Chapter 13

Subtyping and Polymorphism

We need a quote for this chapter.

— *Mark A. Sheldon*

13.1 Subtyping

13.1.1 Motivation

The typing rules presented for FL/X are rather restrictive. For example, consider a `get-name` procedure that extracts the contents of the `name` field of a record:

```
(define get-name (-> ((recordof (name string))) string)
  (lambda ((r (recordof (name string))))
    (select name r)))
```

According to the FL/X typing rules, the following use of `get-name` does not type check:

```
(get-name (record (name "Paula Morwicz") (age 35) (student? #f)))
```

The problem is that the given record has three fields, but the type of `get-name` dictates that the argument record must have exactly one field. Yet the presence of the extra fields does not compromise the type safety of the expression. We can reliably extract a string from the `name` field of *any* record whose type binds `name` to `string`. No constraints on the number or nature of other fields is implied by the extraction of the `name` field.

Situations like `get-name` can be addressed by the notion of **type inclusion** (also called **subtyping**). We say that S is a **subtype** of T (written $S \sqsubseteq T$) if all expressions of type S can be used (in a type-safe manner) in every situation where an expression of type T is used. Viewing types as sets, $S \sqsubseteq T$ means that $S \subseteq T$. If S is a subtype of T, we can also say that T is a **supertype** of S.

13.1.2 FL/XS

We shall consider a variant of FL/X, called FL/XS (the "S" stands for "Subtypes"), that supports subtyping. The typing rules of FL/XS are the same as those for FL/X except for the addition of the [*inclusion*] rule of Figure 13.1. This rule formalizes the notion that a subtype element can be used in any situation where a supertype element is expected.

$$\frac{(A \vdash E : T)\ ;\ (T \sqsubseteq T')}{A \vdash E : T'} \qquad [\textit{inclusion}]$$

Figure 13.1: Additional typing rule for FL/XS. This rule augments the typing rules for FL/X.

Subtyping is a relation on type expressions that can be defined by a collection of rules. The subtyping rules for FL/XS appear in Figure 13.2. The [*reflexive*-

$$T \sqsubseteq T \qquad [\textit{reflexive-}\sqsubseteq]$$

$$\frac{T_1 \sqsubseteq T_2 \quad ; \quad T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3} \qquad [\textit{transitive-}\sqsubseteq]$$

$$\frac{\forall \mathrm{i}\ .\ \exists \mathrm{j}\ .\ ((I_i = J_j) \wedge (S_j \sqsubseteq T_i))}{\begin{array}{c}\texttt{(recordof (}J_1\ S_1\texttt{) ... (}J_m\ S_m\texttt{))} \\ \sqsubseteq \texttt{(recordof (}I_1\ T_1\texttt{) ... (}I_n\ T_n\texttt{))}\end{array}} \qquad [\textit{recordof-}\sqsubseteq]$$

$$\frac{\forall \mathrm{j}\ .\ \exists \mathrm{i}\ .\ ((J_j = I_i) \wedge (S_j \sqsubseteq T_i))}{\texttt{(oneof (}J_1\ S_1\texttt{) ... (}J_m\ S_m\texttt{))} \sqsubseteq \texttt{(oneof (}I_1\ T_1\texttt{) ... (}I_n\ T_n\texttt{))}} \qquad [\textit{oneof-}\sqsubseteq]$$

$$\frac{\forall \mathrm{i}\ .\ (T_i \sqsubseteq S_i) \quad ; \quad S_{body} \sqsubseteq T_{body}}{\texttt{(-> (}S_1\ \ldots\ S_n\texttt{)}\ S_{body}\texttt{)} \sqsubseteq \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [\texttt{->}\text{-}\sqsubseteq]$$

$$\frac{\forall T\ .\ ([T/I_1]T_1 \sqsubseteq [T/I_2]T_2)}{\texttt{(recof}\ I_1\ T_1\texttt{)} \sqsubseteq \texttt{(recof}\ I_2\ T_2\texttt{)}} \qquad [\textit{recof-}\sqsubseteq]$$

Figure 13.2: Subtyping rules for FL/XS.

$\sqsubseteq$] and [*transitive*-$\sqsubseteq$] rules guarantee that subtyping is a reflexive, transitive closure of the relation induced by the other rules. The [*recordof*-$\sqsubseteq$] rule says that a record type S is a subtype of a record type T if (1) S has at least the fields of T and (2) for each of the field names in T, the field types in S are in a subtype relation with the corresponding field types of T. Condition (1) says that extra fields in S can't hurt, since they will be ignored by any code that extracts only the fields mentioned in T. Condition (2) says that subtyping is allowed among the corresponding component types of the fields named by T. When corresponding type components are related via subtyping in the same direction as the entire type, the subtyping of the components is said to be **monotonic**. Thus, the second condition for record subtyping could be rephrased as "for each of the field names in T, the corresponding field types are related monotonically."

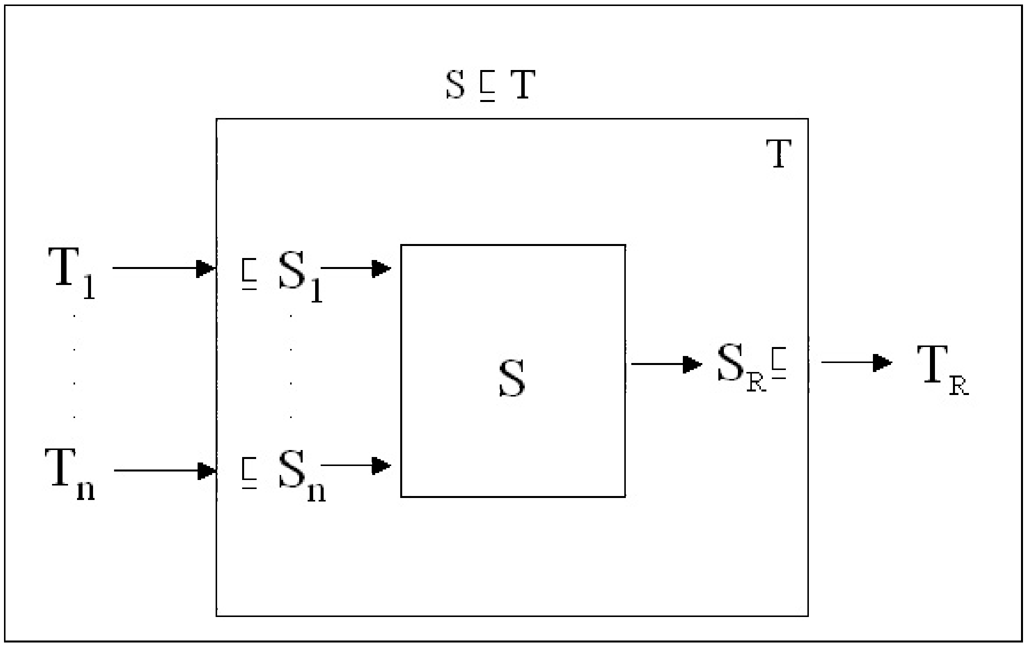

Figure 13.3: Procedure subtyping is monotonic on result of types and anti-monotonic on input types.

The [*oneof*-$\sqsubseteq$] rule is a dual of the [*record*-$\sqsubseteq$] rule: A oneof type S is a subtype of a oneof type T if it has *fewer* tags. The types of the shared tags are related monotonically. This makes sense because if a program is prepared to handle all the cases of the supertype (T), then it is prepared for the fewer possible cases of the subtype (S).

In the rule for procedure subtyping, the return types are monotonic, but the parameter types are **anti-monotonic** — i.e., they are related via subtyping in a direction opposite to the subtyping of the procedure type as a whole. As shown in Figure 13.3, procedures are supposed to handle any element in the class specified by the type of the formal parameter. Thus if a procedure expects type S_1, it can be used in a context where it will be passed elements from the smaller class $T_1 \sqsubseteq S_1$. (Alternatively, one may always safely use a procedure that is defined on **more** values.) On the other hand, it is always safe to use a procedure that returns elements of type S_R where T_R is expected if $S_R \sqsubseteq T_R$.

The rule for the subtyping of `recof` types says that `recof` type S is a subtype of another `recof` type T if the result of instantiating the body of S with any type is a subtype of the result of instantiating the body of T with the same type.

Note that there are no special subtyping rules for `cellof`, `listof`, and `vectorof` types. This is not an oversight; types of these forms are only in a subtype relation if they are type equivalent! The following monotonic rule for `cellof` subtyping seems natural, but it is actually *incorrect*:

$$\frac{T_1 \sqsubseteq T_2}{\texttt{(cellof } T_1\texttt{)} \sqsubseteq \texttt{(cellof } T_2\texttt{)}} \qquad [\textit{incorrect-cellof-}\sqsubseteq]$$

It is possible to show expressions that are well-typed using this rule, but that would raise a run-time type error in the corresponding dynamically typed system. We leave the generation of such an example as an exercise for the reader. Corresponding rules for `listof` and `vectorof` subtyping suffer the same problem as the `cellof` rule above. In all cases, the fundamental problem is due to side effects. In fact, the expected monotonic rule for these types is valid if they are immutable.

Finally, in a language with subtyping, it is reasonable to define type equivalence as mutual inclusion.

13.1.3 Discussion

The [*inclusion*] rule is a simple way of extending the FL/X typing rules with subtyping, but it harbors some problems. In FL/X, every expression has exactly one type (actually, an expression may have many types, but they are all type equivalent). The [*inclusion*] rule destroys this unique typing property allowing a single expression to have many (non-equivalent) types. For instance, in FL/XS, it is possible to prove that the expression

```
(record (a 3) (b #t))
```

has each of the following types:

```
(recordof (a int) (b bool))
(recordof (a int))
(recordof (b bool))
(recordof)
```

The lack of unique typing is not in itself a problem, but it can complicate other analysis. In the case of FL/XS, the lack of unique typing makes it difficult to write a type checker. The problem is that a straightforward type checker needs to choose one of many possible types before enough information is known to make a correct decision. Consider type checking the following simple expression:

```
(let ((c (cell (record (a 3) (b #t)))))
  (begin (cell-set! c (record (a 4)))
         (select a (cell-ref c))))
```

This expression is well-typed according to the typing rules of FL/XS. But the proof of well-typedness requires invoking the [*inclusion*] rule to hide the `b` field of the first `record` so that `c` has the type `(cellof (recordof (a int)))`. A straightforward type checker needs to decide upon the type of `c` before it examines the rest of the program. At this point the type checker does not "know" which fields of the record content of `c` may be accessed later and how `c` will be mutated. (In fact, such details are undecidable in general.) But without such knowledge, the type checker may make an inappropriate choice. For instance, upon encountering the `cell` expression, it seems prudent to assume that `c` has the type

```
(cellof (recordof (a int) (b bool)))
```

Unfortunately, the program is not well-typed under this assumption. In this case, the correct type for `c` is

```
(cellof (recordof (a int)))
```

but this is only OK because it so happens that the program does not later extract the `b` field. Without backtracking or some sophisticated mechanism for managing constraints, simple expressions like the one above will not be type checked properly.

It is possible to restore unique typing and make type checking easier by restricting the contexts in which subtyping is allowable. Figure 13.4 presents an alternate set of type rules that can be used in place of the [*inclusion*] rule. The [*call-inclusion*] rule permits actual arguments to be subtypes of the formal parameters expected by the called procedure. This rule pinpoints procedure call boundary as the most useful spot where the power of subtyping is used implicitly. Implicit coercion is common in other languages: JAVA allows methods to accept arguments of a subclass of the expected class, and numerous languages allow implicit coercion among numeric types (e.g., converting an integer to a floating

$$\frac{\begin{array}{c} A \vdash E_{rator} : \texttt{(->}\ \texttt{(}T_1 \ \ldots T_n\texttt{)}\ T_{body}\texttt{)} \\ \forall \text{i}\ ((A \vdash E_i : S_i) \wedge (S_i \sqsubseteq T_i)) \end{array}}{A \vdash \texttt{(}E_{rator}\ E_1 \ \ldots E_n\texttt{)} : T_{body}} \quad [\textit{call-inclusion}]$$

$$\frac{\begin{array}{c} A \vdash E : S \\ S \sqsubseteq T \end{array}}{A \vdash \texttt{(the}\ T\ E\texttt{)} : T} \quad [\textit{the-inclusion}]$$

Figure 13.4: Modified typing rules for FL/XS

point number).[1]

The alternate set of type rules also includes the [*the-inclusion*] for handling `the`. Under this rule, `the` is no longer merely a type declaration, but a means of **type coercion** — that is, a means of making a value appear to have as its type a supertype of its actual type. In FL/XS, an item can only be coerced to an object of a supertype, and thus no type loophole can arise. Some languages support arbitrary coercion as a deliberate type loophole. C's *casts* are a prime example.

```
;; This type checks
(select a (the (recordof (a int))
               (record (a 3) (b #t))))

;; This fails to type check, because B is hidden by coercion
(select b (the (recordof (a int))
               (record (a 3) (b #t))))
```

If the [*inclusion*], [*call*], and [*the*] rules are replaced with the rules in 13.4, then every FL/XS expression has a single type. This is because implicit subtyping is limited to argument positions while all other coercions must be explicitly made by the programmer. This limitation also makes it possible to implement a straightforward type checker that embodies the rules; the situations in which subtyping needs to be employed are very constrained. Of course, the price of increased simplicity is a reduction in the power of the type system. Under the alternate set of typing rules, some expressions well-typed under the [*inclusion*] rule are no longer well-typed. For example, reconsider an example from above:

[1]In the case of numeric coercion, there is an actual runtime change in representation that must be inserted.

```
(let ((c (cell (record (a 3) (b #t)))))
  (begin (cell-set! c (record (a 4)))
         (select a (cell-ref c))))
```

Under the alternate rules, this expression is not well-typed. The variable `c` is found to have the type

`(cellof (recordof (a int) (b bool)))`

However, because the [*cell-set*] rule requires the new value to have the same type as that stored in the cell, type checking fails at the `cell-set!` expression.

▷ **Exercise 13.1** Show that the following subtyping rule for `cellof` types is unsound:

$$\frac{T_1 \sqsubseteq T_2}{\texttt{(cellof } T_1\texttt{)} \sqsubseteq \texttt{(cellof } T_2\texttt{)}} \qquad [\textit{incorrect-cellof-}\sqsubseteq]$$

Do this by exhibiting an expression that is well-typed under this rule, but which would raise a run-time type error in a dynamically-typed version of FL/XS. ◁

▷ **Exercise 13.2** Suppose that FL/XS were extended to include immutable lists of type `(ilistof` *T*`)` with operations `icons`, `icar`, and `icdr`. Argue that the following subtyping rule for immutable list types is sound:

$$\frac{T_1 \sqsubseteq T_2}{\texttt{(ilistof } T_1\texttt{)} \sqsubseteq \texttt{(ilistof } T_2\texttt{)}} \qquad [\textit{immutable-listof-}\sqsubseteq]$$

◁

▷ **Exercise 13.3** The typing rules in Figure 13.4 can be extended to handle limited subtyping for certain cell, list, and vector operations while still maintaining the unique typing property. For example, if `lst` is defined as

```
(define lst (cons (record (a 3)) (null (recordof (a int)))))
```

then it seems reasonable that

```
(cons (record (a 7) (b #t)) lst)
```

should type check with `(listof (recordof (a int)))` as its type.

Extend the rules of Figure 13.4 to include special subtyping rules for cell, list, and vector special forms, where they make sense. Argue that your rules are (1) safe and (2) preserve the unique typing property of expressions. ◁

▷ **Exercise 13.4** Ben Bitddidle has decided to improve FL/XS by allowing user defined procedures to return multiple values instead of just one. Here's an example of a program that uses Ben's new improvement:

```
(let ((f (lambda ((x int)) (result (* x x) (< x 10)))))
  (result-bind (f 4) (i b)
    (if b (+ i 1) 2)))
```
$\xrightarrow[eval]{}$ *17*

Multiple values are returned from a procedure by `result` and they are bound at the point of application with `result-bind`. In the above example, `i` is bound to 16, and `b` is bound to `#t`. The syntax of FL/XS expressions is expanded to include `result` and `result-bind`:

E ::= ... | `(result` E^*`)` | `(result-bind` E_{app} `(`I^*`)` E_{body}`)`

Values created by `result` can only be used with `result-bind`. In particular, E and `(result` E`)` do not have the same type and are not equivalent. Procedures do not have to return multiple values. The type domain includes a new type constructor for multiple-value return values that are created by `result`:

T ::= ... | `(result-of` T^*`)`

a. Give the typing rules for the `result` and `result-bind` constructs.

b. What are the subtyping rules for `result-of` types? ◁

▷ **Exercise 13.5** Bud Lojack decides to add exceptions with termination semantics to FL/X. He extends the grammar as follows:

E ::= ... | `(raise` I_{except} E_{val}`)` | `(handle` I_{except} E_{handle} E_{body}`)`

Recall the informal semantics of `raise` and `handle` from Exercise 9.12 on page 407:

- `(raise` I_{except} E_{val}`)` evaluates E_{val} and applies the current handler for the exception named I_{except} to the resulting value. This application takes place in the handler environment and continuation of the handler.
- `(handle` I_{except} E_{handle} E_{body}`)` first evaluates E_{handle}. It is an error if the value of E_{handle} is not a procedure of one argument. Otherwise the (procedure) value of E_{handle} is installed as the current handler of the exception named I_{except}, and E_{body} is evaluated. If E_{body} returns normally, the value of E_{handle} is removed as the handler of I_{except}, and the value of E_{body} is returned.

Bud wants to modify the type system of FL/X to support the newly introduced constructs. First, he extends the type grammar to include a new type for exception handlers:

T ::= ... | `(handlerof` T`)`

Next, he suggests the following typing rules for `handle` and `raise`:

$$\frac{\begin{array}{c}A \vdash E_{handle} : \texttt{(-> (}T_1\texttt{)}\ T_2\texttt{)}\\ A[I_{except}\texttt{:(handlerof}\ T_1\texttt{)}] \vdash E_{body} : T_2\end{array}}{A \vdash \texttt{(handle}\ I_{except}\ E_{handle}\ E_{body}\texttt{)} : T_2} \quad [handle]$$

$$\frac{A \vdash I_{except} : \texttt{(handlerof}\ T\texttt{)},\ A \vdash E_{val} : T}{A \vdash \texttt{(raise}\ I_{except}\ E_{val}\texttt{)} : T'} \quad [raise]$$

a. Show that Bud's typing rules result in an unsound type system by providing expressions for E_{first} and E_{second} in the following expression such that the expression is well-typed by Bud's rules, but will generate a dynamic type error.

```
(handle an-exn E_first
   (let ((f (lambda () (raise an-exn 17))))
     (handle an-exn E_second
        (f))))
```

Scared by this initial failure, Bud calls his more skilled friend Ty Pingnut and gives him the task of defining a sound type system for `raise` and `handle`. Ty makes the following change to the grammar of types:

$$\begin{array}{lcl} T & ::= & \ldots \text{ normal FL/X types except for } \texttt{->} \ldots \\ & | & \texttt{(->}\ S\ \texttt{(}T^*\texttt{)}\ T\texttt{)} \\ & | & \texttt{void} \\ \\ S & \in & \text{Exn-Spec} \\ S & ::= & \{ \langle I_1, T_1\rangle, \ldots, \langle I_n, T_n\rangle \} \end{array}$$

The type system is changed to have judgments of the form

$$A \vdash E : T\ \$\ S$$

This can be read, "under type environment A, expression E has type T and may raise exceptions as specified by S." An Exn-Spec S is a set of $\langle I_i, T_i\rangle$ pairs that indicates exceptions that may be raised when E is evaluated, and the type of the value raised for each exception. For example, the judgment

$$A \vdash E : \texttt{bool}\ \$\ \{\langle \texttt{x}, \texttt{bool}\rangle, \langle \texttt{y}, \texttt{int}\rangle\}$$

indicates that if E returns normally, its value will have type `bool`; and that evaluation of E could cause the exception `x` to be raised with a value of type `bool`, or the exception `y` to be raised with a value of type `int`. Ty's type system guarantees that no other exceptions can be raised by E. Note that in Ty's system, a procedure type includes an Exn-Spec S that describes the *latent* exceptions that might be raised *when the procedure is applied.*

Moreover, Ty uses *exception masking* to remove exceptions from judgments when it is clear that they will be handled:

$$A \vdash \texttt{(handle x (lambda ((z bool)) z)}\ E\texttt{)} : \texttt{bool}\ \$\ \{\langle \texttt{y}, \texttt{int}\rangle\}.$$

Ty uses `void` as the type of a `raise` expression:

$$\vdash \texttt{(raise x 4)} : \texttt{void} \ \$ \ \{\langle \texttt{x}, \texttt{int}\rangle\}$$

It is a general property of Ty's system that any expression of type `void` is guaranteed to raise an exception, and therefore, will never return a result. Since an expression of type `void` can never return, it makes sense to think of `void` as a subtype of every type. Ty uses this idea to define a subtyping relation, $\sqsubseteq$, that is similar to the subtyping relation for FL/XS, but has the following additional rule:

$$\texttt{void} \sqsubseteq T \qquad [\texttt{void-}\sqsubseteq]$$

Also, the procedure subtyping rule has been modified to be monotonic on the set of possible exceptions:

$$\frac{\forall \mathrm{i}\ .\ (T_i{}' \sqsubseteq T_i),\ \ T_{body} \sqsubseteq T_{body}{}',\ \ S \subseteq S'}{\texttt{(-> } S\ \texttt{(}T_1 \ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)} \sqsubseteq \texttt{(-> } S'\ \texttt{(}T_1{}' \ \ldots\ T_n{}'\texttt{)}\ T_{body}{}'\texttt{)}} \qquad [\texttt{->-}\sqsubseteq]$$

All other subtyping rules for FL/XS are unchanged in Ty's system.

Here are some of Ty's typing rules:

$$A \vdash N : \texttt{int} \ \$ \ \{\} \qquad [int]$$

$$A \vdash B : \texttt{bool} \ \$ \ \{\} \qquad [bool]$$

$$\frac{\begin{array}{c} A \vdash E_1 : \texttt{bool} \ \$ \ S_1 \\ A \vdash E_2 : T \ \$ \ S_2 \\ A \vdash E_3 : T \ \$ \ S_3 \end{array}}{A \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : T \ \$ \ S_1 \cup S_2 \cup S_3} \qquad [if]$$

$$\frac{A \vdash E : T \ \$ \ S,\ \ T \sqsubseteq T',\ \ S \subseteq S'}{A \vdash E : T' \ \$ \ S'} \qquad [inclusion]$$

Note in particular the rule [*inclusion*], which is crucial in typing the following examples:

$$\vdash \texttt{(if \#t \#f (raise x 4))} : \texttt{bool} \ \$ \ \{\langle \texttt{x}, \texttt{int}\rangle\}$$
$$\vdash \texttt{(if \#f (raise x 4) (raise x \#t))} : \texttt{void} \ \$ \ \{\langle \texttt{x}, \texttt{int}\rangle, \langle \texttt{x}, \texttt{bool}\rangle\}$$

In the second example, values of two incompatible types (`int` and `bool`) are raised for the same exception `x`. Because we are working in a language without polymorphism, it is impossible to write a handler for both values.

b. Give the typing rule for `raise`.

c. Give the typing rule for `handle`.

d. Suppose we alter the syntax of `error` to be `(error` *Y*`)`. What is the new typing rule for `error`? ◁

13.2 Polymorphic Types

Monomorphic type systems are easy to reason about, but they hinder the development of reusable code. In particular, monomorphic languages prevent the programmer from expressing **polymorphic** values — values (typically procedures) that can have different types in different contexts. In this section, we develop a type system that allows the expression of polymorphic values.

As an example of a polymorphic value, consider a `map` procedure written in FL:

```
(define map
  (lambda (fn lst)
    (if (null? lst)
        (null)
        (cons (fn (car lst)) (map fn (cdr lst))))))
```

We have seen that aggregate data operators like `map` are a powerful means of composing programs out of reusable, mix-and-match parts. In large part, this power is due to the fact that the same operator works over many types of operands. The `map` procedure, for instance, can be viewed as having an infinite number of possible types, including:

```
(-> ((-> (int) int) (listof int)) (listof int))

(-> ((-> (int) bool) (listof int)) (listof bool))

(-> ((-> (bool) int) (listof bool)) (listof int))

(-> ((-> (bool) bool) (listof bool)) (listof bool))

(-> ((-> ((listof int)) int) (listof (listof int))) (listof int))

(-> ((-> (int) (-> (bool) int)) (listof int))
    (listof (-> (bool) int)))
```

The type of `map` for any particular call depends on the types of its arguments. So, in the call

```
(map (lambda (x) (* x x)) (list 1 2 3)) ,
```

`map` effectively has type

```
(-> ((-> (int) int) (listof int)) (listof int)) ,
```

while in the call

```
(map (lambda (x) (< x 17)) (list 23 13 29)) ,
```

it has the type

```
(-> ((-> (int) bool) (listof int)) (listof bool)) .
```

Other common examples of useful polymorphic procedures include the identity function (`(lambda (x) x)`) and general sorting utilities.

Unfortunately, the type system of FL/X requires the type of values like `map` to be specified where it is created, not where it is called. A programmer wishing to use `map` on different types of arguments must write a different version of `map` for every different set of argument types. For example, here are two FL/X versions of `map` that correspond to the two calls mentioned above:

```
(define map (-> ((-> (int) int) (listof int)) (listof int))
  (lambda ((fn (-> (int) int)) (lst (listof int)))
    (if (null? lst)
        (null int)
        (cons (fn (car lst)) (map fn (cdr lst))))))
(define map (-> ((-> (int) bool) (listof int)) (listof bool))
  (lambda ((fn (-> (int) bool)) (lst (listof int)))
    (if (null? lst)
        (null bool)
        (cons (fn (car lst)) (map fn (cdr lst))))))
```

Except for type information, the two definitions are exactly the same.

Any language like FL/X that forces the programmer to reimplement functionality in order to satisfy the type system thwarts the goal of writing reusable software components. There is a broad class of general-purpose functions and data structures that are inexpressible in such languages due to the shackles of the type system. This lack of expressiveness is indicative of the price that programmers may have to pay for types. In fact, the primary limitations of languages such as PASCAL and C stem from their monomorphic type systems.

Polymorphism can be introduced into a language by generalizing the types of values where they are created, and then specializing these types where the values are used. Reconsider the types of `map` listed above. All of them are instances of a common pattern:

`(-> ((-> (`S`) `T`) (listof `S`)) (listof `T`))`

We would like to be able to declare that `map` has this general type, but then specialize this type (by specifying S and T) wherever `map` is applied.

We embody this approach in a polymorphic language FL/XSP (the "P" stands for "Polymorphism") by adding two new expression constructs and one new type construct to FL/XS:

E ::= . . . | `(plambda (`I*`) `E`)` | `(pcall `E T*`)`
T ::= . . . | `(forall (`I*`) `T`)`

`(plambda (`I*`) `E`)` creates a polymorphic value that is parameterized over the type variables I*.

`(pcall `E T*`)` instantiates, or **projects**, the type variables of the polymorphic object denoted by E. `pcall` is the "call" that supplies "arguments" to values created by `plambda`.

`(forall (`I*`) `T`)` is the type of a polymorphic value. In the literature, polymorphic types are often written using ∀ notation and referred to as "universally quantified." For example, the type of the mapping procedure,

```
(forall (s t) (-> ((-> s t) (listof s)) (listof t)))
```

is typically rendered

$$\forall s,t\,.\ \texttt{(s} \ \to\ \texttt{t)}\ \times\ \texttt{(listof s)}\ \to\ \texttt{(listof t)}$$

Here is a polymorphic version of `map` written in FL/XSP:

```
(define map (forall (s t)
              (-> ((-> (s) t) (listof s)) (listof t)))
  (plambda (s t)
    (lambda ((fn (-> (s) t)) (lst (listof s)))
      (if ((pcall null? s) lst)
          ((pcall null t))
          ((pcall cons t) (fn ((pcall car s) lst))
                          ((pcall map s t)
                           fn ((pcall cdr s) lst)))))))
```

The `(plambda (s t)` . . .`)` creates a polymorphic value (in this case, a procedure) whose type is abstracted over the type variables `s` and `t`. The `pcall` construct specializes the type of a polymorphic value by filling in the types of these variables:

```
(the (-> ((-> (int) int) (listof int)) (listof int))
     (pcall map int int))

(the (-> ((-> (int) bool) (listof int)) (listof bool))
     (pcall map int bool))
```

Projection allows a polymorphic procedure to be used with different types of arguments:

```
((pcall map int int) (lambda (x) (* x x)) (list 1 2 3))

((pcall map int bool) (lambda (x) (< x 17)) (list 23 13 29))
```

`plambda` and `pcall` have a similar contract to `lambda` and procedure call. But whereas `lambda` and procedure call imply computation at run time, `plambda` and `pcall` imply computation during type checking. That is, `plambda` builds abstractions over types during static analysis; these abstractions are also unwound by `pcall` during static analysis. Every polymorphic value must have its types instantiated (via `pcall`) before it can be used.

FL/XSP requires the explicit projection of polymorphic values via `pcall`. But some polymorphic languages support **implicit projection**, in which the projected types are automatically deduced from context. Implicit projection makes polymorphic programming more palatable by removing some of the overhead of writing explicit types.

In a polymorphic language, general operations on data structures like cells, sums, products, and lists can once again be treated as first-class procedures rather than as special forms. For example, here are the types of the list operators in FL/XSP:

```
(the (forall (t) (-> (t (listof t)) (listof t))) cons)
(the (forall (t) (-> ((listof t)) t))  car)
(the (forall (t) (-> ((listof t)) (listof t))) cdr)
(the (forall (t) (-> ((listof t)) bool)) null?)
(the (forall (t) (-> () (listof t))) null)
```

In FL/XSP, it is even possible to have a polymorphic empty list `nil` with type `(forall (t) (listof t))`. This underscores the fact that polymorphism can be used with all values, not only procedures.

In order to type check expressions involving `plambda` and `pcall`, it is necessary to extend the typing rules, type inclusion rules, and type equivalence as shown in Figure 13.5.

The [$p\lambda$] rule gives a `forall` type to a `plambda`, while the [*project*] rule specifies a beta substitution in the type domain. The [$p\lambda$] rule includes a restriction on the identifiers that `plambda` can abstract over. The restriction uses

Typing Rules

$$\frac{A \vdash E : T}{A \vdash \texttt{(plambda (}I_1 \ \ldots \ I_n\texttt{)} \ E\texttt{)} : \texttt{(forall (}I_1 \ \ldots \ I_n\texttt{)} \ T\texttt{)}} \quad [p\lambda]$$

where $\forall_{i=1}^{n} . I_i \notin (FTV(FreeIds[\![E]\!])A)$
and E is pure. *[purity restriction]*

$$\frac{A \vdash E : \texttt{(forall (}I_1 \ \ldots \ I_n\texttt{)} \ T_E\texttt{)}}{A \vdash \texttt{(pcall} \ E \ T_1 \ \ldots \ T_n\texttt{)} : ([T_i/I_i]_{i=1}^{n}) \ T_E} \quad [project]$$

Type Inclusion Rules

$$\frac{([I_i/J_i]_{i=1}^{n}) \ S \sqsubseteq T, \quad \forall \text{i} \ (I_i \notin FreeIds[\![S]\!])}{\texttt{(forall (}J_1 \ \ldots \ J_n\texttt{)} \ S\texttt{)} \sqsubseteq \texttt{(forall (}I_1 \ \ldots \ I_n\texttt{)} \ T\texttt{)}} \quad [forall\text{-}\sqsubseteq]$$

Type Equivalence

$$\frac{\begin{array}{c}(T_1 \sqsubseteq T_2) \\ (T_2 \sqsubseteq T_1)\end{array}}{T_1 \equiv T_2} \quad [\equiv]$$

Figure 13.5: New rules to handle polymorphism in FL/XSP.

a function *FTV*(which stands for Free Type Variables). (*FTV* I^* *A*) returns the collection of type variables that appear free in the type assignments that *A* gives to the identifiers I^*. This restriction prohibits a subtle form of variable capture. Consider the following example:

```
(define polytest
  (plambda (t)
    (lambda ((x t))
      (plambda (t) x))))
```

What is the type of `polytest`? To say that it is

```
(forall (t) (-> (t) (forall (t) t)))
```

is incorrect, because the `t` introduced by the outer `plambda` has been captured by the inner one. Because of this name capture, the following expression would not type check even though it should:

```
(pcall ((pcall polytest int) 3) bool)
```

In the $[p\lambda]$ rule, we simply outlaw such situations. An implementation could insist programmers enforce the rule, or it could α-rename type variables to guarantee that no capture is possible no matter what names the programmer used.

Note that the rule for type equivalence is broadened to allow equivalence of `forall` types that are the same except for the names chosen for their variables. E.g., this rule allows us to show:

```
(forall (s) (-> (s) s)) ≡ (forall (t) (-> (t) t))
```

▷ **Exercise 13.6** Alyssa P. Hacker wants to remove `error` from the language as a special syntactic construct. She suggests that we add an `error` procedure to the standard environment.

a. Specify the type of the `error` procedure.

b. Illustrate its use by filling in the box in the following example to produce a well-typed expression:

```
(lambda ((x int) (y int))
  (if (= x 0)
      ┌──────────────┐
      │              │
      └──────────────┘
      (/ y x)))
```

◁

▷ **Exercise 13.7** Louis Reasoner has had a hard time implementing `letrec` in a call-by-name version of FL/XSP, and has decided to use the fixed point operator **fix** instead. For example, here is the correct definition of factorial in Louis's approach:

```
(let ((fact-gen
        (lambda ((fact (-> (int) int)))
          (lambda ((n int))
            (if (= n 0) 1 (* n (fact (- n 1))))))))
  ((pcall fix (-> (int) int)) fact-gen))
```

Thus, `fix` is a procedure that computes the fixed point of a generating function. Ben Bitdiddle has been called on the scene to help, and he has ensured that Louis's FL/XSP supports recursive types using `recof`.

a. What is the type of `fact-gen`?

b. What is the type of `fix`?

c. What is the type of `((pcall fix (-> (int) int)) fact-gen)`?

Ben Bitdiddle defined the call-by-name version of `fix` to be:

```
(let ((fix (plambda (t)
             (lambda ((f T1))
               ((lambda ((x T2)) (f (x x)))
                (lambda ((x T2)) (f (x x))))))))
    ... fix can be used here ...
    )
```

d. What is T_1?

e. What is T_2?

f. Louis has decided that he would like `(fix` E`)` to be a standard expression in his language. What is the typing rule for `(fix` E`)`? ◁

13.3 Descriptions

The ability to abbreviate types with `tlet` is not sufficiently powerful to express many desirable abstractions. For example, the `define-type` construct in FL/X is too weak to simplify the definition of `make-tree`, the polymorphic version of `make-int-tree` shown in Figure 13.8. Here the tree type expressions cannot be replaced by some globally named type because they are parameterized over the type `t`, which is local to the definition of `make-tree`. What we'd like in this situation is a `lambda`-like construct in the type domain that would allow the construction of **type abstractions**.[2] In this case, we'd like to define a `treeof` operator in the type domain that would allow us to rewrite `make-tree` as:

[2]Not to be confused with abstract types.

```
(define make-int-tree
        (-> ((recof tree
               (oneof
                 (leaf int)
                 (node (recordof (left tree)
                                 (right tree)))))
             (recof tree
               (oneof
                 (leaf int)
                 (node (recordof (left tree)
                                 (right tree))))))
         (recof tree
           (oneof (leaf int)
                  (node (recordof (left tree)
                                  (right tree))))))
        (lambda ((left-branch (recof tree
                                (oneof
                                  (leaf int)
                                  (node (recordof
                                          (left tree)
                                          (right tree))))))
                 (right-branch (recof tree
                                 (oneof
                                   (leaf int)
                                   (node (recordof
                                           (left tree)
                                           (right tree)))))))
          (one
            (recof tree
              (oneof
                (leaf int)
                (node (recordof (left tree)
                                (right tree)))))
            node (record (left left-branch)
                         (right right-branch)))))
```

Figure 13.6: Lack of type abstraction greatly complicates the definition of a `make-int-tree` procedure.

```
(define-type int-tree
  (recof tree
    (oneof (leaf int)
           (node (recordof (left tree) (right tree))))))

(define make-int-tree (-> (int-tree int-tree) int-tree)
  (lambda ((left-branch int-tree) (right-branch int-tree))
    (one int-tree
         node
         (record (left left-branch)
                 (right right-branch)))))
```

Figure 13.7: Type abstractions simplify the definition of `make-int-tree`.

```
(define make-tree (forall (t) (-> ((treeof t) (treeof t)) (treeof t)))
  (plambda (t)
    (lambda ((left-branch (treeof t))
             (right-branch (treeof t)))
      (one (treeof t)
           node
           (record (left left-branch)
                   (right right-branch))))))
```

Note that a type operator such as `treeof` cannot be created by `lambda` or `plambda`; whereas `lambda` creates procedures that map values to values and `plambda` creates procedures that map types to values, a type operator maps types to types. Therefore, a new kind of `lambda` is needed.

In order to address the issues raised by the above examples, we consider a new language FL/XSPD that is a generalized version of FL/XSP. The grammar for FL/XSPD is given in Figures 13.9 and 13.10.[3] Whereas all type expressions in FL/XSP (generated by nonterminal T) denote types, the type expressions in FL/XSPD (generated by nonterminal D) denote **descriptions**. Descriptions encompass not only types, but also operators on types and, in fact, operators on arbitrary descriptions.[4]

The `define-desc` construct can be used to name descriptions globally. Thus,

[3]The grammar for `program` specifies that all `define-desc`s must precede all `define`s. In spite of this, we will assume that these two forms can be freely intermingled in practice. It is easy to imagine that the FL/X parser translates the more liberal form of `program` into the restricted form specified by the grammar.

[4]Descriptions can be extended to include other information as well, such as **effects**, which indicate the allocation, reading, or writing of a mutable data structure. FX uses descriptions with effects to perform static side-effect analysis on programs.

```
(define make-tree
        (forall (t)
          (-> ((recof tree
                 (oneof
                   (leaf t)
                   (node (recordof (left tree)
                                   (right tree)))))
               (recof tree
                 (oneof
                   (leaf t)
                   (node (recordof (left tree)
                                   (right tree))))))
         (recof tree
           (oneof (leaf t)
                  (node (recordof (left tree)
                                  (right tree)))))))
        (plambda (t)
          (lambda ((left-branch (recof tree
                                  (oneof
                                    (leaf t)
                                    (node (recordof
                                            (left tree)
                                            (right tree))))))
                   (right-branch (recof tree
                                   (oneof
                                     (leaf t)
                                     (node (recordof
                                             (left tree)
                                             (right tree)))))))
            (one
              (recof tree
                (oneof
                  (leaf t)
                  (node (recordof (left tree)
                                  (right tree)))))
              node (record (left left-branch)
                           (right right-branch))))))
```

Figure 13.8: `make-tree`, a version of `make-int-tree` parameterized over the leaf type.

Abstract Syntax:

$P \in$ Program
$I, J \in$ Identifier
$E \in$ Exp
$Y \in$ Symlit
$L \in$ Lit $=$ Unitlit $\cup$ Boollit $\cup$ Intlit $\cup$ Stringlit $\cup$ Symlit
$D \in$ Description

E ::= L | I | `(if` E_1 E_2 E_3`)` | `(begin` E_1 E_2`)` | `(the` D E`)`
  | `(lambda ((`I D`)*)` E_{body}`)` | `(`E_{proc} E_{args}`*)`
  | `(let ((`I E`)*)` E_{body}`)` |`(letrec ((`I D E`)*)` E_{body}`)`
  | `(record (`I E`)*)` | `(with` E_{rec} E_{body}`)`
  | `(one` D I_{tag} E_{val}`)` | `(tagcase` E_{disc} `(`I_{tag} I_{val} E_{body}`)`$^{+}$`)`
  | `(tagcase` E_{disc} `(`I_{tag} I_{val} E_{body}`)`$^{+}$ `(else` $E_{default}$`))`
  | `(plambda (`I`*)` E`)` | `(pcall` E D`*)`
  | `(plet ((`I D`)*)` E_{body}`)` | `(pletrec ((`I D`)*)` E_{body}`)`
  | `(error` D Y`)`

D ::= `int` | `unit` | `bool` | `string` | I
  | `(-> (`D_{arg}`*)` D_{body}`)`
  | `(recordof (`I_{field} D_{val}`)*)` | `(oneof (`I_{tag} D_{val}`)*)` | `(cellof` D`)`
  | `(forall (`I`*)` D`)`
  | `(dlambda (`I`*)` D_{body}`)` | `(`D_{rator} D_{rand}`*)` | `(dlet ((`I D`)*)` D`)`
  | `(drecof` I D`)` | `(dletrec ((`I D`)*)` D_{body}`)`

Figure 13.9: A kernel grammar for FL/XSPD.

Syntactic Sugar:

P ::= `(program` E_{body} `(define-desc` I D`)`* `(define` I D E`)`*`)`

E ::= ... | `(begin` E_1 E_2 ... E_n`)`
  | `(cond (`E_{test} $E_{consequent}$`)`* `(else` $E_{default}$`))`
  | `(letrec (`I D E`)`* E_{body}`)`

```
(program E_body
  (define-desc I_d_1 D_1) ... (define-desc I_d_m D_m)
  (define I_v_1 E_1) ... (define I_v_n E_n))
=
(plet (I_d_1 D_1)
       ⋮
  (plet (I_d_m D_m)
    (letrec ((I_v_1 E_1) ... (I_v_n E_2))
      E_body)) ... )
```

The usual desugarings for `begin`, `cond`, and `letrec`.

Figure 13.10: A grammar for FL/XSPD's syntactic sugar.

it acts like the hypothetical `define-type` in the integer binary tree examples above:

```
(define-desc int-tree
  (drecof t
    (oneof (leaf int)
           (node (recordof (left t) (right t))))))
```

The `dlambda` construct denotes a description operator that takes descriptions as arguments and returns a description as a result. Using `dlambda`, the `treeof` type operator suggested above could be written as

```
(define-desc treeof
  (dlambda (leaf-type)
    (drecof tree
      (oneof (leaf leaf-type)
             (node (recordof (left tree) (right tree)))))))
```

This description operator can then be applied to another description. For example, an alternate definition of the `int-tree` type described above is:

```
(define-desc int-tree (treeof int))
```

Since `listof` can be defined in a way similar to `treeof`, `listof` need not be a primitive type constructor in FL/XSPD. It is worth noting in the above examples that `define-desc` can be used to name both types and operators on types.

Because the arguments and results of description operators may include arbitrary descriptions, it is possible to have higher-order description operators. As an example where this power can be put to use, suppose we are defining mapping procedures for several different homogeneous aggregate data structures. In particular, suppose that `listof` and `vectorof` are type constructors for lists and vectors, respectively. Then the types of the procedures `list-map` and `vector-map` would be as follows:

```
  list-map : (forall (in-type out-type)
               (-> ((-> (in-type) out-type)
                    (listof in-type))
                   (listof out-type)))

vector-map : (forall (in-type out-type)
               (-> ((-> (in-type) out-type)
                    (vectorof in-type))
                   (vectorof out-type)))
```

Clearly there is a common pattern in the types of the two mapping procedures. We can capture this pattern by creating a description operator `map-type`.

```
(define-desc map-type
  (dlambda (type-constructor)
    (forall (in-type out-type)
      (-> ((-> (in-type) out-type)
           (type-constructor in-type))
          (type-constructor out-type)))))
```

Then the types of `list-map` and `vector-map` can be written more succinctly:

```
  list-map : (map-type listof)

vector-map : (map-type vectorof)
```

The `dlet` construct names a description in a local scope. The `drecof` and `dletrec` constructs are used for creating recursive descriptions, such as `treeof`. We have seen versions of these before in FL/XSP, where `drecof` was called `recof` and `dletrec` was called `rectype`. `plet` and `pletrec` are similar to `dlet` and `dletrec` except that they return values rather than descriptions; e.g.,

```
;; DLET returns a description
(dlet ((pairof (dlambda (d1 d2)
                 (recordof (first d1) (second d2)))))
  (pairof int (pairof bool string)))

;; PLET returns a value
(plet ((pairof (dlambda (d1 d2)
                 (recordof (first d1) (second d2)))))
  (the (pairof int (pairof bool string))
       (record (first 3)
               (second (record (first #f)
                               (second "Alyssa"))))))
```

Just as `let` in a typed language cannot be desugared into a `lambda` combination (because type information is lost), it is similarly the case that `plet` cannot be desugared into `plambda` plus `pcall`, nor can `dlet` be desugared into `dlambda` plus a description application.

Intuitively, constructs in the description domain (`define-desc`, `dlambda`, description operator application, `dlet`, and `dletrec`) have a close correspondence with value domain constructs (`define`, `lambda`, value procedure application, `let`, and `letrec`). But how do we formally describe the meanings of the new description expressions that we have introduced? Two new typing rules are needed (see Figure 13.11), but these are for the value-producing `plet` and `pletrec`, not the description producing `dlambda`, `dlet`, `drecof`, and `dletrec`.

$$\frac{A \vdash ([D_i/I_i]_{i=1}^{n})E_{body} : D_{body}}{A \vdash \texttt{(plet ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ E_{body}\texttt{)} : D_{body}} \quad [plet]$$

$$\frac{A \vdash ([I_i : \texttt{(dletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_i\texttt{)}/I_i]_{i=1}^{n})E_{body} : D_{body}}{A \vdash \texttt{(pletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ E_{body}\texttt{)} : D_{body}} \quad [pletrec]$$

Figure 13.11: New typing rules needed for FL/XSPD.

In order to perform type checking in the presence of general descriptions, we require **description equivalence** rules that tell us when two descriptions are the same. Earlier, we saw some type equivalence rules, including ones for `recof` and `rectype`. We need to extend those rules to handle arbitrary descriptions. Figure 13.12 shows the description equivalence rules that are necessary for FL/XSPD.

Some of the the description equivalence rules correspond to the α, β, and η conversion rules of the lambda calculus. Consider the following examples:

$$D \equiv D \qquad [\textit{reflexivity}]$$

$$\frac{D_1 \equiv D_2}{D_2 \equiv D_1} \qquad [\textit{symmetry}]$$

$$\frac{D_1 \equiv D_2 \; ; \; D_2 \equiv D_3}{D_1 \equiv D_3} \qquad [\textit{transitivity}]$$

$$\frac{\forall \text{i} \,.\, (D_i \equiv D_i{}') \;\; ; \;\; D_B \equiv D_B{}'}{\texttt{(-> (}D_1 \ldots D_n\texttt{)}\ D_B\texttt{)} \equiv \texttt{(-> (}D_1{}' \ldots D_n{}'\texttt{)}\ D_B{}'\texttt{)}} \qquad [\texttt{->}\text{-}\equiv]$$

$$\frac{\exists \text{ permuation } \pi \text{ such that } \forall \text{i} \,.\, ((I_i = I'\pi(i)) \wedge (D_i \equiv D_{\pi(i)}{}'))}{\texttt{(recordof (}I_1\ D_1\texttt{)} \ldots \texttt{(}I_n\ D_n\texttt{))} \equiv \texttt{(recordof (}I_1{}'\ D_1{}'\texttt{)} \ldots \texttt{(}I_n{}'\ D_n{}'\texttt{))}} \qquad [\textit{recordof-}\equiv]$$

$$\frac{\exists \text{ permuation } \pi \,. \text{ such that } \forall \text{i} \,.\, ((I_i = I'\pi(i)) \wedge (D_i \equiv D_{\pi(i)}{}'))}{\texttt{(oneof (}I_1\ D_1\texttt{)} \ldots \texttt{(}I_n\ D_n\texttt{))} \equiv \texttt{(oneof (}I_1{}'\ D_1{}'\texttt{)} \ldots \texttt{(}I_n{}'\ D_n{}'\texttt{))}} \qquad [\textit{oneof-}\equiv]$$

$$\frac{D \equiv D'}{\texttt{(refof }D\texttt{)} \equiv \texttt{(refof }D'\texttt{)}} \qquad [\textit{refof-}\equiv]$$

$$\frac{\forall \text{i} \,.\, (J_i \notin \textit{FreeIds}[\![D_B]\!])}{\texttt{(forall (}I_1 \ldots I_n\texttt{)}\ D_B\texttt{)} \equiv \texttt{(forall (}J_1 \ldots J_n\texttt{)}\ ([J_i/I_i]_{i=1}^{n})D_B\texttt{)}} \qquad [\textit{forall-}\equiv]$$

$$\frac{\forall \text{i} \,.\, (J_i \notin \textit{FreeIds}[\![D_B]\!])}{\texttt{(dlambda (}I_1 \ldots I_n\texttt{)}\ D_B\texttt{)} \equiv \texttt{(dlambda (}J_1 \ldots J_n\texttt{)}\ ([J_i/I_i]_{i=1}^{n})D_B\texttt{)}} \qquad [\textit{dlambda-}\equiv]$$

$$\frac{D_P \equiv D_P{}' \;\; ; \;\; \forall \text{i} \,.\, (D_i \equiv D_i{}')}{\texttt{(}D_P\ D_1 \ldots D_n\texttt{)} \equiv \texttt{(}D_P{}'\ D_1{}' \ldots D_n{}'\texttt{)}} \qquad [\textit{dapply-}\equiv]$$

$$\texttt{((dlambda (}I_1 \ldots I_n\texttt{)}\ D_B\texttt{)}\ D_1 \ldots D_n\texttt{)} \equiv ([D_i/I_i]_{i=1}^{n})D_B \qquad [\textit{dbeta-}\equiv]$$

$$\frac{\forall \text{i} \,.\, (I_i \notin \textit{FreeIds}[\![D_P]\!])}{\texttt{(dlambda (}I_1 \ldots I_n\texttt{) (}D_P\ I_1 \ldots I_n\texttt{))} \equiv D_P} \qquad [\textit{deta-}\equiv]$$

$$\texttt{(dlet ((}I_1\ D_1\texttt{)} \ldots \texttt{(}I_n\ D_n\texttt{))}\ D_B\texttt{)} \equiv ([D_i/I_i]_{i=1}^{n})D_B \qquad [\textit{dlet-}\equiv]$$

$$\texttt{(drecof }I\ D\texttt{)} \equiv [D/I]D \qquad [\textit{drecof-}\equiv]$$

$$\begin{array}{l} \texttt{(dletrec ((}I_1\ D_1\texttt{)} \ldots \texttt{(}I_n\ D_n\texttt{))}\ D_B\texttt{)} \\ \equiv ([\texttt{(dletrec ((}I_1\ D_1\texttt{)} \ldots \texttt{(}I_n\ D_n\texttt{))}\ D_i\texttt{)}/I_i]_{i=1}^{n})\ D_B \end{array} \qquad [\textit{dletrec-}\equiv]$$

Figure 13.12: Description equivalence rules for FL/XSPD.

```
(dlambda (t) t)   ≡   (dlambda (s) s)                       ; alpha

; Assume pairof defined as above
(pairof int bool)   ≡   (recordof (left int) (right bool))  ; beta

(dlambda (s t) (pairof s t))   ≡   pairof                   ; eta
```

It is enlightening to compare the three kinds of abstraction that exist in FL/XSPD:

Abstraction Constructor	Arguments	Results
`lambda`	values	values
`plambda`	descriptions	values
`dlambda`	descriptions	descriptions

It is also possible to imagine a fourth kind of abstraction that takes values as arguments and returns descriptions. These can result in what are called **dependent descriptions** — descriptions that contain values. The array type constructor in Pascal is a simple example of a dependent type; every array type has an integer which indicates the length of the array. Of course, in order to ensure static type checking, the argument values to such an abstraction would have to be statically determinable.

It is disturbing that there are three different constructs that are so similar in intent. The need for the differing constructs arises from the fact that we have maintained a rigid distinction between types and values. In the interest of conceptual economy, some languages, such as PEBBLE, blur the distinction between types and values; in these languages, a single operator constructor can do the job of `lambda`, `plambda`, and `dlambda`. Since types can be treated as values in these languages, however, type checking can generally not be performed statically. Instead, it may have to be interleaved with the execution of the program; in such cases, type checking is effectively dynamic. In fact, in some languages with first-class types, type checking might never even terminate!

13.4 Kinds and Kind Checking: FL/XSPDK

It is important to note that only a subset of descriptions serve as types of values. For example, there is no value that has the type `(dlambda (t) t)` or the type `listof`. Furthermore, many expressions generated by the grammar for descriptions are nonsensical. The description `(int bool)`, for instance, indicates that `int` is being applied to `bool` as a description operator. But since `int` is the type of a value and not a description operator, such an application is not meaningful. Even the typing rules in Figure 13.11 are problematic as stated; the notation $I : D$ only makes sense if D is a type.

We'd like to ensure that descriptions make sense, both intrinsically and in context. This problem seems an awful lot like the one we already solved via types; showing that `(int bool)` is not a meaningful description is rather similar to showing that `(1 2)` is not a meaningful expression. Just as we had types for expressions, we'd like to have something akin to types for descriptions. These are called **kinds**; kinds are the types of descriptions.

We incorporate the notion of kinds into the language FL/XSPDK, which is just an extension of FL/XSPD. The grammatical changes necessary to extend FL/XSPD into FL/XSPDK are presented in Figure 13.13. The nonterminal K generates kind expressions, which are now required in `plambda`, `forall`, and `dlambda`.

E ::= ... | `(plambda ((`I K`)*)` E`)`

D ::= ... | `(forall ((`I K`)*)` D`)` | `(dlambda ((`I K`)*)` D_{body}`)`

K ::= `type` | `(->> (`K`*)` K`)`

Figure 13.13: The grammar for FL/XSPDK (the parts not listed are the same as in FL/XSPD).

The simplest kind is the base kind `type`. All legal FL/XSP types have kind `type`. For example, the following expressions all have kind `type`:

```
int
(-> (bool) string)
(recordof (name string) (age int))
```

Description operators have a kind that reflects the kinds of the operator's arguments and the kind of the operator's results. `listof`, for example, has kind `(->> (type) type)` because it takes a type and returns a type. Note the double arrow `->>` is used in the kind of a description operator, whereas `->` is used in the type of a procedure. This notational difference is not strictly necessary but serves to emphasize the distinction between the two levels.

A description is **well-kinded** if it can be assigned a kind according to a set of kind checking rules. Kind checking is analogous to type checking; the notation

$$B \vdash D :: K$$

means that kind environment B assigns kind K to description D. Figure 13.14 includes the kind checking rules for FL/XSPD. ϕ_k indicates the empty kind environment.

$$\phi_k \vdash \texttt{unit} :: \texttt{type}, \phi_k \vdash \texttt{bool} :: \texttt{type}, \phi_k \vdash \texttt{int} :: \texttt{type}, \phi_k \vdash \texttt{string} :: \texttt{type}, \phi_k \vdash \texttt{sym} :: \texttt{type} \quad [\mathit{literal}]$$

$$[\ldots, I :: K, \ldots] \vdash I :: K \quad [\mathit{var}]$$

$$\frac{\forall \mathrm{i}\ .\ (B \vdash D_i :: \texttt{type}) \qquad B \vdash D_{body} :: \texttt{type}}{B \vdash \texttt{(-> (}D_1 \ \ldots\ D_n\texttt{)}\ D_{body}\texttt{)} :: \texttt{type}} \quad [\texttt{->}]$$

$$\frac{\forall \mathrm{i}\ .\ (B \vdash D_i :: \texttt{type})}{B \vdash \texttt{(recordof (}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))} :: \texttt{type}} \quad [\mathit{recordof}]$$

$$\frac{\forall \mathrm{i}\ .\ (B \vdash D_i :: \texttt{type})}{B \vdash \texttt{(oneof (}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))} :: \texttt{type}} \quad [\mathit{oneof}]$$

$$\frac{B \vdash D :: \texttt{type}}{B \vdash \texttt{(refof}\ D\texttt{)} :: \texttt{type}} \quad [\mathit{refof}]$$

$$\frac{B[I_1 :: K_1,\ \ldots,\ I_n :: K_n] \vdash D_B :: \texttt{type}}{B \vdash \texttt{(forall ((}I_1\ K_1\texttt{)}\ \ldots\ \texttt{(}I_n\ K_n\texttt{))}\ D_B\texttt{)} :: \texttt{type}} \quad [\mathit{forall}]$$

$$\frac{B[I_1 :: K_1,\ \ldots,\ I_n :: K_n] \vdash D_B :: K_B}{B \vdash \texttt{(dlambda ((}I_1\ K_1\texttt{)}\ \ldots\ \texttt{(}I_n\ K_n\texttt{))}\ D_B\texttt{)} :: \texttt{(->> (}K_1\ \ldots\ K_n\texttt{)}\ K_B\texttt{)}} \quad [d\lambda]$$

$$\frac{B \vdash D_P :: \texttt{(->> (}K_1\ \ldots\ K_n\texttt{)}\ K_B\texttt{)} \qquad \forall \mathrm{i}\ .\ (B \vdash D_i :: K_i)}{B \vdash \texttt{(}D_P\ D_1\ \ldots\ D_n\texttt{)} :: K_B} \quad [\mathit{dapply}]$$

$$\frac{\forall \mathrm{i}\ .\ (B \vdash D_i :: K_i) \qquad B[I_1 :: K_1,\ \ldots,\ I_n :: K_n] \vdash D_B :: K_B}{B \vdash \texttt{(dlet ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_B\texttt{)} :: K_B} \quad [\mathit{dlet}]$$

$$\frac{B[I :: \texttt{type}] \vdash D :: \texttt{type}}{B \vdash \texttt{(drecof}\ I\ D\texttt{)} :: \texttt{type}} \quad [\mathit{drecof}]$$

$$\frac{\mathrm{B'} = \mathrm{B}[I_1 :: \texttt{type},\ \ldots,\ I_n :: \texttt{type}] \qquad \forall \mathrm{i}\ .\ (B' \vdash D_i :: \texttt{type}) \qquad B' \vdash D_B :: K_B}{B \vdash \texttt{(dletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_B\texttt{)} :: K_B} \quad [\mathit{dletrec}]$$

Figure 13.14: Kind checking rules for FL/XSPDK

How do kinds and kind checking interact with types and type checking? Not only must all user-supplied descriptions in an expression be well-kinded, but the descriptions may also be used in a context that requires them to be of a particular kind, typically kind `type`. For example, the descriptions annotating the formal parameters to a `lambda` expression must be of kind `type`. We express these relationships by including constraints on kinds in the antecedents of type checking rules; see, for example, the type checking rules for FL/XSPDK shown in Figure 13.15. The notation

$$A, B \vdash E : D$$

means that type environment A assigns E type D in the presence of kind environment B. (Note that a double colon is used for the "has-kind" relation, whereas a single colon is used for the "has-type" relation.) The rules in Figure 13.15 suggest that the type checking and kind checking processes can be interleaved into a single process that uses both a type environment and a kind environment. Of course, it is also possible to perform kind checking and type checking in separate phases.

Several of the rules in Figure 13.15 extend the type environment with some bindings. Kind checking is used in these situations to guarantee that the extensions bind identifiers to types and not arbitrary descriptions. That is, the notation

$$A[I_1 : D_1 \ \ldots \ I_n : D_n]$$

only makes sense when all of the D_i have kind `type`.

The substitution $([D_i/I_i]_{i=1}^{n})E_B$ in the rule for `plet` is assumed to do the "right thing." That is, only occurrences of I in descriptions (not value expressions) are substituted for. We leave the formal definition of substitution in this situation as an exercise for the reader.

A desirable goal for typechecking is that it should be guaranteed to terminate. Has the introduction of general descriptions compromised this goal? For example, it is possible to imagine description operators which go into infinite loops when applied. Type checking an expression containing such a description might never terminate.

The kind checking and type checking rules we have presented are carefully constructed so that this situation can never occur. The `drecof`, `dletrec`, and `pletrec` constructs are constrained so that the descriptions they introduce must be of kind `type`. With these kind constraints, descriptions in FL/XSPDK have a property called **strong normalization**. This property means that all descriptions can be reduced to normal form in a finite number of steps.[5] Note

[5]There are typed versions of the lambda calculus that have the strong normalization property; in these systems it is impossible to write a Y operator.

$$\frac{\begin{array}{c}\forall \mathrm{i}\ .\ (B \vdash D_i\ ::\ \texttt{type})\\ A[I_1 : D_1\texttt{,}\ \ldots\texttt{,}\ I_n : D_n], B \vdash E_{body} : D_{body}\end{array}}{A, B \vdash \texttt{(lambda ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ E_{body}\texttt{)} : \texttt{(-> (}D_i\ \ldots\ D_n\texttt{)}\ D_{body}\texttt{)}}\ [\lambda]$$

$$\frac{\begin{array}{c}\forall \mathrm{i}\ .\ (B \vdash D_i\ ::\ \texttt{type})\\ \mathrm{A'} = \mathrm{A}[I_1 : D_1\texttt{,}\ \ldots\texttt{,}\ I_n : D_n]\\ \forall \mathrm{i}\ .\ (A', B \vdash E_i : D_i)\\ A', A \vdash E_{body} : D_{body}\end{array}}{A, B \vdash \texttt{(letrec ((}I_1\ D_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\ E_n\texttt{))}\ E_{body}\texttt{)} : D_{body}}\ [\mathit{letrec}]$$

$$\frac{\begin{array}{c}A, B[I_1 :: K_1\ \ldots\ I_n :: K_n] \vdash E : D\\ \forall \mathrm{i}\ .\ (I_i \notin FTV\ \ (\mathit{FreeIds}[\![(]\!]E)))\end{array}}{A, B \vdash \texttt{(plambda ((}I_1\ K_1\texttt{)}\ \ldots\ \texttt{(}I_n\ K_n\texttt{))}\ E\texttt{)} : \texttt{(forall ((}I_1\ K_1\texttt{)}\ \ldots\ \texttt{(}I_n\ K_n\texttt{))}\ D\texttt{)}}\ [p\lambda]$$

$$\frac{\begin{array}{c}A, B \vdash E : \texttt{(forall ((}I_1\ K_1\texttt{)}\ \ldots\ \texttt{(}I_n\ K_n\texttt{))}\ D_{body}\texttt{)}\\ \forall \mathrm{i}\ .\ (B \vdash D_i\ ::\ K_i)\end{array}}{A, B \vdash \texttt{(pcall}\ E\ D_1\ \ldots\ D_n\texttt{)} : ([D_i/I_i]_{i=1}^{n})\ D_{body}}\ [\mathit{project}]$$

$$\frac{A, B \vdash ([D_i/I_i]_{i=1}^{n})E_B : D_{body}}{A, B \vdash \texttt{(plet ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ E_{body}\texttt{)} : D_{body}}\ [\mathit{plet}]$$

$$\frac{\begin{array}{c}\forall \mathrm{i}\ .\ (B \vdash \texttt{(dletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_i\texttt{)}\ ::\ \texttt{type})\\ A' = A[I_1 : \texttt{(dletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_1\texttt{)}\ \ldots\ I_n : \texttt{(dletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_n\texttt{)}]\\ A', B[I_1 :: \texttt{type}\ \ldots\ I_n :: \texttt{type}] \vdash E_{body} : D_{body}\end{array}}{A, B \vdash \texttt{(pletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ E_{body}\texttt{)} : D_{body}}\ [\mathit{pletrec}]$$

Figure 13.15: Type checking rules for FL/XSPDK. Rules not shown are analogous to those in FL/XSP.

that strong normalization implies that it is impossible to write the Y operator in the description language. Intuitively, this is due to the simplicity of the kind system; there are no recursive kind constructs. Thus, in FL/XSPDK, it is possible to write Y as an expression, but not as a description.

If you're wondering whether it's possible for kinds themselves have something similar to types or kinds, the answer is yes. The types of kinds are sometimes called **sorts**; all kind expressions we have examined are of sort `kind`. But it is possible to consider operators on kinds — kind operators — that would have more interesting sorts. Similarly, we could construct a "typing" system for sorts that distinguished sorts from operators on sorts. Clearly this process could be repeated ad infinitum (and ad nauseum!), giving rise to an infinite "tower" of typing systems. However, only the lowest levels of the tower — types and kinds — are useful in most practical situations.

Reading

The polymorphic typed lambda calculus was invented by Girard and later reinvented by Reynolds [Rey74]. See [Hue90] for some papers on the polymorphic lambda calculus.

For work on types in object-oriented programming, see [GM94].