# Part XIII

# Modalities

# Chapter 37

# Computational Effects

In this chapter we study the use of types to segregate expressions that use computational effects from those that do not. While it is difficult to be precise about what constitutes a computational effect, a rough-and-ready rule is *any behavior that constrains the order of execution beyond that the requirements imposed by the flow of data*. For example, since the order in which input or output is performed clearly matters to the meaning of a program, these operations may be classified as effects. Similarly, mutation of data structures (as described in Chapter 35) is clearly sensitive to the order in which they are executed, and so mutation should also be classified as an effect.

The trouble with computational effects is precisely that they constrain the order of evaluation. This inhibits the use parallelism (Chapter 44) or laziness (Chapter 42), and generally makes it harder to reason about the behavior of a piece of code. But it should ideally be possible to take advantage of these concepts when effects are *not* used, rather than always planning for the possibility that they might be used. One way to achieve this is to introduce a distinction, called a *modality*, between two modes of expression:

1. The *pure* expressions that are executed solely for their value, and that may engender no effects.

2. The *impure* expressions, or *commands*, that are executed for their value and their effect.

The mode distinction gives rise to a new form of type, called the *lax modality*, or *monad*, whose elements are unevaluated commands. These commands can be passed as pure data, or activated for use by a special form of command.

## 37.1 A Modality for Effects

The syntax of $\mathcal{L}\{\texttt{cmd}\}$ is given by the following grammar:

| *Category* | *Item* | | *Abstract* | *Concrete* |
|---|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{cmd}(\tau)$ | $\tau\ \texttt{cmd}$ |
| Expr | $e$ | ::= | $x$ | $x$ |
| | | $\mid$ | $\texttt{cmd}(m)$ | $\texttt{cmd}(m)$ |
| Comm | $m$ | ::= | $\texttt{return}(e)$ | $\texttt{return}(e)$ |
| | | $\mid$ | $\texttt{letcmd}(e; x.m)$ | $\texttt{let cmd}(x)\ \texttt{be}\ e\ \texttt{in}\ m$ |

The language $\mathcal{L}\{\texttt{cmd}\}$ distinguishes two modes of expression, the pure (effect-free) *expressions*, and the impure (effect-capable) *commands*. The modal type $\texttt{cmd}(\tau)$ consists of suspended commands that, when evaluated, yield a value of type $\tau$. The expression $\texttt{cmd}(m)$ introduces an unevaluated command as a value of modal type. The command $\texttt{return}(e)$ returns the value of $e$ as its value, without engendering any effects. The command $\texttt{letcmd}(e; x.m)$ activates the suspended command obtained by evaluating the expression $e$, then continue by evaluating the command $m$. This form sequences evaluation of commands so that there is no ambiguity about the order in which effects occur during evaluation.

The static semantics of $\mathcal{L}\{\texttt{cmd}\}$ consists of two forms of typing judgement, $e : \tau$, stating that the expression $e$ has type $\tau$, and $m \sim \tau$, stating that the command $m$ only yields values of type $\tau$. Both of these judgement forms are considered with respect to hypotheses of the form $x : \tau$, which states that a variable $x$ has type $\tau$. The rules defining the static semantics of $\mathcal{L}\{\texttt{cmd}\}$ are as follows:

$$\frac{\Gamma \vdash m \sim \tau}{\Gamma \vdash \texttt{cmd}(m) : \texttt{cmd}(\tau)} \tag{37.1a}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}(e) \sim \tau} \tag{37.1b}$$

$$\frac{\Gamma \vdash e : \texttt{cmd}(\tau) \quad \Gamma, x : \tau \vdash m \sim \tau'}{\Gamma \vdash \texttt{letcmd}(e; x.m) \sim \tau'} \tag{37.1c}$$

The dynamic semantics of an instance of $\mathcal{L}\{\texttt{cmd}\}$ is specified by two transition judgements:

1. *Evaluation* of expressions, $e \mapsto e'$.
2. *Execution* of commands, $m \mapsto m'$.

The rules of expression evaluation are carried over from the effect-free setting without change. There is, however, an additional form of value, the encapsulated command:

$$\frac{}{\texttt{cmd}(m)\ \mathsf{val}} \tag{37.2}$$

Observe that $\texttt{cmd}(m)$ is a value regardless of the form of $m$. This is because the command is not executed, but only encapsulated as a form of value.

The rule of execution enforce the sequential execution of commands.

$$\frac{e \mapsto e'}{\texttt{return}(e) \mapsto \texttt{return}(e')} \tag{37.3a}$$

$$\frac{e\ \mathsf{val}}{\texttt{return}(e)\ \mathsf{final}} \tag{37.3b}$$

$$\frac{e \mapsto e'}{\texttt{letcmd}(e; x.m) \mapsto \texttt{letcmd}(e'; x.m)} \tag{37.3c}$$

$$\frac{m_1 \mapsto m_1'}{\texttt{letcmd}(\texttt{cmd}(m_1); x.m_2) \mapsto \texttt{letcmd}(\texttt{cmd}(m_1'); x.m_2)} \tag{37.3d}$$

$$\frac{\texttt{return}(e)\ \mathsf{final}}{\texttt{letcmd}(\texttt{cmd}(\texttt{return}(e)); x.m) \mapsto [e/x]m} \tag{37.3e}$$

Rules (37.3a) and (37.3c) specify that the expression part of a `return` or `let` command is to be evaluated before execution can proceed. Rule (37.3b) specifies that a `return` command whose argument is a value is a final state of command execution. Rule (37.3d) specifies that a `letcomp` activates an encapsulated command, and Rule (37.3e) specifies that a completed command passes its return value to the body of the `let`.

## 37.2 Imperative Programming

The bind construct imposes a sequential evaluation order on commmands, according to which the encapsulated command is executed prior to execution of the body of the bind. This gives rise to a familiar programming idiom, called *sequential composition*, which we now derive from the lax modality.

Since there are only two constructs for forming commands, the bind and the return command, it is easy to see that a command of type $\tau$ always has the form

$$\texttt{let cmd}(x_1)\ \texttt{be}\ e_1\ \texttt{in} \ldots \texttt{let cmd}(x_n)\ \texttt{be}\ e_n\ \texttt{in return}(e),$$

where $e_1 : \tau_1\,\mathtt{cmd}$, ..., $e_n : \tau_n\,\mathtt{cmd}$, and $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$. The dynamic semantics of $\mathcal{L}\{\mathtt{cmd}\}$ specifies that this is evaluated by evaluating the expression, $e_1$, to an encapsulated command, $m_1$, then executing $m_1$ for its value and effects, then passing this value to $e_2$, and so forth, until finally the value determined by the expression $e$ is returned.

To execute $m_1$ and $m_2$ in sequence, where $m_2$ may refer to the value of $m_1$ via a variable $x_1$, we may write

$$\mathtt{let\ cmd}(x_1)\ \mathtt{be\ cmd}(m_1)\ \mathtt{in}\ m_2.$$

This encapsulates, and then immediate activates, the command $m_1$, binding it value to $x_1$, and continuing by executing $m_2$. More generally, to execute a sequence of commands in order, passing the value of each to the next, we may write

$$\mathtt{let\ cmd}(x_1)\ \mathtt{be\ cmd}(m_1)\ \mathtt{in} \ldots \mathtt{let\ cmd}(x_{k-1})\ \mathtt{be\ cmd}(m_{k-1})\ \mathtt{in}\ m_k.$$

Notationally, this quickly gets out of hand. We therefore introduce the *do syntax*, which is reminiscent of the notation used in many imperative programming languages. The binary `do` construct, $\mathtt{do}\,\{x \leftarrow m_1\ ;\ m_2\}$, stands for the command

$$\mathtt{let\ cmd}(x)\ \mathtt{be\ cmd}(m_1)\ \mathtt{in}\ m_2,$$

which executes the commands $m_1$ and $m_2$ in sequence, passing the value of $m_1$ to $m_2$ via the variable $x$. The general `do` construct,

$$\mathtt{do}\,\{x_1 \leftarrow m_1\ ; \ldots ;\ x_k \leftarrow m_k\ ;\ \mathtt{return}(e)\},$$

is defined by iteration of the binary `do` as follows:

$$\mathtt{do}\,\{x_1 \leftarrow m_1\ ; \ldots \mathtt{do}\,\{x_k \leftarrow m_k\ ;\ \mathtt{return}(e)\} \ldots\}.$$

This notation is remiscent of that used in many well-known programming languages. The point here is that sequential composition of commands arises from the presence of the lax modality in the language. In other words conventional imperative programming languages are implicitly structured by this type, even if the connection is not made explicit.

## 37.3 Integrating Effects

The modal separation of expressions from commands ensures that the semantics of expression evaluation is not compromised by the possibility of

effects. One consequence of this restriction is that it is impossible to define an expression $x : \tau\,\texttt{cmd} \vdash \texttt{run}\,x : \tau$ whose behavior is to unbundle the command bound to $x$, execute it, and return its value as the value of the entire expression. For if such an expression were to exist, expression evaluation would engender effects, ruining the very distinction we are trying to preserve!

The only way for a command to occur inside of an expression is for it to be encapsulated as a value of modal type. To execute such a command it is necessary to bind it to a variable using the bind construct, which is itself a form of command. This is the essential means by which effects are confined to commands, and by which expressions are ensured to remain pure. Put another way, it is impossible to define an *expression* $\texttt{run}\,e$ of type $\tau$, where $e : \tau\,\texttt{cmd}$, whose value is the result of running the command encapsulated in the value of $e$. There is, however, a *command* $\texttt{run}\,e$ defined by

$$\texttt{let cmd}(x)\,\texttt{be}\,e\,\texttt{in return}(x),$$

which executes the encapsulated command and returns its value.

Now consider the extension of $\mathcal{L}\{\texttt{cmd}\}$ with function types. Recall from Chapter 13 that a function has the form $\lambda(x{:}\tau.\,e)$, where $e$ is a (pure) expression. In the context of $\mathcal{L}\{\texttt{cmd}\}$ this implies that no function may engender an effect when applied! For example, it is not possible to write a function of the form $\lambda(x{:}\texttt{unit}.\,\texttt{print "hello"})$ that, when applied, outputs the string `hello` to the screen.

This may seem like a serious limitation, but this apparent "bug" is actually an important "feature." To see why, observe that the type of the foregoing function would, in the absence of the lax modality, be something like $\texttt{unit} \to \texttt{unit}$. Intuitively, a function of this type is either the identity function, the constant function returning the null tuple (this is, in fact, the identity function), or a function that diverges or incurs an error when applied (in the presence of such possibilities). But, above all, it *cannot* be the function that prints `hello`.

However, let us consider the closely related type $\texttt{unit} \to (\texttt{unit}\,\texttt{cmd})$. This is the type of functions that, when applied, yield an *encapsulated command*, of type `unit`. One such function is

$$\lambda(x{:}\texttt{unit}.\,\texttt{cmd}(\texttt{print "hello"})).$$

This function *does not* output to the screen when applied, since no pure function can have an effect, but it *does* yield a command that, when executed, performs this output. Thus, if $e$ is the above function, then the

command

$$\texttt{let}\ \texttt{cmd}(\_)\ \texttt{be}\ e(\langle\rangle)\ \texttt{in}\ \texttt{return}(\langle\rangle) \tag{37.4}$$

executes the encapsulated command yielded by $e$ when applied, engendering the intended effect, and returning the trivial element of unit type.

The importance of this example lies in the distinction between the type $\texttt{unit} \to \texttt{unit}$, which can only contain uninteresting functions such as the identity, and the type $\texttt{unit} \to (\texttt{unit}\ \texttt{cmd})$, which reveals in its type that the result of applying it is an encapsulated command that may, when executed, engender an arbitrary effect. In short, the type reveals the reliance on effects. The function type retains its meaning, and, in combination with the lax modality, provides a type of *procedures* that yield a command when applied. A *procedure call* is implemented by combining function application with the modal bind operation in the manner illustrated by expression (37.4).

## 37.4 Exercises

# Chapter 38

# Monadic Exceptions

As we saw in Chapter 37, if an expression can raise an exception, then the order of evaluation of sub-expressions of an expression is significant. For example, the expression $e_1 + e_2$ is not in general equivalent to $e_2 + e_1$, even though addition is commutative. This is so because in the presence of exceptions an expression of type `nat` need not evaluate to a number—it can, instead, raise an exception. If $e_1$ is `raise(L)` and $e_2$ is `raise(R)`, then we may use a handler to distinguish the two addition expressions from each other, yielding, say, zero in the one case and one in the other.

The semantics of expressions may be preserved even in the presence of exceptions if we confine them to the monad by making the primitives for raising and handling exceptions commands, rather than expressions. In this chapter we study a variation on $\mathcal{L}\{\texttt{cmd}\}$ in which exceptions are treated as an impurity to be confined to commands. It should be noted, however, that this approach is unsatisfactory for two related reasons. First, because the monad imposes a strict sequential execution order on commands, the programmer must specify an evaluation order whenever an exception might be raised. Second, if any exception can appear *somewhere* in a program, then it must be structured as though an exception could appear *anywhere*. This is because there is no means of "escaping the monad"—an impurity somewhere infects all parts of the program that depend on its result.

## 38.1 Monadic Exceptions

The most natural formalization of exceptions in the monadic framework is to regard an exception as an alternative outcome of evaluation of a com-

mand. That is, a command, when executed, may engender effects, and then either return a value (as in $\mathcal{L}\{\texttt{cmd}\}$) or raise an exception. The language $\mathcal{L}\{\texttt{comm exc}\}$ is an extension of $\mathcal{L}\{\texttt{cmd}\}$ to account for this additional outcome of execution. The following grammar specifies the characteristic features of $\mathcal{L}\{\texttt{comm exc}\}$:

| *Category* | *Item* | | *Abstract* | *Concrete* |
|---|---|---|---|---|
| Comm | $m$ | $::=$ | $\texttt{raise[}\tau\texttt{](}e\texttt{)}$ | $\texttt{raise(}e\texttt{)}$ |
| | | $\mid$ | $\texttt{letcomp(}e\texttt{;}x\texttt{.}m_1\texttt{;}y\texttt{.}m_2\texttt{)}$ | $\texttt{let cmd(}x\texttt{) be }e\texttt{ in }m_1\texttt{ ow(}y\texttt{) in }m_2$ |

This grammar extends that of $\mathcal{L}\{\texttt{cmd}\}$ with a new primitive command, $\texttt{raise(}e\texttt{)}$, that raises an exception with value $e$. It also modifies the grammar of $\mathcal{L}\{\texttt{cmd}\}$ to generalize the monadic bind construct to include an exception handler. The command

$$\texttt{let cmd(}x\texttt{) be }e\texttt{ in }m_1\texttt{ ow(}y\texttt{) in }m_2$$

executes the encapsulated command determined by evaluation of $e$. If it returns normally, then the return value is bound to $x$ and the command $m_1$ is executed. If, instead, it raises an exception, the exception value is bound to $y$ and the command $m_2$ is executed instead. The monadic bind construct of $\mathcal{L}\{\texttt{cmd}\}$ is to be regarded as short-hand for the command

$$\texttt{let cmd(}x\texttt{) be }e\texttt{ in }m\texttt{ ow(}y\texttt{) in raise(}y\texttt{),}$$

which behaves as before in the case of a normal return, and propagates any exception in that case of an exceptional return.

The static semantics of these constructs is given by the following rules:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise[}\tau\texttt{](}e\texttt{)} \sim \tau} \tag{38.1a}$$

$$\frac{\Gamma \vdash e : \texttt{cmd(}\tau\texttt{)} \quad \Gamma, x : \tau \vdash m_1 \sim \tau' \quad \Gamma, y : \tau_{exn} \vdash m_2 \sim \tau'}{\Gamma \vdash \texttt{letcomp(}e\texttt{;}x\texttt{.}m_1\texttt{;}y\texttt{.}m_2\texttt{)} \sim \tau'} \tag{38.1b}$$

The dynamic semantics of these commands consists of a transition system of the form $m \mapsto m'$ defined by the following rules:

$$\frac{e \mapsto e'}{\texttt{return(}e\texttt{)} \mapsto \texttt{return(}e'\texttt{)}} \tag{38.2a}$$

$$\frac{e \mapsto e'}{\texttt{raise[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{raise[}\tau\texttt{](}e'\texttt{)}} \tag{38.2b}$$

$$\frac{e \mapsto e'}{\texttt{letcomp}(e; x.m_1; y.m_2) \mapsto \texttt{letcomp}(e'; x.m_1; y.m_2)} \tag{38.2c}$$

$$\frac{m \mapsto m'}{\texttt{letcomp}(\texttt{cmd}(m); x.m_1; y.m_2) \mapsto \texttt{letcomp}(\texttt{cmd}(m'); x.m_1; y.m_2)} \tag{38.2d}$$

$$\frac{e\ \mathsf{val}}{\texttt{letcomp}(\texttt{cmd}(\texttt{return}(e)); x.m_1; y.m_2) \mapsto [e/x]m_1} \tag{38.2e}$$

$$\frac{e\ \mathsf{val}}{\texttt{letcomp}(\texttt{cmd}(\texttt{raise}[\tau](e)); x.m_1; y.m_2) \mapsto [e/y]m_2} \tag{38.2f}$$

## 38.2 Programming With Monadic Exceptions

The chief virtue of monadic exceptions is also its chief vice. A value of type `nat` $\rightarrow$ `nat` remains a function that, when applied to a natural number, returns a natural number (or, in the case of partial functions, may diverge). If a function can raise an exception when called, then it must be given the weaker type `nat` $\rightarrow$ `nat cmd`, which specifies that, when applied, it yields an encapsulated computation that, when executed, may raise an exception. Two such functions cannot be directly composed, since their types are no longer compatible. Instead we must explicitly sequence their execution. For example, to compose $f$ and $g$ of this type, we may write

$$\lambda(x{:}\texttt{nat}.\,\texttt{do}\,\{y \leftarrow \texttt{run}\,g(x); z \leftarrow \texttt{run}\,f(y); \texttt{return}(z)\}).$$

Here we have used the `do` syntax introduced in Chapter 37, which according to our conventions above, implicitly propagates exceptions arising from the application of $f$ and $g$ to their surrounding context.

This distinction may be regarded as either a boon or a bane, depending on how important it is to indicate in the type whether a function might raise an exception when called. For programmer-defined exceptions one may wish to draw the distinction, but the situation is less clear for other forms of run-time errors. For example, if division by zero is to be regarded as a form of exception, then the type of division must be

$$\texttt{nat} \rightarrow \texttt{nat} \rightarrow \texttt{nat}\,\texttt{cmd}$$

to reflect this possibility. But then one cannot then use division in an ordinary arithmetic expression, because its result is not a number, but an encapsulated command. One response to this might be to consider division by zero, and other related faults, not as handle-able exceptions, but rather as fatal errors that abort computation. In that case there is no difference between such an error and divergence: the computation never terminates, and this condition cannot be detected during execution. Consequently, operations such as division may be regarded as partial functions, and may therefore be used freely in expressions without taking special pains to manage any errors that may arise.

## 38.3 Exercises

# Chapter 39

# Monadic State

In Chapter 35 we introduced the type of cells of a given type so as to distinguish mutable from immutable data structures. In that chapter we left open the question of how to integrate mutation into a full-scale language. There are two main methods of doing so, one that permits great flexibility in the use of mutable storage at the expense of weakening the meaning of the typing judgement considerably, and one that retains the meaning of the typing judgement, but impairs the use of storage effects considerably. As this description suggest, each approach has its benefits and drawbacks, with neither being clearly preferable to the other in all circumstances.

The simplest, and most obvious, approach, which we will call the *integral* style, is to enrich a purely functional language, such as $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ or its extensions, with the mechanisms of $\mathcal{L}\{\texttt{ref}\}$. This results in an integration of imperative and functional programming in which the programmer may, at will, use or eschew mutation at any point within a program. For example, if we start with a purely functional data structure such as a tree structure represented as a recursive type, and then we wish to instrument this structure with, say, a reference count for profiling purposes, we may simply revise the definition of the type to, say, attach a mutable cell to each node that maintains the profiling information. It is usually straightforward to extend the implementation of the tree operations to account for the additional information at the nodes.

The chief drawback of the integral approach is that the meaning of the typing judgement changes drastically. The judgement $e : \tau$ no longer means "if $e$ evaluates to a value $v$, then $v$ is a value of type $\tau$." Instead, the judgement now means that, in addition, that evaluation of $e$ can engender arbitrary *side effects* on any data structure to which $e$ has (direct or indirect)

access. (Indeed, side effects are so-called precisely because they act "on the side," without their influence being reflected in the type of the expression.) Consequently, the type $\texttt{nat} \to \texttt{nat}$ can no longer be understood as the type of partial functions on the natural numbers, but must also admit the possibility of side effects during its execution. As a case in point, in a language without mutation the type $\texttt{unit} \to \texttt{unit}$ is quite trivial, containing only the identity and the divergent function, whereas in a language with integral mutation, this type contains arbitrarily complex functions that mutate storage, with the type revealing nothing about this behavior.

The integral approach works best with a strict language, in which the order of evaluation of sub-expressions is fully determined by its form, and is not sensitive to the evolution of the computation. Laziness complicates the integral approach because it makes it much harder to predict when expressions are evaluated. In the absence of storage effects this is of no concern (at least for correctness, if not efficiency), but in the presence of storage effects, the indeterminacy of evaluation order is disastrous. It is difficult to tell exactly when, or how often, a cell will be allocated or assigned, greatly complicating reasoning about program correctness.

The *monadic* approach to storage effects is to confine operations that may affect storage to the command level of $\mathcal{L}\{\texttt{cmd}\}$. This ensures that the expression level remains pure, so that it is compatible with both an eager and a lazy interpretation. The chief benefit of the monadic style is that it makes explicit in the types any reliance on storage effects. The chief drawback of the monadic style is that it makes explicit in the types any reliance on storage effects. While it can be useful to document the use of storage effects, it can also be a hindrance to program development. For example, if we wish to instrument a piece of pure code with code for profiling, then we must restructure it to permit modifications to the store for profiling purposes, even though its functionality has not changed.

## 39.1 Storage Effects

The language $\mathcal{L}\{\texttt{cmd}\,\texttt{ref}\}$ is an extension of $\mathcal{L}\{\texttt{cmd}\}$ (described in Chapter 37) with mutable references into $\mathcal{L}\{\texttt{cmd}\}$. The syntax of $\mathcal{L}\{\texttt{cmd}\,\texttt{ref}\}$

extends that of $\mathcal{L}\{\texttt{cmd}\}$ with the following constructs:

| *Category* | *Item* | | *Abstract* | *Concrete* |
|---|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{ref}(\tau)$ | $\tau\,\texttt{ref}$ |
| Expr | $e$ | ::= | $l$ | $l$ |
| Comm | $m$ | ::= | $\texttt{ref}(e)$ | $\texttt{ref}(e)$ |
| | | \| | $\texttt{get}(e)$ | $\texttt{!}\,e$ |
| | | \| | $\texttt{set}(e_1;e_2)$ | $e_1\,\texttt{<-}\,e_2$ |

Locations are pure expressions, whereas the primitives for reference cells are forms of command.

The static semantics of $\mathcal{L}\{\texttt{cmd}\,\texttt{ref}\}$ extends that of $\mathcal{L}\{\texttt{cmd}\}$ with the following rules:

$$\frac{}{\Lambda, l : \tau\,\Gamma \vdash l : \texttt{ref}(\tau)} \tag{39.1a}$$

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \texttt{ref}(e) \sim \texttt{ref}(\tau)} \tag{39.1b}$$

$$\frac{\Lambda\,\Gamma \vdash e : \texttt{ref}(\tau)}{\Lambda\,\Gamma \vdash \texttt{get}(e) \sim \tau} \tag{39.1c}$$

$$\frac{\Lambda\,\Gamma \vdash e_1 : \texttt{ref}(\tau) \quad \Lambda\,\Gamma \vdash e_2 : \tau}{\Lambda\,\Gamma \vdash \texttt{set}(e_1;e_2) \sim \tau} \tag{39.1d}$$

Here we make explicit the location typing assumptions, $\Lambda$, as well as the variable typing assumptions, $\Gamma$.

The dynamic semantics of $\mathcal{L}\{\texttt{cmd}\,\texttt{ref}\}$ is structured into two parts:

1. A transition relation $e \mapsto e'$ for expressions.
2. A transition relation $m \mathbin{@} \mu \mapsto m' \mathbin{@} \mu'$ for commands.

Expressions are evaluated without regard to context, since they engender no effects, whereas commands are evaluated relative to a memory, on which they may have an effect.

The rules defining the dynamic semantics of the monad constructs are as follows.

$$\frac{}{\texttt{cmd}(m)\ \mathsf{val}} \tag{39.2a}$$

$$\frac{e \mapsto e'}{\texttt{return}(e) \mathbin{@} \mu \mapsto \texttt{return}(e') \mathbin{@} \mu} \tag{39.2b}$$

$$\frac{e \mapsto e'}{\texttt{letcmd}(e; x\,.\,m) \mathbin{@} \mu \mapsto \texttt{letcmd}(e'; x\,.\,m) \mathbin{@} \mu} \tag{39.2c}$$

$$\frac{m_1 \mathbin{@} \mu \mapsto m_1' \mathbin{@} \mu'}{\texttt{letcmd(cmd(}m_1\texttt{);}x\texttt{.}m_2\texttt{)} \mathbin{@} \mu \mapsto \texttt{letcmd(cmd(}m_1'\texttt{);}x\texttt{.}m_2\texttt{)} \mathbin{@} \mu'} \tag{39.2d}$$

$$\frac{e\ \mathsf{val}}{\texttt{letcmd(cmd(return(}e\texttt{));}x\texttt{.}m\texttt{)} \mathbin{@} \mu \mapsto [e/x]m \mathbin{@} \mu} \tag{39.2e}$$

The transition rules for the monadic elimination form is somewhat unusual. First, the expression $e$ is evaluated to obtain a suspended command. Once such a command has been obtained, execution continues by evaluating it in the current memory, updating that memory as appropriate during its execution. This process ends once the suspended command is a `return` statement, in which case this value is passed to the body of the `letcomp`.

The transition rules for evaluation of storage commands are as follows:

$$\frac{}{l\ \mathsf{val}} \tag{39.3a}$$

$$\frac{e \mapsto e'}{\texttt{ref(}e\texttt{)} \mathbin{@} \mu \mapsto \texttt{ref(}e'\texttt{)} \mathbin{@} \mu} \tag{39.3b}$$

$$\frac{e\ \mathsf{val}}{\texttt{ref(}e\texttt{)} \mathbin{@} \mu \mapsto \texttt{return(}l\texttt{)} \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{39.3c}$$

$$\frac{e \mapsto e'}{\texttt{get(}e\texttt{)} \mathbin{@} \mu \mapsto \texttt{get(}e'\texttt{)} \mathbin{@} \mu} \tag{39.3d}$$

$$\frac{e\ \mathsf{val}}{\texttt{get(}l\texttt{)} \mathbin{@} \mu \otimes \langle l : e \rangle \mapsto \texttt{return(}e\texttt{)} \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{39.3e}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{set(}e_1\texttt{;}e_2\texttt{)} \mathbin{@} \mu \mapsto \texttt{set(}e_1'\texttt{;}e_2\texttt{)} \mathbin{@} \mu} \tag{39.3f}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{set(}e_1\texttt{;}e_2\texttt{)} \mathbin{@} \mu \mapsto \texttt{set(}e_1\texttt{;}e_2'\texttt{)} \mathbin{@} \mu} \tag{39.3g}$$

$$\frac{e\ \mathsf{val}}{\texttt{set(}l\texttt{;}e\texttt{)} \mathbin{@} \mu \otimes \langle l : e' \rangle \mapsto \texttt{return(}e\texttt{)} \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{39.3h}$$

## 39.2 Integral versus Monadic Effects

The chief motivation for introducing monads is to make explicit in the types any reliance on computational effects. In the case of storage effects this is not always an advantage. The problem is that any use of storage forces the computation to be within the monad, and there is no way to get back out—once in the monad, always in the monad. This rules out the use of so-called *benign effects*, which may be used internally in some computation that is, for all outward purposes, entirely pure. One example of this is provided by splay trees, which may be used to implement a functional dictionary abstraction, but which rely heavily on mutation for their implementation in order to ensure efficiency. A simpler example, which we consider in detail, is provided by the use of backpatching to implement recursion as described in Chapter 35.

When formulated using monads to expose the use of storage effects, the backpatching implementation, *fact*, of the factorial function is as follows:

```
do {
  r ← return (ref (λ n:nat. comp(return (n))))
; f ← return (λ n:nat. ...)
; _ ← set (r, f)
; return f
}
```

where the elided $\lambda$-abstraction is given as follows:

```
λ(n:nat.
  ifz(n,
      comp(return(1)),
      n'.comp(
           do {
             f'← get(r)
           ; return (n*f'(n'))
           })))
```

Observe that each branch of the conditional test returns a suspended command. In the case that the argument is zero, the command simply returns the value 1. Otherwise, it fetches the contents of the associated reference cell, applies this to the predecessor, and returns the result of the appropriate calculation.

We may check that that *fact* $\sim$ `nat` $\rightarrow$ `(nat cmd)`, which exposes two aspects of this code:

1. The command that builds the recursive factorial function is impure, because it allocates and assigns to the reference cell used to implement backpatching.

2. The body of the factorial function is itself impure, because it accesses the reference cell to effect the recursive call.

The consequence is that the factorial function may no longer be used as a (pure) function! In particular, we cannot apply *fact* to an argument in an expression; it must be executed as a command. We must write

```
do {
  f ← return(fact)
; x ← let comp (x:nat) be f(n) in return x
; return x
}
```

to bind the function computed by the expression *fact* to the variable `f`; apply this to $n$, yielding the result; and return this to the caller.

The difficulty is that the use of a reference cell to implement recursion is a benign effect, one that does not affect the purity of the function expression itself, nor of its applications. But the type system for effects studied here is incapable of recognizing this fact, and for good reason. It is extremely difficult, in general, to determine whether or not the use of effects in some region of a program is benign. As a stop-gap measure, one way around this is to introduce an operation of type $\tau\,\texttt{cmd} \to \tau$, which may be used to exit the monad. But this ruins the very distinction we are trying to enforce, to segregate pure expressions from impure commands.

## 39.3 Exercises

# Chapter 40

# Comonads

Monads arise naturally for managing effects that both *influence* and are *influenced by* the context in which they arise. This is particularly clear for storage effects, whose context is a memory mapping locations to values. The semantics of the storage primitives makes reference to the memory (to retrieve the contents of a location) and makes changes to the memory (to change the contents of a location or allocate a new location). These operations must be sequentialized in order to be meaningul (that is, the precise order of execution matters), and we cannot expect to escape the context since locations are values that give rise to dependencies on the context. As we shall see in Chapter 46 other forms of effect, such as input/output or interprocess communication, are naturally expressed in the context of a monad.

By contrast the use of monads for exceptions as in Chapter 38 is rather less natural. Raising an exception does not influence the context, but rather imposes the requirement on it that a handler be present to ensure that the command is meaningful even when an exception is raised. One might argue that installing a handler influences the context, but it does so in a nested, or stack-like, manner. A new handler is installed for the duration of execution of a command, and then discarded. The handler does not persist across commands in the same sense that locations persist across commands in the case of the state monad. Moreover, installing a handler may be seen as restoring purity in that it catches any exceptions that may be raised and, assuming that the handler does not itself raise an exception, yields a pure value. A similar situation arises with fluid binding (as described in Chapter 34). A reference to a symbol imposes the demand on the context to provide a binding for it. The binding of a symbol may be changed, but only

for the duration of execution of a command, and not persistently. Moreover, the reliance on symbol bindings within a specified scope confines the impurity to that scope.

The concept of a *comonad* captures the concept of an effect that *imposes a requirement* on its context of execution, but that does not persistently alter that context beyond its execution. Computations that rely on the context to provide some capability may be thought of as impure, but the impurity is confined to the extent of the reliance—outside of this context the computation may be once again regarded as pure. One may say that monads are appropriate for *global*, or *persistent*, effects, whereas comonads are appropriate for *local*, or *ephemeral*, effects.

## 40.1 A Comonadic Framework

The central concept of the comonadic framework for effects is the *constrained typing judgement*, $e : \tau\,[\chi]$, which states that an expression $e$ has type $\tau$ (as usual) provided that the context of its evaluation satisfies the constraint $\chi$. The nature of constraints varies from one situation to another, but will include at least the trivially true constraint, $\top$, and the conjunction of constraints, $\chi_1 \wedge \chi_2$. We sometimes write $e : \tau$ to mean $e : \tau\,[\top]$, which states that expression $e$ has type $\tau$ under no constraints.

The syntax of the comonadic framework, $\mathcal{L}\{\texttt{comon}\}$, is given by the following grammar:

| *Category* | *Item* | | *Abstract* | *Concrete* |
|---|---|---|---|---|
| Type | $\tau$ | ::= | `box[`$\chi$`](`$\tau$`)` | $\Box_\chi\,\tau$ |
| Const | $\chi$ | ::= | `tt` | $\top$ |
| | | $\mid$ | `and(`$\chi_1$`;`$\chi_2$`)` | $\chi_1 \wedge \chi_2$ |
| Expr | $e$ | ::= | `box(`$e$`)` | `box(`$e$`)` |
| | | $\mid$ | `unbox(`$e$`)` | `unbox(`$e$`)` |

A type of the form $\Box_\chi\,\tau$ is called a *comonad*; it represents the type of unevaluated expressions that impose constraint $\chi$ on their context of execution. The constraint $\top$ is the trivially true constraint, and the constraint $\chi_1 \wedge \chi_2$ is the conjunction of two constraints. The expression `box(`$e$`)` is the introduction form for the comonad, and the expression `unbox(`$e$`)` is the corresponding elimination form.

The judgement $\chi$ `true` expresses that the constraint $\chi$ is satisfied. This judgement is partially defined by the following rules, which specify the

meanings of the trivially true constraint and the conjunction of constraints.

$$\frac{}{\mathtt{tt}\ \mathsf{true}} \tag{40.1a}$$

$$\frac{\chi_1\ \mathsf{true} \quad \chi_2\ \mathsf{true}}{\mathtt{and}(\chi_1;\chi_2)\ \mathsf{true}} \tag{40.1b}$$

$$\frac{\mathtt{and}(\chi_1;\chi_2)\ \mathsf{true}}{\chi_1\ \mathsf{true}} \tag{40.1c}$$

$$\frac{\mathtt{and}(\chi_1;\chi_2)\ \mathsf{true}}{\chi_2\ \mathsf{true}} \tag{40.1d}$$

We will make use of hypothetical judgements of the form $\chi_1\ \mathsf{true}, \ldots, \chi_n\ \mathsf{true} \vdash \chi\ \mathsf{true}$, where $n \geq 0$, expressing that $\chi$ is derivable from $\chi_1, \ldots, \chi_n$, as usual.

The static semantics is specified by generic hypothetical judgements of the form

$$x_1 : \tau_1\,[\chi_1], \ldots, x_n : \tau_n\,[\chi_n] \vdash e : \tau\,[\chi].$$

As usual we write $\Gamma$ for a finite set of hypotheses of the above form.

The static semantics of the core constructs of $\mathcal{L}\{\mathtt{comon}\}$ is defined by the following rules:

$$\frac{\chi' \vdash \chi}{\Gamma, x : \tau\,[\chi] \vdash x : \tau\,[\chi']} \tag{40.2a}$$

$$\frac{\Gamma \vdash e : \tau\,[\chi]}{\Gamma \vdash \mathtt{box}(e) : \Box_\chi\,\tau\,[\chi']} \tag{40.2b}$$

$$\frac{\Gamma \vdash e : \Box_\chi\,\tau\,[\chi'] \quad \chi' \vdash \chi}{\Gamma \vdash \mathtt{unbox}(e) : \tau\,[\chi']} \tag{40.2c}$$

Rule (40.2b) states that a boxed computation has comonadic type under an arbitrary constraint. This is valid because a boxed computation is a value, and hence imposes no constraint on its context of evaluation. Rule (40.2c) states that a boxed computation may be activated provided that the ambient constraint, $\chi'$, is at least as strong as the constraint $\chi$ of the boxed computation. That is, any requirement imposed by the boxed computation must be met at the point at which it is unboxed.

Rules (40.2) are formulated to ensure that the constraint on a typing judgement may be strengthened arbitrarily.

**Lemma 40.1** (Constraint Strengthening)**.** *If* $\Gamma \vdash e : \tau\,[\chi]$ *and* $\chi' \vdash \chi$*, then* $\Gamma \vdash e : \tau\,[\chi']$.

*Proof.* By rule induction on Rules (40.2). □

Intuitively, if a typing holds under a weaker constraint, then it also holds under any stronger constraint as well.

At this level of abstraction the dynamic semantics of $\mathcal{L}\{\texttt{comon}\}$ is trivial.

$$\frac{}{\texttt{box}(e)\ \texttt{val}} \tag{40.3a}$$

$$\frac{e \mapsto e'}{\texttt{unbox}(e) \mapsto \texttt{unbox}(e')} \tag{40.3b}$$

$$\frac{}{\texttt{unbox}(\texttt{box}(e)) \mapsto e} \tag{40.3c}$$

In specific applications of $\mathcal{L}\{\texttt{comon}\}$ the dynamic semantics will also specify the context of evaluation with respect to which constraints are to interpreted.

The role of the comonadic type in $\mathcal{L}\{\texttt{comon}\}$ is explained by considering how one might extend the language with, say, function types. The crucial idea is that the comonad isolates the dependence of a computation on its context of evaluation so that such constraints do not affect the other type constructors. For example, here are the rules for function types expressed in the context of $\mathcal{L}\{\texttt{comon}\}$:

$$\frac{\Gamma, x : \tau_1\,[\texttt{tt}] \vdash e_2 : \tau_2\,[\texttt{tt}]}{\Gamma \vdash \texttt{lam}[\tau_1](x.e_2) : \texttt{arr}(\tau_1;\tau_2)\,[\chi]} \tag{40.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau\,[\chi] \quad \Gamma \vdash e_2 : \tau_2\,[\chi]}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau\,[\chi]} \tag{40.4b}$$

These rules are formulated so as to ensure that constraint strengthening remains admissible. Rule (40.4a) states that a $\lambda$-abstraction has type $\tau_1 \to \tau_2$ under any constraint $\chi$ provided that its body has type $\tau_2$ under the trivially true constraint, assuming that its argument has type $\tau_1$ under the trivially true constraint. By demanding that the body be well-formed under no constraints we are, in effect, insisting that its body be boxed if it is to impose a constraint on the context at the point of application. Under a call-by-value evaluation order, the argument $x$ will always be a value, and hence imposes no constraints on its context.

Let the expression $\texttt{unbox\_app}(e_1;e_2)$ be an abbreviation for $\texttt{unbox}(\texttt{ap}(e_1;e_2))$, which applies $e_1$ to $e_2$, then activates the result. The derived static semantics

for this construct is given by the following rule:

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \Box_\chi\, \tau\,[\chi'] \quad \Gamma \vdash e_2 : \tau_2\,[\chi'] \quad \chi' \vdash \chi}{\Gamma \vdash \texttt{unbox\_app}(e_1; e_2) : \tau\,[\chi']} \tag{40.5}$$

In words, to apply a function with impure body to an argument, the ambient constraint must be strong enough to type the function and its argument, and must be at least as strong as the requirements imposed by the body of the function. We may view a type of the form $\tau_1 \to \Box_\chi\, \tau_2$ as the type of functions that, when applied to a value of type $\tau_1$, yield a value of type $\tau_2$ engendering local effects with requirements specified by $\chi$.

Similar principles govern the extension of $\mathcal{L}\{\texttt{comon}\}$ with other types such as products or sums.

## 40.2 Comonadic Effects

In this section we discuss two applications of $\mathcal{L}\{\texttt{comon}\}$ to managing local effects. The first application is to exceptions, using constraints to specify whether or not an exception handler must be installed to evaluate an expression so as to avoid an uncaught exception error. The second is to fluid binding, using constraints to specify which symbols must be bound during execution so as to avoid accessing an unbound symbol. The first may be considered to be an instance of the second, in which we think of the exception handler as a distinguished symbol whose binding is the current exception continuation.

### 40.2.1 Exceptions

To model exceptions we extend $\mathcal{L}\{\texttt{comon}\}$ as follows:

| *Category* | *Item* | | *Abstract* | *Concrete* |
|---|---|---|---|---|
| Const | $\chi$ | ::= | $\uparrow$ | $\uparrow$ |
| Expr | $e$ | ::= | $\texttt{raise[}\tau\texttt{]}(e)$ | $\texttt{raise}(e)$ |
| | | \| | $\texttt{handle}(e_1; x.e_2)$ | $\texttt{try}\, e_1\, \texttt{ow}\, x \Rightarrow e_2$ |

The constraint $\uparrow$ specifies that an expression may raise an exception, and hence that its context is required to provide a handler for it.

The static semantics of $\mathcal{L}\{\texttt{comon}\}$ is extended with the following rules:

$$\frac{\Gamma \vdash e : \tau_{exn}\,[\chi] \quad \chi \vdash \uparrow}{\Gamma \vdash \texttt{raise[}\tau\texttt{]}(e) : \tau\,[\chi]} \tag{40.6a}$$

$$\frac{\Gamma \vdash e_1 : \tau\,[\chi \wedge \uparrow] \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau\,[\chi]}{\Gamma \vdash \texttt{handle}(e_1; x.e_2) : \tau\,[\chi]} \tag{40.6b}$$

Rule (40.6a) imposes the requirement for a handler on the context of a `raise` expression, in addition to any other conditions that may be imposed by its argument. (The rule is formulated so as to ensure that constraint strengthening remains admissible.) Rule (40.6b) transforms an expression that requires a handler into one that may or may not require one, according to the demands of the handling expression. If $e_2$ does not demand a handler, then $\chi$ may be taken to be the trivial constraint, in which case the overall expression is pure, even though $e_1$ is impure (may raise an exception).

The dynamic semantics of exceptions is as given in Chapter 28. The interesting question is to explore the additional assurances given by the comonadic type system given by Rules (40.6). Intuitively, we may think of a stack as a constraint transformer that turns a constraint $\chi$ into a constraint $\chi'$ by composing frames, including handler frames. Then if $e$ is an expression of type $\tau$ imposing constraint $\chi$ and $k$ is a $\tau$-accepting stack transforming constraint $\chi$ into constraint $\top$, then evaluation of $e$ on $k$ cannot yield an uncaught exception. In this sense the constraints reflect the reality of the execution behavior of expressions.

To make this precise, we define the judgement $k : \tau\,[\chi]$ to mean that $k$ is a stack that is suitable as an execution context for an expression $e : \tau\,[\chi]$. The typing rules for stacks are as follows:

$$\frac{}{\epsilon : \tau\,[\top]} \tag{40.7a}$$

$$\frac{k : \tau'\,[\chi'] \quad f : \tau\,[\chi] \Rightarrow \tau'\,[\chi']}{k; f : \tau\,[\chi]} \tag{40.7b}$$

Rule (40.7a) states that the empty stack must not impose any constraints on its context, which is to say that there must be no uncaught exceptions at the end of execution. Rule (40.7b) simply specifies that a stack is a composition of frames. The typing rules for frames are easily derived from the static semantics of $\mathcal{L}\{\texttt{comon}\}$. For example,

$$\frac{x : \tau_{exn} \vdash e : \tau\,[\chi]}{\texttt{handle}(-; x.e) : \tau\,[\chi \wedge \uparrow] \Rightarrow \tau\,[\chi]} \tag{40.8}$$

This rule states that a handler frame transforms an expression of type $\tau$ demanding a handler into an expression of type $\tau$ that may, or may not, demand a handler, according to the form of the handling expression.

The formation of states is defined essentially as in Chapter 27.

$$\frac{k : \tau\,[\chi] \quad e : \tau\,[\chi]}{k \rhd e\ \mathsf{ok}} \tag{40.9a}$$

$$\frac{k : \tau\,[\chi] \quad e : \tau\,[\chi] \quad e\ \mathsf{val}}{k \lhd e\ \mathsf{ok}} \tag{40.9b}$$

Observe that a state of the form $\epsilon \rhd \mathtt{raise}(e)$, where $e$ val, is ill-formed, because the empty stack is well-formed only under no constraints on the context.

Safety ensures that no uncaught exceptions can arise. This is expressed by defining final states to be only those returning a value to the empty stack.

$$\frac{e\ \mathsf{val}}{\epsilon \lhd e\ \mathsf{final}} \tag{40.10}$$

In contrast to Chapter 28, we *do not* consider an uncaught exception state to be final!

**Theorem 40.2** (Safety)**.** *1. If $s$ ok and $s \mapsto s'$, then $s'$ ok.*

*2. If $s$ ok then either $s$ final or there exists $s'$ such that $s \mapsto s'$.*

*Proof.* These are proved by rule induction on the dynamic semantics and on the static semantics, respectively, proceeding along standard lines. □

### 40.2.2 Fluid Binding

Using comonads we may devise a type system for fluid binding that ensures that no unbound symbols are accessed during execution. This is achieved by regarding the mapping of symbols to their values to be the context of execution, and introducing a form of constraint stating that a specified symbol must be bound in the context.

Let us consider a comonadic static semantics for $\mathcal{L}\{\mathtt{fluid}\}$ defined in Chapter 34. For this purpose we consider atomic constraints of the form $\mathtt{bd}(a)$, stating that the symbol $a$ has a binding.

The static semantics of fluid binding consists of judgements of the form $\Sigma\,\Gamma \vdash e : \tau\,[\chi]$, where $\Sigma$ consists of hypotheses of the form $a : \tau$ assigning a type to a symbol.

$$\frac{\Sigma \vdash a : \tau \quad \chi \vdash \mathtt{bd}(a)}{\Sigma\,\Gamma \vdash \mathtt{get}[a] : \tau\,[\chi]} \tag{40.11a}$$

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e_1 : \tau\,[\chi] \quad \Sigma\,\Gamma \vdash e_2 : \tau\,[\chi \wedge \mathtt{bd}(a)]}{\Sigma\,\Gamma \vdash \mathtt{put}[a]\,(e_1; e_2) : \tau\,[\chi]} \tag{40.11b}$$

Rule (40.11a) records the demand for a binding for the symbol $a$ incurred by retrieving its value. Rule (40.11b) propagates the fact that the symbol $a$ is bound to the body of the fluid binding.

The dynamic semantics is as specified in Chapter 34. The safety theorem for the comonadic type system for fluid binding states that no unbound symbol error may ever arise during execution. We define the judgement $\theta \models \chi$ to mean that $a \in dom(\theta)$ whenever $\chi \vdash \mathtt{bd}(a)$.

**Theorem 40.3** (Safety). *1. If $e : \tau\,[\chi]$ and $e \mapsto_\theta e'$, then $e' : \theta\,[\chi]$.*

*2. If $e : \tau\,[\chi]$ and $\theta \models \chi$, then either $e$ val or there exists $e'$ such that $e \mapsto_\theta e'$.*

The comonadic static semantics for $\mathcal{L}\{\mathtt{fluid}\}$ may be extended to $\mathcal{L}\{\mathtt{fluid\,new}\}$, which also permits dynamic symbol generation. The main difficulty is to manage the interaction between the scopes of symbols and their occurrences in types. First, it is straightforward to define the judgement $\Sigma \vdash \chi$ constr to mean that $\chi$ is a constraint involving only those symbols $a$ such that $\Sigma \vdash a : \tau$ for some $\tau$. Using this we may also define the judgement $\Sigma \vdash \tau$ type analogously. This judgement is used to impose a restriction on symbol generation to ensure that symbols do not escape their scope:

$$\frac{\Sigma, a : \sigma\,\Gamma \vdash e : \tau \quad \Sigma \vdash \tau\ \mathsf{type}}{\Sigma\,\Gamma \vdash \mathtt{new}[\sigma]\,(a.e) : \langle\sigma\rangle\tau} \tag{40.12}$$

This imposes the requirement that the result type of a computation involving a dynamically generated symbol must not mention that symbol. Otherwise the type $\langle\sigma\rangle\tau$ would involve a symbol that makes no sense with respect to the ambient symbol context, $\Sigma$. In practical terms this means that the expression, $e$, must ensure that its type imposes no residual requirements involving the symbol $a$ introduced by the binder.

For example, an expression such as

$$\mathtt{gen}(\nu(a{:}\mathtt{nat}.\mathtt{put}\,a\,\mathtt{is}\,\mathtt{z}\,\mathtt{in}\,\lambda(x{:}\mathtt{nat}.\mathtt{box}(\ldots\mathtt{get}\,a\ldots))))$$

is necessarily ill-typed. The type of the $\lambda$-abstraction must be of the form $\mathtt{nat} \to \Box_\chi\,\tau$, where $\chi \vdash \mathtt{bd}(a)$, reflecting the dependence of the body of the function on the binding of $a$. This type is propagated through the fluid binding for $a$, since it holds only for the duration of evaluation of the $\lambda$-abstraction itself, which is immediately returned as its value. Since

the type of the $\lambda$-abstraction involves the symbol $a$, the second premise of Rule (40.12) is not met, and the expression is ill-typed. This is as it should be, for we cannot guarantee that the dynamically generated symbol replacing $a$ during evaluation will, in fact, be bound when the body of the function is executed.

However, if we move the binding for $a$ into the scope of the $\lambda$-abstraction,

$$\texttt{gen}(\nu(a\texttt{:nat}.\lambda(x\texttt{:nat}.\texttt{box}(\texttt{put}\,a\,\texttt{is}\,\texttt{z}\,\texttt{in}\,\ldots\texttt{get}\,a\,\ldots)))),$$

then the type of the $\lambda$-abstraction may have the form $\texttt{nat} \to \Box_\chi\,\tau$, where $\chi$ need not constrain $a$ to be bound. The reason is that the fluid binding for $a$ discharges the obligation to bind $a$ within the body of the function. Consequently, the condition on Rule (40.12) is met, and the expression is well-typed. Indeed, each evaluation of the body of the $\lambda$-abstraction initializes the fresh copy of $a$ generated during evaluation, so no unbound symbol error can arise during execution.

## 40.3 Exercises