is sym=?.

For simplicity, unparameterized datatypes are required to be written as applications of nullary type constructors. For instance, a geometric shape type could be declared as

(define-datatype (shape)
  (square int)
  (rectangle int int)
  (triangle int int int))

in which case the figure type would be (shape) (the application of a nullary type constructor) and not shape (which is a nullary type constructor, not a type). It is left as an exercise to extend the language to support declarations of types in addition to type constructors.

15.5.2.3 Example: A Parameterized Module

As a non-trivial example of a module, consider the parameterized table module expression E_makeTableModule in Figure 15.15, which implements an immutable but updatable table as a linked list of key/value pairs. The module declares a tableof type constructor parameterized over the value type t. Additionally, we
want to abstract our table implementation over the type of the key, which must admit equality testing via a \texttt{key=?} procedure. We achieve this simply by using \texttt{lambda} to parameterize the table module over a key module \texttt{key-mod}. Since the type of \texttt{key-mod} cannot be inferred, it is explicitly provided. When we call this procedure on a particular key module, we get back a table module that is polymorphic in the type of value stored in the table.

Most languages would require that we write the table module in terms of a key module that must be supplied at either compile time or link time. There would be a separate language for specifying the relationship between the table and key modules. For example, in SML we would use a \texttt{functor} (a linking language function) to abstract the table module over the key module. But because FL/RM has first class modules, the relationship can be expressed via ordinary procedural abstraction.

\subsection*{15.5.2.4 Semantics of \texttt{module} and \texttt{open}}

The static and dynamic semantics of the module constructs is defined in Figures 15.11–15.13. From the type erasure for \texttt{module} in Figure 15.13, we see that at run time a module is just a record of values denoted by recursively scoped definition expressions that are evaluated in the scope of the constructor and deconstructor procedures introduced by the \texttt{define-datatype} declarations. This scoping information is also apparent in the \texttt{[module]} type rule, where the value bindings in the module are analyzed with respect to a type environment that not only includes the types of all constructor and deconstructor procedures declared in the \texttt{define-datatype} declarations (via \texttt{©}) but also includes the types of the defined expressions. In addition to types for the value bindings, the \texttt{moduleof} type for a \texttt{module} expression includes abstract types of the form \((I_{tc} \: I_{p_1} : \ldots : I_{p_k})\) declared by the \texttt{define-datatype} declarations. Only the abstract type constructor name \(I_{tc}\) is a binding occurrence; the type parameters \(I_{p_1} : \ldots : I_{p_k}\) are provided only to indicate arity information (the number of type arguments for the type constructor).

As indicated by the definition of \texttt{©} in Figure 15.12, constructors create values with the abstract type and deconstructors decompose values with the abstract type. Thus, they play the role of \texttt{up} and \texttt{down} in the typing rule for \texttt{pack-depend} in Figure 15.9 (see Exercise 15.9). As indicated by the \texttt{[module]} type rule and the \texttt{module} type erasure rule, the constructor and deconstructor procedures are not exported by a module unless the programmer includes explicit value definitions for them. Thus, the programmer has complete control over how abstract values are constructed and deconstructed. If the constructors for an abstract type are not exported, clients cannot create forgeries that possibly violate representation
(lambda ((key-mod (moduleof ((key))
  (key=? (-> ((key) (key)) bool))))
  (open key-mod
    (module
      (define-datatype (tableof t)
        (empty)
        (non-empty (pairof (pairof (key) t)
                        (tableof t))))
      (define make-table empty)
      (define lookup
        (lambda (k tbl succ fail)
          (match tbl
            ((empty) (fail))
            ((non-empty (pair (pair ak x)) rest)
             (if (key=? k ak)
                (succ x)
                (lookup rest k succ fail))))))
      (define insert
        (lambda (newk newval tbl)
          (lookup tbl k
            (lambda (x) (error alreadyInTable))
            (lambda () (non-empty (pair newk newval) tbl))))))
      (define delete
        (lambda (k tbl)
          (match tbl
            ((empty) tbl)
            ((non-empty (pair (pair ak x)) rest)
             (if (key=? k ak)
                rest
                (non-empty (pair (pair ak x)
                                  (delete rest k))))))))))
  )))

Figure 15.15: An expression $E_{makeTableModule}$ denoting a procedure that takes a key module and returns a table module.
invariants. If the deconstructor for an abstract type is not exported, then clients cannot directly manipulate the concrete representation of an abstract value.

For example, in the table module in Figure 15.15, the empty constructor is exported by the module under a different name (make-table), but the non-empty constructor and the empty* and non-empty* deconstructors are not exported. These are used only to define the lookup, insert, delete operations. So it is impossible for a client to create a non-empty table or manipulate the bindings of a table except through these operations.

For simplicity, we consider two moduleof types to be equal when their abstract types are exactly the same (including order and parameter names) and their bindings have equal types in the same order. This is overly restrictive. The reader is encouraged to develop a more lenient type equality rule as well as subtyping rules for moduleof.

The type erasure rule for (open $E_{mod}$ $E_{body}$) indicates that it dynamically makes the value bindings of the module denoted by $E_{mod}$ available in the body expression $E_{body}$. Note that the type erasure rule needs to “know” the type of $E_{mod}$ in order to determine the field names needed by with-fields. The [open] typing rule is similar to the [dunpack] rule for dependent packages in that it substitutes a dependent type for all occurrences of abstract type constructors in the body expression. As in [dunpack], the expression on which a dependent type depends must be pure. The easiest way to guarantee this is to require $E_{mod}$ in (dselect $I_c$ $E_{mod}$) to be a syntactic value. However, we will see shortly that this solution has fundamental drawbacks in the presence of parameterized modules.

An example of the result of applying the [module] and [open] type rules is the type $T_{makeTableModule}$ (Figure 15.16) of the expression $E_{makeTableModule}$ studied earlier. Each of the procedures exported by the module has a type schema that parameterizes over unification variables introduced by type reconstruction. Note how all instances of the (key) type have been replaced by ((dselect key key-mod)) due to the substitution in the [open] rule.
15.5.2.5 Loading Modules

The load construct supports the development and construction of large programs by allowing separately developed program modules to refer to one another. In our simple system, \( (\text{load } S) \) causes the desugared expression named by the unique name \( S \) to replace \( (\text{load } S) \). The loaded expression is called \( \text{contents}[S] \) in our rules.

Because module dependencies must be acyclic, it is not possible to have modules that directly load each other. Nevertheless, modules with recursive dependencies can be parameterized over their dependencies and expressed within FL/RM; see Exercise 15.26.

Unifying the programming and linking languages via load and first-class modules is very powerful. As illustrated above, the creation, instantiation, and linking of parameterized modules is easily accomplished via lambda and application. It is also possible to choose which modules to load at run time using if, as in the following procedure:

\[
(\lambda \text{matrix} \\
  \ (\text{open if (sparse? matrix)} \\
    \ (\text{load "sparse-matrix-module-v3.22.cmp"}) \\
    \ (\text{load "dense-matrix-module.cmp-v4.5"})) \\
  \ ... \text{ code that manipulates matrix ...})
\]

The ability to use arbitrary computation when linking program components permits idioms that are not expressible in most linking languages. Some down-

\[
\begin{align*}
& (\Rightarrow ((\text{key-mod} \ (\text{moduleof} \ ((\text{key})))) \\
  \ (\text{key} = ? (\Rightarrow ((\text{key} \ (\text{key})) \text{ bool})))) \\
  \ (\text{make-table} \ (\text{generic} \ (a) \ (\Rightarrow () \ (\text{tableof} \ a)))) \\
  \ (\text{lookup} \ (\text{generic} \ (b \ c) \\
    \ (\Rightarrow (((\text{dselect} \ \text{key} \ \text{key-mod})) \ (\text{tableof} \ b) \\
      \ (\Rightarrow (b \ c) \ (\Rightarrow () \ c)) \\
      \ c))) \\
  \ (\text{insert} \ (\text{generic} \ (d) \\
    \ (\Rightarrow (((\text{dselect} \ \text{key} \ \text{key-mod})) \ d \ (\text{tableof} \ d)) \\
      \ (\text{tableof} \ d)))) \\
  \ (\text{delete} \ (\text{generic} \ (e) \\
    \ (\Rightarrow (((\text{dselect} \ \text{key} \ \text{key-mod})) \ (\text{tableof} \ e) \\
      \ (\text{tableof} \ e))))
\end{align*}
\]

Figure 15.16: The type \( T_{\text{makeTableModule}} \) of the expression \( E_{\text{makeTableModule}} \).
sides are that linking is not a separate phase from computation and it may not terminate.

The simple module facility described here still has many important shortcomings.

- Module types can quickly become very awkward to write. Often, we don’t need all the functionality of a module, only particular features. The table module above can use any type with equality as a key. Subtyping can supply the necessary machinery, but it is still useful for languages to provide special syntax for specifying that a type parameter must support certain operations. CLU’s where mechanism was designed to solve this problem.

- Explicitly abstracting over modules is a tedious operation, and results in overly complex code when, for example, two modules must share a common definition. SML’s sharing specification was designed to address this issue.

- Having programmers essentially encode all version information in a manifest string constant is very inconvenient. It is possible to have the programmer specify just a name, like "make-table-module.cmp", have the compiler use the most recent version of the file, and have the compiler and runtime system ensure that the code available during type checking is in fact the source for the object code loaded at run time. FX used the desugaring process to introduce a unique stamp from the file system for this purpose. Whenever a module is modified, any other modules that load that module name must be recompiled. Exercise ?? explores more sophisticated approaches to the value store.

- In the presence of parameterized modules, there is a fundamental problem with using the crude syntactic value test to conservatively approximate which module expressions do not have side effects. To see this, suppose that we replace the body of the open subexpression in Figure ?? by just the insert application. In this case, the type of both the insert and open subexpressions would be `((dselect tableof tbl-mod) bool)`, but the inner let would have type

\[
((dselect tableof (mk-tbl-mod int-key-mod)) bool)
\]

and the outer let would have a type like

\[
((dselect tableof
  (load "make-table-module-v1.3.cmp")
  (load "int-key-module-v2.0.cmp"))
bool)
\]
The problem with the last two types is that both module expressions in the `dselects` are applications and therefore not syntactic values (expansive in the literature). Even though both expressions are in fact referentially transparent, the conservative syntactic test of non-expansiveness fails and causes type checking to fail. This sort of failure will occur whenever an attempt is made to export a type containing an abstract type from a parameterized module outside the scope of an application of that module.

Thus, the syntactic value test for expression purity imposes a kind of export restriction on abstract values, thereby reducing the power of the module system. This problem can be mitigated somewhat by using `let` to introduce names for applications, as in the `let` binding of `tbl-mod`, without which even the type of the `insert` expression would contain the application `(mk-tbl-mod int-key-mod)`. But `let` only locally increases the scope in which subexpressions are well-typed and cannot remove the effective export restriction. What we really need is a better way to determine the purity of an expression, which is the subject of the next chapter.

▷ **Exercise 15.19**  In the table implementation in Figure 15.15, the `lookup` procedure takes success and failure continuations, and is polymorphic in the return type of the continuations. Alternatively, `lookup` could be modified to return either the value stored under the key or some entity indicating the value was not found. Define a new datatype to express this return type, and modify the table implementation to use it.  ◁

▷ **Exercise 15.20**  Sam Antix notices that the `load` syntax requires the value’s name to be a manifest constant. He suggests that `load` should be a primitive procedure that takes a string argument, i.e., one could apply `load` to any expression that returns a string. Is this a good idea? Why or why not?  ◁

▷ **Exercise 15.21**  The abstract type names (and their parameters) introduced by `define-datatype` and used in `moduleof` types are binding occurrences. Extend the definition of `FTV` and type substitution to properly handle these type names.  ◁

▷ **Exercise 15.22**  Is the following FL/RM expression well-typed? If so, give the type reconstructed for `test` and the type of the whole expression. If not, explain.
(let ((mk-tbl-mod (load "make-table-module-v1.3.cmp")))
  (int-key-mod (load "int-key-module-v2.0.cmp"))
  (let ((tbl-mod (mk-tbl-mod int-key-mod)))
    (open tbl-mod
      (let ((test (lambda (k v)
                    (lookup k
                      (insert k v (make-table))
                      (lambda (x) x)
                      (error shouldntHappen)))))
       (pair (test 1 true) (test 2 3))))))

Exercise 15.23  It would be convenient if the module system were extended to support the declaration of types in addition to type constructors. For example, after the extension, an alternative way to define the geometric shape type discussed in the text would be

(define-datatype shape
  (square int)
  (rectangle int int)
  (triangle int int int))

and (square 3) would have the type shape.

a. Extend the type syntax, typing rules, and the definition of \$\oplus\$ so that \textit{define-datatype} can declare types in addition to type constructors.

b. An alternative strategy is to transform all declarations and uses of user-defined types to declarations and uses of nullary type constructors. Define a program transformation that implements this strategy.

Exercise 15.24  Write a type equality rule for \textit{moduleof} type expressions that (1) permits the type components to be in any order; (2) permits the value components to be in any order; and (3) ignores the names (but not the number!) of the type parameters for each type constructor.

Exercise 15.25  The module system described above uses static dependent types. Write \textit{[proc]}, \textit{[let]}, \textit{[letrec]}, and \textit{[apply]} typing rules for this language, being careful to carry out all necessary substitutions. You may want to refer to Figure 15.9 and Exercise 15.16.

Exercise 15.26  Consider the following three expressions:
15.5. MODULES

\[ E_A = \text{(module} \]
\[ \quad (f \text{(lambda} (x) \]
\[ \quad \quad (\text{if} \ (\equiv \ x \ 0) \]
\[ \quad \quad \quad 0 \]
\[ \quad \quad \quad (\text{(open} \text{(load} "B.cmp") \ g) \ (- \ x \ 1)))))) \]
\[ E_B = \text{(open} \text{(load} "A.cmp") \]
\[ \quad \text{(module} \]
\[ \quad \quad (g \text{(lambda} (x) \]
\[ \quad \quad \quad (\text{if} \ (\equiv \ x \ 0) \]
\[ \quad \quad \quad 1 \]
\[ \quad \quad \quad (+ \ (\text{(open} \text{(load} "A.cmp") \ f) \ (- \ x \ 1)) \]
\[ \quad \quad \quad \quad (g (- \ x \ 1)))))))) \]
\[ E_C = \text{(let} (\text{(amod} \text{(load} "A.cmp")) \]
\[ \quad \quad \quad (\text{(bmod} \text{(load} "B.cmp"))) \]
\[ \quad \quad \quad (+ \ (\text{(open} \text{amod} \ f \ 3)) \]
\[ \quad \quad \quad \quad (\text{(open} \text{bmod} \ g \ 3))))) \]

a. Suppose that we want to compile \( E_A \) to the file "A.cmp", \( E_B \) to the file "B.cmp", and \( E_C \) to the file "C.cmp". Explain why there is no compilation order that can be chosen that will allow us to eventually execute the code in \( E_C \) that will use \( E_A \) for "A.cmp" and \( E_B \) for "B.cmp". Consider the case where the file system contains pre-existing files named "A.cmp" and "B.cmp".

b. It is possible to change \( E_A \) and \( E_B \) into parameterized modules that do not directly load modules from particular files, but instead load modules from a parameter that is a (thunk of) a module. Based on this idea, rewrite \( E_A \), \( E_B \), and \( E_C \) in such a way that all three files can be compiled and executing \( E_C \) will return the desired result.

\[ Exercise \ 15.27 \quad \text{Modify the type reconstruction algorithm from Chapter ?? to handle the module, open, load, lambda, and letrec constructs.} \]

Reading

- CLU\[L^+79\]
- SML and revised\[AM87, MTH90, MTHM97\]
- MESA\[MMS78\]
- Benjamin Pierce's book\[Pie02\]
- John Mitchell's books\[Mit96, Mit03\]
• [CW85] discusses polymorphism, bounded quantification, existential types, compares existential types to data abstraction in Ada.

• Mitchell and Plotkin’s existential types [MP84]. Impredicative Strong Existential Equivalent to Type:Type [HH86].

• MacQueen on Modules [Mac84, Mac88]. Dependent types to express modular structure [Mac86]

• Harper on modules [Har86, HMM90]

• [CHD01][CHP99]

• [Ler95]

[Ada] [Parnas?] [ML sharing]

For more information on existential types, see John Mitchell’s textbook, [Mit96]. Luca Cardelli’s QUEST language [Car89] employed first-class existential types.

Static dependent types are due to Mark Sheldon and David Gifford [SG90].

For a somewhat different of view in which a type is its operation set, see the programming language RUSSELL [BDD80].

The programming language Pebble [BL84] included strong existential types (also known as strong sums) and dependent types that could contain any value. Type checking in Pebble could fail to terminate if values in dependent types looped.

Putting type declarations into a language with type reconstruction, as we did with our final module system design, can lead to some surprising results. For example, it is easy to make type checking undecidable. To see how inferable and non-inferable types can be combined in a decidable type system, see James O’Toole’s work in [OG89].

[Recent work on dependent types: Cayenne, Hongwei’s work.]