Chapter 16

Effects Describe Program Behavior

Nothing exists from whose nature some effect does not follow.

— Ethics, I, proposition 36, Benedict Spinoza

16.1 Types, Effects, and Regions - What, How, and Where

We have seen that types are a powerful tool for reasoning about the ideas presented in the FL, Naming, and Data Chapters (Chapter **??**), yet types do not help us reason in detail about the ideas introduced in the State, Control, and Concurrency Chapters (Chapter **??**). A formal system called an **effect system** allows us to reason about many of the state, control, and storage issues that arise in practical programs.

In this chapter, we introduce effect systems and explore their applications. An effect system produces a concise description of the observable actions of an expression, and this description is called the **effect** of the expression. Example effects include writing into a region of the store or jumping to a non-local label. An effect is a **dual** to a type. Just as a type describes **what** an expression computes, an effect describes **how** an expression computes.

As we shall see, effects describe a wide variety of properties about a program that are usefull to programmers, compiler writers, and language designers. Effect systems provide three benefits to the programmer: improved documentation, safety, and execution efficieny. Documentation and saftey improvements include the ability to better understand the behavior of code, including the ability to detemine how modules developed by others may modify state, modify files, perform non-local tranfers, or be the target of non-local jumps. Efficiency improvements include parallel expression scheduling, remote procedure call scheduling, and storage management. For example, when expressions do not share store dependencies, they can be reordered or executed in parallel, subject to their input values being available.

To make effects precise we introduce the idea of **regions** that describe **where** objects reside. In our effect system every object resides in a single region, and this region is described by the type of the object. We can think of regions as colors (red, blue, green, etc.) or as distinct memory banks (Bank 1, Bank 2, Bank 3, etc.), or even machines (mit.edu, cmu.edu, etc.). However, regions are logical locations, and may or may not correspond to physical locations in a given implementation. For example, control points can be assigned to regions that represent locations in code as opposed to regions in a store.

When two objects are in distinct regions mutations to one of the objects will not cause changes to the other object. This is a consequence of our invariant that an object is only in a single region. Thus regions can be used to prove that object references do not alias one another. *Aliasing* occurs when two references refer to the same object. Aliasing can inhibit important compiler optimizations such as caching the values of mutable objects in registers.

To produce an accurate accounting of effects we include three key innovations in our type system. First, the type of every mutable object includes the object's region. Second, we will account for the effect of a procedure in the type of the procedure as a **latent effect** that is realized when the procedure is called. Latent effects communicate the effects of a procedure from the point of the procedure's definition to its points of use. Third, we introduce the idea of effect and region polymorphism to permit procedures to have effects that depend on their input parameters.

Although in this chapter we discuss an interwoven system for effects in regions, it is possible to have effects without regions and regions without effects. In the absence of regions our effect system would be coarse, and would simply report a limited repitore of broad effects. In the absence of an effect system, a region system alone can not deduce when a particular region is accessed or when it becomes inaccessible. Although decoupling effects and regions is possible, we will show that there is no advantage to doing so because we can hide the complexity of a simulatenous effect and region system from programmers.

Ultimately, programmers must find an effect system easy to use and it must produce valuable results. In this chapter we will make an effect system easy to use by making it invisible to programmers. We will make it invisible by performing effect and region reconstruction without programmer declarations or assistance. Early experiments with effect systems showed that programmers had a difficult time composing appropriate effect declarations, and thus the existance of a sound effect reconstruction algorithm is necessary for the practical application of a novel effect system you might think of creating!

Since the types of procedures include effects, effect reconstruction naturally depends upon type reconstruction and vice-versa. We will use FL/R as our base langauge in this chapter, and demonstrate how to reconstruct effects fully automatically in this language context.

In this chapter we introduce two classes of example effects. The first class, store effects (read, write, and create), describe the creation or observation of store based state. The second class, control effects (comefrom, goto), describe the creation of control points (labels) and the transfer of control to control points (gotos). As described above, both store and control effects are subscripted by a region that delinates the scope of the effect. A store region describes a set of cells (usually one), and a control region describes a set of control points (usually one). We will use store and control effects for concreteness, but new effects are readily introduced in the effect system framework, and abstract effects that encapsulate base effects are also possible.

We will introduce store effects with a few short examples, and then provide a complete set of effect system rules. We begin with the standard operations on cells:

E ::= ...	| `(cell` E`)`	[Allocate and initialize a cell]
	| `(:=` E E`)`	[Cell set]
	| `(^` E`)`	[Cell read]

The `:=` and `^` procedures are respectively implemented with the `cell-ref` and `cell-set!` primitives that we previously defined in Chapter 8.

Our effect system will produce a summary of how we use these procedures in an expression. The simple effect system we discuss here does not keep track of the ordering or number of times a particular effect is used. However, we will keep track of what region of the store is subject to an effect. We will use "`!`" as the "has effect" relation for expressions as a complement to the "`:`" relation for "has type."

First, we create a mutable cell that contains an integer. The expression that creates the cell is assigned an `init` (initialize) effect in a new store region named `?r-1`:

```
(cell 1) : (cellof int ?r-1) ! (init ?r-1)
```

Next, we create a boolean cell, set it to true, and read out the contents of the cell. Note that the effects on this boolean cell are in a new store region called

`?r-2`. Whenever possible, we will use a new region for each object we create:

```
(let ((x (cell #f)))
  (begin
    (:= x #t)
    (^ x))) : bool ! (maxeff (init ?r-2) (write ?r-2) (read ?r-2))
```

Higher order procedures, such as the `apply-twice` procedure below, can be polymorphic both in type and effect. In this example, the application of `apply-twice` has the store effects `(read ?r-3)` and `(write ?r-3)`:

```
(let ((apply-twice (lambda (f x) (f (f x))))
      (add-one (lambda (c)
                 (begin (:= c (+ (^ c) 1))
                        c)))
      (counter (cell 0)))
  (begin (apply-twice add-one counter)
         (^ counter)))))  :  int
                          ! (maxeff (init ?r-3)
                                    (write ?r-3)
                                    (read ?r-3))
```

The type schema of `apply-twice` in this example is

```
apply-twice : (generic (tf ft)   ; tf is input and output type of f
                                 ; ft is latent effect of f
                 (-> ((-> (tf) ft tf)  ; f
                      tf)              ; x
                     ft          ; latent effect of apply-twice
                     tf))        ; result
```

Note in this instance that effect polymorphism carries the effect of the procedure provided to `apply-twice` to `apply-twice` itself.

Procedure types in standard environment now have latent effects. Here are the entries in the standard type envirnoment for the free variables in the above example:

```
+    : (-> (int int) pure int)
cell : (generic (t r) (-> (t) (init r) (cellof t r)))
:=   : (generic (t r) (-> ((cellof t r) t) (write r) unit))
^    : (generic (t r) (-> ((cellof t r)) (read r) t))
```

When we generalize over a region in a type schema we are indicating that any region can be assigned. For example, every time that `cell` is used we assign a new region variable to the newly created cell. Thus we try to maximize the number of distinct regions used in a program to provide a fine grained accounting of storage (or other assets). However, keep in mind that we assign a single region for each static occurance of `cell`, and all of the dynamically created cells from a single static occurance of `cell` will wind up in the same region. In addition, when cells are used interchangably, their types will be unified, forcing them to be in the same region.

Of course, many expressions will not have any effects:

```
(+ 1 2) : int ! pure
```

When an expression has no effects we say that the expression is `pure`, and conversely when it has effects we call it **impure**. The `pure` effect is a shorthand for the effect `(maxeff)`. A pure expression is guaranteed to be **referentially transparent**. An expression is referentially transparent when different syntactic occurrences of the expression are guaranteed to have the same value assuming identical bindings for all of the free variables in the expression. We have already discussed the idea of referential transparency in the chapter on state (Chapter 8). Programming languages do not guarantee referential transparency when expressions observe mutable state with expressions that have effects. Thus when an expression is pure, it will be referentially transparent becuase it can not observe mutable state.

Advanced properties of effect systems are beyond the scope of this book, including effect algebras that can associate execution times or storage costs with expressions. These advanced effect algebras require different approaches to type and effect reconstruction than the one we discuss below. The interested reader can consult the bibliography at the end of this chapter for reserach papers on these topics.

In the rest of this chapter, we introduce rules for assigning effects to expressions (Section 16.2), we will discuss how effects can be used to analyze program behavior (Section 16.3), and how effects can be reconstructed as an integral part of a type and effect system (Section 16.4).

16.2 An Effect System for FL/R

Formally, an *effect system* is a set of rules for assigning an effect and a type to an expression. Our effect system needs to assign types to expressions to permit us to analyze the behavior of user defined procedures. Thus, we first extend the syntax of procedure types to include latent effects:

`(-> (`T*`)` F T`)`

where the T^* are the types of the procedure's parameters, F describes the procedure's latent effect, and T is the output type of the procedure.

Figure 16.1 shows the grammar for FL/R language with types that include latent effects. An effect system inherits the names of effects from the latent effects of the primitive operators and procedures that are available in the standard top-level environment. We combine effects with the effect union operator `maxeff` that is associative (A), commutative (C), unitary (U), and has an identity (I) (`pure` is the identity element). Thus, effect combination is an ACUI algebra.

Our algebra of effects is important to consider because the equality of latent effects that occur in procedure types must be considered with respect to our ACUI effect algebra. Because effect combination is commutative, the order in which effects occur is not preserved when effects are combined. Thus procedures that perform equivent operations in different orders will have the same effect according to our algebra of effects.

```
 E ::= L | (if E E E) | (primop O E*) | (let ((I E)*) E)
     | (letrec ((I E)*) E) | (lambda (I*) E) | (E E*)

 T ::= I | (I T) | (-> (T*) F T) | (cellof T R)

 F ::= pure | (maxeff F*) | (I R)

 R ::= I

TS ::= (generic (I*) T)
```

Figure 16.1: Grammar for FL/R with latent effects.

We will now introduce a set of rules for assigning types and effects to FL/R expressions. The rules show us how to deduce the type T and effect F of an expression E given a type environment A:

$$A \vdash E \ : \ T \, ! \, F$$

The standard environment A includes a library of standard procedures (such as `^`, `:=`, etc.), and the types of these procedures include latent effects that describe their actions.

The effect system shown in Figure 16.2 consists of rules that simultaneously compute the type and effect of an expression. FL/R's [*lambda*] and application [*app*] typing rules are extended to permit latent effects to move in and out of procedure types. The [*lambda*] rule moves the effect of a procedure's body into the procedure type of the `lambda` expression, and the [*app*] rule moves the latent effect of a procedure type into the effect of a procedure application. Latent effects and these rules are the mechanism that communicates effect information from the point of procedure definition to the point of procedure call. Other rules such as [*if*] simply combine the effects of all subexpressions of E to compute a conservative approximation of the effects of E.

When our typing rules require that two types T_1 and T_2 be equal, any latent effects that are in identical positions in T_1 and T_2 must be equivalent according to our effect algebra. This occurs when T_1 and T_2 are procedure types since procedure types include latent effects. Recall that effect equivalence is considered with respect to the ACUI algebra for effects, and thus the order of effects does not matter. For example the effects `(maxeff (read ?r-1) (write ?r-2))` and `(maxeff (write ?r-2) (read ?r-1))` are identical.

However, even with our algebra of equivalence over effects, it is a simple matter to construct a program that does not "effect check." We say a program does not effect check when two effects are compared during type checking and the effects do not match. For example

```
(if #t ^ (lambda (c) 1))
```

is not well typed in a strict sense since the cell reference operator `^` and the `lambda` must have identical types but their types do not contain identical latent effects. Note that we do not insist that the consequent and alternative of an `if` have the same effect. Only effects in the types of the consequent and alternative must be the same, and this will only occur when `if` is returning a type that contains a procedure type.

We can make the latent effects in two procedure types equivalent by permitting expressions to take on more effects than they may actually cause to ensure that programs always effect check. Thus our effect system for FL/R includes a subeffecting rule called [*does*] that permits effect expansion. For example, the [*does*] rule permits our example

```
(if #t ^ (lambda (c) 1))
 : (-> ((cellof int ?r-1)) (read ?r-1) int)  ! pure
```

to be well typed by expanding the latent effect of `(lambda (c) 1)` to be `(read ?r-1)` to match the latent effect of the cell reference operator `^`.

Henceforth when we refer to the **effect of an expression**, we will mean the smallest effect that can be proven by our rules. This is because [*does*] permits an expression to take on many possible effects. Effects form a lattice under `maxeff` and thus the notion of a smallest effect is well defined. Later in this chapter when we discuss effect reconstruction (Section 16.4), we will show how to compute the smallest effect allowed by the rules.

Our typing rules for `let` have two forms, [*impure-let*] and [*pure-let*]. As we discussed in our introduction to FL/R, only `let` bindings that do not have side effects can be generalized. In the literature, such let expressions are called "non-expansive." For simplicity we will assume that any expression that is not a `lambda` that includes an application is expansive. It would seem logical to use our own effect system to determine which `let` expressions are pure and thus can be generalized. We leave this extension to the interested reader.

In our typing rules we use FV to denote a function that returns the free type, effect, or region variables in a description expression (type or effect), and FDV to denote a function that returns all of the free type, effect, or region variables in a type environment.

16.3 Using Effects to Analyze Program Behavior

Our exploration of the application of effects will consider how effects can be made to disappear with effect masking, how effects can be used to describe the actions of applets, how effects can be used to describe control transfers, and how static storage allocation can use effects.

16.3.1 Effect Masking Hides Invisible Effects

Effect masking is an important tool for encapsulation. It allows effects to be erased from an expression when the effects cannot be observed from outside of the expression. For example, let's reconsider the effect of the expression we introduced above:

```
(let ((apply-twice (lambda (f x) (f (f x))))
      (add-one (lambda (c) (begin (:= c (+ (^ c) 1)) c)))
      (counter (cell 0)))
  (begin
    (apply-twice add-one counter)
    (^ counter)))))
        : int ! (maxeff (init ?r-3) (write ?r-3) (read ?r-3))
```

Since region `?r-3` is not in the type of this expression and is not in the types of the free variables of this expression (`:=, +,` $\wedge$`, cell`), we know that region `?r-3` is invisible outside of the expression. It is impossible for any context for this expression to determine if the expression has performed any side effects to `?r-3`. Thus, effects on `?r-3` can be erased, leaving this expression with no effect.

$$A[I\!:T] \vdash I : T \ \texttt{!}\ \texttt{pure} \qquad [id]$$

$$[\ldots, I\texttt{:(generic (}I_1 \ \ldots\ I_n\texttt{)}\ T_{body}\texttt{)}, \ldots] \vdash I : ([T_i/I_i]_{i=1}^{n})\,T_{body}\ \texttt{!}\ \texttt{pure} \qquad [genvar]$$

$$\frac{A[I_1 : T_1 \ \ldots\ I_n : T_n] \vdash E : T_r \ \texttt{!}\ F}{A \vdash \texttt{(lambda (}I_1 \ \ldots\ I_n\texttt{)}\ E\texttt{)} : \texttt{(-> (}T_1 \ \ldots\ T_n\texttt{)}\ F\ T_r\texttt{)}\ \texttt{!}\ \texttt{pure}} \qquad [lambda]$$

$$\frac{A \vdash E_o : \texttt{(-> (}T_1 \ldots T_n\texttt{)}\ F_p\ T_r\texttt{)}\ \texttt{!}\ F_o \qquad \forall_{i=1}^{n}\ .\ A \vdash E_i : T_i\ \texttt{!}\ F_i}{A \vdash \texttt{(}E_o\ E_1 \ldots E_n\texttt{)} : T_r\ \texttt{!}\ \texttt{(maxeff}\ F_o\ F_1 \ldots F_n\ F_p\texttt{)}} \qquad [app]$$

$$\frac{A \vdash E_1 : \texttt{bool}\ \texttt{!}\ F_1 \qquad A \vdash E_2 : T\ \texttt{!}\ F_2 \qquad A \vdash E_3 : T\ \texttt{!}\ F_3}{A \vdash \texttt{(if}\ E_1\ E_2\ E_3\texttt{)} : \ T\ \texttt{!}\ \texttt{(maxeff}\ F_1\ F_2\ F_3\texttt{)}} \qquad [if]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n}\ .\ A \vdash E_i : T_i\ \texttt{!}\ \texttt{pure} \\ A[I_1 : \textit{\texttt{Gen}}\texttt{(}T_1\texttt{,}\ A\texttt{)},\ \ldots\ I_n : \textit{\texttt{Gen}}\texttt{(}T_n\texttt{,}\ A\texttt{)}] \vdash E_b : T_b\ \texttt{!}\ F_b\end{array}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_b\texttt{)} : T_b\ \texttt{!}\ F_b} \qquad [pure\text{-}let]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n}\ .\ A \vdash E_i : T_i\ \texttt{!}\ F_i \\ A[I_1 : T_1\texttt{,}\ \ldots\ I_n : T_n] \vdash E_b : T_b\ \texttt{!}\ F_b\end{array}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_b\texttt{)} : T_b\ \texttt{!}\ F_b} \qquad [impure\text{-}let]$$

$$\frac{A \vdash E : T\ \texttt{!}\ F' \qquad F' \sqsubseteq F}{A \vdash E : T\ \texttt{!}\ F} \qquad [does]$$

$$\frac{\begin{array}{c}A_{standard} \vdash O : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ F\ T\texttt{)}\ \texttt{!}\ \texttt{pure} \\ \forall_{i=1}^{n}\ .\ A \vdash E_i : T_i\ \texttt{!}\ F_i\end{array}}{A \vdash \texttt{(primop}\ O\ E_1\ \ldots\ E_n\texttt{)} : T\ \texttt{!}\ \texttt{(maxeff}\ F_1\ \ldots\ F_n\ F\texttt{)}} \qquad [primop]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n}\ .\ A[I_1 : T_1\texttt{,}\ \ldots\ I_n : T_n] \vdash E_i : T_i\ \texttt{!}\ \texttt{pure} \\ A[I_1 : \textit{\texttt{Gen}}\texttt{(}T_1\texttt{,}\ A\texttt{)},\ \ldots\ I_n : \textit{\texttt{Gen}}\texttt{(}T_n\texttt{,}\ A\texttt{)}] : E_b\!: T_b\ \texttt{!}\ F_b\end{array}}{A \vdash \texttt{(letrec ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_b\texttt{)} : T_b\ \texttt{!}\ F_b} \qquad [letrec]$$

$$Gen(T, A) = \texttt{(generic (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}, \text{where } \{I_i\} = FV(T) - FDV(A)$$

Figure 16.2: FL/R Type and Effect Rules

The general rule for effect erasure is

$$\frac{\begin{array}{rcl} A & \vdash & E \,:\, T \,!\, F \\ F' & = & F - \{F_1 \ldots F_n\} \\ \textit{where for all } R_j & \in & FV(F_i),\ R_j \notin FV(T) \ \ [\textit{Export restriction}] \\ \textit{and for all } V_i & \in & \textit{FreeIds}[\![E]\!],\ R_j \notin FV(A[V_i]) \ \ [\textit{Import Restriction}] \end{array}}{A \ \vdash\ E \,:\, T \,!\, F'}$$

The effect erasure rule will detect that certain expressions, while internally impure, are in fact externally pure, and thus are referentially transparent. Thus it permits impure expressions to be included in functional programs. Thus even in functional programs selected program expressions can take advantage of local side effects for efficiency without losing their referential transparency. It also allows effects that denote control transfers (as from `cwcc`) to be masked, indicating that an expression may perform internal control transfers that are not observable outside of the expression. (See Section 16.3.3 for more on control effects.)

16.3.2 Effects Describe the Actions of Applets

One application of effects is to provide applet security by labeling trusted primitive operations with latent effects that describe their actions. For example, all procedures that write on the executing computer's disk could carry a `write-disk` latent effect. Other latent effects could be assigned to display and networking procedures. These effects create a verifiable, succinct summary of the actions of an imported applet. The effects of an applet could be presented to a security checker — such as a user dialog box — that would accept or reject applets on the basis of their effects. In such a system, the vocabulary of effects is defined by the client machine and its effect system, and not by the imported applet. Thus the client security system is able to verify the potential actions of an applet in client defined terms.

An essential part of using types and effects for mobile code security is the ability of the recepient of code to rapidly verify the code's purported type and effect. This is because the type and effect of an applet effectively document its output and observable behavior, and the well-typedness of an applet guarantees that the applet will not perform illegal run-time operations. In a proof carrying code framework, a producer of code provides a series of assetions about code that can be rapidly verified by a code consumer. In proof carrying code, these assertions document saftey properties of the code because the underlying language (e.g. assembly language) may be inherently unsafe. In our framework we too can provide assertions with exported code. When an applet is provided to a consumer we can include a parse tree of the applet with explicit types and effects attached to each expression. With such assertions for every expression it is a simple matter to run our type and effect rules in linear time over the code to verify the purported type and effect of the provided applet.

16.3.3 Effects Describe Control Transfers

Effects can be used to analyze non-local control transfers such as the behavior produced by call-with-current-continuation (`cwcc`). `cwcc` is a procedure that creates a local control point, continues execution, but permits the executed code to invoke the created control point and exit from the body of the `cwcc`. For example, consider the following FL/R example:

```
(lambda (x y)
  (+ 1 (cwcc (lambda (exit)
               (if (= y 0) (exit 0) x)))))
```

In this example, if `y` is 0 the outer `lambda` will return 1, and if `y` is not 0 the outer `lambda` will return `x+1`.

`cwcc` can be understood by consdering the procedure *P* that `cwcc` takes as its only input. *P* receives as its single parameter a continuation procedure *C* that, when called with value *V*, will cause the computation to return from `cwcc` with *V* as its value (see Section 9.4 for more). If *C* is never called, the value returned by *P* is returned by `cwcc`. Whew! Now read that one more time following along with the example above.

The type schema of `cwcc` is rather complicated:

```
(generic (t r t2 f)
  (-> ((-> ((-> (t) (goto r) t2)) f  t))
      (maxeff f (comefrom r))
      t))
```

This type schema shows that `cwcc` takes a procedure *P* with type

```
(-> ((-> (t) (goto r) t2)) f t)
```

that has a latent effect `f` and returns a value of type `t`. Procedure *P* will receive the current continuation, *C*, as an argument. *C* has type `(-> (t) (goto r) t2)`. `cwcc` will return with a value when the value is either provided as an input to *C* or is the return value from *P*. Since either of these choices will cause `cwcc` to return with the provided value, both of these options must insist upon the same type for the value as can be seen in the above type schema.

`cwcc` creates a `comefrom` effect to indicate that control may be transfered back to the return point of `cwcc`. If the continuation C is called after `cwcc` returns, the `cwcc` will return again! Thus the continuation procedure C has a latent `goto` effect that is specific to the continuation's region. Other effects of P (represented by the variable `f`) are assigned to `cwcc`. `(goto r)` will only show up in `f` if P actually invokes C. The type `t2` is generic to allow C to be called in any context because C never returns.

Returning to our example

```
(lambda (x y)
  (+ 1 (cwcc (lambda (exit)
               (if (= y 0) (exit 0) x)))))
```

the expression `(exit 0)` will have a `(goto ?r-4)` effect (where `?r-4` is a new region), and the call to `cwcc` will have the effect `(comefrom ?r-4)`. The `comefrom` effect is used because `cwcc` has established the `exit` control point that permits control to materialize at a later time at the return from the `cwcc`. The effect name `comefrom` is a play on the name `goto`. Finally, in this example, effect masking can be used to drop the `(goto ?r-4)` and `(comefrom ?r-4)` from the `cwcc` expression and thus the latent effect of the `lambda`.

As we have just seen in our small example, effect masking works for all effects including the control effects `comefrom` and `goto`. When a control effect in region R is masked from expression E, it means that any context for expression E will not be subject to unexpected control transfers with respect to the continuation in R. Effect masking of control effects is powerful because it allows module implementors to use control transfers internally, while allowing clients of the modules to insist that these internal control transfers do not alter the clients' control flow. A client can guarantee this invariant by ensuring that it does not call module procedures with control effects.

16.3.4 Effects Can Be Used to Deallocate Storage

In implementations of FL/R class languages, a cell is typically reclaimed by a garbage collector (see Chapter **??**) because it is difficult to statically determine when a cell can no longer be reached. With regions, it is possible to do limited forms of static allocation and deallocation of memory. Assuming we are considering deallocating a region R that occurs in the type of an expression, the rule for storage deallocation is

$$\frac{\begin{array}{c} A \vdash E \; : \; T \; ! \; F \\ R \notin FV(F) \\ R \notin FV(T) \\ \textit{for all } R_i \; (\texttt{comefrom } R_i) \; \notin \; F \\ \textit{for all } I_i \; \in \; \textit{FreeIds}[\![E]\!], \; R \notin \; FV(A[I_i]) \; , \; \textit{for all } R_i \; (\texttt{comefrom } R_i), \; \notin \; A[I_i] \end{array}}{\textit{Deallocate all storage allocated in } R \textit{ after } E}$$

We rely upon our effect system to make this rule sound. Consider the following expression:

```
(let ((my-cell (cell 47))
      (your-cell (cell 48)))
       (lambda () (^ my-cell))).
```

We cannot deallocate `my-cell` after this expression as the latent effect of the procedure returned will contain the region for `my-cell`. This latent effect prevents us from deallocating storage that can be accessed by a procedure. However, we are free to deallocate `your-cell` after this expression returns because it meets the test of our deallocation rule above and thus will no longer be accessible.

Effects can also be used to manage the deallocation of `lambda` storage by associating a region with every procedure type. We leave the details as an exercise for the reader. Note that there are no allocation effects for `lambda` because FL/R does not provide comparison operators on procedures. Thus it is not possible in FL/R to distinguish procedure instances, and thus procedure allocation is not an obervable effect.

16.4 Reconstructing Types and Effects

Our treatment of type reconstruction in Chapter **??** introduced type schemas to permit an identifier to have different types in different contexts. For example, the identity function `(lambda (x) x)` can be used on any type of input. When it is `let` bound to an identifier, it has the type schema `(generic (t) (-> (t) t))`. The job of a type schema is to describe all of the possible types of an identifier by identifying type variables that can be generalized.

Effect and region reconstruction requires us to further elaborate a type schema with effect and region variables. These type schemas also carry along a set of constraints on the effects they describe. We call a type schema that includes a constraint set an **algebraic type schema** [JG91]. A **constraint set** (C) is a set of assertions (A) between effects

C ::= `(`A*`)`
A ::= `(>=` F_1 F_2`)`

where (`>=` F_1 F_2) means that effect F_1 contains all of the effects of effect F_2. The general form of an algebraic type schema is:

TS ::= `(generic (`I^*`)` *T* *C*`)`

For example, the algebraic type schema for a procedure that implements the cell assignment operation (`:=`) is:

```
(generic (t e r)
         (-> ((cellof t r)) e unit)     ; type
         ((>= e (write r)))             ; effect constraints
```

There are three parts to this algebraic type schema. The variables

```
(t e r)
```

describe the type, effect, and region variables that can be generalized in the type schema. The procedure type

```
          (-> ((cellof t r)) e unit)
```

describes the cell assignment operation and notes that its application will have effect `e`. The constraint

```
((>= e (write r)))
```

describes the constraints upon the effect variable `e`. In this case, the assignment operation can have any effect as long as it is larger than `(write r)`.

An integer cell incrementing procedure would have the following algebraic type schema:

```
(generic  (e r)
  (-> ((cellof int r)) e unit)           ; type
  ((>= e (read r)) (>= e (write r))))    ; effect constraints
```

The constraint set in the above type schema constrains the latent effect of the increment procedure to include read and write effects for the region of the cell being incremented.

Type schemas that include effects and constraints are central to Algorithm *Z*, our algorithm for type and effect reconstruction (Figure **??**). Algorithm *Z* is similar to Algorithm *R* from Chapter **??**, except that it simultaneously computes the type and effect of an expression. The unification algorithm U is inherited unchanged from Algorithm *R*. A key intuitive insight into Algorithm *Z* is that constraints are used to keep track of the effects that expressions must have, and the constraints on an expression are solved after the type and effect of the expression is computed. Thus types and effects returned by *Z* can include effect variables that are subject to effect constraints returned by *Z*.

The type and effect reconstruction algorithm Z takes as input an expression E, a type environment A_C, and a starting substitution S. Algorithm Z outputs the type T of the expression, the effect F of the expression, the final substitution S', and the constraint set C on effect variables in T, F, and S'.

$$(Z[\![E]\!]\ A_C\ \ S) = \langle T, F, S', C\rangle$$

Note that in the type environment A_C type schemas include effect constraints as descrbed above.

An expression E is well typed if Z does not fail and if C is solvable. A constraint set C is solvable if there is a concrete assignment of effects M to the effect variables in C that satisfies all of the constraints in C. The substitution of concrete effects to effect variables that satisfies C is called a model. We write that substitution M is a model of C as $M \models C$. In our case, the [*does*] rule allows us to increase the effect of any expression and thus we will always be able to find a model for C.

Algorithm Z in Figures 16.4–16.5 is defined such that its results and a corresponding model M result in a provable type and effect

$$(M\ (S'A_C)) \vdash\ E : (M\ (S'\ \ T))\ \ !\ (M\ (S'\ \ F))$$

(S X) or (M X) means the result of respectively applying the substitution S or M to X, where X can be a type, effect, or a type environment. In the case of a type environment the substitution is applied to all of the identifiers bound in the environment.

An integral part of Algorithm Z is the *Zgen* algorithm for creating algebraic type schemas. The *Zgen* algorithm is identical to the *Rgen* algorithm from our type reconstruction algorithm R, with the key addition of detecting generic effect and region variables and accounting for them by carrying along a copy of a constraint set that can later be instantiated. When a type schema is instantiated, the constraint set carried in the type schema is updated to replace the generic effect and region variables and is returned from Z.

$$\texttt{Zgen}(T,\ A_C,\ S,\ C) = (\texttt{generic}\ (I_1 \ldots I_n)\ \ T\ \ C)$$
$$\{I_1 \ldots I_n\} = FV((S\ \ T)) + FV((S\ \ C)) - FDV((S\ \ A_C))$$

In our definition of `Zgen`, we have used FV to denote a function that returns the free type, effect or region variables in a type or constraint expression, and FDV to denote a function that returns all of the free type, effect, or region variables in a type environment .

An example helps to clarify the role of constraints kept in type schemas. Consider the type schema of the integer cell incrementing procedure `plus-one`

```
(generic (e r)
         (-> ((cellof int r)) e unit)
         ((>= e (read r)) (>= e (write r))))
```

and assume that `c` has type

```
c: (cellof int ?r-5)
```

Then

```
(plus-one c) : unit ! ?e-1
```

The constraints created by `(plus-one c)` will be

```
((>= ?e-1 (read ?r-5)) (>= ?e1 (write ?r-5)))
```

and thus an application of `plus-one` will have `read` and `write` effects. Note that the effect of the `plus-one` procedure is a variable. Our algorithm for creating procedure types always uses a unification variable to represent the effect of a procedure in a procedure type. The effects of the procedure's body are placed into the constraint set to bound the newly introduced variable. The advantage of this approach is that two procedure types always have compatible effect components because they can be unified together. The consequence of unifying the latent effects of two procedures is that both of the procedure bodies will have their effect bounds combined.

An expression E is well typed if Z does not fail and if the resulting constraint set C is solvable. A minimal solution for a constraint set C that assigns concrete effects to effect variables can be found using Algorithm Solve shown in Figure 16.3. Solve is used to solve the constraint set C after the final substitution produced by Z is applied to the constraints. Solve will always succeed because of the [*does*] rule, and thus every expression that is well typed without effects will have both a type and a conservative effect in our type and effect system. Another way to see this is that if two types contain different effects that must be made equal, the [*does*] rule allows us to choose their least upper bound as a common effect. This is precisely what Algorithm Solve does.

▷ **Exercise 16.1** Complete Algorithm Z to handle *impure* let. ◁

▷ **Exercise 16.2** Imagine that `lambda` is extended in FL/R to create a procedure in a region. Thus every procedure type will have a region that identifies where the procedure is located.

a. Give a revised type grammar for FL/R.

b. Give a revised typing rule for `lambda`.

c. Give the revised portion of Algorithm Z for `lambda`.

```
Set all effect variables $I_j$ := pure;
Changed := true;
While Changed
    Changed := false;
    For every constraint $C_i$
        ; Constraint $C_i$ is of the form (>= $I_j$ $E_j$) on variable $I_j$ with effect $E_j$
        ; $E_j$ is evaluated with respect to current effect variable assignments
        If $I_j \neq$ (maxeff $E_j$ $I_j$) then begin
            Changed := true;
            $I_j$ := (maxeff $E_j$ $I_j$);
            End If;
    End For;
End While;
```

Figure 16.3: Algorithm Solve.

d. Procedures can be *stack allocated* when they are *not* returned out of the context where they are created or stored in a cell. Give a rule using the regions in procedure types that identifes procedure regions that can be stack allocated. ◁

▷ **Exercise 16.3** Costs

Sam Antics has a new idea for a type system that is intended to help programmers estimate the running time of their programs. His idea is to develop a set of static rules that will assign every expression a *cost* as well as a type. The cost of an expression is a conservative estimate of how long the expression will take to evaluate.

Sam has developed a new language, called Discount, that uses his cost model. Discount is a call-by-value, statically typed functional language with type reconstruction. Discount is based on FL/R, and inherits its types, with one major difference: a function type in Discount includes the *latent cost* of the function, that is, the cost incurred when the function is called on some arguments.

For example, the Discount type `(-> (int int) 4 int)` is the type of a function that takes two `int`s as arguments, returns an `int` as its result, and has cost at most 4 every time it is called.

The grammar of Discount is shown in Figure 16.1 except that procedure types are altered to include latent costs instead of latent effects:

C ::= `loop` | I | `(sum` C*`)` | `(max` C*`)` | `0` | `1` | `2` | ...

T ::= `int` | `bool` | I | `(-> (`T*`)` C T_{body}`)`

Sam has formalized his system by defining type/cost rules for Discount. The rules allow judgments of the form

$$A \vdash E : T \,\$\, C,$$

```
$(Z[\![\texttt{\#u}]\!]\ A\ S)\ =\ \langle \texttt{unit}, \texttt{pure}, S, \texttt{()}\rangle$

$(Z[\![I]\!]\ A[\ldots,\ I\!:T,\ \ldots]\ S)\ =\ \langle T, \texttt{pure}, S, \texttt{()}\rangle$

$(Z[\![I]\!]\ A[\ldots,\ I\!:\texttt{(generic (}I_1\texttt{..}I_n\texttt{)}\ T\ C\texttt{)},\ \ldots]\ S)\ =$
    $\langle [?D_i/I_i]T, \texttt{pure}, S, [?D_i/I_i]C\rangle$
    All $?D_i$ are fresh and represet type, effect, or region variables


$(Z[\![I]\!]\ A\ S)\ =\ \texttt{fail}$, where $I$ unbound in $A$

$(Z[\![\texttt{(if}\ E_{test}\ E_{con}\ E_{alt}\texttt{)}]\!]\ A\ S)\ =$
  let $\langle T_{test}, F_{test}, S_{test}, C_{test}\rangle$ be $(Z[\![E_{test}]\!]\ A\ S)$ in
   let $S_{test}{}'$ be $U(T_{test}, \texttt{bool}, S_{test})$ in
    let $\langle T_{con}, F_{con}, S_{con}, C_{con}\rangle$ be $(Z[\![E_{con}]\!]\ A\ S_{test}{}')$ in
     let $\langle T_{alt}, F_{alt}, S_{alt}, C_{alt}\rangle$ be $(Z[\![E_{alt}]\!]\ A\ S_{con})$ in
      let $S_{alt}{}'$ be
          $U(T_{con}, T_{alt}, S_{alt})$ in
       $\langle T_{alt}, \texttt{(maxeff}\ F_{test}\ F_{con}\ F_{alt}\texttt{)}, S_{alt}{}', C_{test}\texttt{+}C_{con}\texttt{+}C_{alt}\rangle$

$(Z[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E\texttt{)}]\!]\ A\ S)\ =$
  let $\langle T, F, S, C\rangle$ be $(Z[\![E]\!]\ A[I_1\!:\!\texttt{?v}_1\ \ldots\ I_n\!:\!\texttt{?v}_n]\ S)$ in
   $\langle \texttt{(-> (?v}_1\ \ldots\ \texttt{?v}_n\texttt{) ?e}_n\ T\texttt{)}, \texttt{pure}, S, C\texttt{+((>= ?e}\ F\texttt{))}\rangle$


$(Z[\![\texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)}]\!]\ A\ S)\ =$
   let $\langle T_{rator}, F_{rator}, S_{rator}, C_{rator}\rangle$ be $(Z[\![E_{rator}]\!]\ A\ S)$ in
    let $\langle T_1, F_1, S_1, C_1\rangle$ be $(Z[\![E_1]\!]\ A\ S_{rator})$ in
        $\vdots$
     let $\langle T_n, F_n, S_n, C_n\rangle$ be $(Z[\![E_n]\!]\ A\ S_{n-1}\ C_{n-1})$ in
      let $S_{final}$ be $U(T_{rator}, (->\ (T_1\ \ldots\ T_n)\ \texttt{?e ?t}))$ in
       $\langle \texttt{?t}, \texttt{(maxeff}\ F_{rator}\ F_1\ldots F_n\ \texttt{?e)}, S_{final}, C_{rator}\texttt{+}C_1\texttt{+}\ldots\texttt{+}C_n\rangle$


$(Z[\![\texttt{(let ((}I_1\ E_1\texttt{)..(}I_n\ E_n\texttt{))}\ E\texttt{)}]\!]\ A\ S)\ =$   ; Pure LET
  let $\langle T_1, F_1, S_1, C_1\rangle$ be $(Z[\![E_1]\!]\ A\ S)$ in
        $\vdots$
   let $\langle T_n, F_n, S_n, C_n\rangle$ be $(Z[\![E_n]\!]\ A\ S_{n-l})$ in
    let $\langle T, F, S, C\rangle$ be
        $(Z[\![E]\!]$
         $A[I_1\!:\!\texttt{Zgen}(T_1,\ A,\ S_n,\ C_n),\ \ldots,\ I_n\!:\!\texttt{Zgen}(T_n,\ A,\ S_n,\ C_n)]$
         $S_n)$ in
     $\langle T, \texttt{(maxeff}\ F\ F_1\ldots F_n, S, C\texttt{+}C_1\texttt{+}\ldots\texttt{+}C_n\rangle$
```

Figure 16.4: Algorithm Z reconstructs Types, Regions, and Effects, Part I

```
$(Z[\![$(letrec (($I_1$ $E_1$) ... ($I_n$ $E_n$)) $E$)$]\!]$ $A$ $S$) =
   let $A_1$ be $A$[$I_1$:?t$_1$,...,$A_n$:?t$_n$] in
    let $\langle T_1, F_1, S_1, C_1\rangle$ be $(Z[\![E_1]\!]\ A_1\ S)$ in
        ⋮
     let $\langle T_n, F_n, S_n, C_n\rangle$ be $(Z[\![E_n]\!]\ A_1\ S_{n-1}\ C_{n-1})$ in
      let $S_b$ be $U$((?t$_1$ ...?t$_n$),($T_1$ ... $T_n$),$S_n$) in
       let $\langle T, F, S, C\rangle$ be
          $(Z[\![E]\!]$
            $A$[$I_1$:Zgen($T_1$, $A$, $S_n$, $C_n$), ..., $I_n$:Zgen($T_n$, $A$, $S_n$, $C_n$)]
            $S_b$) in
          $\langle T$, (maxeff $F$ $F_1$...$F_n$, $S$, $C$+$C_1$+...+$C_n\rangle$
```

Figure 16.5: Algorithm Z reconstructs Types, Regions, and Effects, Part II

which is pronounced, "in the type environment A, expression E has type T and cost C."

For example, here are Sam's type/cost rules for literals and(non-generic) identifiers:

$$A \vdash U : \texttt{int}\ \$\ 1$$
$$A \vdash B : \texttt{bool}\ \$\ 1$$
$$A[I : T] \vdash I : T\ \$\ 1$$

That is, Sam assigns both literals and identifiers a cost of 1. In addition:

- The cost of a `lambda` expression is 2.
- The cost of an `if` expression is 1 plus the cost of the predicate expression plus the maximum of the costs of the consequent and alternate.
- The cost of an N argument application is the sum of the cost of the operator, the cost of each argument, the latent cost of the operator, and N.
- The cost of an N argument primop application is the sum of the cost of each argument, the latent cost of the primop, and N. The latent cost of the primop is determined by a *signature* Σ, a function from primop names to types. For example,

$$\Sigma(\texttt{+}) = \texttt{(-> (int int) 1 int)}.$$

Here are some example judgments that hold in Sam's system:

$$A \vdash \texttt{(primop + 2 1)} : \texttt{int}\ \$\ 5$$
$$A \vdash \texttt{(primop + (primop + 1 2) 4)} : \texttt{int}\ \$\ 9$$
$$A \vdash \texttt{(primop + 2 ((lambda (y) (primop + y 1)) 3))} : \texttt{int}\ \$\ 13$$

`Loop` is the cost assigned to expressions that may diverge. For example, the expression

```
(letrec ((my-loop (lambda () (my-loop))))
  (my-loop))
```

is assigned cost `loop` in Discount. Because it is undecidable whether an arbitrary expression will diverge, we cannot have a decidable type/cost system in which *exactly* the diverging expressions have cost `loop`. We will settle for a system that makes a *conservative approximation*: every program that diverges will be assigned cost `loop`, but some programs that do not diverge will also be assigned `loop`.

Because Discount has non-numeric costs, like `loop` and cost identifiers (which we won't discuss), it is not so simple to define what we mean by statements like "the cost is the sum of the costs of the arguments. . . ." That is the purpose of the costs `(sum` C_1 C_2`)` and `(max` C_1 C_2`)`. Part of Sam's system ensures that `sum` and `max` satisfy sensible cost equivalent axioms, such as the following:

$$\begin{array}{rcl}
(\texttt{sum}\ U_1\ U_2) & = & U_1 + U_2 \\
(\texttt{sum loop}\ U) & = & \texttt{loop} \\
(\texttt{sum}\ U\ \texttt{loop}) & = & \texttt{loop} \\
(\texttt{sum loop loop}) & = & \texttt{loop} \\
(\texttt{max}\ U_1\ U_2) & = & \text{the max of } U_1 \text{ and } U_2 \\
(\texttt{max loop}\ U) & = & \texttt{loop} \\
(\texttt{max}\ U\ \texttt{loop}) & = & \texttt{loop} \\
(\texttt{max loop loop}) & = & \texttt{loop}
\end{array}$$

You do not have to understand the details of how cost equivalences are proved in order to solve this problem.

a. Give a type/cost rule for `lambda`.

b. Give a type/cost rule for application.

c. Give a type/cost rule for `if`.

◁

Reading

The first paper on effect systems outlined the need for a new kind of static analysis [LG88], and this early effect system was later extended to include regions in the FX-89 programming language [?]. Region and and effect inference were developed next [JG91]. A wide variety of effect systems have been developed, from systems for cost accounting [DJG92, RG94], to control effects [JG89], to region based memory management [?]. The FX-91 programming language [GJSO92] included all of these features.