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.[1] This separation of concerns is especially

[1] However, the client may need to recompile existing code in order to use a modified implementation.

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:

- `(make-pt` x y`)`: Create a point whose x coordinate is the integer x and whose y coordinate is the integer y.
- `(pt-x` p`)`: Return the x coordinate of the given point p.
- `(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, `(pt-x (make-pt` n_1 n_2`))` evaluates to n_1.
2. For any integers n_1 and n_2, `(pt-y (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.

```
(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 `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 `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).

```
(define test-point-impl
  (lambda (point-impl)
    (with-fields (make-pt pt-x pt-y) point-impl
      (let ((transpose (lambda (p) (make-pt (pt-y p) (pt-x p))))
            (point->pair (lambda (p) (pair (pt-x p) (pt-y p)))))
        (point->pair (transpose (make-pt 1 2)))))))
```

The result of invoking `test-point-impl` on a valid point implementation should be the pair value $\langle 2, 1 \rangle$.

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 the client might write `(left p)` rather than `(pt-x p)` to extract the x coordinate of a point `p`, or might create a point "forgery" using `(pair 1 2)` in place of `(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 implementation of a rational number abstraction represents a rational number as a pair of two integers, then a rational number could be dissected with `pt-x` and `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 abstractions 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:

```
(define make-pair-point
  (lambda (x y)
    (let ((point (pair x y)))
      ;; Return a message dispatcher
      (lambda (msg)
        (cond
          ((sym=? msg 'pt-x) (left point))
          ((sym=? msg 'pt-y) (right point))
          (else (error unrecognized-message))
          )))))

(define make-proc-point
  (lambda (x y)
    (let ((point (lambda (b) (if b x y))))
      ;; Return a message dispatcher
      (lambda (msg)
        (cond
          ((sym=? msg 'pt-x) (point true))
          ((sym=? msg 'pt-y) (point false))
          (else (error unrecognized-message))
          )))))
```

How secure is this approach to hiding data abstraction representations? What kinds of abstraction violations are prevented by this technique? What kinds of abstraction violations can still occur? ◁

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))
```

```
(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))))))))
```

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 `cluster` macro that abstracts over the pattern used in the point implementations:

$\mathcal{D}_{\text{exp}}[\![$`(cluster (`$I$ E`)*)`$]\!] =$
`(let ((`I_{key} `(new-key))) ;` I_{key} `fresh`
`(let ((up (lambda (x) (lock` I_{key} `x)))`
`(down (lambda (x) (unlock` I_{key} `x))))`
`(recordrec (`I E`)*)))`

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

```
(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))))))
```

Note that `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.

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

- `(key?` *thing*`)` determines if *key* is a key value.
- `(key=?` *key1 key2*`)` returns true if *key1* is the same key value as *key2* and returns false otherwise. Signals an error if either *key1* or *key2* is not a key value.
- `(locked?` *thing*`)` determines if *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 `gcd` function (which calculates the greatest common divisor of two numbers) is intended to be an unexported local recursive function used by `make-rat`.

```
(define rat-impl
  (cluster
    (make-rat (lambda (x y)
                (let ((g (gcd x y)))
                  (up (pair (div x g) (div y g))))))
    (numer (lambda (r) (left (down r))))
    (denom (lambda (r) (right (down r))))
    (gcd (lambda (a b)
           (if (= b 0)
               a
               (gcd b (rem a b)))))
    ))
```

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

`(cluster (`I_{exp}`*) (`I E`)*)`

where `(`I_{exp}`*)` is an explicit list of **exports** – those bindings we wish to be included in the resulting record. For instance, if we use `(make-rat numer denom)` as the export list in `rat-impl`, then `gcd` would not appear in the resulting record. Modify the desugaring of `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 `new-key`, `lock`, and `unlock`.

b. We can add a `cluster` form to FL/X using the syntax

$$\texttt{(cluster } T_{rep} \texttt{ (}I_1\ T_1\ E_1\texttt{) ... (}I_n\ T_n\ E_n\texttt{))},$$

where T_{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 `cluster` form.

c. Why is it necessary to include T_{rep} and the T_i in the explicitly typed `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:

```
(define-type pair-point-impl-type
  (recordof
    (make-pt (-> (int int) (pairof int int)))
    (pt-x (-> ((pairof int int)) int))
    (pt-y (-> ((pairof int int)) int))))

(define-type proc-point-impl-type
  (recordof
    (make-pt (-> (int int) (-> (bool) int)))
    (pt-x (-> ((-> (bool) int)) int))
    (pt-y (-> ((-> (bool) int)) int))))
```

These two types are the same except for the concrete type used to represent an abstract point value: `(pairof int int)` in the first case and `(-> (bool) 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, packof$_{exist}$, to introduce an abstract type name, `point`, that stands for the concrete type used in a particular implementation:

```
(define-type pt-eface
  (packof_exist point
    (recordof
      (make-pt (-> (int int) point))
      (pt-x (-> (point) int))
      (pt-y (-> (point) int)))))
```

We informally read the above (packof$_{exist}$ ...) type as "there exists a concrete point representation (call it `point`) such that there are `make-pt`, `pt-x`, and `pt-y` procedures with the specified types that manipulate this representation." Such a type is called an **existential type** because it posits the existence of an abstract type and indicates how it is used without saying anything about its concrete representation.[2] In the following discussion, we will often refer to this particular existential type, so we have given it the name `pt-eface`, where `eface` is short for **e**xistential inter**face**.

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

```
(packof_exist q
  (recordof
    (make-pt (-> (int int) q))
    (pt-x (-> (q) int))
    (pt-y (-> (q) int))))
```

is equivalent to the existential type using `point` above.

Values of existential type, which we shall call **existential packages**, are created by the form (pack$_{exist}$ I_{abs} T_{rep} E_{impl}). The type identifier I_{abs} is a type name that is used to hide the concrete representation type T_{rep} within the type of the implementation expression E_{impl}. For example, Figure 15.4 shows two existential packages that implement the type contract specified by `pt-eface`. In the first package, the abstract name `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 pack$_{exist}$ form implicitly introduces `up` and `down` procedures that convert between the concrete and abstract values.

[2] In the literature, such types are often written with ∃ or `exists` just as ∀ and `forall` are used for universal polymorphism. For example, a more standard syntax for the `pt-eface` type is: ∃ `point` . {`make-pt: int*int` → `point, pt-x: point` → `int, pt-y: point` → `int`}.

Syntax

$E ::= \ldots \mid$ `(pack`$_{exist}$ I_{abs} T_{rep} E_{impl}`)` [Existential Introduction]
$\quad \mid$ `(unpack`$_{exist}$ E_{pkg} I_{ty} I_{impl} E_{body}`)` [Existential Elimination]

$T ::= \ldots \mid$ `(packof`$_{exist}$ I_{abs} T_{impl}`)` [Existential Type]

Type Rules

$$\frac{A[\texttt{up:(-> (}T_{rep}\texttt{) }I_{abs}\texttt{), down:(-> (}I_{abs}\texttt{) }T_{rep}\texttt{)}] \vdash E_{impl} : T_{impl}}{A \vdash \texttt{(pack}_{exist}\ I_{abs}\ T_{rep}\ E_{impl}\texttt{)} : \texttt{(packof}_{exist}\ I_{abs}\ T_{impl}\texttt{)}} \quad [epack]$$

where $I_{abs} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E_{impl}]\!]\}$ *[import restriction]*

$$\frac{\begin{array}{c}A \vdash E_{pkg} : \texttt{(packof}_{exist}\ I_{abs}\ T_{impl}\texttt{)} \\ A[I_{impl} : [I_{ty}/I_{abs}]T_{impl}] \vdash E_{body} : T_{body}\end{array}}{A \vdash \texttt{(unpack}_{exist}\ E_{pkg}\ I_{ty}\ I_{impl}\ E_{body}\texttt{)} : T_{body}} \quad [eunpack]$$

where $I_{ty} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E_{body}]\!]\}$ *[import restriction]*
$I_{ty} \notin (FTV\ T_{body})$ *[export restriction]*

Type Equality

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

Type Erasure

$\lceil \texttt{(pack}_{exist}\ I_{abs}\ T_{rep}\ E_{impl}\texttt{)} \rceil$
$= \texttt{(let ((up (lambda (x) x))}$
$\texttt{(down (lambda (x) x)))}$
$\lceil E_{impl} \rceil\texttt{)}$

$\lceil \texttt{(unpack}_{exist}\ E_{pkg}\ I_{ty}\ I_{impl}\ E_{body}\texttt{)} \rceil = \texttt{(let ((}I_{impl}\ \lceil E_{pkg} \rceil\texttt{))}\ \lceil E_{body} \rceil\texttt{)}$

Figure 15.3: The essence of existential types in FL/XSP.

```
(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)))))))
```

Figure 15.4: Two existential packages that implement `pt-eface`.

The [*epack*] type rule in Figure 15.3 specifies how different implementations can have exactly the same existential type. The implementation expression E_{impl} is checked in a type environment where `up` converts from the concrete representation type T_{rep} to the abstract type name I_{abs} and `down` converts from I_{abs} to T_{rep}. These conversions allow the implementer to hide the concrete representation type with an **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.[3] If type checking of a $\texttt{pack}_{exist}$ expression succeeds, we have a proof that there is at least one representation for I_{abs} (namely T_{rep}) and one implementation using this representation (namely E_{impl}) that satisfies the implementation type T_{impl}. This knowledge is recorded with the type $(\texttt{packof}_{exist}\ I_{abs}\ T_{impl})$, in which any implementation details related to T_{rep} and E_{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 `up` and `down` are identity operations. The remaining parts of the expression (I_{abs} and T_{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.

[3] `up` and `down` are just one way to distinguish concrete and abstract types in an existential type. Some alternative approaches are explored in Exercise 15.9.

The type erasure of this expression — `(let ((`I_{impl} $\lceil E_{pkg} \rceil$`))` $\lceil E_{body} \rceil$`)` — 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 `unpack`$_{exist}$, consider the following procedure, which is a typed version of the `test-point-impl` procedure presented in Section 15.1.

```
(define test-point-epkg (-> (pt-eface) (pairof int int))
  (lambda ((point-epkg pt-eface))
    (unpack_exist point-epkg pt point-ops
      (with point-ops
        (let ((transpose (lambda ((p pt))
                           (make-pt (pt-y p) (pt-x p))))
              (point->pair (lambda ((p pt))
                             (pair (pt-x p) (pt-y p)))))
          (point->pair (transpose (make-pt 1 2))))))))
```

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

In the [*eunpack*] type rule, it is assumed that the package expression E_{pkg} has type `(packof`$_{exist}$ 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 `unpack`$_{exist}$ example, where `pt` is the local abstract type name, the `make-pt` procedure has type `(-> (int int) pt)`. The fact that the result type is `pt` rather than `point` is essential for matching up the return type of `transpose` and the declared argument type of `point->pair`.

In the [*epack*] rule, there is an **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:

```
(plambda (t)
  (lambda ((z t))
    (pack_exist t int (down z))))
```

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

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

Without the export restriction, the [*eunpack*] rule would be unsound. Consider the following example of what would go wrong if the restriction were removed:

```
(let ((p (unpack_exist proc-point-epkg t point-ops1
           (with point-ops1 (make-pt 1 2))))
      (f (unpack_exist pair-point-epkg t point-ops2
           (with point-ops2 pt-x))))
  (f p)).
```

The first `unpack`$_{exist}$ makes a procedural point whose type within the `unpack`$_{exist}$ is the local abstract type `t`. This point escapes from the `unpack`$_{exist}$ and is `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 `unpack`$_{exist}$ unpackages a pair point implementation and returns its `pt-x` operation, which is renamed `f`. Since `t` is also used as the local abstract type in the second `unpack`$_{exist}$, the type of `f` is `(-> (t) int)`, where `t` again is actually an unbound type variable. Since `f` has type `(-> (t) 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 `unpack`$_{exist}$, the local type name has no meaning outside the scope of the `unpack`$_{exist}$ and so cannot be allowed to escape.

The export restriction fundamentally limits the usefulness of existential types in practice. For instance, in the `test-point-epkg` procedure studied above, it would be more natural to return the transposed point directly, but then

the type of the `unpack`$_{exist}$ 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 `(-> (pt-eface) (-> (`I_{abs}`) `I_{abs}`))`, where I_{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`$_{exist}$ 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`$_{exist}$ 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`$_{exist}$, 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.

```
(define-type new-pt-eface
  (packof_exist point
    (recordof
      (make-pt (-> (int int) point))
      (pt-x (-> (point) int))
      (pt-y (-> (point) int))
      (transpose (-> (point) point))
      (point->pair (-> (point) (pairof int int)))
      (origin point)
      )))

(define extend-point-epkg (-> (pt-eface) new-pt-eface)
  (lambda ((point-epkg pt-eface))
    (unpack_exist point-epkg pt point-ops
      (with point-ops
        (pack_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->pair (lambda ((p newpt))
                           (pair (pt-x (down p)) (pt-y (down p)))))
            (origin (up (make-pt 0 0)))))))))
```

Figure 15.5: The `extend-point-epkg` procedure shows how values mentioning an abstract type can be passed outside `unpack`$_{exist}$ 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 ith 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

```
(define-type point-object
  (packof_exist point
    (recordof
      (state point)
      (methods (recordof
                 (make-pt (-> (int int) point))
                 (pt-x (-> (point) int))
                 (pt-y (-> (point) int)))))))

(define make-pair-point (-> (int int) point-object)
  (lambda ((x int) (y int))
    (pack_exist point (pairof int int)
      (let ((make-pt (lambda ((x int) (y int)) (up (pair x y)))))
        (record
          (state (make-pt x y))
          (methods (record
                     (make-pt make-pt)
                     (pt-x (lambda ((p point)) (left (down p))))
                     (pt-y (lambda ((p point)) (right (down p))))))))))))

(define make-proc-point (-> (int int) point-object)
  (lambda ((x int) (y int))
    (pack_exist point (-> (bool) int)
      (let ((make-pt (lambda ((x int) (y int))
                       (up (lambda ((b bool)) (if b x y))))))
        (recordof
          (state (make-pt x y))
          (methods (record
                     (make-pt make-pt)
                     (pt-x (lambda ((p point)) ((down p) true)))
                     (pt-y (lambda ((p point)) ((down p) false)))))))))))

(define transpose (-> (point-object) point-object)
  (lambda ((pobj point-object))
    (unpack_exist pobj pt impl
      (with impl
        (with methods
          (pack_exist newpt pt
            (record
              (state (up (make-pt (pt-y state) (pt-x state))))
              (methods
                (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))))))))))))))
```

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 $\texttt{pack}_{exist}$ 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 $\texttt{pack}_{exist}$. 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 $\texttt{pack}_{exist}$ to have the form $(\texttt{pack}_{exist}\ I_{abs}\ T_{rep}\ T_{impl}\ E_{impl})$, 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}:

```
(packexist point (pairof int int)
  (recordof (make-pt (-> (int int) point))
            (pt-x (-> (point) int))
            (pt-y (-> (point) int)))
  (record
    (make-pt (lambda ((x int) (y int)) (pair x y)))
    (pt-x (lambda ((p point)) (left p)))
    (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 $\texttt{pack}_{exist}$. 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 $\texttt{pack}_{exist}$ 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.

c. Yet another way to convert between concrete and abstract types is to interpret the `define-datatype` form in a creative way. (The module system in Section 15.5 follows this approach.) Each constructor can be viewed as performing a conversion up to an opaque abstract type and each deconstructor can be viewed as performing a conversion down from this type. For example, here is a point-as-pair existential package declared via an alternative syntax for pack$_{exist}$ that replaces I_{abs} and T_{rep} by a `define-datatype` declaration:

```
(pack_exist
  (define-datatype point (pt (pairof int int)))
  (record
    (make-pt (lambda ((x int) (y int)) (pt (pair x y))))
    (pt-x (lambda ((p point)) (match p ((pt (pair x _)) x))))
    (pt-y (lambda ((p point)) (match p ((pt (pair _ y)) y))))))
```

Give a typing rule for this modified form of pack$_{exist}$.

d. Express the examples in Figure 15.5 and Figure 15.6 using the alternative approaches to existential types introduced above. ◁

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$_{exist}$ 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 $T_{point-npkg}$ of one implementation of a point abstraction might be the nonce package type

```
(packof_nonce #1729
  (recordof
    (make-pt (-> (int int) #1729))
    (pt-x (-> (#1729) int))
    (pt-y (-> (#1729) int)))),
```

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$_{nonce}$ 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`$_{exist}$ I_{abs} T_{impl}) is a binding construct declaring that the name I_{abs} may be used in the scope of T_{impl}, (`packof`$_{nonce}$ ν_{abs} T_{impl}) is not a binding construct. Rather, it effectively pairs the nonce type ν_{abs} with an implementation type in such a way that the two components can be unbundled by an elimination form (`unpack`$_{nonce}$). Like I_{abs}, ν_{abs} is an opaque name that hides a concrete representation type. But unlike I_{abs}, which has no meaning outside the scope of the `packof`$_{exist}$, ν_{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 `pack`$_{nonce}$ and `unpack`$_{nonce}$) and typing nonce packages (`packof`$_{nonce}$) parallels the syntax for existential packages in order to facilitate comparisons.

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

```
(pack_nonce 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)))))))).
```

According to the [*npack*] typing rule, $E_{pair-point-npkg}$ could have the `packof`$_{nonce}$ type $T_{point-npkg}$ given earlier. Each application of the [*npack*] rule introduces a fresh nonce type ν (in this case `#1729`) that is not used in any other application of the [*npack*] rule. This nonce type replaces all occurrences of the programmer-specified abstract type name I_{abs} (in this case `point`) in E_{impl}. 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 $E_{pair-point-test}$ is a use of the example package that

Syntax

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

$\nu \quad \in \quad$ Nonce-Type

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

Type Rules

$$\frac{A[\texttt{up:(->}\ (T_{rep})\ \nu)\texttt{, down:(->}\ (\nu)\ \ T_{rep})] \vdash [\nu/I_{abs}]E_{impl} : T_{impl}}{A \vdash (\texttt{pack}_{nonce}\ \ I_{abs}\ \ T_{rep}\ \ E_{impl}) : (\texttt{packof}_{nonce}\ \ \nu\ \ T_{impl})} \quad [npack]$$

where ν is a fresh nonce type *[freshness condition]*
T_{rep} does not contain any `plambda`-bound identifiers *[rep restriction]*

$$\frac{\begin{array}{c}A \vdash E_{pkg} : (\texttt{packof}_{nonce}\ \ \nu\ \ T_{impl}) \\ A[I_{impl} : T_{impl}] \vdash [\nu/I_{ty}]E_{body} : T_{body}\end{array}}{A \vdash (\texttt{unpack}_{nonce}\ \ E_{pkg}\ \ I_{ty}\ \ I_{impl}\ \ E_{body}) : T_{body}} \quad [nunpack]$$

Type Equality

No new type equality rules.

Type Erasure

Same as for $\texttt{pack}_{exist}/\texttt{unpack}_{exist}$.

Figure 15.7: The essence of nonce types in FL/XSP.

is possible with nonce packages but not with existential packages:

```
(let ((pair-point-npkg E_pair−point−npkg))
  (let ((transpose (unpack_nonce pair-point-npkg t pair-point-ops
                     (with pair-point-ops
                       (lambda ((p t))
                         (make-pt (pt-y p) (pt-x p))))))
        (pt (unpack_nonce pair-point-npkg t pair-point-ops
                          (with pair-point-ops
                            (make-pt 1 2)))))
    (transpose pt))).
```

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

Although [*nunpack*] has no restrictions, there are two restrictions in [*npack*]. The **freshness condition** requires that a different nonce type be used for each occurrence of $\texttt{pack}_{nonce}$ 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 [*npack*] 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 [*npack*] rule also has a **rep restriction** that prohibits the concrete representation type T_{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 $\texttt{pack}_{nonce}$ expression. For example, consider the following expression, which would unsoundly be well-typed without the restriction:

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

If `#251` is used as the nonce type in the `pack`$_{nonce}$ 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 `unpack`$_{nonce}$ 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 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 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 SML, where the nonce-based `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

```
(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 rops (select numer rops))
       ((unpack_nonce leftist-rat lty lops (select make-rat lops)) 1 2))
```

Figure 15.8: A form of abstraction violation that can occur with nonce types.

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. *Insufficient abstraction.* Nonce packages are sound in the sense that there are no **representation violations** — a well-typed program cannot encounter a run time type error. However, there is still a form of **abstraction violation** that can occur with nonce packages. An example of this is shown in Figure 15.8. The `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 `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 `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 `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 $\texttt{pack}_{nonce}$ and there is only one of these in the example, the two dynamic invocations of `make-rat-impl` yield implementations that use the same nonce type for `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 ensuring compatible representations) enables the abstraction and the clients to rely on important invariants that, among other things, ensure correctness of programs.

▷ **Exercise 15.10**

a. Would the expression in Figure 15.8. be well-typed if all occurrences of $\texttt{pack}_{nonce}$ and $\texttt{unpack}_{nonce}$ were replaced by $\texttt{pack}_{exist}$ and $\texttt{unpack}_{exist}$? Explain.

b. Below are three replacements for the body of the inner `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[5]. For each case, discuss whether you think the type system does the "right thing" in that case.

i.
```
((unpack leftist-rat lty1 lops1 (select numer lops2))
   ((unpack leftist-rat lty2 lops2 (select make-rat lops1)) 1 2))
```

ii.
```
(unpack leftist-rat lty lops
  (unpack rightist-rat rty rops
    ((select numer rops) ((select make-rat lops) 1 2))))))
```

iii.
```
(unpack leftist-rat lty1 lops1
  (unpack leftist-rat lty2 lops2
    ((select numer lops2) ((select make-rat lops1) 1 2))))
```
◁

▷ **Exercise 15.11** As noted above, the inability to write down nonce types is incompatible with top level `define` declarations in FL/XSP, which require an explicit type to handle potentially recursive definitions. Design a top level definition mechanism for FL/XSP that enables the declaration of non-recursive global values. Illustrate how your mechanism can be used to give the global name `pair-point-npkg` to $E_{pair-point-npkg}$. ◁

[5]Assume that the occurrence of $\texttt{pack}_{nonce}$ in the figure is changed to $\texttt{pack}_{exist}$ for the existential case.

▷ **Exercise 15.12** If the abstract type name I_{abs} in (`pack`$_{nonce}$ I_{abs} T_{rep} E_{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 `pack`$_{nonce}$ and `unpack`$_{nonce}$ that are consistent with this interpretation. In this interpretation, there is no need for the identifier I_{ty} in `unpack`$_{nonce}$, since the unique name I_{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 `pack`$_{nonce}$ are necessarily different. For example, two nonce packages implementing a point abstraction would both have types of the form

```
(packof_nonce ν_point
  (recordof
    (make-pt (-> (int int) ν_point))
    (pt-x (-> (ν_point) int))
    (pt-y (-> (ν_point) int))))
```

but the nonce type ν_{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 `packof`$_{nonce}$ to be (`packof`$_{nonce}$ T T_{impl}) but do not change the typing rules [*npack*] and [*nunpack*] in any way. Given this change, write a `make-transpose` procedure that takes any nonce package implementing a point abstraction and returns a coordinate swapping procedure for that implementation. (Hint: use `plambda` to abstract over the nonce type.)

b. Explain why it is necessary to modify the syntax of `packof`$_{nonce}$ in order to write `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 (`pack`$_{nonce}$ E_{impl}) and nonce types.

   i. Give an FL/R typing rule for the modified `pack`$_{nonce}$ form.

   ii. Describe how to extend the FL/R reconstruction algorithm to reconstruct the modified `pack`$_{nonce}$ form.

   iii. The `unpack`$_{nonce}$ expression and `packof`$_{nonce}$ type are not necessary in FL/R for most programs. Explain why.

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 (pack$_{exist}$ E_{impl}) and (unpack$_{exist}$ E_{pkg} T_{pkg} I_{impl} E_{impl}) and the type (packof$_{exist}$ I_{abs} T_{impl}). In the modified unpack$_{exist}$ form, the type T_{pkg} is the type of the expression E_{pkg}.

i. Give FL/R typing rules for the modified pack$_{exist}$ and unpack$_{exist}$ forms.

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

iii. Explain why it is necessary for the reconstruction system to be explicitly given the type T_{pkg} of the existential package expression E_{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? ◁

▷ **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 `define-cluster` form. For simplicity, we assume that programs have the form:

```
(program
  (define-cluster I_impl_1 I_abs_1 T_rep_1 E_impl_1)
  ⋮
  (define-cluster I_impl_k I_abs_k T_rep_k E_impl_k)
  E_body)
```

a. One interpretation of `define-cluster` is given by the following desugaring for the above `program` form:

```
(let ((I_impl_1 (pack_exist I_abs_1 T_rep_1 E_impl_1))
      ⋮
      (I_impl_k (pack_exist I_abs_k T_rep_k E_impl_k)))
  (unpack_exist I_impl_1 I_abs_1 I_impl_1
    ⋮
    (unpack_exist I_impl_k I_abs_k I_impl_k
      E_body)))
```

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

b. Give a direct typing rule for the `program` form with `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_1 (pack_exist I_abs_1 T_rep_1 E_impl_1)))
  (unpack_exist I_impl_1 I_abs_1 I_impl_1
    ⋮
    (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:

```
(let-clusters ((I_clust I_abs T_rep E_impl)*) E_body)
```

Discuss the design issues involved in specifying the semantics of `let-clusters`. ◁

15.4 Dependent Types

As we saw with existential packages, the inability to express "the type exported by *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 `(dtype` E_{pkg}`)` to mean "the type exported by the package denoted by E_{pkg}."[6] It is an error if 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 **carrier**.

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

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 $T_{pair-point}$ in the following definition?

Syntax

$E ::= \ldots \mid$ `(pack`$_{depend}$ I_{abs} T_{rep} E_{impl}`)` [Dependent Package Introduction]
$\mid$ `(unpack`$_{depend}$ E_{pkg} I_{ty} I_{impl} E_{body}`)` [Dependent Package Elimination]

$T ::= \ldots \mid$ `(-> ((`I T`)*)` T`)` [Dependent Arrow Type]
$\mid$ `(packof`$_{depend}$ I_{abs} T_{impl}`)` [Dependent Package Type]
$\mid$ `(dtype` E_{pkg}`)` [Dependent Type Selection]

Type Rules

$$\frac{A[\texttt{up:(-> ((x } T_{rep}\texttt{))} \ I_{abs}\texttt{), down:(-> ((x } I_{abs}\texttt{))} \ T_{rep}\texttt{)}] \vdash E : T}{A \vdash \texttt{(pack}_{depend}\ I_{abs}\ T_{rep}\ E\texttt{)} : \texttt{(packof}_{depend}\ I_{abs}\ T\texttt{)}} \quad [dpack]$$

where $I_{abs} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E]\!]\}$ *[import restriction]*

$$\frac{\begin{array}{c}A \vdash E_{pkg} : \texttt{(packof}_{depend}\ I_{abs}\ T_{impl}\texttt{)} \\ A[I_{impl} : [\texttt{(dtype } E_{pkg}\texttt{)}/I_{abs}]\ T_{impl}] \vdash [\texttt{(dtype } E_{pkg}\texttt{)}/I_{ty}]\ E_{bdy} : T_{bdy}\end{array}}{A \vdash \texttt{(unpack}_{depend}\ E_{pkg}\ I_{ty}\ I_{impl}\ E_{bdy}\texttt{)} : T_{bdy}} \quad [dunpack]$$

where $I_{abs} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E_{body}]\!]\}$ *[import restriction]*
E_{pkg} must be pure *[purity restriction]*

$$\frac{A[I_1 : T_1\texttt{,}\ \ldots\texttt{,}\ I_n : T_n] \vdash E : T}{A \vdash \texttt{(lambda ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ E\texttt{)} : \texttt{(-> ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ T\texttt{)}} \quad [\lambda]$$

$$\frac{\begin{array}{c}A \vdash E_{rator} : \texttt{(-> ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ T_{body}\texttt{)} \\ \forall_{i=1}^{n} .\ A \vdash E_i : T_i\end{array}}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)} : [_{i=1}^{n} E_i/I_i]\,T_{body}} \quad [apply]$$

where $I_i \in FreeIds[\![T_{body}]\!]$ implies E_i is pure. *[purity restriction]*

$$\frac{\begin{array}{c}\forall_{i=1}^{n} .\ A \vdash E_i : T_i \\ A[I_1 : T_1\texttt{,}\ \ldots\texttt{,}\ I_n : T_n] \vdash E_{body} : T_{body}\end{array}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)} \ldots \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} : [_{i=1}^{n} E_i/I_i]\,T_{body}} \quad [let]$$

where $I_i \in FreeIds[\![T_{body}]\!]$ implies E_i is pure *[purity restriction]*

Rules for `letrec` and `with` are similar and left as exercises.

Type Erasure Same as for `pack`$_{exist}$/`unpack`$_{exist}$ and `pack`$_{nonce}$/`unpack`$_{nonce}$.

Type Equality

$$\frac{\forall_{i=1}^{n} .\ I_i' \notin FreeIds[\![T_{body}]\!]}{\begin{array}{l}\texttt{(-> ((}I_1\ T_1\texttt{)}\ldots\texttt{(}I_n\ T_n\texttt{))}\ T_{body}\texttt{)} \\ \equiv \texttt{(-> ((}I_1'\ T_1\texttt{)}\ldots\texttt{(}I_n'\ T_n\texttt{))}[_{i=1}^{n} I_i/I_i']\,T_{body}\texttt{)}\end{array}} \quad [\texttt{->}=]$$

$$\texttt{(packof}_{depend}\ I\ T\texttt{)} = \texttt{(packof}_{depend}\ I'\ [I'/I]\,T\texttt{)} \quad [dpackof=]$$

$$\frac{E_1 =_{depends} E_2}{\texttt{(dtype } E_1\texttt{)} = \texttt{(dtype } E_2\texttt{)}} \quad [dtype=]$$

where $=_{depends}$ is discussed in the text.

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

```
(define pair-point T_pair-point
  (unpack_depend pair-point-dpkg point point-ops
    (with point-ops (make-pt 1 2)))
```

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

The expressive power of dependent types is illustrated by the `make-transpose` procedure:

```
(define make-transpose T_make-transpose
  (lambda ((point-dpkg point-dface))
    (unpack_depend point-dpkg pt point-ops
      (with point-ops
        (lambda ((p pt))
          (make-pt (pt-y p) (pt-x p)))))))
```

What is the type $T_{make-transpose}$ of this procedure? It is a procedure that takes a package that implements the `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:

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

It should now be clear why the `->` type constructor is a binding form in a dependent type system: the return type can depend on the value of a parameter (such as `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 `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 $E_{transpose-test}$:

```
((make-transpose pair-point-dpkg) pair-point-dpkg)
```

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

```
(-> ((p (dtype point-dpkg))) (dtype point-dpkg))
```

to give the type

```
(-> ((p (dtype pair-point-dpkg))) (dtype pair-point-dpkg))
```

Since `pair-point` has type `(dtype pair-point-dpkg)`, $E_{transpose-test}$ is a well-typed expression denoting a value with type `(dtype pair-point-dpkg)`.

Similarly, free references to `let`-bound variables are substituted away in the result type of a `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 `letrec` and `with` are similar to the [*apply*] and [*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:

```
((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))
```

With dependent types, the first expression is ill-typed because an attempt is made to apply a procedure of type `(-> ((r (dtype rightist-rat))) int)` to a value of type `(dtype leftist-rat)`. However, the second expression is well-typed since the procedure parameter and the argument point both have type `(dtype 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 `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 `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 **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.

In the system we have studied so far, dependent packages have a single type that is extracted via `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 `(dtype` E_1`)` considered equal to `(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, `(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 `(dtype` E_1`)` to be the same as `(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 [*dtype*=] rule of Figure 15.9, which is parameterized over a notion of equality ($=_{depends}$) that is not defined in the figure. There are two broad approaches to defining $=_{depends}$:

- *Value equality:* At one end of the spectrum, we can interpret two expressions to be the same under $=_{depends}$ 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 $\equiv_{depends}$ that is statically computable. The easiest solution is to say that two `dtype`s 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 $\equiv_{depends}$ besides textual equality. We could, for example, allow the expressions in equivalent `dtype`s to admit α-renaming. We could allow certain substitutions to take place (say if expressions are equivalent after their `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 `proc-point-dpkg` cannot be passed to an operation from `pair-point-dpkg`.

One requirement is that programs must be α-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:

```
(let ((trans (make-transpose pair-point-dpkg))
      (pair-point-dpkg proc-point-dpkg))
  (trans (unpack_depend pair-point-dpkg point pt-ops
           (with pt-ops
             (make-pt 1 2)))))
```

If the new binding of `pair-point-dpkg` were not α-renamed, then it would be confused with the `pair-point-dpkg` that occurs free in the type of `trans`, which would be unsound.

A second requirement is that in a dependent type `(dtype` E_{pkg}`)`, the expression E_{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:

```
(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 [λ] rule does not need to change to support implicitly curried parameters.[8] The [*apply*] rule, however, must change. Write a new [*apply*] rule that supports procedure formals that refer to previous parameters. E.g., a procedure of type `uncurried-make-transpose-type` must be applied to two arguments where the type of the second argument depends on the value of the first. ◁

▷ **Exercise 15.17** Del Sharkmon suggests the following alternative [*dunpack'*] type rule that does dependent type substitutions on the way out of $\texttt{unpack}_{depend}$ rather than on the way in.

$$\frac{\begin{array}{c} A \vdash E_{pkg} \ : \ (\texttt{packof}_{depend} \ \ I_{abs} \ \ T_{impl}) \\ A[I_{impl} : [I_{ty}/I_{abs}]T_{impl}] \vdash E_{bdy} \ : \ T_{bdy} \end{array}}{A \vdash (\texttt{unpack}_{depend} \ \ E_{pkg} \ \ I_{ty} \ \ I_{impl} \ \ E_{bdy}) \ : \ [(\texttt{dtype} \ \ E_{pkg})/I_{ty}]T_{bdy}} \qquad [dunpack']$$

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

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

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

▷ **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 [*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] In a system with kinds, the [λ] rule *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.[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

[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_{body} via the `(open` E_{mod} E_{body}`)` form, which is the module analog of the `with` form for typed records. Programmers load separately compiled modules from an external storage system via the `load` construct. Loading is described in more detail in Section 15.5.2.5.

Dependent types have the form `(dselect` I_{tc} E_{mod}`)`, which selects the abstract type constructor named I_{tc} from the module denoted by E_{mod}. As in Section 15.4, dependent procedure types of the form `(->` `((`I 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 `(`TC T`*)`, where a type constructor 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 `define-datatype` or a predefined type constructor name like `listof`.

Conventional type reconstruction cannot, in general, infer module types. For this reason, optional declarations have been added to the syntax for `lambda` and `letrec` expressions and `define` declarations. (We use the convention that syntax enclosed by square brackets is optional.) Whenever an identifier introduced by `lambda`, `letrec`, or `define` denotes a module (or a value whose type includes a module type), that identifier *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 `define-datatype` form is a typed version of the `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,

```
(define-datatype (treeof t)
  (leaf)
  (node t (treeof t) (treeof t)))
```

declares a binary tree type constructor, `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 `define-datatype` declaration are formalized by the $\oplus$ operator (see Figure 15.12), which extends

Syntax

E ::= `...` | `(lambda (`LF`*) `E`)`
| `(letrec ((`I `[`T`] `E`)*) `E`)`
| `(match `E_{disc} `(`P E`)*)`
| `(module `DD`* `D`*)` [Module introduction]
| `(open `E_{mod} E_{body}`)` [Module elimination]
| `(load `S`)` [Load compiled code]

LF ::= I | `(`I T`)` [Lambda Formals]
P ::= L | I | `_` | `(`I P`*)` [Patterns]
DD ::= `(define-datatype `AB `(`I_{cnstr} T_{comp}`*)*)` [Datatype Definition]
D ::= `(define `I `[`T`] `E`)` [Value Definition]
AB ::= `(`I_{tc} I_{param}`*)` [Abstract Type]
UID ::= System dependent [Unique File Identifier]

T ::= `...` | `(moduleof (`AB`*) (`I TS`)*)` [Module Type]
| `(`TC T`*)` [Type Constructor Application]
| `(-> ((`I T`)*) `T`)` [Dependent Proc Type]
| `(-> (`T`*) `T`)` [Non-dependent Proc Type]
| `(dselect `I_{tc} E_{mod}`)` [Dependent Type]
TC ::= I_{tc} | `(dselect `I_{tc} E_{mod}`)` [Type Constructor]
TS ::= T | `(generic (`I`*) `T`)` [Type Schema]

Sugar

The usual FL/R desugaring function $\mathcal{D}$ is extended as follows:

$$\mathcal{D}[\![\texttt{(match } E_{disc} \texttt{ (}P\ E\texttt{)*)}]\!] = \mathcal{D}[\![\mathcal{D}_{\text{match}}[\![\texttt{(match } E_{disc} \texttt{ (}P\ E\texttt{)*)}]\!]]\!]$$

where $\mathcal{D}_{\text{match}}$ is the pattern matching desugarer presented in Figure 10.30 with the modification to $\texttt{equal}_L$ described in the text.

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

Type Rules

$$\frac{\forall_{j=1}^{m} \ . \ (A \oplus [DD_1, \ldots, DD_n])[I_1 : T_1, \ldots, I_m : T_m] \vdash E_j \ : \ T_j}{\begin{array}{l} A \vdash \texttt{(module } DD_1 \ \ldots \ DD_n \\ \qquad \texttt{(define } I_1 \ [T_1] \ E_1\texttt{)} \ldots \texttt{(define } I_m \ [T_m] \ E_m\texttt{))} \\ \quad : \texttt{(moduleof ((}I_{tc_1} \ I_{p_{1,1}} \ldots I_{p_{1,a_1}}\texttt{)} \ldots \texttt{(}I_{tc_n} \ I_{p_{n,1}} \ldots I_{p_{n,a_n}}\texttt{))} \\ \qquad\qquad \texttt{(}I_1 \ (mgen \ T_1)\texttt{)} \ldots \texttt{(}I_m \ (mgen \ T_m)\texttt{)))} \end{array}} \quad [module]$$

where $DD_i = \texttt{(define-datatype (}I_{tc_i} \ I_{p_{i,1}} \ldots I_{p_{i,a_i}}\texttt{) ...)}$
$(mgen \ T) = \texttt{(generic (}J^*\texttt{) } T\texttt{)}$
$J^* = FTV \ (T) - FTE(A) - \{I_{tc_1}, \ldots, I_{tc_n}\}$

$$\frac{\begin{array}{c} A \vdash E_m \ : \ \texttt{(moduleof ((}I_{tc_1} \ldots\texttt{)} \ldots \texttt{(}I_{tc_n} \ldots\texttt{)) (}I_1 \ TS_1\texttt{)} \ldots \texttt{(}I_m \ TS_m\texttt{))} \\ A[I_1 : sub(TS_1), \ldots, I_m : sub(TS_m)] \vdash sub(E_b) \ : \ T_b \end{array}}{A \vdash \texttt{(open } E_m \ E_b\texttt{)} \ : \ T_b} \quad [open]$$

where $sub(X) = [_{i=1}^{n} \texttt{(dselect } I_{tc_i} \ E_m\texttt{)}/I_{tc_i}]\text{X}$ *[Dependent type introduction]*
E_m is pure *[Purity restriction]*

$$\frac{\vdash \texttt{contents [}S\texttt{]} \ : \ T}{A \vdash \texttt{(load } S\texttt{)} \ : \ T} \quad [load]$$

The [*proc*], [*apply*], [*let*], and [*letrec*] rules are similar to those in Section 15.4 and are left as exercises.

Type Equality

$$\frac{\forall_{i=1}^{n} \ . \ T_i \ = T_i\,'}{\begin{array}{c} \texttt{(moduleof (}AB_1 \ \ldots AB_k\texttt{) ((}I_1 \ T_1\texttt{) } \ldots\texttt{(}I_n \ T_n\texttt{)))} \\ = \texttt{(moduleof (}AB_1 \ \ldots AB_k\texttt{) ((}I_1 \ T_1\,'\texttt{) } \ldots\texttt{(}I_n \ T_n\,'\texttt{)))} \end{array}} \quad [moduleof{=}]$$

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

$$\frac{E_{m_1} =_{depend} E_{m_2}}{\texttt{(dselect } I \ E_{m_1}\texttt{)} = \texttt{(dselect } I \ E_{m_2}\texttt{)}} \quad [dselect{=}]$$

where $=_{depend}$ is textual equality and all programs are appropriately α-renamed.

$$\frac{\begin{array}{c} \{I_1, \ldots, I_n\} = \{I_1\,', \ldots, I_n\,'\} \\ T = T' \end{array}}{\texttt{(generic (}I_1 \ldots I_n\texttt{) } T\texttt{)} = \texttt{(generic (}I_1\,' \ldots I_n\,'\texttt{) } T'\texttt{)}} \quad [generic{=}]$$

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

The [->=] 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

$$(A \oplus DD)(I) = \begin{cases} \texttt{(generic (}I_{p_1}\ \ldots\ I_{p_m}\texttt{)} \\ \quad \texttt{(->\ (}T_1\ \ldots\ T_n\texttt{)\ (}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)))}, \\ \quad \textbf{if } I = I_{con} \textbf{ and } DD = \texttt{(define-datatype (}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)} \\ \qquad\qquad \ldots\ \texttt{(}I_{con}\ T_1\ \ldots\ T_n\texttt{)}\ \ldots\texttt{)} \\ \\ \texttt{(generic (}I_{ret}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)\ ;\ }I_{ret}\texttt{ is fresh} \\ \quad \texttt{(->\ ((}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)\ \ \ ; datum} \\ \qquad \texttt{(->\ (}T_1\ \ldots\ T_n\texttt{)}\ I_{ret}\texttt{)\ \ ; success continuation} \\ \qquad \texttt{(->\ ()}\ I_{ret}\texttt{))\ \ \ ; failure continuation} \\ \qquad I_{ret}\texttt{))}, \\ \quad \textbf{if } I = I_{con}{\bowtie}^{\sim} \textbf{ and } DD = \texttt{(define-datatype (}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)} \\ \qquad\qquad \ldots\ \texttt{(}I_{con}\ T_1\ \ldots\ T_n\texttt{)}\ \ldots\texttt{)} \\ \\ A(I), \text{otherwise} \end{cases}$$

$A \oplus [DD_1, \ldots, DD_n]$ 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

$\lceil$`(define-datatype (`$I_{tc}\ I_{p_1}\ \ldots\ I_{p_k}$`)`
$\quad$`(`$I_{cnstr_1}\ T_{1,1}\ \ldots T_{1,m_1}$`) ... (`$I_{cnstr_n}\ T_{n,1}\ \ldots T_{n,m_n}$`))`$\rceil$
$=$`(define-data `I_{tc} `(`I_{cnstr_1} `x`$_1$ `... x`$_{m_1}$`) ... (`I_{cnstr_n} `x`$_1$ `... x`$_{m_n}$`))`

$\lceil$`(module `$DD_1\ \ldots\ DD_k$ `(define `$I_1\ [T_1]\ E_1$`) ...(define `$I_n\ [T_n]\ E_n$`))`$\rceil$
$=$ `(let ((`$I_{dd_1}\ E_{dd_1}$`) ... (`$I_{dd_m}\ E_{dd_m}$`))`
`(recordrec (`$I_1\ \lceil E_1 \rceil$`) ...(`$I_n\ \lceil E_n \rceil$`)))`
where $\mathcal{D}_{\text{def}}[\![\lceil DD_1 \rceil]\!] \ @\ \cdots\ @\ \mathcal{D}_{\text{def}}[\![\lceil DD_k \rceil]\!]$
$= [$`(define `$I_{dd_1}\ E_{dd_1}$`)...(define `$I_{dd_m}\ E_{dd_m}$`)`$]$
using the $\mathcal{D}_{\text{def}}$ function from Figure 10.24.

$\lceil$`(open `$E_{mod}\ E_{body}$`)`$\rceil$ $=$ `(with-fields (`$I_1\ \ldots\ I_n$`) `$\lceil E_{mod} \rceil\ \lceil E_{body} \rceil$`)`
where E_{mod} : `(moduleof (`AB^*`) (`$I_1\ T_1$`) ... (`$I_n\ T_n$`))`

Operational Semantics

`(load `S`)` $\Rightarrow$ `contents[`S`]` `[load]`

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

```
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.

a type environment with these types.

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 $\texttt{equal}_L$ must be an equality operation appropriate for the type of the literal L rather than the generic `equal?`. For instance, $\texttt{equal}_{17}$ is `=`, $\texttt{equal}_{true}$ is `bool=?`, and $\texttt{equal}_{'foo}$ 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 `key=?` procedure. We achieve this simply by using `lambda` to parameterize the table module over a key module `key-mod`. Since the type of `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 **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.

15.5.2.4 Semantics of `module` and `open`

The static and dynamic semantics of the module constructs is defined in Figures 15.11–15.13. From the type erasure for `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 `define-datatype` declarations. This scoping information is also apparent in the [*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 `define-datatype` declarations (via $\oplus$) but also includes the types of the defined expressions. In addition to types for the value bindings, the `moduleof` type for a `module` expression includes abstract types of the form `(`I_{tc} I_{p_1} ... I_{p_k}`)` declared by the `define-datatype` declarations. Only the abstract type constructor name I_{tc} is a binding occurrence; the type parameters I_{p_1} ... 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 $\oplus$ 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 `up` and `down` in the typing rule for $\texttt{pack}_{depend}$ in Figure 15.9 (see Exercise 15.9). As indicated by the [*module*] type rule and the `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_{tc} 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.

```
(-> ((key-mod (moduleof ((key))
                (key=? (-> ((key) (key)) bool)))))
  (moduleof ((tableof t))
    (make-table (generic (a) (-> () (tableof a))))
    (lookup (generic (b c)
              (-> (((dselect key key-mod)) (tableof b)
                   (-> (b) c) (-> () c))
                  c)))
    (insert (generic (d)
              (-> (((dselect key key-mod)) d (tableof d))
                  (tableof d))))
    (delete (generic (e)
              (-> (((dselect key key-mod)) (tableof e))
                  (tableof e))))
    ))
```

Figure 15.16: The type $T_{makeTableModule}$ of the expression $E_{makeTableModule}$.

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, `(load` S`)` causes the desugared expression named by the unique name S to replace `(load` S`)`. The loaded expression is called `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 (matrix)
  (open (if (sparse? matrix)
            (load "sparse-matrix-module-v3.22.cmp")
            (load "dense-matrix-module.cmp-v4.5"))
    ... 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-

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 `dselect`s 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 `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 `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 [*proc*], [*let*], [*letrec*], and [*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:

E_A =
```
(module
  (f (lambda (x)
       (if (= x 0)
           0
           ((open (load "B.cmp") g) (- x 1))))))
```

E_B =
```
(open (load "A.cmp")
  (module
    (g (lambda (x)
         (if (= x 0)
             1
             (+ ((open (load "A.cmp") f) (- x 1))
                (g (- x 1))))))))
```

E_C =
```
(let ((amod (load "A.cmp"))
      (bmod (load "B.cmp")))
  (+ (open amod (f 3))
     (open 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** 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.]