Practical Foundations for Programming Languages

SECOND EDITION

Robert Harper
Carnegie Mellon University

Copyright © 2016 by Robert Harper.

All Rights Reserved.

Contents

Part I

Judgments and Rules

Part II

Statics and Dynamics

Part III

Total Functions

Part IV

Finite Data Types

Part V

Types and Propositions

Part VI

Infinite Data Types

Part VII

Variable Types

Part VIII

Partiality and Recursive Types

Part IX

Dynamic Types

Part X

Subtyping

Part XI

Dynamic Dispatch

Part XII

Control Flow

Part XIII

Symbolic Data

Part XIV

Mutable State

Part XV

Parallelism

Part XVI

Concurrency and Distribution

Part XVII

Modularity

Part XVIII

Equational Reasoning

Part XIX

Appendices

Appendix A

Answers to the Exercises

Chapter 1

1.1. Because $\mathcal{X} \subseteq \mathcal{Y}$, any variable in $\mathcal{A}[\mathcal{X}]$ is also a variable in $\mathcal{A}[\mathcal{Y}]$. Inductively, if $a_i \in \mathcal{A}[\mathcal{X}]$, then $a_i \in \mathcal{A}[\mathcal{Y}]$, and therefore $o(a_1, \ldots, a_n) \in \mathcal{A}[\mathcal{Y}]$.

1.2. Extending the solution to the preceding exercise, we need only account for abstractors. Suppose that $\vec{x}\,.\,a \in \mathcal{A}[\mathcal{X}]$, and we are to show that $\vec{x}\,.\,a \in \mathcal{A}[\mathcal{Y}]$. Pick any $\pi : \vec{x} \leftrightarrow \vec{x}'$ such that $\vec{x}' \notin \mathcal{Y}$. Noting that $\vec{x}' \notin \mathcal{X}$, because $\mathcal{X} \subseteq \mathcal{Y}$, we have that $\widehat{\pi}(a) \in \mathcal{A}[\mathcal{X}, \vec{x}']$, and hence inductively in $\mathcal{A}[\mathcal{Y}, \vec{x}']$ as well, which suffices for the result.

1.3. (Omitted.)

1.4. A standard solution is to represent back edges by *de Bruijn indices*: a bound variable occurrence is represented by $\texttt{bv}[\,i\,]$, where i is a positive natural number designating the ith enclosing abstractor. An abstractor for abg's has the form $.g$, where the "dot" indicates the introduction of an (unnamed) bound variable. Letting $\mathcal{G}[\mathcal{X}]_n$ stand for the abg's with n enclosing abstractors, we may say that $.g$ is in $\mathcal{G}[\mathcal{X}]_{n+1}$ if $g \in \mathcal{G}[\mathcal{X}]_n$, and that $\texttt{bv}[\,i\,] \in \mathcal{G}[\mathcal{X}]_n$ if $1 \leq i \leq n$.

Chapter 2

2.1. One possible definition of the judgment $\texttt{max}(m\,;n\,;p)$ is given by the following rules:

$$\frac{}{\texttt{max}(m\,;\texttt{zero}\,;m)} \tag{A.1a}$$

$$\frac{}{\texttt{max}(\texttt{zero}\,;\texttt{succ}(\,n\,)\,;\texttt{succ}(\,n\,))} \tag{A.1b}$$

$$\frac{\mathsf{max}(m\,;n\,;p)}{\mathsf{max}(\texttt{succ}(\,m\,)\,;\texttt{succ}(\,n\,)\,;\texttt{succ}(\,p\,))} \tag{A.1c}$$

One may show by rule induction that to every two natural number inputs there corresponds a natural number output. A second nested pair of rule inductions shows that if $\mathsf{max}(m\,;n\,;p)$ and $\mathsf{max}(m\,;n\,;q)$, then p is q. This proof establishes that the three-place relation defines a total function of its first two arguments.

2.2. Assume that t tree and n nat. As in Solution **2.1** it is easiest to prove first that $\mathsf{hgt}(t\,;n)$ relates every tree to at least one height. Then one may prove by rule induction that if $\mathsf{hgt}(t\,;m)$ and $\mathsf{hgt}(t\,;n)$, then m is n.

2.3. The judgments t tree and f forest may be simultaneously defined by the following rules:

$$\frac{f\ \mathsf{forest}}{\texttt{node}(\,f\,)\ \mathsf{tree}} \tag{A.2a}$$

$$\frac{}{\texttt{nil}\ \mathsf{forest}} \tag{A.2b}$$

$$\frac{t\ \mathsf{tree} \quad f\ \mathsf{forest}}{\texttt{cons}(\,t\,;f\,)\ \mathsf{forest}} \tag{A.2c}$$

The empty tree may be thought of as $\texttt{node}(\,\texttt{nil}\,)$, the node with no children.

2.4. The judgments $\mathsf{hgtT}(t\,;n)$, stating that the variadic tree t has height n, and $\mathsf{hgtF}(f\,;n)$, stating that the variadic forest f has height n, may be simultaneously inductively defined by the following rules:

$$\frac{\mathsf{hgtF}(f\,;n)}{\mathsf{hgtT}(\texttt{node}(\,f\,)\,;\texttt{succ}(\,n\,))} \tag{A.3a}$$

$$\frac{}{\mathsf{hgtF}(\texttt{nil}\,;\texttt{zero})} \tag{A.3b}$$

$$\frac{\mathsf{hgtT}(t\,;m) \quad \mathsf{hgtF}(f\,;n) \quad \mathsf{max}(m\,;n\,;p)}{\mathsf{hgtF}(\texttt{cons}(\,t\,;f\,)\,;p)} \tag{A.3c}$$

The required modes may be proved as outlined in the preceding exercises, albeit by a simultaneous rule induction for each case.

2.5. The judgment n bin stating that n is a natural number in binary may be defined by the following rules:

$$\frac{}{\texttt{zero}\ \mathsf{bin}} \tag{A.4a}$$

$$\frac{n\ \mathsf{bin}}{\texttt{twice}(\,n\,)\ \mathsf{bin}} \tag{A.4b}$$

$$\frac{n\ \mathsf{bin}}{\texttt{twiceplus1}(\,n\,)\ \mathsf{bin}} \tag{A.4c}$$

Clearly zero represents the number 0, and there are two forms of "successor", corresponding to doubling, and to doubling and adding one. The (unique) representation of the number $5 = 2 \times (2 \times 1) + 1$ is therefore

$$\texttt{twiceplus1}(\,\texttt{twice}(\,\texttt{twiceplus1}(\,\texttt{zero}\,)\,)\,).$$

2.6. The sum of two numbers in binary, represented in binary, is given by the judgment $\mathsf{sum}(m\,;n\,;p)$ defined in terms of an auxiliary judgment $\mathsf{succ}(m\,;n)$, as follows:

$$\frac{}{\mathsf{sum}(\texttt{zero}\,;\texttt{zero}\,;\texttt{zero})} \tag{A.5a}$$

$$\frac{\mathsf{sum}(m\,;n\,;p)}{\mathsf{sum}(\texttt{twice}(\,m\,)\,;\texttt{twice}(\,n\,)\,;\texttt{twice}(\,p\,))} \tag{A.5b}$$

$$\frac{\mathsf{sum}(m\,;n\,;p)}{\mathsf{sum}(\texttt{twice}(\,m\,)\,;\texttt{twiceplus1}(\,n\,)\,;\texttt{twiceplus1}(\,p\,))} \tag{A.5c}$$

$$\frac{\mathsf{sum}(m\,;n\,;p)}{\mathsf{sum}(\texttt{twiceplus1}(\,m\,)\,;\texttt{twice}(\,n\,)\,;\texttt{twiceplus1}(\,p\,))} \tag{A.5d}$$

$$\frac{\mathsf{sum}(m\,;n\,;p) \quad \mathsf{succ}(p\,;q)}{\mathsf{sum}(\texttt{twiceplus1}(\,m\,)\,;\texttt{twiceplus1}(\,n\,)\,;\texttt{twiceplus1}(\,q\,))} \tag{A.5e}$$

The auxiliary computation of the successor is defined as follows:

$$\frac{}{\mathsf{succ}(\texttt{zero}\,;\texttt{twiceplus1}(\,\texttt{zero}\,))} \tag{A.6a}$$

$$\frac{}{\mathsf{succ}(\texttt{twice}(\,n\,)\,;\texttt{twiceplus1}(\,n\,))} \tag{A.6b}$$

$$\frac{\mathsf{succ}(n\,;p)}{\mathsf{succ}(\texttt{twiceplus1}(\,n\,)\,;\texttt{twice}(\,p\,))} \tag{A.6c}$$

For correctness we must check that if n bin then there exists p bin such that $\mathsf{succ}(n\,;p)$, where p is the representation of the successor of n in binary, and that if m bin and n bin then there exists p bin such that $\mathsf{sum}(m\,;n\,;p)$ and p is the representation of the sum of m and n in binary. These may both be proved by induction on the foregoing rules, making use of such elementary facts as

$$(2 \times m + 1) + (2 \times n + 1) = 2 \times (m + n) + 2 = 2 \times (m + n + 1).$$

The sample solution above uses the successor to compute one more than the sum of m and n, which is obtained recursively. This suggests the alternative solution in which one defines simultaneously both the sum of m and n and one more than the sum of m and n. Each calls the other because

$$(2 \times m + 1) + (2 \times n) + 1 = 2 \times (m + n) + 2 = 2 \times (m + n + 1).$$

Chapter 3

3.1. As usual, give an inductive definition of the two-place judgment $\mathsf{len}(a\,;n)$, where a comb and n nat, and show that it relates every combinator a to a unique number n by induction on the given rules $\mathcal{C}$ defining a comb.

3.2. Pick a renaming x' of x, and extend rules $\mathcal{C}$ with the axiom x' comb. Proceed by rule induction on this extended rule set, replacing x' comb by a_1 comb at the base case, and otherwise proceeding inductively.

3.3. The required derivation is suggested by the following equivalences:

$$\begin{aligned} \mathtt{s}\,\mathtt{k}\,\mathtt{k}\,x &\equiv (\,\mathtt{k}\,x\,)\,(\,\mathtt{k}\,x\,) \\ &\equiv x \end{aligned}$$

The first is justified by the S axiom, the second by the K axiom.

3.4. The formulation of the question suggests to fix x and define a judgment $\mathsf{abs}_x\ a$ is a' and show that it defines a function:

$$\frac{}{\mathsf{abs}_x\ x\ \mathsf{is}\ \mathtt{s}\,\mathtt{k}\,\mathtt{k}} \tag{A.7a}$$

$$\frac{}{\mathsf{abs}_x\ \mathtt{k}\ \mathsf{is}\ \mathtt{k}\,\mathtt{k}} \tag{A.7b}$$

$$\frac{}{\mathsf{abs}_x\ \mathtt{s}\ \mathsf{is}\ \mathtt{k}\,\mathtt{s}} \tag{A.7c}$$

$$\frac{\mathsf{abs}_x\ a_1\ \mathsf{is}\ a_1' \quad \mathsf{abs}_x\ a_2\ \mathsf{is}\ a_2'}{\mathsf{abs}_x\ a_1\,a_2\ \mathsf{is}\ \mathtt{s}\ a_1'\,a_2'} \tag{A.7d}$$

It is easy to check that the required equivalence holds, noting that the axioms governing k and s have been chosen precisely to make the proof go through without complication.

3.5. Simply redefine bracket abstraction so that $[\,x\,]\,a \triangleq \mathtt{ap}(\,\mathtt{k}\,;a\,)$ when $x \notin a$. This formulation generalizes the original case, where $a = y \neq x$, to avoid altering any combinator in which x does not occur. Then prove that $\{a/y\}[\,x\,]\,b = [\,x\,]\,\{a/y\}b$ under the stated conditions by induction on the derivation of $x\,y \mid x$ comb y comb $\vdash b$ comb.

3.6. The following rules define the generalized form of the judgment:

$$\frac{(1 \leq i \leq n)}{x_1,\ldots,x_k \mid x_1\ \mathsf{closed},\ldots,x_k\ \mathsf{closed} \vdash x_i\ \mathsf{closed}} \tag{A.8a}$$

$$\frac{x_1,\ldots,x_k \mid x_1\ \mathsf{closed},\ldots,x_k\ \mathsf{closed} \vdash a_1\ \mathsf{closed} \quad x_1,\ldots,x_k \mid x_1\ \mathsf{closed},\ldots,x_k\ \mathsf{closed} \vdash a_2\ \mathsf{closed}}{x_1,\ldots,x_k \mid x_1\ \mathsf{closed},\ldots,x_k\ \mathsf{closed} \vdash \mathsf{ap}(\,a_1\,;a_2\,)\ \mathsf{closed}} \tag{A.8b}$$

$$\frac{x_1, \ldots, x_k, x \mid x_1 \text{ closed}, \ldots, x_k \text{ closed}, x \text{ closed} \vdash a \text{ closed}}{x_1, \ldots, x_k \mid x_1 \text{ closed}, \ldots, x_k \text{ closed} \vdash \lambda(\, x \,.\, a \,) \text{ closed}} \tag{A.8c}$$

The "trick" is that the local variables $x_1, \ldots, x_k$ of the generality judgment are disjoint from the ambient variables $\mathcal{X}$ that are also available. There being no hypotheses governing the ambient variables, $\mathcal{X}$, it is impossible to derive x closed for any $x \in \mathcal{X}$. But when descending into the scope of an abstractor, it is temporarily postulated that the bound variable x is closed so that its occurrences within the scope of the abstractor are properly regarded as closed.

This exercise drives home the principle that *variables are pronouns*, and *are not nouns*. The assumption x closed does not say of a "thing in itself" x that is closed; were variables "things" such a hypothesis would be senseless. But variables are *not* things, they *refer* to things. So a hypothesis x closed expresses a constraint on to what the pronoun x refers—that is, it constrains what can be substituted for it. It makes perfect sense to hypothesize that only closed abts may be substituted for a given variable.

Chapter 4

4.1. Many variations are possible. Here is an illustrative fragment of a solution that incorporates some of the suggestions given in Exercise **4.2**.

$$\frac{}{\Gamma\, x \uparrow \tau \vdash x \uparrow \tau} \tag{A.9a}$$

$$\frac{\Gamma \vdash e \uparrow \tau}{\Gamma \vdash e \downarrow \tau} \tag{A.9b}$$

$$\frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash \texttt{cast}[\,\tau\,](\,e\,) \uparrow \tau} \tag{A.9c}$$

$$\frac{}{\Gamma \vdash \texttt{num}[\,n\,] \downarrow \texttt{num}} \tag{A.9d}$$

$$\frac{\Gamma \vdash e_1 \downarrow \texttt{num} \quad \Gamma \vdash e_2 \downarrow \texttt{num}}{\Gamma \vdash \texttt{plus}(\,e_1\,;e_2\,) \uparrow \texttt{num}} \tag{A.9e}$$

$$\frac{}{\Gamma \vdash \texttt{str}[\,s\,] \downarrow \texttt{str}} \tag{A.9f}$$

$$\frac{\Gamma \vdash e_1 \uparrow \tau_1 \quad \Gamma, x \uparrow \tau_1 \vdash e_2 \downarrow \tau_2}{\Gamma \vdash \texttt{let}(\,e_1\,; x \,.\, e_2\,) \downarrow \tau_2} \tag{A.9g}$$

The separation of synthetic from analytic typing resolves the difficulty with the type of the defined term in a definition expression.

4.2. The main difficulty is to ensure that you do not preclude programs that ought to be allowed, or that can only be expressed very awkwardly. Within these constraints there are many possible variations on Solution **4.1**.

Chapter 5

5.1. Proceed by rule induction on Rules (5.10).

5.2. Proceed by rule induction on the first premise.

5.3. The definitions of multi-step and k-step transition are chosen so as to make this proof a routine induction as indicated. Because it is obvious that if $s \longmapsto^{k} s'$ and $s' \longmapsto^{k'} s''$, then $s \longmapsto^{k+k'} s''$, Solution **5.1** may be obtained as a corollary of this solution.

5.4. Proceed by rule induction on rules (5.10). The suggested strengthening ensures that rule (5.10f) can be proved without complication. The assumptions on e_i and e_i' are preserved when passing to the premises of rule (5.10f), and these assumptions are needed when considering reflexivity for a variable x_i. The rest of the proof is routine.

Chapter 6

6.1. The remaining cases follow along the same lines as those given in the proof of Theorem 6.2.

6.2. The remaining cases follow along the same lines as those given in the proof of Theorem 6.4.

6.3. The suggested case analysis ensures that errors are propagated properly by each construct. The proof as a whole ensures that there are no well-typed "stuck" expressions other than values and checked errors.

Chapter 7

7.1. Proceed by a simultaneous rule induction on rules (7.1).

7.2. Proceed along the same lines as those steps already given.

7.3. The second part proceeds by a rule induction on rules (5.1), appealing to the lemma in the inductive step.

7.4. The difficulty is that the progress theorem would allow an unchecked, as well as a checked, error in its statement. Moreover, Theorem 7.5 is no longer valid in the presence of a checked error, so safety is no longer a corollary of progress. The most obvious alternative is to introduce *two* forms of error checks, one for unchecked errors (solely to express safety), and one for checked errors (to allow for run-time errors arising from well-typed expressions). Such a formulation becomes rather baroque.

7.5. Besides the given rule for variables, the rule for definitions should be given as follows:

$$\frac{\Delta \vdash e_1 \Downarrow v_1 \quad \Delta, x_1 \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\Delta \vdash \texttt{let}(\, e_1\, ;\, x\, .\, e_2\,) \Downarrow v_2}$$

The remaining rules are self-evident.

The left-to-right direction of the correctness proof is proved by induction on the rules defining the environmental evaluation dynamics. The right-to-left direction must be proved by induction on the structure of e, rather than on the derivation of $\{v_1, \ldots, v_n / x_1, \ldots, x_n\}e \Downarrow v$ so that it is clear when a variable is to be evaluated.

Chapter 8

8.1. Introduce a new judgment form, $f \Downarrow x \,.\, e$, and allow judgments of this form as hypotheses of evaluation. The evaluation rule for (call-by-name) application takes the form

$$\frac{\Delta \vdash \{e'/x\}e \Downarrow e''}{\Delta, f \Downarrow x \,.\, e \vdash \texttt{apply}[\, f\,](\, e'\,) \Downarrow e''} \tag{A.10}$$

More provocatively, the atomic judgment $f \Downarrow x \,.\, e$ can be understood instead as the generic judgment

$$x \mid \texttt{apply}[\, f\,](\, x\,) \Downarrow e.$$

Admitting such a judgment as an assumption extends the framework given in Chapter 3 to admit *higher-order* judgment forms. Doing so requires some additional machinery that we do not develop further in this book.

8.2. The difficulty is how to specify the evaluation of a λ-abstraction, which may contain free variables governed by the hypotheses:

$$\frac{}{\Delta \vdash \lambda[\,\tau\,](\, x\,.\, e\,) \Downarrow \textbf{???}}$$

The value of the λ cannot be itself, as would be the case in a substitution-based evaluation dynamics; to do so would be to lose the connection between the binding of a variable and its subsequent usage, amounting to a form of dynamic scope.

One natural solution is to replace each free variable in the λ-abstraction by its binding in Δ at the time that the λ is evaluated:

$$\frac{}{x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash \lambda[\,\tau\,](\, x\,.\, e\,) \Downarrow \lambda[\,\tau\,](\, x\,.\, \{\vec{v}/\vec{x}\}e\,)}\ .$$

(We assume, without loss of generality, that x is not already governed by an assumption in Δ, so that no confusion of distinct variables may occur.) But doing so defeats the purpose of the environmental dynamics; we may use the substitutional evaluation dynamics instead.

A variation on this approach is to regard $\{\vec{v}/\vec{x}\}e$ as a form of expression, called an *explicit substitution*, or *closure*. At application we must perform a "context switch" from the ambient hypotheses to the hypotheses encoded in the closure:

$$\frac{\Delta \vdash e_1 \Downarrow \{\vec{v}/\vec{x}\}e \quad x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v}{\Delta \vdash \mathtt{ap}(\, e_1\, ; e_2\,) \Downarrow v}\ .$$

Both approaches suffer from an abuse of the framework of inductive definitions in Chapter 3. Check, for example, that the formulation using closures does not admit weakening, a basic requirement for a well-defined a hypothetical judgment.

Chapter 9

9.1. Proceed by rule induction on the statics of **T**.

9.2. Decompose the safety theorem into a preservation and progress lemma, which are proved along standard lines, appealing to Lemma 9.2 in the progress proof.

9.3. The proof breaks down more or less immediately. For example, even if $e_1(\, e_2\,) : \mathtt{nat}$, the expression e_1 is of function type, and the theorem as stated provides no inductive hypothesis for it. But the termination of the application clearly depends on the termination of the function being applied!

9.4. The proof breaks down at application, for even if $e_1 : \tau_2 \to \tau$ is terminating and $e_2 : \tau_2$ is terminating, it does not follow directly that the application $e_1(\, e_2\,)$ is terminating. For example, e_1 might evaluate to a function that, when applied, fails to terminate. The inductive hypothesis provides no information with which to rule out this possibility.

9.5. The stronger inductive hypothesis is sufficient to handle applications: if $e_1 : \tau_2 \to \tau$ is hereditarily terminating, and $e_2 : \tau_2$ is hereditarily terminating, then so is $e_1(\, e_2\,)$, by definition. But how are we to show that $\lambda\, (\, x : \tau_1\,)\, e_2$ is hereditarily terminating at type $\tau_1 \to \tau_2$? We must show that if e_1 is hereditarily terminating at type τ_1, then the application $(\, \lambda\, (\, x : \tau_1\,)\, e_2\,)(\, e_1\,)$ is hereditarily terminating at type τ_2. The restriction to closed terms prevents us from applying the inductive hypothesis to e_2, because, in general, it has a free variable x occurring within it. There is as yet no way to proceed.

9.6. Proceed by induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result is immediate by definition of hereditary termination at $\mathtt{nat}$. If $\tau = \tau_1 \to \tau_2$, let e_1 be hereditarily terminating at type τ_1, and observe that $e'(\, e_1\,) \longmapsto e(\, e_1\,)$. But the latter expression is hereditarily terminating, and so, by induction, is the former.

Returning to Solution **9.5**, it suffices to show that $\{e_1/x\}e_2$ is hereditarily terminating at type τ_2. This term is closed if $\lambda\, (\, x : \tau_1\,)\, e_2$ is closed, but we still do not have justification to conclude that the latter expression is hereditarily terminating, because the inductive hypothesis does not apply to open terms.

9.7. The final strengthening given in the exercise is now sufficient to show that every well-typed open term is open hereditarily terminating in the stated sense. The original result follows by considering a closed term of type `nat`, which is thereby shown to be hereditary terminating, and hence terminating.

Chapter 10

10.1. Let $\sigma = \langle \tau_i \rangle_{i \in I}$ be a database schema. A database on this schema may be considered to be a value of type $\texttt{nat} \times (\texttt{nat} \to \sigma)$ whose elements consist of pairs $\langle n, s \rangle$ such that the sequence s is defined on all natural numbers less than n, and is undefined otherwise. Using this representation, the *project* function sending the database $\langle n, s \rangle$ onto $I' \subseteq I$ is given by the pair $\langle n, s' \rangle$, where s' is the sequence

$$\lambda\,(\,k : \texttt{nat}\,)\,\langle i' \hookrightarrow s(\,k\,) \cdot i' \mid i' \in I' \rangle$$

that selects the columns specified by I' from each row of the given database. The standard *select* and *join* operations on databases are similarly defined.

10.2. Negative in terms of positive:

$$\begin{aligned}
\langle e_1, e_2 \rangle &\triangleq (\,\lambda\,(\,_ : \texttt{unit}\,)\,e_1\,) \otimes (\,\lambda\,(\,_ : \texttt{unit}\,)\,e_2\,) \\
e \cdot \texttt{l} &\triangleq \texttt{split}\,e\,\texttt{as}\,x_1 \otimes _\,\texttt{in}\,x_1(\,\langle\rangle\,) \\
e \cdot \texttt{r} &\triangleq \texttt{split}\,e\,\texttt{as}\,_ \otimes x_2\,\texttt{in}\,x_2(\,\langle\rangle\,)
\end{aligned}$$

Positive in terms of negative:

$$\begin{aligned}
e_1 \otimes e_2 &\triangleq \texttt{let}\,x_1\,\texttt{be}\,e_1\,\texttt{in}\,\texttt{let}\,x_2\,\texttt{be}\,e_2\,\texttt{in}\,x_1 \otimes x_2 \\
\texttt{split}\,e_0\,\texttt{as}\,x_1 \otimes x_2\,\texttt{in}\,e &\triangleq \texttt{let}\,x_1\,\texttt{be}\,e_0 \cdot \texttt{l}\,\texttt{in}\,\texttt{let}\,x_2\,\texttt{be}\,e_0 \cdot \texttt{r}\,\texttt{in}\,e
\end{aligned}$$

10.3. The introduction form would remain the same; the elimination form would be a degenerate form of decomposition:

$$\frac{\Gamma \vdash e_0 : \langle\rangle \quad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{check}(\,e_0\,;e\,) : \tau}$$

The principal argument e_0 should always be evaluated. By the canonical forms lemma, it must be $\langle\rangle$, and hence we may continue by evaluating e.

There would be little point in formulating a positive unit type apart from the desire to achieve uniformity among all finite positive products.

Chapter 11

11.1. Follow the example of the Booleans given in Section 11.3.2, which are just a finite enumeration type with two elements.

11.2. The option type cannot be simulated in the manner described. Here is a reasonable attempt that corresponds to Hoare's intended practice:

$$\begin{aligned} \texttt{null} &\triangleq \langle \texttt{false}, \texttt{null} \rangle \\ \texttt{just}(\, e \,) &\triangleq \langle \texttt{true}, e \rangle \\ \texttt{ifnull}\, e \, \{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(\, x \,) \hookrightarrow e_2\} &\triangleq \texttt{if}\, e \cdot \texttt{l}\, \texttt{then}\, \{e \cdot \texttt{r}/x\} e_2 \,\texttt{else}\, e_1 \end{aligned}$$

The solution makes use of `null` as the "null" inhabitant of type τ. But doing so conflicts with the existence of empty types, such as `void`, that do not have a value. Worse, regardless of the setting of the flag, the second component of the pair is always accessible and may be used in a computation. It is a matter of convention not to do this, but experience shows that, whether by mistake or by malice, it is often used inappropriately. By contrast the option type requires no special "null" value, and precludes the abuses just mentioned.

11.3. It would be considerably more flexible to generalize schemas from product types to, at least, *products of sums* of atomic types. Null values are naturally represented using options, and heterogeneous values are just homogeneous values of a sum type. More generally, one might wish to consider admitting, for example, *nested* databases, in which an attribute of a tuple might be a database itself.

11.4. The combinational logic problems are all straightforward programming exercises involving case analyses on bits. Try to optimize your solutions by producing the shortest program you can think of to exhibit the required behavior. The nested case analyses are called *binary decision diagrams*, or *bdd*'s for short. Finding optimal bdd's that exhibit a specified input/output behavior is a well-known problem in hardware logic design.

11.5. At the present stage of development there is no enough machinery available to define signals formally. Signals are typically self-referential in that their inputs is defined in terms of their own outputs at an earlier stage. The passage of time is fundamental to defining signals. For example, the signal whose value at time t is the negation of its value at time t is clearly ill-defined and does not exist. But one can clearly define a signal whose value at time $t > 0$ is the negation of its value at time $t - 1$. Generally, a signal definition is *causal* if its value at later times only depends on its value at earlier times. Thus, the passage of time required to compute the output of a combinational circuit is critically important for specifying well-defined signals.

$$\begin{aligned} e_{\mathrm{RS}} &\triangleq \lambda\,(\,\langle r, s\rangle : \texttt{signal} \times \texttt{signal}\,)\, \lambda\,(\,t : \texttt{nat}\,)\, e'_{\mathrm{RS}} \\ e'_{\mathrm{RS}} &\triangleq \texttt{rec}\, t\, \{\texttt{z} \hookrightarrow \langle \texttt{true}, \texttt{false} \rangle \mid \texttt{s}(\, t' \,)\, \texttt{with}\, \langle r', s' \rangle \hookrightarrow \langle e_{\mathrm{NOR}}(\, \langle r, s' \rangle\,), e_{\mathrm{NOR}}(\, \langle r', s \rangle\,) \rangle\} \end{aligned}$$

Chapter 12

12.1. Informally, to prove $\neg\neg(\phi \vee \neg\phi)$ true, assume $\neg(\phi \vee \neg\phi)$ true and derive a contradiction. To do so, we prove $\phi \vee \neg\phi$. (Why is this plausible, given that LEM cannot be expected to hold for a general ϕ?) To prove a disjunction, it suffices to prove one of its disjuncts; in this case, prove $\neg\phi$ true. To do so, assume ϕ true and derive a contradiction. You now have two assumptions, $\neg(\phi \vee \neg\phi)$ true, and ϕ true. From the latter it follows that $\phi \vee \neg\phi$ true, which contradicts the former. By discharging the second assumption derive $\neg\phi$ true. But then $\phi \vee \neg\phi$ true, again contradicting the first assumption. Discharging the first assumption, derive $\neg\neg(\phi \vee \neg\phi)$. Formally, the proof term $\lambda\,(x)\,x(\mathtt{r}\cdot\lambda\,(y)\,x(\mathtt{l}\cdot y))$ has the required type.

12.2. Informally, suppose that LEM holds universally, let ϕ be an arbitrary proposition, and assume $\neg\neg\phi$ true with the intent to derive ϕ true. By LEM instantiated with ϕ we have $\phi \vee \neg\phi$ true. In the former case we have ϕ true by assumption; in the latter we have a contradiction of the assumption ϕ true, and hence by false elimination we have ϕ true as well. Formally, the proof term $\lambda\,(y)\,\mathtt{case}\,\mathrm{LEM}_\phi\,\{\mathtt{l}\cdot y_1 \hookrightarrow y_1 \mid \mathtt{r}\cdot y_2 \hookrightarrow \mathtt{case}\,y(y_2)\,\{\,\}\}$ has the required type.

12.3. The required properties all follow more or less directly from the definition of entailment in constructive logic. First, check that A_1 true, $\ldots, A_n$ true $\vdash A$ true iff $A_1 \wedge \ldots A_n$ true $\vdash A$ true. Second, check the required properties of conjunction and truth, and of disjunction and falsehood, respectively. Third, check that $\phi \supset \psi$ is such that $\phi \wedge \phi \supset \psi \leq \psi$. Suppose that $\phi \wedge \rho \leq \psi$; we are to show that $\rho \leq \phi \supset \psi$. It is convenient to appeal to the Yoneda Lemma (Exercise **25.2**). It is enough to show that if $\gamma \leq \rho$, then $\gamma \leq \phi \supset \psi$. Given $\gamma \leq \rho$, it follows from the assumption that $\phi \wedge \gamma \leq \psi$. But then $\gamma \leq \phi \supset \psi$, by implication introduction.

Consider the equivalence $\phi \wedge (\psi_1 \vee \psi_2) \equiv (\phi \wedge \psi_1) \vee (\phi \wedge \psi_2)$. Let λ stand for the left-hand side, and ρ stand for the right-hand side. Because λ is a meet, it suffices to show $\rho \leq \phi$ and $\rho \leq \psi_1 \vee \psi_2$. The former is immediate, the latter almost immediate, and so $\rho \leq \lambda$. To show that $\lambda \leq \rho$, use the exponential property to reduce the problem to showing $\phi \leq \rho^{\psi_1 \vee \psi_2}$, which is to say that $\phi \leq \rho^{\psi_1}$ and $\phi \leq \rho^{\psi_2}$. But for these we need only show $\phi \wedge \psi_1 \leq \rho$ and $\phi \wedge \psi_2$, both of which are immediate. The "dual duality" is proved dually, and is left as an exercise.

It makes sense that the exponential is used in the preceding argument, because not all lattices are distributive.

12.4. It is elementary to check that the truth tables define a Boolean algebra with the exponential $\phi \supset \psi$ given by $\neg\phi \vee \phi$. The first de Morgan duality law may be proved for any Heyting algebra, but the second requires LEM (or one of its many equivalents).

Chapter 13

13.1. It is helpful to derive the negation and implication elimination forms from constructive logic as follows. For negation, if $p' : \neg\phi$ true and $p : \phi$ true, then $p'(p) : \bot$, where $p'(p) \triangleq$ $\texttt{ccr}(u.(\texttt{not}(p) \,\texttt{\#}\, p'))$. For implication, a similar derivation shows that if $\phi \supset \psi$ true and ϕ true, then ψ true.

(a) $\lambda(x)\,\texttt{ccr}(u.(\texttt{exfalso} \,\texttt{\#}\, x(\texttt{not}(u)))) : (\neg\neg\phi) \supset \phi$.

(b) $\lambda(x)\,\lambda(x_1)\,\texttt{ccr}(u_2.(\texttt{exfalso} \,\texttt{\#}\, x(\texttt{not}(u_2))(x_1))) : (\neg\phi_2 \supset \neg\phi_1) \supset (\phi_1 \supset \phi_2)$.

(c) $\lambda(x)\,\langle\texttt{not}(\texttt{ccp}(x_1.(\texttt{exfalso} \,\texttt{\#}\, x(\texttt{l}\cdot x_1)))),\texttt{not}(\texttt{ccp}(x_2.(\texttt{exfalso} \,\texttt{\#}\, x(\texttt{r}\cdot x_2))))\rangle$ is a proof of $\neg(\phi_1 \vee \phi_2) \supset (\neg\phi_1 \wedge \neg\phi_2)$.

Compare these proof terms to Solution **30.2**, which amount to the same thing, under the identification of the proof $\texttt{not}(k)$, where $k \div \phi$, with the continuation $\texttt{cont}(k)$, where $k \div \tau$. The ability to create a machine state directly avoids the ruses used in Solution **30.2** to throw a continuation for use elsewhere in the computation.

13.2. Section 13.5 sketches the main ideas of the proof. It is left to you to compare the "compiled" and "hand-written" proof of the doubly negated LEM; it depends on the details of your particular formulation of the translation.

Chapter 14

14.1. Closure under substitution may be shown by structural induction on τ'.

14.2. Proceed by induction on the structure of τ. Observe that, for example, if $\tau = \tau_1 \times \tau_2$, then e will be transformed into $\langle e\cdot\texttt{l}, e\cdot\texttt{r}\rangle$. But by the canonical forms property for closed values of product type, e will itself be a pair $\langle e_1, e_2\rangle$, where e_1 and e_2 are closed values of type τ_1 and τ_2, respectively. Thus $\langle e\cdot\texttt{l}, e\cdot\texttt{r}\rangle$ evaluates (under an eager dynamics) to $\langle e_1, e_2\rangle$, which is e itself. The complication with function types $\tau = \tau_1 \to \tau_2$ is that the transformation will yield $\lambda(x:\tau_1)\,e(x)$, which is a value that is not identical with e, but only interchangeable with it in the sense of Chapter 47.

14.3. Let the database schema σ be a finite product type $\langle\tau_i\rangle_{i\in I}$, where $\texttt{first}$ and $\texttt{last}$ are elements of I, and for which $\tau_{\texttt{first}}$ and $\tau_{\texttt{last}}$ are both $\texttt{str}$. Let σ' be the finite product type $\langle\tau_i'\rangle_{i\in I}$ that agrees with σ on each each attribute, except that $\tau'_{\texttt{first}}$ and $\tau'_{\texttt{last}}$ are both chosen to be the type variable t, indicating the positions of the intended transformation. We have, by construction, that $\{\texttt{str}/t\}\sigma'$ is σ, the database schema.

Let d be a database on the schema σ, which, according to Solution **10.1**, is a value of type

$$\texttt{nat} \times (\texttt{nat} \to \sigma).$$

To perform the required transformation, it suffices to use the generic extension of $t.\sigma'$ applied to the capitalization function c and the database d:

$$\texttt{map}[t.\texttt{nat} \times (\texttt{nat} \to \sigma')](x.c(x))(d).$$

Keeping in mind Exercise **14.2**, we may see at a glance that the size of the database remains fixed, that the only columns that are transformed are those specified by the occurrences of t in σ', and that the resulting database replaces the value v of each row at these columns with $c(v)$, as required.

14.4. The judgments $t\,.\,\tau$ non-neg and $t\,.\,\tau$ neg are defined simultaneously on the structure of τ. The key clauses of the definitions are as follows:

$$\frac{}{t\,.\,t\ \mathsf{non\text{-}neg}} \tag{A.11a}$$

$$\frac{t\,.\,\tau_1\ \mathsf{neg} \quad t\,.\,\tau_2\ \mathsf{non\text{-}neg}}{t\,.\,\tau_1 \rightarrow \tau_2\ \mathsf{non\text{-}neg}} \tag{A.11b}$$

$$\frac{t\,.\,\tau_1\ \mathsf{non\text{-}neg} \quad t\,.\,\tau_2\ \mathsf{neg}}{t\,.\,\tau_1 \rightarrow \tau_2\ \mathsf{neg}} \tag{A.11c}$$

Observe that the argument variable of the type operator cannot be judged to occur negatively, and the definitions for function types swaps polarities in the domain, and preserves them in the range. The remaining cases are defined similarly, preserving the polarity in all positions.

It is easy to give a derivation of $t\,.\,(\,t \rightarrow \mathtt{bool}\,) \rightarrow \mathtt{bool}$ non-neg according to the above rules.

14.5. The dynamics of the two forms of generic extension are given by the following key rules:

$$\frac{}{\mathtt{map}^{--}[\,t\,.\,t\,](\,x\,.\,e'\,)(\,e\,) \longmapsto \{e/x\}e'} \tag{A.12a}$$

$$\frac{}{\mathtt{map}^{--}[\,t\,.\,\tau_1 \rightarrow \tau_2\,](\,x\,.\,e'\,)(\,e\,) \longmapsto \lambda\,(\,x_1 : \{\rho'/t\}\tau_1\,)\,\mathtt{map}^{--}[\,t\,.\,\tau_2\,](\,x\,.\,e'\,)(\,e(\,\mathtt{map}^{-}[\,t\,.\,\tau_1\,](\,x\,.\,e'\,)(\,x_1\,)\,)\,)} \tag{A.12b}$$

$$\frac{}{\mathtt{map}^{-}[\,t\,.\,\tau_1 \rightarrow \tau_2\,](\,x\,.\,e'\,)(\,e\,) \longmapsto \lambda\,(\,x_1 : \{\rho/t\}\tau_1\,)\,\mathtt{map}^{-}[\,t\,.\,\tau_2\,](\,x\,.\,e'\,)(\,e(\,\mathtt{map}^{--}[\,t\,.\,\tau_1\,](\,x\,.\,e'\,)(\,x_1\,)\,)\,)} \tag{A.12c}$$

The non-negative generic extension of the non-negative operator $t\,.\,(\,t \rightarrow \mathtt{bool}\,) \rightarrow \mathtt{bool}$ on $x\,.\,e'$ sends a function f of type $(\,\rho \rightarrow \mathtt{bool}\,) \rightarrow \mathtt{bool}$ to the function

$$\lambda\,(\,g : \rho' \rightarrow \mathtt{bool}\,)\,f(\,\lambda\,(\,x : \rho\,)\,g(\,e'\,)\,)$$

of type $(\,\rho' \rightarrow \mathtt{bool}\,) \rightarrow \mathtt{bool}$.

Chapter 15

15.1. Define i using inductive recursion on the natural numbers in terms of the auxiliary expressions $\tilde{\texttt{z}} : \texttt{conat}$ and $\tilde{\texttt{s}} : \texttt{conat} \rightarrow \texttt{conat}$ as follows:

$$\lambda\,(\,x:\texttt{nat}\,)\,\texttt{rec}\,x\,\{\texttt{z} \hookrightarrow \tilde{\texttt{z}} \mid \texttt{s}(\,x\,)\,\texttt{with}\,y \hookrightarrow \tilde{\texttt{s}}(\,y\,)\}.$$

The expression $\tilde{\texttt{z}}$ of type conat is the "coinductive zero", $\texttt{gen}[\,.\,](\,x\,.\,(\,\texttt{l}\cdot x\,)\,;\langle\rangle\,)$, and $\tilde{\texttt{s}}$ is the "coinductive successor",

$$\lambda\,(\,y:\texttt{conat}\,)\,\texttt{gen}[\,.\,](\,x\,.\,(\,\texttt{r}\cdot x\,)\,;y\,).$$

A few simple calculations show that the required properties hold. The function i may also be defined by coinductive generation as follows:

$$\lambda\,(\,n:\texttt{nat}\,)\,\texttt{gen}[\,.\,](\,x\,.\,\texttt{ifz}\,x\,\{\texttt{z} \hookrightarrow \texttt{l}\cdot\langle\rangle \mid \texttt{s}(\,x'\,) \hookrightarrow \texttt{r}\cdot x'\}\,;n\,).$$

Again, a few simple calculations shows that it exhibits the required behavior.

15.2. Define $\texttt{iter}\,e\,\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(\,x\,) \hookrightarrow e_1\}$ to be the expression

$$\texttt{rec}_{\texttt{nat}}(\,e\,.\,y\,;\texttt{case}\,y\,\{\texttt{l}\cdot\_ \hookrightarrow e_0 \mid \texttt{r}\cdot x \hookrightarrow e_1\}\,).$$

Then check that the dynamics of the iterator given in Chapter 9 is derivable from this definition.

15.3. Define $\texttt{gen}_{\texttt{stream}}\,x\,\texttt{is}\,e\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow e_1,\texttt{tl} \hookrightarrow e_2\rangle$ to be the expression

$$\texttt{gen}_{\texttt{stream}}(\,e\,.\,x\,;\langle e_1,e_2\rangle\,).$$

Then check that the dynamics, as given in Section 15.1 is derivable from this definition.

15.4. The required transformation is given by the function

$$\lambda\,(\,q:\texttt{seq}\,)\,\texttt{gen}_{\texttt{stream}}\,x\,\texttt{is}\,\texttt{z}\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow q(\,x\,),\texttt{tl} \hookrightarrow \texttt{s}(\,x\,)\rangle.$$

The nth tail of the stream associated to q is the stream

$$\texttt{gen}_{\texttt{stream}}\,x\,\texttt{is}\,\overline{n}\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow q(\,x\,),\texttt{tl} \hookrightarrow \overline{n+1}\rangle.$$

It's head is therefore $q(\,\overline{n}\,)$, as required. The two transformation are, informally, mutually inverse, showing that stream and seq are isomorphic types.

15.5. Define the lists as follows:

$$\texttt{natlist} \triangleq \mu(\,t\,.\,\texttt{unit} + (\,\texttt{nat} \times t\,)\,)$$
$$\texttt{nil} \triangleq \texttt{fold}(\,\texttt{l}\cdot\langle\rangle\,)$$
$$\texttt{cons}(\,e_1\,;e_2\,) \triangleq \texttt{fold}(\,\texttt{r}\cdot\langle e_1,e_2\rangle\,)$$
$$\texttt{rec}_{\texttt{list}}\,e\,\{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(\,x\,;y\,) \hookrightarrow e_1\} \triangleq \texttt{rec}[\,.\,](\,z\,.\,\texttt{case}\,z\,\{\texttt{l}\cdot\_ \hookrightarrow e_0 \mid \texttt{r}\cdot u \hookrightarrow \{u\cdot\texttt{l},u\cdot\texttt{r}/x,y\}e_1\}\,;e\,)$$

Then check that the requisite statics and dynamics are derivable under these definitions.

15.6. The key rule of the dynamics is the inversion principle given by the rule

$$\frac{}{\texttt{view}(\,\texttt{gen}_{\texttt{itree}}\,x\,\texttt{is}\,e\,\texttt{in}\,e'\,) \longmapsto \texttt{map}[\,t\,.\,(\,t\times t\,)\,\texttt{opt}\,](\,y\,.\,\texttt{gen}_{\texttt{itree}}\,x\,\texttt{is}\,z\,\texttt{in}\,e'\,)(\,\{e/x\}e'\,)} \tag{A.13}$$

The generic extension operation makes it convenient to apply the recursive calls, as necessary, to the result of the state transformation.

The type `itree` may be coinductively defined to be the type $\nu(\,t\,.\,(\,t\times t\,)\,\texttt{opt}\,)$. The derivation of the introduction and elimination forms from this definition follows directly from this characterization.

15.7. Define `signal` to be the coinductive type $\nu(\,t\,.\,(\,\texttt{bool}\times\texttt{bool}\,)\times t\,)$, the type of infinite streams of pairs booleans. The definition of an RS latch as transducer of such streams as follows:

$$\begin{aligned} e_{\mathrm{RS}} &\triangleq \lambda\,(\,\langle r,s\rangle:\texttt{signal}\,)\,\texttt{gen}[\,.\,](\,\langle r',s'\rangle\,.\,e'_{\mathrm{RS}}\,;\langle\texttt{true},\texttt{false}\rangle\,)\\ e'_{\mathrm{RS}} &\triangleq \langle e_{\mathrm{NOR}}(\,\langle r,s'\rangle\,), e_{\mathrm{NOR}}(\,\langle r',s\rangle\,)\rangle \end{aligned}$$

In this formulation the passage of time is strictly a mater of the propagation of the signals through the gates involved.

Chapter 16

16.1. The requested definitions and types are

$$\begin{aligned} &\texttt{s} \triangleq \Lambda(\,s\,)\,\Lambda(\,t\,)\,\Lambda(\,u\,)\,\lambda\,(\,x:s\to t\to u\,)\,\lambda\,(\,y:s\to t\,)\,\lambda\,(\,z:s\,)\,(\,x(\,z\,)\,)(\,y(\,z\,)\,)\\ &\quad:\forall(\,s\,.\,\forall(\,t\,.\,\forall(\,u\,.\,(\,s\to t\to u\,)\to(\,s\to t\,)\to s\to u\,)\,)\,)\\ &\texttt{k} \triangleq \Lambda(\,s\,)\,\Lambda(\,t\,)\,\lambda\,(\,x:s\,)\,\lambda\,(\,y:t\,)\,x\\ &\quad:\forall(\,s\,.\,\forall(\,t\,.\;s\to t\to s\,)\,). \end{aligned}$$

16.2. Define `bool` to be the type $\forall(\,t\,.\,t\to t\to t\,)$. Then define the introduction and elimination forms as follows:

$$\begin{aligned} \texttt{true} &\triangleq \Lambda(\,t\,)\,\lambda\,(\,x:t\,)\,\lambda\,(\,y:t\,)\,x\\ \texttt{false} &\triangleq \Lambda(\,t\,)\,\lambda\,(\,x:t\,)\,\lambda\,(\,y:t\,)\,y\\ \texttt{if}\,e\,\texttt{then}\,e_0\,\texttt{else}\,e_1 &\triangleq e[\,\rho\,](\,e_0\,)(\,e_1\,), \end{aligned}$$

where ρ is the result type of the conditional. Check that the statics and dynamics of these operations are derivable according to these definitions.

16.3. The type `natlist` may be defined in **F** as follows:

$$\forall(\,t\,.\,t\to(\,\texttt{nat}\to t\to t\,)\to t\,).$$

The introduction and elimination forms may then be defined as follows:

$$\texttt{nil} \triangleq \Lambda(t)\,\lambda\,(n:t)\,\lambda\,(c:\texttt{nat}\to t\to t)\,n$$
$$\texttt{cons}(e_0;e_1) \triangleq \Lambda(t)\,\lambda\,(n:t)\,\lambda\,(c:\texttt{nat}\to t\to t)\,c(e_0)(e_1[t](n)(c))$$
$$\texttt{rec}_{\texttt{list}}\,e\,\{\texttt{nil}\hookrightarrow e_0 \mid \texttt{cons}(x;y)\hookrightarrow e_1\} \triangleq e[\rho](e_0)((\lambda\,(x:\rho)\,\lambda\,(y:\texttt{nat}\to\rho\to\rho)\,e_1)).$$

Check that the statics and dynamics are derivable according to these definitions.

16.4. The inductive type $\mu(t.\tau)$, where $t.\tau$ pos, may be defined in **F** by the equation

$$\mu(t.\tau) \triangleq \forall(t.(\tau\to t)\to t).$$

The introduction and elimination forms are defined as follows:

$$\texttt{fold}(e) \triangleq \Lambda(t)\,\lambda\,(f:\tau\to t)\,f(\texttt{map}[t.\tau](y.y[t](f))(e))$$
$$\texttt{rec}[\,.\,](x.e';e) \triangleq e[\rho](\,\lambda\,(x:\tau)\,e'),$$

wherein ρ is the result type of the recursor. One may check that the statics and dynamics are derivable from these definitions. It is very instructive to check that this definition essentially coincides with the definition of the natural numbers given in Chapter 16 under the identification of nat with $\mu(t.\texttt{unit}+t)$, and the definition of sum types within **F**.

16.5. Fix τ and $l_0 : \tau\,\texttt{list}$. Define $\mathcal{P}_{l_0}$ to hold of $z : \tau$ iff z is among the elements of l_0. By parametricity the function f must preserve $\mathcal{P}$, which means that if its input has elements among those of l_0, then so must its output. But l_0 has just such elements, so $f[\tau](l_0)$ must have elements among those of l_0 as well. Thus, among other things, f could be the constantly nil function, or the list reversal function, or the function that drops every other element of its input. But it cannot, for example, transform the elements of its input in any way.

Chapter 17

17.1. Define the type stream as the following existential type:

$$\texttt{stream} \triangleq \exists(t.(t\to(\texttt{nat}\times t))\times t).$$

The introduction and elimination forms for streams are defined as follows:

$$\texttt{gen}_{\texttt{stream}}\,x\,\texttt{is}\,e\,\texttt{in}\,\langle\texttt{hd}\hookrightarrow e_0,\texttt{tl}\hookrightarrow e_1\rangle \triangleq$$
$$\texttt{pack}\,\tau\,\texttt{with}\,\langle\lambda\,(x:\tau)\,\langle e_0,e_1\rangle,e\rangle\,\texttt{as}\,\texttt{stream}$$
$$\texttt{hd}(e) \triangleq \texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle f:t\to(\texttt{nat}\times t),x:t\rangle\,\texttt{in}\,f(x)\cdot\texttt{l}$$
$$\texttt{tl}(e) \triangleq \texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle f:t\to(\texttt{nat}\times t),x:t\rangle\,\texttt{in}\ldots,\ \text{where}$$
$$\ldots \triangleq \texttt{gen}_{\texttt{stream}}\,x\,\texttt{is}\,f(x)\cdot\texttt{r}\,\texttt{in}\,\langle\texttt{hd}\hookrightarrow f(x)\cdot\texttt{l},\texttt{tl}\hookrightarrow f(x)\cdot\texttt{r}\rangle.$$

17.2. The coinductive type $\nu(\,t\,.\,\tau\,)$, where $t\,.\,\tau$ pos, may be defined in **FE** by the type

$$\exists(\,t.(\,t \to \tau\,) \times t\,).$$

The associated introduction and elimination forms may be defined as follows:

$$\begin{aligned}\texttt{gen}[\,.\,](\,x\,.\,e'\,;e\,) &\triangleq \texttt{pack}\,\sigma\,\texttt{with}\,\langle\lambda\,(\,x:t\,)\,e',e\rangle\,\texttt{as}\,\nu(\,t\,.\,\tau\,)\\ \texttt{unfold}(\,e\,) &\triangleq \texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle g:t\to\tau, x:t\rangle\,\texttt{in}\ldots,\text{ where}\\ \ldots &\triangleq \texttt{map}[\,t\,.\,\tau\,](\,y\,.\,\texttt{gen}[\,.\,](\,z\,.\,n(\,z\,)\,;y\,)\,)(\,n(\,x\,)\,).\end{aligned}$$

One may check that Solution **17.1** is a special case of this representation under the identification of `stream` with the coinductive type $\nu(\,t\,.\,\texttt{nat} \times t\,)$. It is fascinating, if a bit unnerving, to expand the definition of the existential type, and its associated operations, to obtain a representation of coinductive types in **F**.

17.3. Recalling the definition of $\exists(\,t.\tau\,)$ in **FE** as the type $\forall(\,u\,.\,\forall(\,t\,.\,\tau \to u\,) \to u\,)$ in **F**, the abstract type of queues becomes the polymorphic type

$$\forall(\,u\,.\,\forall(\,t\,.\,\tau_{queue} \to u\,) \to u\,),$$

where τ_{queue} is the type

$$\langle \texttt{emp} \hookrightarrow t\,, \texttt{ins} \hookrightarrow \texttt{nat} \times t \to t\,, \texttt{rem} \hookrightarrow t \to (\,\texttt{nat} \times t\,)\,\texttt{opt}\rangle.$$

For any choice ρ of result type, the client of the abstraction is therefore of polymorphic type $\forall(\,t\,.\,\tau_{queue} \to \rho\,)$, where ρ is fixed. Now spell out the relational interpretation of this quantified type with the binary relation R given in Section 17.4, and check that it coincides with the conditions given there. Assigning the identity relation to ρ yields the desired result that the two implementations of queues are observably indistinguishable in **FE**.

Chapter 18

18.1. The code carries over largely intact, but for the need for type abstraction around each operation. Definitional equality is required to simplify the instances of the representation constructors, as described in the chapter.

18.2. The equational dynamics of **F** carries over to $\mathbf{F}_\omega$ without change. One may or may not include definitional equality of constructor arguments; no other rules depend on these being in canonical form. Similarly, with a transition dynamics there is no need to simplify c in the instantiation $e[\,c\,]$, because to do so would not influence the evaluation of expressions of observable type.

Chapter 19

19.1. We first define $e_{\texttt{eo}}$ of type

$$\tau_{\texttt{eo}} \triangleq \langle \texttt{even} \hookrightarrow (\, \texttt{nat} \rightharpoonup \texttt{nat} \,)\, , \texttt{odd} \hookrightarrow (\, \texttt{nat} \rightharpoonup \texttt{nat} \,) \rangle$$

from which we obtain the desired functions by projection, $e_{\texttt{eo}} \cdot \texttt{even}$ and $e_{\texttt{eo}} \cdot \texttt{odd}$, respectively.

The expression $e_{\texttt{eo}}$ is defined by general recursion to be

$$\texttt{fix}\,\texttt{this} : \tau_{\texttt{eo}}\ \texttt{is}\ \langle \texttt{even} \hookrightarrow e_{\texttt{ev}}, \texttt{odd} \hookrightarrow e_{\texttt{od}} \rangle,$$

where $e_{\texttt{ev}}$ is the expression

$$\lambda\,(\,\texttt{x} : \texttt{nat}\,)\,\texttt{ifz}\,\texttt{x}\,\{\texttt{z} \hookrightarrow \texttt{s}(\,\texttt{z}\,) \mid \texttt{s}(\,\texttt{y}\,) \hookrightarrow \texttt{this} \cdot \texttt{odd}(\,\texttt{y}\,)\},$$

and $e_{\texttt{od}}$ is the expression

$$\lambda\,(\,\texttt{x} : \texttt{nat}\,)\,\texttt{ifz}\,\texttt{x}\,\{\texttt{z} \hookrightarrow \texttt{z} \mid \texttt{s}(\,\texttt{y}\,) \hookrightarrow \texttt{this} \cdot \texttt{even}(\,\texttt{y}\,)\}.$$

19.2. Using the general fixed point operator, define a search function that repeatedly tests $\phi(m,n)$ on successive values of m, starting with zero, until either $\phi(m,n)$ evaluates to zero. The resulting computation diverges if no such m exists.

19.3. Suppose that e_{halts} were a definition of ϕ_{halts} in **PCF**. Define e_{diag} to be the function

$$\lambda\,(\,n : \texttt{nat}\,)\,\texttt{ifz}\,e_{halts}(\,n\,)\,\{\texttt{z} \hookrightarrow e_{diverge} \mid \texttt{s}(\,\_\,) \hookrightarrow \texttt{z}\},$$

where $e_{diverge}$ diverges always. Let d be $\ulcorner e_{diag} \urcorner$, the Gödel-number of e_{diag}. By the assumption either $e_{halts}(\,d\,)$ evaluates to zero or to one. If the former, then by the definition of e_{diag} we have that $e_{diag}(\,d\,)$ converges, which means that $e_{halts}(\,d\,)$ evaluates to one, which means that $e_{diag}(\,d\,)$ diverges, a contradiction. If the latter, then $e_{diag}(\,d\,)$ diverges, which by the definition of e_{diag} means that $e_{halts}(\,d\,)$ evaluates to zero, which means that $e_{diag}(\,d\,)$ converges, also a contradiction. Therefore ϕ_{halts} is not definable in **PCF**.

19.4. The difficulty is that ϕ might be undefined for certain values of m prior to the first one for which $\phi(m,n)$ is zero. The search process described in Solution **19.2** would diverge prior to finding the first zero of ϕ, violating the specification. One can show that this form of minimization, if definable, could be used to solve the halting problem, in contradiction to Solution **19.3**. To see this consider the function $\phi_{step}(m)(\ulcorner e \urcorner)$, which converges iff $e(\overline{\ulcorner e \urcorner})$ converges in fewer than m steps, and diverges otherwise.

19.5. The "parallel or" function is not definable in **PCF**, yet is, intuitively, computable. The problem is that the dynamics of **PCF** is *sequential* in the sense that it evaluates the arguments of a two-argument function in a definite order, first committing to one, then to the other.

Sequentiality precludes defining the function described in the exercise. One solution is to enrich **PCF** with a form of parallelism, called *dove-tailing*, that interleaves the evaluation of two expressions, returning the result of the first to converge (or, say, the leftmost one if both converge simultaneously).

19.6. First, define a notion of Gödel-numbering that respects α-equivalence, for example by using de Bruijn indices as described in Solution **1.4**. Second, provide operations that allow one to build (the Gödel numbers of) expressions from (the Gödel numbers of) their components, and to decompose (the Gödel numbers of) expressions into (the Gödel numbers of) their components. Third, define the universal function in terms of these primitives, using general recursion to define the interpretation of general recursion, and functions to define the interpretation of functions, and so forth. Such a function is called a *metacircular interpreter* because it defines the interpretation of the constructs of a language in terms of those constructs themselves!

Chapter 20

20.1. The following definitions suffice:

$$\begin{aligned} \mathtt{k} &\triangleq \mathtt{fold}(\,\lambda\,(\,x:\mathtt{D}\,)\,\mathtt{fold}(\,\lambda\,(\,y:\mathtt{D}\,)\,x\,)\,) \\ \mathtt{s} &\triangleq \mathtt{fold}(\,\lambda\,(\,x:\mathtt{D}\,)\,\mathtt{fold}(\,\lambda\,(\,y:\mathtt{D}\,)\,\mathtt{fold}(\,\lambda\,(\,z:\mathtt{D}\,)\,(\,x\cdot z\,)\cdot(\,y\cdot z\,)\,)\,)\,) \\ x\cdot y &\triangleq (\,\mathtt{unfold}(\,x\,)\,)(\,y\,). \end{aligned}$$

Surprisingly, this structure is sufficient to represent every partial computable function on the natural numbers as an element of D!

20.2. Let ρ be the result type of the recursor and the state type of the generator in the chart below:

$$\begin{aligned} \mathtt{fold}[\,t\,.\,\tau'\,](\,e\,) &\triangleq \mathtt{fold}(\,e\,) \\ \mathtt{rec}[\,t\,.\,\tau'\,](\,x\,.\,e'\,;e\,) &\triangleq (\,\mathtt{fix}\,r\,\mathtt{is}\,\lambda\,(\,u:\tau\,)\,e_{rec}\,)(\,e\,),\text{ where} \\ e_{rec} &\triangleq \{\mathtt{map}[\,t\,.\,\tau'\,](\,x\,.\,r(\,x\,)\,)(\,\mathtt{unfold}(\,u\,)\,)/x\}e' \\ \mathtt{unfold}[\,t\,.\,\tau'\,](\,e\,) &\triangleq \mathtt{unfold}(\,e\,) \\ \mathtt{gen}[\,t\,.\,\tau'\,](\,x\,.\,e'\,;e\,) &\triangleq (\,\mathtt{fix}\,g\,\mathtt{is}\,\lambda\,(\,u:\rho\,)\,e_{gen}\,)(\,e\,),\text{ where} \\ e_{gen} &\triangleq \mathtt{fold}(\,\mathtt{map}[\,t\,.\,\tau'\,](\,x\,.\,g(\,x\,)\,)(\,\{u/x\}e'\,)\,) \end{aligned}$$

The dual symmetry of the definitions is striking. Check that the statics of the recursor and generator are derivable under these definitions.

However, the dynamics of these operations is ill-behaved. Under an eager interpretation the generator may not converge, depending on the choice of e', and under a lazy interpretation the recursor may not converge, again depending on the choice of e'. These outcomes are a reflection of the fact that in the eager case the recursive type is inductive, not coinductive, whereas in the lazy case the recursive type is coinductive, not inductive.

20.3. Under a lazy dynamics we may define the type `signal` of signals to be the recursive type $\nu(\, t\, .\, \texttt{bool} \times t\,)$, whereas under an eager dynamics one may use instead $\nu(\, t\, .\, \texttt{unit} \rightharpoonup \texttt{bool} \times t\,)$. For simplicity, assume a lazy dynamics. A NOR gate may be defined as the function of type $(\, \texttt{signal} \times \texttt{signal}\,) \rightharpoonup \texttt{signal}$ given by

$$\lambda\, (\, ab\,)\, \texttt{gen}[\, t\, .\, \texttt{bool} \times t\,](\, \langle a, b\rangle\, .\, \langle e_{nor}(\, \langle \texttt{hd}(\, a\,), \texttt{hd}(\, b\,)\rangle\,), \langle \texttt{tl}(\, a\,), \texttt{tl}(\, b\,)\rangle\rangle\, ;ab\,),$$

wherein we have used the definition of the generator from Exercise **20.2**. The internal state of the gate consists of the two input signals a and b. Whenever an output is required, the heads of a and b are nor'd together, and the new state consists of the tails of a and b, one bit from each having been consumed.

Using the NOR gate just defined, one may then use general recursion to define an RS latch by "cross-feeding" the outputs of each of the NOR gates back into one of the inputs of the other. The remaining inputs are then the r and s signals that reset and set the latch, respectively.

20.4. Define the internal state type ρ of the stream to be the type

$$\langle \texttt{X} \hookrightarrow \texttt{bool}\, , \texttt{Q} \hookrightarrow \texttt{bool}\rangle.$$

Define an RS latch with internal state of this type as follows:

$$e_{rsl} \triangleq \texttt{gen}[\, t\, .\, \tau'_{rsl}\,](\, \langle \texttt{X} \hookrightarrow x, \texttt{Q} \hookrightarrow q\rangle\, .\, e'_{rsl}\, ; \langle \texttt{X} \hookrightarrow \texttt{false}, \texttt{Q} \hookrightarrow \texttt{false}\rangle\,),$$

where

$$e'_{rsl} \triangleq \langle \texttt{X} \hookrightarrow x, \texttt{Q} \hookrightarrow q, \texttt{N} \hookrightarrow \langle \texttt{X} \hookrightarrow e_{nor}(\, \langle s, q\rangle\,), \texttt{Q} \hookrightarrow e_{nor}(\, \langle x, r\rangle\,)\rangle\rangle.$$

The state is arbitrarily initialized with both `X` and `Q` being `false`, and these are the initial outputs of the latch. Then, the next state of the latch is computed using e_{nor} applied to the fixed r and s values and to the current x and q values.

When the generator is expanded according to Solution **20.2**, and the result simplified for clarity, we obtain the following formulation of e_{rsl}:

$$\texttt{fix}\, g\, \texttt{is}\, \lambda\, (\, \langle \texttt{X} \hookrightarrow x, \texttt{Q} \hookrightarrow q\rangle\,)\, \texttt{fold}(\, e''_{rsl}\,),$$

where e''_{rsl} is given by

$$\langle \texttt{X} \hookrightarrow x, \texttt{Q} \hookrightarrow q, \texttt{N} \hookrightarrow g(\, \langle \texttt{X} \hookrightarrow e_{nor}(\, \langle s, q\rangle\,), \texttt{Q} \hookrightarrow e_{nor}(\, \langle x, r\rangle\,)\rangle\,)\rangle.$$

Notice that the state is maintained by the recursive self-reference, much as in Section 20.4.

Chapter 21

21.1. The encoding of finite products is given by the following equations:

$$\begin{aligned}\langle\rangle &\triangleq \lambda\,(\,x\,)\,x\\ \langle u_1,u_2\rangle &\triangleq \lambda\,(\,u\,)\,u(\,u_1\,)(\,u_2\,)\\ u\cdot\mathtt{l} &\triangleq u(\,(\,\lambda\,(\,x\,)\,\lambda\,(\,y\,)\,x\,)\,)\\ u\cdot\mathtt{r} &\triangleq u(\,(\,\lambda\,(\,x\,)\,\lambda\,(\,y\,)\,y\,)\,)\end{aligned}$$

21.2. Using only primitive recursion as provided by the Church numerals:

$$\lambda\,(\,x\,)\,x(\,\overline{1}\,)(\,\lambda\,(\,y\,)\,(\,\mathtt{times}(\,x\,)(\,y\,)\,)\,),$$

where `times` is the Church encoding of multiplication. Using the fixed point combinator we may define factorial as follows:

$$\mathtt{Y}(\,\lambda\,(\,f\,)\,\lambda\,(\,x\,)\,x(\,\overline{1}\,)(\,f(\,\mathtt{pred}(\,x\,)\,)\,)\,).$$

The required equations may be proved by induction on n.

21.3. As the specification implies,

$$\begin{aligned}\mathtt{true} &\triangleq \lambda\,(\,x\,)\,\lambda\,(\,y\,)\,x\\ \mathtt{false} &\triangleq \lambda\,(\,x\,)\,\lambda\,(\,y\,)\,y\\ \mathtt{if}\,u\,\mathtt{then}\,u_1\,\mathtt{else}\,u_2 &\triangleq u(\,u_1\,)(\,u_2\,)\end{aligned}$$

21.4. Define sums along similar lines as follows:

$$\begin{aligned}\mathtt{l}\cdot u &\triangleq \lambda\,(\,l\,)\,\lambda\,(\,r\,)\,l(\,u\,)\\ \mathtt{r}\cdot u &\triangleq \lambda\,(\,l\,)\,\lambda\,(\,r\,)\,r(\,u\,)\\ \mathtt{case}\,u\,\{\mathtt{l}\cdot x_1 \hookrightarrow u_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow u_2\} &\triangleq u(\,\lambda\,(\,x_1\,)\,u_1\,)(\,\lambda\,(\,x_2\,)\,u_2\,)\end{aligned}$$

The booleans are a special case in which u_1 and u_2 are always $\lambda\,(\,x\,)\,x$.

21.5. Define lists by their elimination form as follows:

$$\begin{aligned}\mathtt{nil} &\triangleq \lambda\,(\,n\,)\,\lambda\,(\,c\,)\,n\\ \mathtt{cons}(\,u_1\,;u_2\,) &\triangleq \lambda\,(\,n\,)\,\lambda\,(\,c\,)\,c(\,u_1\,)(\,u_2\,)\\ \mathtt{rec}_{\mathtt{list}}\,u\,\{\mathtt{nil} \hookrightarrow u_0 \mid \mathtt{cons}(\,x_1\,;x_2\,) \hookrightarrow u_1\} &\triangleq u(\,u_0\,)(\,\lambda\,(\,x_1\,)\,\lambda\,(\,x_2\,)\,u_1\,).\end{aligned}$$

21.6. Dually to lists, define streams by their introduction form:

$$\begin{aligned}\mathtt{gen}_{\mathtt{stream}}\,u\,\mathtt{is}\,x\,\mathtt{in}\,\langle\mathtt{hd} \hookrightarrow u_1,\mathtt{tl} \hookrightarrow u_2\rangle &\triangleq (\,\lambda\,(\,x\,)\,\langle u_1,u_2\rangle\,)(\,u\,)\\ \mathtt{hd}(\,u\,) &\triangleq u\cdot\mathtt{l}\\ \mathtt{tl}(\,u\,) &\triangleq (\,\lambda\,(\,x\,)\,\langle u_1,u_2\rangle\,)(\,u\cdot\mathtt{r}\,).\end{aligned}$$

The encoding relies on binary products as defined in Solution **21.1**.

21.7. The translation is given as follows:

$$\begin{aligned} x^* &\triangleq x \\ (u_1(\,u_2\,))^* &\triangleq \texttt{ap}(\,u_1^*\,;u_2^*\,) \\ (\lambda\,(\,x\,)\,u)^* &\triangleq [\,x\,]\,u^* \end{aligned}$$

Compositionality of the translation follows directly from Solution **3.5**.

Then proceed by induction on rules (21.2). The only difficult case is rule (21.2f), stating that

$$(\ \lambda\,(\,x\,)\,u_1\,)(\,u_2\,) \equiv \{u_2/x\}u_1.$$

This equation is handled as follows:

$$\begin{aligned} ((\ \lambda\,(\,x\,)\,u\,)(\,u_2\,))^* &= (\ \lambda\,(\,x\,)\,u\,)^*(\,u_2^*\,) \\ &= (\,[\,x\,]\,u^*\,)(\,u_2^*\,) \\ &\equiv \{u_2^*/x\}u^* \\ &= (\{u_2/x\}u)^*. \end{aligned}$$

The second-to-last equation is Exercise **3.4**.

Chapter 22

22.1. Here is one possible definition of `plus` in **DPCF**:

$$\lambda\,(\,a\,)\,\texttt{fix}\,p\,\texttt{is}\,\lambda\,(\,b\,)\,\texttt{ifz}\,b\,\{\texttt{zero} \hookrightarrow a \mid \texttt{succ}(\,x\,) \hookrightarrow \texttt{succ}(\,p(\,x\,)\,)\}.$$

Examine the transition sequence $\texttt{plus}(\,\overline{5}\,)(\,\overline{7}\,)$ carefully, and observe these points:

(a) Each recursive call to `plus` requires a run-time check to ensure that it is, in fact, a function, even though it cannot fail to be so because of the definition of `plus`.

(b) Each iteration requires examination and removal of the numeric tag from the argument b to determine whether or not it is zero.

(c) Each iteration but the last involves computation of the successor, which requires checking, removing, and re-attaching the numeric tag from its argument.

None of this takes place in the dynamics of **PCF** for the analogous definition of addition.

22.2. Follow the pattern for the natural numbers given in Section 22.1, but with `nil` and `cons` forming separate classes of values. It is not clear whether `cons` should impose any class restrictions on its arguments; conventionally, it does not. The behavior of `append` suffers from similar deficiencies to those outlined in Solution **22.1**, for largely the same reasons.

22.3. Follow the same pattern as the treatment of the class of numbers in **DPCF**, with `nil` and `cons` playing the roles of zero and successor, respectively. In this case `cons` should require its second argument to be of the class `list`, which introduces additional overhead to the dynamics of `append`. Appending a list to a non-list will result in run-time failure at the last step, after the first list has been traversed.

22.4. Consideration of multiple arguments and multiple results amounts to an admission that more than one type is necessary in a practical language. Many issues arise, with no fully satisfactory solutions.

(a) There is no way to express the restriction to a particular number of arguments in a dynamic language, which has only one type. At bottom the arguments must be considered to be a tuple whose components are accessed by projections that may fail at run-time if the call site provides too few arguments. What happens with too many arguments is highly dependent on the implementation of tuples and projections.

(b) Multi-argument functions are often "optimized" by means that amount to a very special case of pattern matching. Match failure can still occur at run-time, rather than be caught at compile time. The core issue is not efficiency, but rather expressiveness—precise expression of invariants admits efficient implementation, but no amount of implementation tricks can make up for lack of expressive power.

(c) The arguments must be processed as a list of unbounded size. Access to the arguments requires a recursive traversal of this list. Many languages provide *ad hoc* forms of pattern matching to, say, allow one to name the first $k \geq 0$ elements of this list. Argument mismatch fails at run-time.

(d) Keyword argument passing is simply a mode of use of pattern matching for labeled product types. Here again in a dynamic setting any mismatches would result in run-time, rather than compile-time, failures.

(e) Multiple results are problematic. In a dynamic setting it is a matter of indifference whether the number of results is fixed or varying because the only choice is to return a value whose structure is determined dynamically. The caller must know the structure of the result, and arrange to access its parts by an unstated convention. Sometimes special syntax is introduced to cover common cases such as finite tuples of results, but without static typing it remains error-prone and difficult to remember the prevalent convention.

Chapter 23

23.1. Add two new classes, `nil` and `cons`, and extend the statics **HPCF** as follows:

$$\frac{\Gamma \vdash e : \texttt{unit}}{\Gamma \vdash \texttt{new}[\,\texttt{nil}\,](\,e\,) : \texttt{dyn}} \tag{A.14a}$$

$$\frac{\Gamma \vdash e : \texttt{dyn} \times \texttt{dyn}}{\Gamma \vdash \texttt{new}[\,\texttt{cons}\,](\,e\,) : \texttt{dyn}} \tag{A.14b}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[nil]}(e) : \texttt{unit}} \tag{A.14c}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[cons]}(e) : \texttt{dyn} \times \texttt{dyn}} \tag{A.14d}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{inst[nil]}(e) : \texttt{bool}} \tag{A.14e}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{inst[cons]}(e) : \texttt{bool}} \tag{A.14f}$$

The dynamics may be extended by following the pattern in Chapter 23.

Using these extensions we may make the following definitions of the null and pairing primitives of **DPCF**:

$$\begin{aligned}
\texttt{nil}^\dagger &\triangleq \texttt{nil}\,!\,\langle\rangle \\
\texttt{cons}(d_1\,;d_2)^\dagger &\triangleq \texttt{cons}\,!\,\langle d_1^\dagger, d_2^\dagger\rangle \\
\texttt{car}(d)^\dagger &\triangleq (d^\dagger\,@\,\texttt{cons})\cdot\texttt{l} \\
\texttt{cdr}(d)^\dagger &\triangleq (d^\dagger\,@\,\texttt{cons})\cdot\texttt{r} \\
\texttt{nil?}(d)^\dagger &\triangleq \texttt{if}\,\texttt{nil}\,?\,d^\dagger\,\texttt{then}\,\texttt{cons}(\texttt{nil};\texttt{nil})^\dagger\,\texttt{else}\,\texttt{nil}^\dagger \\
\texttt{cons?}(d)^\dagger &\triangleq \texttt{if}\,\texttt{cons}\,?\,d^\dagger\,\texttt{then}\,\texttt{cons}(\texttt{nil};\texttt{nil})^\dagger\,\texttt{else}\,\texttt{nil}^\dagger \\
\texttt{cond}(d\,;d_0\,;d_1) &\triangleq \texttt{if}\,\texttt{nil}\,?\,d\,\texttt{then}\,d_1\,\texttt{else}\,d_0
\end{aligned}$$

The choice of value $\texttt{cons}(\texttt{nil};\texttt{nil})^\dagger$ is arbitrary; it can be anything other than $\texttt{nil}^\dagger$.

23.2. Define dyn in **FPC** to be the recursive type

$$\texttt{rec}\,t\,\texttt{is}\,[\texttt{num} \hookrightarrow \texttt{nat}\,,\texttt{fun} \hookrightarrow t \rightharpoonup t\,,\texttt{nil} \hookrightarrow \texttt{unit}\,,\texttt{cons} \hookrightarrow t \times t].$$

The null and pairing primitives given in Section 22.2 may then be defined directly in **FPC**. For example, $\texttt{cond}(d\,;d_0\,;d_1)$ may be defined as

$$\texttt{case}\,\texttt{unfold}(d)\,\{\texttt{nil}\cdot\_ \hookrightarrow d_0 \mid \texttt{cons}\cdot\_ \hookrightarrow d_1 \mid \texttt{num}\cdot\_ \hookrightarrow d_1 \mid \texttt{fun}\cdot\_ \hookrightarrow d_1\}.$$

It is apparent from this definition that $\texttt{cond}(d\,;d_0\,;d_1)$ throws away useful information in dispatching to either d_0 or d_1 without passing any other information to either branch.

23.3. The translation of the append function of **DPCF** into **HPCF** is as follows:

$$\texttt{fix}\,a:\texttt{dyn}\,\texttt{is}\,\texttt{fun}\,!\,\lambda\,(x:\texttt{dyn})\,\texttt{fun}\,!\,\lambda\,(y:\texttt{dyn})\,e_{a,x,y},$$

where

$$a:\texttt{dyn}, x:\texttt{dyn}, y:\texttt{dyn} \vdash e_{a,x,y}:\texttt{dyn}$$

is the expression

$$\texttt{if}\ \texttt{nil}\ ?\ x\ \texttt{then}\ \texttt{cons}\ !\ \langle (\ x\ @\ \texttt{cons}\ ) \cdot \texttt{l}, e'_{a,x,y} \rangle\ \texttt{else}\ y,$$

and where

$$a : \texttt{dyn}, x : \texttt{dyn}, y : \texttt{dyn} \vdash e'_{a,x,y} : \texttt{dyn}$$

is the expression

$$(\ a\ @\ \texttt{fun}\ )(\ (\ x\ @\ \texttt{cons}\ ) \cdot \texttt{r}\ ).$$

This code may be optimized by a process similar to that outlined in Section 23.3.

Chapter 24

24.1. Neither, it is invariant. Formally, the variance rules for function types imply that in the first component τ is covariant, yet in the second component τ is contravariant. It cannot be both, so it is neither. If the composite type were deemed covariant in τ, then one could construct a counterexample to type safety by exploiting the second component, and, conversely, if the composite were deemed contravariant in τ, then one could construct a counterexample to safety by exploiting the first component.

24.2. Attempting to show that $\rho_2 <: \rho_1$ requires showing that

$$t_2 <: t_1 \vdash \langle \texttt{eq} \hookrightarrow (\ t_2 \rightharpoonup \texttt{bool}\ ), \texttt{f} \hookrightarrow \texttt{bool} \rangle <: \langle \texttt{eq} \hookrightarrow (\ t_1 \rightharpoonup \texttt{bool}\ ) \rangle.$$

But this requires showing that $t_2 \rightharpoonup \texttt{bool} <: t_1 \rightharpoonup \texttt{bool}$, which requires showing that $t_1 <: t_2$, which is the opposite of what we have assumed. Then perhaps the suggested subtyping relation is false. Suppose that $\rho_2 <: \rho_1$, for the sake of a contradiction. So if $e : \rho_2$, then $e : \rho_1$ by subsumption, and hence that $\texttt{unfold}(\ e\ ) \cdot \texttt{eq} : \rho_1 \rightharpoonup \texttt{bool}$. Now choose $e : \rho_2$ to be

$$\texttt{fold}(\ \langle \texttt{eq} \hookrightarrow (\ \lambda\ (\ x : \rho_2\ )\ x \cdot \texttt{f}\ ), \texttt{f} \hookrightarrow \texttt{true} \rangle\ ).$$

Consider the application

$$(\ \texttt{unfold}(\ e\ ) \cdot \texttt{eq}\ )(\ e_1\ ) : \texttt{bool},$$

where

$$e_1 \triangleq \texttt{fold}(\ \langle \texttt{eq} \hookrightarrow \lambda\ (\ \_ : \rho_1\ )\ \texttt{true} \rangle\ ).$$

The expression e_1 is chosen to not contain an `f` field; the choice of `eq` field is immaterial and can be any function of type $\rho_1 \rightharpoonup \texttt{bool}$. Evaluation of the application "gets stuck" attempting to access the `f` component of e_1, a violation of type safety. Thus the assumed subtyping $\rho_2 <: \rho_1$ cannot be valid.

The relevance of this example is that it is sometimes thought that if one extends a tuple with a new field, then the type of the extension is always a subtype of the type being extended. But this fails when the tuples are of recursive type, and hence can be self-referential. Extant programming languages nevertheless postulate the incorrect subtyping relationship, and are consequently not type safe.

24.3. One may give an inductive definition of the judgment $\chi : \tau <: \tau'$ specifying that χ witnesses that τ is a subtype of τ'. The following incomplete set of rules illustrate the main ideas:

$$\frac{}{\lambda\,(\,x:\tau\,)\,x : \tau <: \tau} \tag{A.15a}$$

$$\frac{\chi_1 : \tau_1 <: \tau_2 \quad \chi_2 : \tau_2 <: \tau_3}{\lambda\,(\,x_1:\tau_1\,)\,\chi_2(\,\chi_1(\,x_1\,)\,) : \tau_1 <: \tau_3} \tag{A.15b}$$

$$\frac{J \subseteq I}{\lambda\,(\,x:\langle\tau_i\rangle_{i\in I}\,)\,\langle j \hookrightarrow x\cdot j \mid j \in J\rangle : \langle\tau_i\rangle_{i\in I} <: \langle\tau_j\rangle_{j\in J}} \tag{A.15c}$$

Reflexivity is witnessed by the identity function, transitivity by function composition, and width subtyping by creating a narrower tuple on passage to a supertype of a tuple type. To complete the definition fill in the rules that express the variance principles for product, sum, and function types, and check that the coercion is indeed an expression of function type mapping the subtype to the supertype.

The full coercion interpretation of product subtyping is coherent, but the proof requires an extension of the methods developed in Chapter 47 and will not be given here.

Chapter 25

25.1. Suppose that $\phi_1 \leq \phi_1'$ and $\phi_2 \leq \phi_2'$. We are to show that $\phi_1 \wedge \phi_2 \leq \phi_1' \wedge \phi_2'$. By rule (25.2f) it suffices to show that $\phi_1 \wedge \phi_2 \leq \phi_1'$ and that $\phi_1 \wedge \phi_2 \leq \phi_2'$. By rules (25.2d) and (25.2e) we have $\phi_1 \wedge \phi_2 \leq \phi_1$ and $\phi_1 \wedge \phi_2 \leq \phi_2$, respectively. But then by the assumptions and transitivity (rule (25.2b)), the result follows.

25.2. The forward direction is an immediate consequence of transitivity of entailment (rule (25.2b)). Suppose that $\phi'' \leq \phi$ implies $\phi'' \leq \phi'$ for every ϕ''. In particular we may take ϕ'' to be ϕ, because by reflexivity of entailment (rule (25.2a)) we have $\phi \leq \phi$. But then $\phi \leq \phi'$, as desired.
An alternative to Solution **25.1** making use of the Yoneda Lemma is as follows. Assuming $\phi_1 \leq \phi_1'$ and $\phi_2 \leq \phi_2'$, let us further assume that $\phi \leq \phi_1 \wedge \phi_2$. Then $\phi \leq \phi_1$ and $\phi \leq \phi_2$ by transitivity, and hence $\phi \leq \phi_1'$ and $\phi \leq \phi_2'$ by the assumptions, so $\phi \leq \phi_1' \wedge \phi_2'$, from which the result follows by the Yoneda Lemma.

25.3. The refinement $\texttt{fold}(\,\phi\,)$ is defined to refine a recursive type by the rule

$$\frac{\phi \sqsubseteq \{\texttt{rec}\,t\,\texttt{is}\,\tau/t\}\tau}{\texttt{fold}(\,\phi\,) \sqsubseteq \texttt{rec}\,t\,\texttt{is}\,\tau} \tag{A.16}$$

Satisfaction of this refinement may be defined by the rules

$$\frac{\Phi \vdash e \in_{\{\texttt{rec}\,t\,\texttt{is}\,\tau/t\}\tau} \phi}{\Phi \vdash \texttt{fold}(\,e\,) \in_{\texttt{rec}\,t\,\texttt{is}\,\tau} \texttt{fold}(\,\phi\,)} \tag{A.17a}$$

$$\frac{\Phi \vdash e \in_{\{\texttt{rec}\,t\,\texttt{is}\,\tau/t\}\tau} \texttt{fold}(\phi)}{\Phi \vdash \texttt{unfold}(e) \in_{\texttt{rec}\,t\,\texttt{is}\,\tau} \phi} \tag{A.17b}$$

25.4. Summand refinements may be used to improve the expressiveness of type refinements by extending the satisfaction rules for refinements.

(a) Summand refinements are stronger than general sum refinements, are contradictory with one another, and are covariant:

$$\frac{}{\texttt{l} \cdot \phi_1 \leq \phi_1 + \phi_2} \tag{A.18a}$$

$$\frac{}{\texttt{r} \cdot \phi_2 \leq \phi_1 + \phi_2} \tag{A.18b}$$

$$\frac{}{\texttt{l} \cdot \phi_1 \wedge \texttt{r} \cdot \phi_2 \leq \phi} \tag{A.18c}$$

$$\frac{\phi_1 \leq \phi_1'}{\texttt{l} \cdot \phi_1 \leq \texttt{l} \cdot \phi_1'} \tag{A.18d}$$

$$\frac{\phi_2 \leq \phi_2'}{\texttt{r} \cdot \phi_2 \leq \texttt{l} \cdot \phi_2'} \tag{A.18e}$$

(b) The introduction forms may be given summand refinements:

$$\frac{\Phi \vdash e_1 \in_{\tau_1} \phi_1}{\Phi \vdash \texttt{l} \cdot e_1 \in_{\tau_1+\tau_2} \texttt{l} \cdot \phi_1} \tag{A.19a}$$

$$\frac{\Phi \vdash e_2 \in_{\tau_2} \phi_2}{\Phi \vdash \texttt{r} \cdot e_2 \in_{\tau_1+\tau_2} \texttt{r} \cdot \phi_2} \tag{A.19b}$$

(c) Unreachable branches of case may be ignored:

$$\frac{\Phi \vdash e \in_{\tau_1+\tau_2} \texttt{l} \cdot \phi_1 \quad \Phi, x_1 \in_{\tau_1} \phi_1 \vdash e_1 \in_\tau \phi}{\Phi \vdash \texttt{case}\, e\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \in_\tau \phi} \tag{A.20a}$$

$$\frac{\Phi \vdash e \in_{\tau_1+\tau_2} \texttt{r} \cdot \phi_2 \quad \Phi, x_2 \in_{\tau_2} \phi_2 \vdash e_2 \in_\tau \phi}{\Phi \vdash \texttt{case}\, e\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \in_\tau \phi} \tag{A.20b}$$

(d) The learned information may be propagated by adding the hypothesis $e \in_{\tau_1+\tau_2} \texttt{l} \cdot \phi_1$ while refinement checking e_1, and correspondingly adding $e \in_{\tau_1+\tau_2} \texttt{r} \cdot \phi_2$ while refinement checking e_2. In the event of a further case analysis on e within either branch, one of the preceding two rules would apply.

The relevance to Boolean blindness stems from the identification of `bool` with $\mathtt{unit} + \mathtt{unit}$, and the application of the foregoing rules. Propagating $e \in_{\mathtt{bool}} \mathtt{true}$ and $e \in_{\mathtt{bool}} \mathtt{false}$ into the `then` and `else` branches is weaker than one might expect because doing so records information about the expression e itself, but not about any expression equivalent to e.

25.5. Recall from Chapter 23 the definition

$$\mathtt{dyn} \triangleq \mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{num} \hookrightarrow \mathtt{nat}\, ,\, \mathtt{fun} \hookrightarrow t \rightharpoonup t]$$

of the type `dyn` as a recursive type. With this in mind we may make the following definitions of the refinements of type `dyn`:

$$\mathtt{num}\,!\,\phi \triangleq \mathtt{fold}(\, \mathtt{num} \cdot \phi\,)$$
$$\mathtt{fun}\,!\,\phi \triangleq \mathtt{fold}(\, \mathtt{fun} \cdot \phi\,),$$

using a labeled form of summand refinements.

25.6. The verification consists of exhibiting a derivation composed of the rules of refinement satisfaction given in Chapter 25. The description of the behavior of the addition function is absurdly complicated, but accurately reflects the behavior of even so simple a function as addition when formulated using dynamic typing. The surface syntax of **DPCF**, in particular, is highly deceptive because it obscures the needless complexity of the underlying code that is exposed by type refinements (and by the optimization process carried out in detail in Chapter 23).

25.7. To establish that the dynamic addition function satisfies the stated refinement necessitates showing that the interior general recursion satisfies the refinement $\phi \triangleq \mathtt{fun}\,!\,\mathtt{num}\,!\,\top \rightharpoonup \mathtt{num}\,!\,\top$. To show this, assume that $p \in_{\mathtt{dyn}} \phi$ and show that its body also satisfies ϕ. The assumption that $p \in_{\mathtt{dyn}} \phi$ amounts to a loop invariant that guarantees that the cast of p to the function class cannot fail (by the safety theorem for type refinements), and hence can be safely eliminated, exactly as described in Section 23.3. Similarly, the assumptions that $x \in_{\mathtt{dyn}} \mathtt{num}\,!\,\top$ and $y \in_{\mathtt{dyn}} \mathtt{num}\,!\,\top$ suffice to underwrite the other optimizations described in that section.

Chapter 26

26.1. Disregarding self-reference, the type of the dispatch matrix allows for some entries to be absent by redefining its type as follows:

$$\tau_{\mathsf{dm}} \triangleq \langle \langle \tau^c \rightharpoonup \rho_d\, \mathtt{opt} \rangle_{d \in D} \rangle_{c \in C}.$$

Thus an entry in the dispatch matrix may either be absent, represented by `null`, or be present, represented by by $\mathtt{just}(\, e^c_d\,)$, where e^c_d is the behavior of method d on instances of class c, as before.

For the class-based implementation the object type ρ must be redefined to reflect the possibility of a "not understood" error:

$$\rho \triangleq \langle \rho_d \, \texttt{opt} \rangle_{d \in D}.$$

Message send is defined as before by projection, $e \Leftarrow d \triangleq e \cdot d$, but now has the type $\rho_d \, \texttt{opt}$ to reflect the possibility that the method d is undefined for the object e. The class vector is defined to be the tuple $e_{\mathsf{cv}} = \langle c \hookrightarrow c \hookrightarrow e^c \mid c \in C \rangle$, where

$$e^c \triangleq \lambda\,(\, u : \tau^c\,)\, \langle d \hookrightarrow d \hookrightarrow \texttt{ifnull}\, e_{\mathsf{dm}} \cdot c \cdot d \, \{\texttt{null} \hookrightarrow \texttt{null} \mid \texttt{just}(\, f\,) \hookrightarrow \texttt{just}(\, f(\, u\,)\,)\} \mid d \in D \rangle.$$

As before, the class vector has type

$$\tau_{\mathsf{cv}} \triangleq \langle \tau^c \rightharpoonup \rho \rangle_{c \in C},$$

albeit for the modified definition of τ_{obj} given above. Instantiation cannot fail, but sending a message to the instantiated object may signal a "not understood" error by returning `null`, rather than $\texttt{just}(\, e\,)$ for some value $e : \rho_d$.

For the method-based implementation, the object type τ remains as before, but the type of the method vector is altered to

$$\tau_{\mathsf{mv}} \triangleq \langle \tau_{\mathsf{obj}} \rightharpoonup \rho_d \, \texttt{opt} \rangle_{d \in D},$$

reflecting the possibility that a message send may fail. The implementation of the method vector is given by the tuple $\langle d \hookrightarrow d \hookrightarrow e_d \mid d \in D \rangle$, where

$$e_d \triangleq \lambda\,(\, \mathit{this} : \tau\,)\, \texttt{case}\, \mathit{this}\, \{c \cdot u \hookrightarrow \texttt{ifnull}\, e_{\mathsf{dm}} \cdot c \cdot d \, \{\texttt{null} \hookrightarrow \texttt{null} \mid \texttt{just}(\, f\,) \hookrightarrow \texttt{just}(\, f(\, u\,)\,)\} \mid c \in C\}.$$

Here again accessing the dispatch matrix checks whether the required entry is present or not, with the result type reflecting the outcome accordingly.

To account for self-reference we must now allow for the possibility that the behavior assigned by the dispatch matrix for a particular class and method may, when present and called, incur a "not understood" error by sending a message to an instance for which it is not defined. The type of the dispatch matrix changes to the following more complex type:

$$\tau_{\mathsf{dm}} \triangleq \langle \langle (\, \forall(\, t_{\mathsf{obj}} \,.\, \tau_{\mathsf{cv}} \rightharpoonup \tau_{\mathsf{mv}} \rightharpoonup \tau^c \rightharpoonup \rho_d \, \texttt{opt}\,)\,)\, \texttt{opt} \rangle_{d \in D} \rangle_{c \in C}.$$

The outermost option in each entry represents, as before, the presence or absence of a behavior for class c and method d. The option at the end reflects the possibility that the behavior may incur a "not understood" error when applied, as is now possible by creating instances and sending them messages using the given class- and method vectors.

The types τ_{cv} and τ_{mv} are given in terms of the abstract type, t, of objects as follows:

$$\tau_{\mathsf{cv}} \triangleq \langle \tau^c \rightharpoonup t_{\mathsf{obj}} \rangle_{c \in C}$$
$$\tau_{\mathsf{mv}} \triangleq \langle t_{\mathsf{obj}} \rightharpoonup \rho_d \, \texttt{opt} \rangle_{d \in D},$$

the latter reflecting the possibility of a "not understood" error upon message send. Within the entries of the dispatch matrix, class instantiation is mediated by the given class vector, without the possibility of error, and message send by the given method vector, with the possibility of error. The implementations of the class- and method vectors for specific choices of object type are given along the lines sketched above.

26.2. (a) The refinement $\texttt{inst}[\,c\,]$ of τ_{obj} is the (generalized) summand refinement $c \hookrightarrow \top_{\tau^c}$ corresponding to $c \in C$, imposing no condition on its instance data. The refinement $\texttt{admits}[\,d\,]$ is defined to be the (generalized) sum refinement $[\top_{\tau^c}]_{c \in C_d}$, which holds of any value of type τ_{obj}, provided that it is an instance of a class that admits method d. It is immediate that $\texttt{inst}[\,c\,] \leq \texttt{admits}[\,d\,]$ when $c \in C_d$, and hence when $d \in D_c$.

(b) The class- and method-vector refinements are chosen as follows:

$$\phi_{\mathsf{cv}} \triangleq \langle \top_{\tau^c} \rightharpoonup \texttt{inst}[\,c\,] \rangle_{c \in C}$$
$$\phi_{\mathsf{mv}} \triangleq \langle \texttt{admits}[\,d\,] \rightharpoonup \texttt{just}(\,\top_{\rho_d}\,) \rangle_{d \in D}$$

Check that $\phi_{\mathsf{cv}} \sqsubseteq \tau_{\mathsf{cv}}$ and $\phi_{\mathsf{mv}} \sqsubseteq \tau_{\mathsf{mv}}$ in the sense of Solution **26.1** with τ_{obj} chosen to be $[\tau^c]_{c \in C}$. It is immediate that $e_{\mathsf{cv}} \in_{\tau_{\mathsf{cv}}} \phi_{\mathsf{cv}}$ and not much harder to check that $e_{\mathsf{mv}} \in_{\tau_{\mathsf{mv}}} \phi_{\mathsf{mv}}$, bearing in mind the assumption that $e_{\mathsf{dm}} \in_{\tau_{\mathsf{dm}}} \phi_{\mathsf{dm}}$, which ensures that the dispatch yields a non-error result.

(c) Because $\texttt{new}[\,c\,](\,e\,) \in_{\tau_{\mathsf{obj}}} \texttt{inst}[\,c\,]$ and $\texttt{inst}[\,c\,] \leq \texttt{admits}[\,d\,]$ whenever $d \in D_c$, it follows that $\texttt{new}[\,c\,](\,e\,) \in_{\tau_{\mathsf{obj}}} \texttt{admits}[\,d\,]$ whenever $d \in D_c$. Consequently, $\texttt{new}[\,c\,](\,e\,) \Leftarrow d \in_{\rho_d} \texttt{just}(\,\top_{\rho_d}\,)$ whenever $d \in D_c$, which is to say that a "not understood" error does not arise for well-refined message send operations.

26.3. The problem is to define the entries $e_{\mathsf{ev}}^{\texttt{num}}$ and $e_{\mathsf{od}}^{\texttt{num}}$ with the specified behavior. These have similar overall form, with different method bodies:

$$e_{\mathsf{ev}}^{\texttt{num}} \triangleq \Lambda(\,t_{\mathsf{obj}}\,)\,\lambda\,(\,cv : \tau_{\mathsf{cv}}\,)\,\lambda\,(\,mv : \tau_{\mathsf{mv}}\,)\,\lambda\,(\,u : \tau^{\texttt{num}}\,)\,e_{\mathsf{ev}}$$
$$e_{\mathsf{od}}^{\texttt{num}} \triangleq \Lambda(\,t_{\mathsf{obj}}\,)\,\lambda\,(\,cv : \tau_{\mathsf{cv}}\,)\,\lambda\,(\,mv : \tau_{\mathsf{mv}}\,)\,\lambda\,(\,u : \tau^{\texttt{num}}\,)\,e_{\mathsf{od}}$$

Their respective method bodies are defined as follows:

$$e_{\mathsf{ev}} \triangleq \texttt{ifz}\,u\,\{\texttt{z} \hookrightarrow \texttt{true} \mid \texttt{s}(\,u'\,) \hookrightarrow mv \cdot \texttt{od}(\,cv \cdot c(\,u'\,)\,)\}$$
$$\texttt{od} \triangleq \texttt{ifz}\,u\,\{\texttt{z} \hookrightarrow \texttt{false} \mid \texttt{s}(\,u'\,) \hookrightarrow mv \cdot \texttt{ev}(\,cv \cdot c(\,u'\,)\,)\}.$$

Message send is effected by projection from mv with the argument being a new instance of c obtained by projection from cv.

26.4. Suppose that the abstract object type t is permitted to occur in the instance type τ^c of some class, or in the result type ρ_d of some method. The alterations required depend on whether we are considering the method-based or the class-based organization.

In the method-based organization the concrete object type τ becomes the recursive sum type

$$\tau \triangleq \texttt{rec}\,t\,\texttt{is}\,[\tau^c]_{c \in C}.$$

Correspondingly, the definition of the method vector e_{mv} of type $\mathtt{self}(\{\tau/t\}\tau_{\mathsf{mv}})$ becomes

$$\mathtt{self}\,mv\,\mathtt{is}\,\langle d \hookrightarrow d \hookrightarrow \lambda\,(\,this:\tau\,)\,\mathtt{case}\,\mathtt{unfold}(\,this\,)\,\{c \cdot u \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot d[\,\tau\,](\,e''_{\mathsf{cv}}\,)(\,e'_{\mathsf{mv}}\,)(\,u\,) \mid c \in C\} \mid d \in D\rangle,$$

wherein e'_{cv} is revised to become

$$e'_{\mathsf{cv}} \triangleq \langle c \hookrightarrow c \hookrightarrow \lambda\,(\,u : \{\tau/t\}\tau^c\,)\,\mathtt{fold}(\,c \cdot u\,) \mid c \in C\rangle : \{\tau/t\}\tau_{\mathsf{cv}}$$

and e'_{mv} remains as in the simple self-referential case. Object creation is redefined similarly, with instance type $\{\tau/t\}\tau^c$ possibly involving the object type,

$$\mathtt{new}[\,c\,](\,e\,) \triangleq \mathtt{fold}(\,c \cdot e\,) : \tau.$$

Message send remains as before, but with a result type that may include the object type:

$$e \Leftarrow d \triangleq \mathtt{unroll}(\,e_{\mathsf{mv}}\,) \cdot d(\,e\,) : \{\tau/t\}\rho_d.$$

In the class-based organization the concrete object type ρ becomes the recursive product type

$$\rho \triangleq \mathtt{rec}\,t\,\mathtt{is}\,\langle\rho_d\rangle_{d \in D}.$$

Correspondingly, the class vector e_{cv} of type $\mathtt{self}(\{\rho/t\}\tau_{\mathsf{cv}})$, becomes

$$\mathtt{self}\,cv\,\mathtt{is}\,\langle c \hookrightarrow c \hookrightarrow \lambda\,(\,u : \{\rho/t\}\tau^c\,)\,\mathtt{fold}(\,\langle d \hookrightarrow d \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot d[\,\rho\,](\,e''_{\mathsf{cv}}\,)(\,e''_{\mathsf{mv}}\,)(\,u\,) \mid d \in D\rangle\,) \mid c \in C\rangle,$$

wherein e''_{mv} is revised to become

$$e''_{\mathsf{mv}} \triangleq \langle d \hookrightarrow d \hookrightarrow \lambda\,(\,this:\rho\,)\,\mathtt{unfold}(\,this\,) \cdot d \mid d \in D\rangle : \{\rho/t\}\tau_{\mathsf{mv}}$$

and e''_{cv} remains as in the chapter. Message send is redefined similarly:

$$e \Leftarrow d \triangleq \mathtt{unfold}(\,e\,) \cdot d : \{\rho/t\}\rho_d.$$

Object creation remains as before, but taking an argument of type $\{\rho/t\}\tau^c$:

$$\mathtt{new}[\,c\,](\,e\,) \triangleq \mathtt{unroll}(\,e_{\mathsf{cv}}\,) \cdot c(\,e\,) : \rho,$$

Chapter 27

27.1. We wish to extend C with a new class c^*, and to define the dispatch matrix entry $e_d^{c^*}$ by inheritance to be e_d^c. The chief difficulty is that the entries are parameterized by the abstract class- and method vectors. Adding a new class c^* extends C, so that the type of the extended class vector is (up to a reordering isomorphism) the product

$$\tau^*_{\mathsf{cv}} \triangleq (\,\langle\tau^c \rightharpoonup t_{\mathsf{obj}}\rangle_{c \in C}\,) \times (\,\tau^{c^*} \rightharpoonup t_{\mathsf{obj}}\,).$$

By product subtyping we may regard $\tau^*_{\mathsf{cv}} <: \tau_{\mathsf{cv}}$ so that e^c_d is applicable when supplied with the extended class vector. In any case we demand that $\tau^{c^*} <: \tau^c$, as before, to ensure that the inherited method may be applied to the instance data of the new class c^*.

The analysis of method extension with inheritance is dual to that of class extension, with covariance on the result type and method vector.

27.2. No, even though the `ev` method is invoked on an instance of `nat*`, the inherited definition of `ev` will send an `od` message to an instance of `num`, and the recursion will continue on `num` instances thereafter. If the revised version of `od` differed in behavior, this fact would not be reflected in the behavior of `ev` on instances of `num*`.

27.3. Work in the context of Solution **26.4**, which permits instance data and results to be objects.

(a) Choose $\tau^{\texttt{num}}$ to be t_{obj}, the abstract type of objects. The intention is that this object be of a class that supports the `ev` and `od` methods; this can be enforced using refinements in the manner of Solution **26.2**. The method bodies for `ev` and `od` are defined as follows:[1]

$$e^{\texttt{num}}_{\texttt{ev}} \triangleq \ldots mv \cdot \texttt{ev}(\, u \,)$$
$$e^{\texttt{num}}_{\texttt{od}} \triangleq \ldots mv \cdot \texttt{ev}(\, u \,).$$

(b) Choose $\tau^{\texttt{zero}}$ to be `unit`, there being no useful instance data. Define `ev` and `od` as follows:

$$e^{\texttt{zero}}_{\texttt{ev}} \triangleq \ldots \texttt{true}$$
$$e^{\texttt{zero}}_{\texttt{od}} \triangleq \ldots \texttt{false}.$$

Choose $\tau^{\texttt{succ}}$ to be t_{obj}, with the intention that it be an object of class `num`, an invariant that may be enforced with refinements in the manner of Solution **26.2**. The `ev` and `od` methods are implemented as follows:

$$e^{\texttt{zero}}_{\texttt{ev}} \triangleq \ldots mv \cdot \texttt{od}(\, u \,)$$
$$e^{\texttt{zero}}_{\texttt{od}} \triangleq \ldots mv \cdot \texttt{ev}(\, u \,).$$

The instance data of the `succ` class is the predecessor, which is assumed to implement the `ev` and `od` methods.

(c) Now introduce a subclass `succ*` of `succ` that overrides the `od` method. Observe that the dynamics of dynamic dispatch ensures that sending `ev` or `od` to any instance of `succ*` will invoke the overridden `od` method.

Chapter 28

[1] The prefixing abstractions over t_{obj}, cv, mv, and u are elided for clarity.

28.1. The preservation proof consists of three parts:

(a) Rule (28.4a): We have $s = k \triangleright \mathtt{ifz}[e_0\,;x\,.\,e_1](e)$ and $s' = k\,;\mathtt{ifz}[e_0\,;x\,.\,e_1](-) \triangleright e$. By inversion of Rules (28.9a) and (28.9a) we have $k \div \tau$ and $\mathtt{ifz}[e_0\,;x\,.\,e_1](e) : \tau$. Therefore by Rules (28.8b), (28.7b), we have s' ok, as required.

(b) Rule (28.4b): We have $s = k\,;\mathtt{ifz}[e_0\,;x\,.\,e_1](-) \triangleleft \mathtt{z}$ and $s' = k \triangleright e$. By inversion of Rules (28.9b), (28.7b) and (28.8b), we have $k \div \tau$, $\mathtt{ifz}[e_0\,;x\,.\,e_1](-) : \mathtt{nat} \rightsquigarrow \tau$, $e : \mathtt{nat}$, and $e_0 : \tau$. But then we have s' ok by Rule (28.9a).

(c) Rule (28.4c): We have $s = k\,;\mathtt{ifz}[e_0\,;x\,.\,e_1](-) \triangleleft \mathtt{s}(e)$ and $s' = k \triangleright \{e/x\}e_1$. By inversion of Rules (28.9b), (28.7b) and (28.8b), we have $k \div \tau$, $\mathtt{ifz}[e_0\,;x\,.\,e_1](-) : \mathtt{nat} \rightsquigarrow \tau$, $\mathtt{s}(e) : \mathtt{nat}$, and hence by inversion and substitution for **PCF**, we have $\{e/x\}e_1 : \tau$. But then the result follows from Rule (28.9a).

The progress proof proceeds as follows. Suppose that s ok. Either $s = k \triangleright e$ or $e = k \triangleleft e$. In either case by inversion of Rules (28.9) we have $k \div \tau$ and $e : \tau$ for some type τ. In addition e val in the case that $e = k \triangleleft e$. By analyzing the statics and canonical forms for **PCF**, we may verify in each case that the state is either final or may make progress.

28.2. First, we need an additional form of frame representing argument evaluation:

$$\frac{e_1\ \mathsf{val}}{\mathtt{ap}(e_1\,;-)\ \mathsf{frame}} \tag{A.21}$$

Second, we must replace rule (28.5c) by these rules:

$$\frac{}{k\,;\mathtt{ap}(-\,;e_2) \triangleleft e_1 \longmapsto k\,;\mathtt{ap}(e_1\,;-) \triangleright e_2} \tag{A.22a}$$

$$\frac{}{k\,;\mathtt{ap}(\lambda[\tau](x\,.\,e)\,;-) \triangleleft e_2 \longmapsto k \triangleright \{e_2/x\}e} \tag{A.22b}$$

Corresponding modifications are required of the proofs of safety and correctness, following along similar lines to those given in Chapter 28.

28.3. Each step of the dynamics involves either extending the stack by one frame, and descending into a sub-expression, or analyzing the outermost form of a value and the topmost frame of a stack to determine how to proceed. These may all be performed in constant time with a suitable representation of stacks as linked data structures and expressions as trees. Determining whether an expression may take time proportional to the size of that value, so determining whether the machine is finished requires time proportional to the size of the resulting value (that is, to "read" the result). If answers are limited to natural numbers, this takes time proportional to the value of the number, because we are using unary representations. The run-time can be improved to logarithmic by using binary representations. Substitution takes time proportional to the size of the expression into which we are substituting, a significant cost.

28.4. The number of transitions taken by the **PCF** machine to compute the value v of an expression e is proportional to the size of the derivation of $e \longmapsto^* v$, taking into account the sub-derivations that check whether an expression is a value. For example, the machine always fully explores a numeral to compute its value, whereas the the value judgment makes this traversal without involving any transitions.

Chapter 29

29.1. The preservation proof is by induction on the machine dynamics, and the progress proof is by induction on the well-formation of states. The proof of preservation relies on there being one universal type `exn` of exception values; otherwise an exception value may be passed to an exception handler expecting a different type. The proof of progress must take into account the transitions that propagate an exception through the frames of the control stack until either a handler is reached, or the stack is exhausted. But in that case the state is final, according to Rules (29.6).

29.2. The following rules illustrate the main ideas:

$$\frac{e \longmapsto e'}{\texttt{raise}(e) \longmapsto \texttt{raise}(e')} \tag{A.23a}$$

$$\frac{e \ \mathsf{val}}{\texttt{ap}(\texttt{raise}(e);e_2) \longmapsto \texttt{raise}(e)} \tag{A.23b}$$

$$\frac{e \ \mathsf{val}}{\texttt{ap}(e_1;\texttt{raise}(e)) \longmapsto \texttt{raise}(e)} \tag{A.23c}$$

$$\frac{e_1 \ \mathsf{val}}{\texttt{try}(e_1;x.e_2) \longmapsto e_1} \tag{A.23d}$$

$$\frac{e \ \mathsf{val}}{\texttt{try}(\texttt{raise}(e);x.e_2) \longmapsto \{e/x\}e_2} \tag{A.23e}$$

29.3. The following rules are representative of an evaluation dynamics for **XPCF**:

$$\frac{v \ \mathsf{val}}{v \Downarrow v} \tag{A.24a}$$

$$\frac{e \Downarrow v}{\texttt{raise}(e) \Uparrow v} \tag{A.24b}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Uparrow v_2}{\texttt{ap}(e_1;e_2) \Uparrow v_2} \tag{A.24c}$$

$$\frac{e_1 \Uparrow v_1}{\mathtt{ap}(\, e_1\, ;e_2\,) \Uparrow v_1} \tag{A.24d}$$

$$\frac{e_1 \Downarrow \lambda\,(\, x : \tau\,)\, e \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e \Downarrow v}{\mathtt{ap}(\, e_1\, ;e_2\,) \Downarrow v} \tag{A.24e}$$

$$\frac{e_1 \Downarrow \lambda\,(\, x : \tau\,)\, e \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e \Uparrow v}{\mathtt{ap}(\, e_1\, ;e_2\,) \Uparrow v} \tag{A.24f}$$

$$\frac{e_1 \Downarrow v_1}{\mathtt{try}(\, e_1\, ; x\, .\, e_2\,) \Downarrow v_1} \tag{A.24g}$$

$$\frac{e_1 \Uparrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\mathtt{try}(\, e_1\, ; x\, .\, e_2\,) \Downarrow v_2} \tag{A.24h}$$

$$\frac{e_1 \Uparrow v_1 \quad \{v_1/x\}e_2 \Uparrow v_2}{\mathtt{try}(\, e_1\, ; x\, .\, e_2\,) \Uparrow v_2} \tag{A.24i}$$

The remaining rules follow a similar pattern.

Chapter 30

30.1. A closed value of type $\tau\,\mathtt{cont}$ has the form $\mathtt{cont}(\, k\,)$, where k is a control stack such that $k \div \tau$. This observation is enough to ensure progress. Preservation is assured because if $k \triangleright \mathtt{letcc}\, x\, \mathtt{in}\, e$ ok, then $k \div \tau$ and $x : \tau\,\mathtt{cont} \vdash e : \tau$ for some type τ. Consequently, $\{\mathtt{cont}(\, k\,)/x\}e : \tau$ by substitution, which is enough for preservation.

30.2. (a) $\lambda\,(\, x : \tau\,\mathtt{cont}\,\mathtt{cont}\,)\,\mathtt{letcc}\, k\,\mathtt{in}\,\mathtt{throw}\, k\,\mathtt{to}\, x$.

(b) $\mathtt{letcc}\, \mathit{ret}\,\mathtt{in}\,\mathtt{r}\cdot(\,\mathtt{letcc}\, r\,\mathtt{in}\,\mathtt{throw}\,\mathtt{l}\cdot(\,\mathtt{letcc}\, l\,\mathtt{in}\,\mathtt{throw}\, l\,\mathtt{to}\, r\,)\,\mathtt{to}\, \mathit{ret}\,)$.

(c) $\lambda\,(\, x : \tau_2\,\mathtt{cont} \to \tau_1\,\mathtt{cont}\,)\,\lambda\,(\, x_1 : \tau_1\,)\,\mathtt{letcc}\, k_2\,\mathtt{in}\,\mathtt{throw}\, x_1\,\mathtt{to}\, x(\, k_2\,)$.

(d) $\lambda\,(\, k : (\, \tau_1 + \tau_2\,)\,\mathtt{cont}\,)\,\langle e_1, e_2\rangle$, where

$$e_1 \triangleq \mathtt{letcc}\, r_1\,\mathtt{in}\,\mathtt{throw}\,\mathtt{l}\cdot(\,\mathtt{letcc}\, k_1\,\mathtt{in}\,\mathtt{throw}\, k_1\,\mathtt{to}\, r_1\,)\,\mathtt{to}\, k, \text{ and}$$
$$e_2 \triangleq \mathtt{letcc}\, r_2\,\mathtt{in}\,\mathtt{throw}\,\mathtt{r}\cdot(\,\mathtt{letcc}\, k_2\,\mathtt{in}\,\mathtt{throw}\, k_2\,\mathtt{to}\, r_2\,)\,\mathtt{to}\, k.$$

30.3. Define $\mathtt{stream} \triangleq \mathtt{rec}\, t\,\mathtt{is}\,(\,\mathtt{nat} \times t\,)\,\mathtt{cont}\,\mathtt{cont}$, so that fold and unfold provide the required isomorphism. The elimination operations, $\mathtt{hd}(\, e\,)$ and $\mathtt{tl}(\, e\,)$, on streams are defined by projection from the client expression

$$\mathtt{letcc}\, c\,\mathtt{in}\,\mathtt{throw}\, c\,\mathtt{to}\,\mathtt{unfold}(\, e\,).$$

The client evaluates to a pair consisting of the first number and the rest of the given stream by passing a return continuation to the stream generator, which provides it with the head and tail of the stream. The introduction generator $\texttt{gen}_{\texttt{stream}}\, x\, \texttt{is}\, e\, \texttt{in}\, \langle \texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2 \rangle$ is defined as the following producer expression:

$$(\ \texttt{fix}\, g\, \texttt{is}\, \lambda\,(\, x : \tau\,)\, \texttt{letcc}\, \mathit{ret}\, \texttt{in}\, \texttt{throw}\, \langle e_1, g(\, e_2\,)\rangle\, \texttt{to}\,(\ \texttt{letcc}\, p\, \texttt{in}\, \texttt{throw}\, \texttt{fold}(\, p\,)\, \texttt{to}\, \mathit{ret}\,)\,)(\, e\,),$$

where τ is the state type of the stream. The innermost parenthesized sub-expression creates a new stream (essentially a continuation), and returns it as the result of the generating function g which takes as argument the current state of the generator. When a client continuation is thrown to that stream, the next number, paired with the remaining numbers, is thrown to it.

Chapter 31

31.1. Simply add a primitive equality test $\texttt{eq} : \rho\,\texttt{sym} \times \rho\,\texttt{sym} \rightharpoonup \texttt{bool}$, and observe that it is applicable only if the two symbol references have the same associated type.

31.2. It is not difficult to program a linear search in **SPCF** using Solution **31.1** to test equality.

31.3. A natural ordering of symbols in a deterministic language such as **SPCF** would be their order of allocation represented by their order of declaration in the signature. The difficulty is that the ordering is not invariant under renaming, because α-equivalence need not respect ordering. One solution involves mutable state, which will be discussed in Chapter 35: maintain a global counter and associate a unique number with each symbol that is used to determine a linear ordering among them.

31.4. The main idea is to define take $'a$ to be the primitive symbol reference, and to extend it to all s-expressions by defining $'\texttt{nil} \triangleq \texttt{nil}$ and $'\texttt{cons}(\, e_1\,;e_2\,) \triangleq \texttt{cons}(\,'e_1\,;'e_2\,)$. So, in particular, $'(\, e_0, \ldots, e_{n-1}\,) \triangleq (\,'e_0, \ldots, 'e_{n-1}\,)$. For example, $'(\, a, b, c, d\,)$ is the list $(\,'a, 'b, 'c, 'd\,)$. Notice that $'\texttt{nil}$ is $\texttt{nil}$, the injection into the recursive sum, and is not a symbol! If numbers are included among s-expressions, then $'n$ would be defined to be n, and not a symbol that happens to have a numeric representation. These, and other related cases, have led to controversy among various Lisp dialects and implementations, in part because of the absence of a rigorous mathematical foundation, leaving only opinion and authority as the determinative criteria.

Chapter 32

32.1. Enrich the **FPCF** machine with transitions corresponding to allocating a symbol, putting a binding for a symbol, and getting a binding for a symbol as follows:

$$\frac{}{k \triangleright \texttt{newsym}\, a \sim \rho\, \texttt{in}\, e \longmapsto k\,;\texttt{newsym}\, a \sim \rho\, \texttt{in} - \triangleright e} \tag{A.25a}$$

$$\frac{}{k\,;\texttt{newsym}\,a \sim \rho\,\texttt{in} - \lhd e \longmapsto k \lhd e} \tag{A.25b}$$

$$\frac{}{k\,;\texttt{newsym}\,a \sim \rho\,\texttt{in} - \blacktriangleleft \;\longmapsto k \blacktriangleleft} \tag{A.25c}$$

$$\frac{}{k \rhd \texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in}\,e_2 \longmapsto k\,;\texttt{put} - \texttt{for}\,a\,\texttt{in}\,e_2 \rhd e_1} \tag{A.25d}$$

$$\frac{}{k\,;\texttt{put} - \texttt{for}\,a\,\texttt{in}\,e_2 \lhd e_1 \longmapsto k\,;\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in} - \rhd e_2} \tag{A.25e}$$

$$\frac{}{k\,;\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in} - \lhd e_2 \longmapsto k \lhd e_2} \tag{A.25f}$$

$$\frac{}{k\,;\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in} - \blacktriangleleft \;\longmapsto k \blacktriangleleft} \tag{A.25g}$$

$$\frac{}{k \rhd \texttt{get}\,a \longmapsto k \geq k\ ?\ a} \tag{A.25h}$$

$$\frac{}{k \geq k'\,;\texttt{put}\,e\,\texttt{for}\,a\,\texttt{in} -\ ?\ a \longmapsto k \lhd e} \tag{A.25i}$$

$$\frac{}{k \geq k'\,;\texttt{newsym}\,a \sim \rho\,\texttt{in} -\ ?\ a \longmapsto k \blacktriangleleft} \tag{A.25j}$$

$$\frac{(f \neq \texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in} -, f \neq \texttt{newsym}\,a \sim \rho\,\texttt{in} -)}{k \geq k'\,; f\ ?\ a \longmapsto k \geq k'\ ?\ a} \tag{A.25k}$$

Rules (A.25a) to (A.25c) mark the allocation of a new symbol by pushing a frame on the stack, and propagate normal and failure return through it. Rules (A.25d) to (A.25g) implement the stack dynamics of put. In particular, rule (A.25f) reveals the underlying problem discussed in Section 32.4: the value e may depend on the symbol a whose binding is being dropped on the transition. Rule (A.25h) initiates the lookup of a binding for a in the stack. The linear search of the stack for the most recent binding of a symbol a is implemented by Rules (A.25i) to (A.25k).

32.2. The following rules implement shallow binding:

$$\frac{}{\mu \parallel k \rhd \texttt{newsym}\,a \sim \rho\,\texttt{in}\,e \longmapsto \mu \otimes a \hookrightarrow \epsilon \parallel k\,;\texttt{newsym}\,a \sim \rho\,\texttt{in} - \rhd e} \tag{A.26a}$$

$$\overline{\mu \otimes a \hookrightarrow \epsilon \parallel k\,;\texttt{newsym}\,a \sim \rho\,\texttt{in}\,- \lhd e \longmapsto \mu \parallel k \lhd e} \tag{A.26b}$$

$$\overline{\mu \otimes a \hookrightarrow \epsilon \parallel k\,;\texttt{newsym}\,a \sim \rho\,\texttt{in}\,- \blacktriangleleft \longmapsto \mu \parallel k \blacktriangleleft} \tag{A.26c}$$

$$\overline{\mu \parallel k \rhd \texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in}\,e_2 \longmapsto \mu \parallel k\,;\texttt{put}\,-\,\texttt{for}\,a\,\texttt{in}\,e_2 \rhd e_1} \tag{A.26d}$$

$$\overline{\mu \otimes a \hookrightarrow s \parallel k\,;\texttt{put}\,-\,\texttt{for}\,a\,\texttt{in}\,e_2 \lhd e_1 \longmapsto \mu \otimes a \hookrightarrow s\,;e_1 \parallel k\,;\texttt{put}\,a\,\texttt{for}\,\rho\,\texttt{in}\,-\,- \rhd e_2} \tag{A.26e}$$

$$\overline{\mu \otimes a \hookrightarrow s\,;e_1 \parallel k\,;\texttt{put}\,-\,\texttt{for}\,a\,\texttt{in}\,- \lhd e_2 \longmapsto \mu \parallel k \lhd e_2} \tag{A.26f}$$

$$\overline{\mu \otimes a \hookrightarrow s\,;e_1 \parallel k\,;\texttt{put}\,-\,\texttt{for}\,a\,\texttt{in}\,- \blacktriangleleft \longmapsto \mu \parallel k \blacktriangleleft} \tag{A.26g}$$

$$\overline{\mu \otimes a \hookrightarrow s\,;e \parallel k \rhd \texttt{get}\,a \longmapsto \mu \otimes a \hookrightarrow s\,;e \parallel k \lhd e} \tag{A.26h}$$

$$\overline{\mu \otimes a \hookrightarrow \epsilon \parallel k \rhd \texttt{get}\,a \longmapsto \mu \otimes a \hookrightarrow \epsilon \parallel k \blacktriangleleft} \tag{A.26i}$$

Obtaining the current binding of a symbol is now immediate (rules (A.26h) and (A.26i)), at the expense of maintaining synchrony between the binding stacks and the control stack.

32.3. Preliminarily, it is important to think carefully about the interaction between the dynamics of fluids and of continuations. The correct behavior of fluids is defined by the deep dynamics given by Solution **32.1**. The bindings of the fluids are determined by the `put` frames present on the target control stack. So, for example, if one seizes a continuation k, then puts a binding for a as an extension to k, then throws back to k, the binding for a is implicitly restored to its state prior to the `put`. The putatively more efficient shallow dynamics given in Solution **32.2** requires a fairly complicated protocol to ensure that when a seized continuation is reactivated, the binding stacks of the active fluids are restored to their proper state.

The implementation of exception handling in terms of fluids and continuations is given as follows:

$$\texttt{try}\,e_1\,\texttt{ow}\,x \hookrightarrow e_2 \triangleq \texttt{letcc}\,\mathit{ret}\,\texttt{in}\,\texttt{let}\,x\,\texttt{be}\,(\,\texttt{letcc}\,k\,\texttt{in}\,\texttt{put}\,k\,\texttt{for}\,\texttt{hdlr}\,\texttt{in}\,\texttt{throw}\,e_1\,\texttt{to}\,\mathit{ret}\,)\,\texttt{in}\,e_2$$

$$\texttt{raise}(\,e\,) \triangleq \texttt{throw}\,e\,\texttt{to}\,(\,\texttt{get}\,\texttt{hdlr}\,)$$

The type associated to the fluid-bound symbol `hdlr` is `exn cont`. The implementation of `handle` restores the proper handler on both normal and exceptional return consistently with the deep dynamics of fluid binding.

Chapter 33

33.1. The implementation of named exception handling is given as follows:

$$\texttt{dclexc}\ a\ \texttt{of}\ \tau\ \texttt{in}\ e \triangleq \texttt{newsym}\ a \sim \tau\ \texttt{in}\ e$$

$$\texttt{raiseexc}\ a \cdot e \triangleq \texttt{raise}(\, a \cdot e\,)$$

$$\texttt{tryexc}\ e\ \texttt{ow}\ a_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid a_n \cdot x_n \hookrightarrow e_n \mid x \hookrightarrow e' \triangleq$$
$$\texttt{try}\ e\ \texttt{ow}\ x \hookrightarrow \texttt{match}\ x\ \texttt{as}\ a_1 \cdot x_1 \hookrightarrow e_1\ \texttt{ow} \hookrightarrow \texttt{match}\ x\ \texttt{as}\ a_n \cdot x_n \hookrightarrow e_n\ \texttt{ow} \hookrightarrow e'$$

Observe that what are called exceptions in a named exception mechanism are just dynamically allocated classes.

33.2. The suggested implementation of dynamic classification treats a classified value as a function of type $\texttt{unit} \rightarrow \texttt{unit}$ that, when activated, assigns the underlying value to its class, which is, as suggested, represented by a free assignable. Allocating a new class amounts to allocating a new free assignable and returning a reference to it. Creating a classified value with a specified class creates an encapsulated assignment to the class, and checking whether a classified value is of a given class is achieved by checking whether the classified value modifies the given class.

For notational convenience decompose the given existential type τ as $\exists\, clsfd :: \texttt{Ty}\,.\,\tau_1$, where τ_1 is $\exists\, cls :: \texttt{Ty} \rightarrow \texttt{Ty}\,.\,\tau_2$ and τ_2 is $\langle \texttt{newcls} \hookrightarrow \tau_{\texttt{newcls}}\,, \texttt{inref} \hookrightarrow \tau_{\texttt{inref}}, \texttt{isinref} \hookrightarrow \tau_{\texttt{isinref}} \rangle$ (whose constituent types are defined in the exercise). An implementation of dynamic classification according to the above strategy is a package e of type τ given by

$$\texttt{pack}\ \texttt{unit} \rightarrow \texttt{unit}\ \texttt{with}\ e_1\ \texttt{as}\ \exists\, clsfd :: \texttt{Ty}\,.\,\tau_1,$$

where e_1 is a package of type τ_1 given by

$$\texttt{pack}\ \lambda\,(\, t :: \texttt{Ty}\,)\,(\, t\ \texttt{opt}\ \texttt{ref}\,)\ \texttt{with}\ e_2\ \texttt{as}\ \exists\, cls :: \texttt{Ty} \rightarrow \texttt{Ty}\,.\,\tau_2.$$

and e_2 is a tuple of type τ_2 given by

$$\langle \texttt{newcls} \hookrightarrow e_{\texttt{newcls}}, \texttt{inref} \hookrightarrow e_{\texttt{inref}}, \texttt{isinref} \hookrightarrow e_{\texttt{isinref}} \rangle.$$

Finally, the operations are implemented as follows:

$$e_{\texttt{newcls}} \triangleq \Lambda(\, t :: \texttt{Ty}\,)\,\texttt{ref}\ \texttt{null}$$
$$e_{\texttt{inref}} \triangleq \Lambda(\, \texttt{Ty} :: t\,)\,\lambda\,(\,\langle c, x\rangle\,)\,\lambda\,(\,\langle\rangle\,)\,\texttt{let}\ \_\ \texttt{be}\ c \mathrel{*}= \texttt{just}(\, x\,)\ \texttt{in}\ \langle\rangle$$
$$e_{\texttt{isinref}} \triangleq \Lambda(\, t :: \texttt{Ty}\,)\,\Lambda(\, u :: \texttt{Ty}\,)\,\lambda\,(\,\langle c, d, f, x\rangle\,)\,\texttt{let}\ \_\ \texttt{be}\ c \mathrel{*}= \texttt{null}\ \texttt{in}\ \texttt{let}\ \_\ \texttt{be}\ d(\,\langle\rangle\,)\ \texttt{in}\ e'_{\texttt{isoinref}}$$
$$e'_{\texttt{isinref}} \triangleq \texttt{ifnull}\,(\, *c\,)\,\{\texttt{null} \hookrightarrow x \mid \texttt{just}(\, y\,) \hookrightarrow f(\, y\,)\}.$$

There is no need to reset the contents of the class after the test, because any future use of a classified value with that class will reset it.

Chapter 34

34.1. To add array assignables to **RMA** proceed as follows:

- Introduce a command to declare an array A, $\mathtt{dcl}\,A[e_1] := e_2\,\mathtt{in}\,m$, where $e_1 : \mathtt{nat}$, $e_2 : \mathtt{nat} \rightharpoonup \tau$, and $m \mathrel{\dot\sim} \rho$. The function e_2 provides the initial values for the elements of the array.
- Introduce a family of commands $|A|$ indexed by array assignables A that returns the number of elements of A.
- Introduce families of commands $A[i]$ and $A[i] := e$ that, respectively, retrieve the ith element of A and update the ith element of A with a new value given by e. It is, of course, a fatal error to exceed the size bound of an array assignable.
- The memory μ maps scalar assignables to scalar values, and it maps array assignables to a size n and a length-n sequence s of scalar values.

34.2. By the miracle of α-conversion each recursive call allocates a *fresh* version of a, which is implicitly renamed to, say, a', before being entered into the signature as a new assignable. The body of the procedure is correspondingly renamed so that a becomes a' wherever it occurs, so that there can be no confusion among the multiple active instances of the "same" assignable declaration.

34.3. The procedure declaration with own assignables

$$\mathtt{proc}\,p(\,x:\tau\,):\rho\,\mathtt{is}\,\{\mathtt{own}\,a := e\,\mathtt{in}\,m\}\,\mathtt{in}\,m'$$

is short-hand for the composite command

$$\mathtt{dcl}\,a := e\,\mathtt{in}\,\mathtt{proc}\,p(\,x:\tau\,):\rho\,\mathtt{is}\,m\,\mathtt{in}\,m',$$

where

$$\mathtt{proc}\,p(\,x:\tau\,):\rho\,\mathtt{is}\,m\,\mathtt{in}\,m'$$

is itself short-hand for

$$\mathtt{bnd}\,p \leftarrow \mathtt{cmd}(\,\mathtt{ret}(\,\mathtt{fix}\,p\,\mathtt{is}\,\lambda\,(\,x:\tau\,)\,\mathtt{cmd}(\,m\,)\,)\,)\,;m'.$$

Thus, within the scope of p, the assignable a, which is private to the body of p, maintains its state across calls. Were the `own` declaration replaced by an ordinary declaration, each call would create a fresh instance of a, with no retention of state across calls.

34.4. Besides the tedium of threading the memory through each step of expression evaluation, you need only add one rule to account for assignables as values:

$$\frac{}{\mu \otimes a \hookrightarrow e \parallel a \xmapsto[\Sigma, a\sim\tau]{} \mu \otimes a \hookrightarrow e \parallel \mathtt{ret}(\,e\,)} \tag{A.27}$$

Final states are defined by the rule

$$\frac{e\ \mathsf{val}}{\mu \parallel e\ \mathsf{final}} \tag{A.28}$$

You may then prove memory invariance by induction on the revised rules, noting that no rule, include this one, alters the memory across a transition.

The class of passive commands may be isolated by a judgment m $\mathsf{passive}$ whose definition need not be given here. The statics for a passive block is given by the following rule:

$$\frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \mathtt{do}\,\{m\} : \tau}\ . \tag{A.29}$$

The dynamics of a passive block is as follows:

$$\frac{\mu \parallel m \underset{\Sigma}{\longmapsto} \mu' \parallel m'}{\mu \parallel \mathtt{do}\,\{m\} \underset{\Sigma}{\longmapsto} \mu' \parallel m}\ . \tag{A.30}$$

$$\frac{\mu \parallel e\ \mathsf{final}}{\mu \parallel \mathtt{do}\,\{\mathtt{ret}\,e\} \underset{\Sigma}{\longmapsto} \mu \parallel e}\ . \tag{A.31}$$

It should be clear that passive commands enjoy memory invariance, so this property extends to expressions that involve passive blocks.

34.5. First, introduce a *class* declaration command that takes an argument that is used to initialize the shared private state. Second, introduce a command to instantiate a class by providing the instance data as argument, and obtaining a tuple of procedures sharing private state in the manner of Solution **34.3**. The tuple of procedures is an object whose components may be called as ordinary procedures as described in the Chapter.

34.6. To use a consolidated stack k, include frames of the form $\mathtt{dcl}\,a := e\ \mathtt{in} -$ in which e represents the current contents of a, taking account of any modifications that may have been made to it since its declaration. To set a to another value you must update the appropriate frame in k to obtain a new stack k' with which to proceed. To get the contents of a you must traverse the stack looking for its declaration and its associated contents. The update and traversal can be optimized using imperative programming methods, at the expense of rendering the control stack to be an ephemeral data structure, creating complications for seizing stacks as continuations.

It is advantageous to separate the control stack from the memory, because it provides direct access to the current contents of an assignable, much as does the shallow binding dynamics of fluids. The control stack frame corresponding to an assignable declaration has the form $\mathtt{dcl}\,a := -\ \mathtt{in} -$, which has a hole for the binding as well as the body, so that it records the declaration of a, but not its contents. The contents of the active assignables is maintained in a separate memory, much as in the structural dynamics. The critical rules for managing the correlation between the declaration and the memory are as follows:

$$\frac{e \longmapsto^{\Sigma} e' \quad e' \ \mathsf{val}_{\Sigma}}{k \parallel \mu \rhd_{\Sigma} \mathtt{dcl}\, a := e \,\mathtt{in}\, m \longmapsto k\,;\mathtt{dcl}\, a := - \,\mathtt{in}\, - \parallel \mu \otimes a \hookrightarrow e' \rhd_{\Sigma, a \sim \tau} m} \tag{A.32a}$$

$$\frac{}{k\,;\mathtt{dcl}\, a := - \,\mathtt{in}\, - \parallel \mu \otimes a \hookrightarrow e' \lhd_{\Sigma} e \longmapsto k \parallel \mu \lhd_{\Sigma} e} \tag{A.32b}$$

On entry to a declaration of a the memory is extended with a new location, a, whose contents is initialized to the value of e. On exit from the declaration that location is deallocated from the memory. This behavior exemplifies the stack-allocation of assignables in **MA**. Indeed, we may view the memory as a stack onto which are pushed and popped bindings for assignables that are not otherwise declared in the memory.

Chapter 35

35.1. Define commands $* \, e_1[e_2]$ and $e_1[e_2] *= e_3$ by the following dynamics:

$$\frac{}{\mu \parallel * (\,\&A\,)[i] \longmapsto \mu \parallel A[i]} \tag{A.33a}$$

$$\frac{e \ \mathsf{val}}{\mu \parallel (\,\&A\,)[i] *= e \longmapsto \mu \parallel A[i] := e} \tag{A.33b}$$

35.2. The following recursive types each in turn satisfy the stated requirements:

(a) $(\,\mathtt{rec}\, t \,\mathtt{is}\, (\,\mathtt{nat} \times t\,)\,\mathtt{opt}\,)\,\mathtt{ref}$.

(b) $\mathtt{rec}\, t \,\mathtt{is}\, (\,(\,\mathtt{nat} \times t\,)\,\mathtt{opt}\,)\,\mathtt{ref}$.

(c) $\mathtt{rec}\, t \,\mathtt{is}\, (\,(\,\mathtt{nat} \times t\,)\,\mathtt{ref}\,)\,\mathtt{opt}$.

(d) $\mathtt{rec}\, t \,\mathtt{is}\, (\,\mathtt{nat} \times (\,t\,\mathtt{ref}\,)\,)\,\mathtt{opt}$.

(e) $\mathtt{rec}\, t \,\mathtt{is}\, (\,(\,\mathtt{nat}\,\mathtt{ref}\,) \times (\,t\,\mathtt{ref}\,)\,)\,\mathtt{opt}$.

(f) $\mathtt{rec}\, t \,\mathtt{is}\, (\,(\,(\,\mathtt{nat}\,\mathtt{ref}\,) \times (\,t\,\mathtt{ref}\,)\,)\,\mathtt{opt}\,)\,\mathtt{ref}$.

Each of these definitions has a claim to being that of a mutable linked list, but no one seems canonical compared to the others.

Chapter 36

36.1. Under the by-name dynamics given in Chapter 19 the value of ω is $\mathtt{s}(\,\omega\,)$, so that evaluation of $\mathtt{ifz}\,\omega\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,) \hookrightarrow e_1\}$ strips off the successor and evaluates $\{\omega/x\}e_1$. Should ω be evaluated again, the same process repeats, with each such evaluation recreating the successor of ω. Under the by-need interpretation the value of ω is $\mathtt{s}(\,@a\,)$ for some cell a containing $\mathtt{s}(\,@a\,)$. A conditional analysis once again strips the successor, and then evaluates $\{@a/x\}e_1$. Any further evaluation of the expression ω will once again result in the stored value $\mathtt{s}(\,@a\,)$, without recreating it. The memo table is self-referential, or circular, in that it contains a value that refers to its own memo cell, sharing the result once for all uses of ω. This behavior is one of the attractions of by-need evaluation; suspension types provide the same benefits in an eager setting.

36.2. The expression

$$(\,\mathtt{fix}\,l\,\mathtt{is}\,\lambda\,(\,x:\mathtt{nat}\,)\,\mathtt{fold}(\,\mathtt{cons}\cdot\langle x, l(\,\mathtt{s}(\,x\,)\,)\rangle\,)\,)(\,\mathtt{z}\,)$$

has the required type

$$\mathtt{rec}\,t\,\mathtt{is}\,[\mathtt{nil} \hookrightarrow \mathtt{unit}\,,\mathtt{cons} \hookrightarrow \mathtt{nat} \times t].$$

36.3. The lazy interpretation in terms of suspensions is given by the following equations:

$$\begin{aligned} \mathtt{unit} &\triangleq \mathtt{unit} \\ \tau_1 \times \tau_2 &\triangleq \widehat{\tau_1} \times \widehat{\tau_2} \\ \mathtt{void} &\triangleq \mathtt{void}\,\mathtt{susp} \\ \tau_1 + \tau_2 &\triangleq (\,\widehat{\tau_1} + \widehat{\tau_2}\,)\,\mathtt{susp} \\ \mathtt{rec}\,t\,\mathtt{is}\,\tau &\triangleq \mathtt{rec}\,t\,\mathtt{is}\,(\,\tau\,\mathtt{susp}\,). \end{aligned}$$

Notice the additional level of suspensions required for sum types that is not required for product types. These suspensions meet the requirement that, in the case of binary products, the summand need not be determined until the value is required. The nullary case is similar: the suspended expression must diverge, but only when its value is requested.

Chapter 37

37.1. The crux of the matter is that the *sequential* dynamics of the parallel let must be chosen so as to match the *parallel* dynamics. Specifically, when evaluating $\mathtt{par}(\,e_1\,;e_2\,;x_1\,.\,x_2\,.\,e\,)$, the sequential dynamics must demand that e_2 is fully evaluated, even if an exception arises while evaluating e_1. If it were to propagate the exception immediately, the parallel dynamics would perform excess work on e_2 until the exception in e_1 arises, losing the required correspondence.

In both the sequential and the parallel dynamics the crucial rules are as follows:

$$\frac{e_2\ \mathsf{val}}{\mathtt{par}(\mathtt{raise}(e_1);e_2;x_1.x_2.e) \longmapsto \mathtt{raise}(e_1)} \tag{A.34a}$$

$$\frac{e_1\ \mathsf{val}}{\mathtt{par}(e_1;\mathtt{raise}(e_2);x_1.x_2.e) \longmapsto \mathtt{raise}(e_2)} \tag{A.34b}$$

It may seem unnecessary to insist in the sequential dynamics that e_2 be a value before propagating an exception arising from e_1. But it is necessary to include this "extra work" in order to match the work done by the parallel dynamics. The sequential case is penalized to suit the parallel case.

37.2. The cost dynamics for parallelism in the presence of exceptions requires accounting for the work performed in a computation that raises an exception, as well as one that does not. The following rules are crucial:

$$\frac{e\ \mathsf{val}}{\mathtt{raise}(e) \Uparrow^{\mathbf{0}} e} \tag{A.35a}$$

$$\frac{e_1 \Downarrow^{c_1} v_1}{\mathtt{try}(e_1;x.e_2) \Downarrow^{c_1 \oplus \mathbf{1}} v_1} \tag{A.35b}$$

$$\frac{e_1 \Uparrow^{c_1} v_1 \quad \{v_1/x\}e_2 \Downarrow^{c_2} v_2}{\mathtt{try}(e_1;x.e_2) \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_2} \tag{A.35c}$$

$$\frac{e_1 \Uparrow^{c_1} v_1 \quad \{v_1/x\}e_2 \Uparrow^{c_2} v_2}{\mathtt{try}(e_1;x.e_2) \Uparrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_2} \tag{A.35d}$$

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad \{e_1, e_2/x_1, x_2\}e \Downarrow^{c} v}{\mathtt{par}(e_1;e_2;x_1.x_2.e) \Downarrow^{(c_1 \otimes c_2) \oplus c \oplus \mathbf{1}} v} \tag{A.35e}$$

$$\frac{e_1 \Uparrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2}{\mathtt{par}(e_1;e_2;x_1.x_2.e) \Uparrow^{(c_1 \otimes c_2) \oplus \mathbf{1}} v_2} \tag{A.35f}$$

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Uparrow^{c_2} v_2}{\mathtt{par}(e_1;e_2;x_1.x_2.e) \Uparrow^{(c_1 \otimes c_2) \oplus \mathbf{1}} v_2} \tag{A.35g}$$

$$\frac{e_1 \Uparrow^{c_1} v_1 \quad e_2 \Uparrow^{c_2} v_2}{\mathtt{par}(e_1;e_2;x_1.x_2.e) \Uparrow^{(c_1 \otimes c_2) \oplus \mathbf{1}} v_1} \tag{A.35h}$$

Observe that in the case that both parallel computations raise an exception, the first one in program order is propagated, in keeping with the structural dynamics. And as with the structural dynamics both parallel computations must complete before the result is propagated.

37.3. Assume that the structural dynamics of **XPCF** has been incorporated as local transitions of the **P** machine, as described in the chapter. There are now four binary join rules, corresponding to which parallel computations raise exceptions:

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau \,\{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1\,;a_2\,](\,x_1\,;x_2\,.\,e\,)\,\} \\ \underset{\mathsf{loc}}{\longmapsto} \\ \nu\, a \sim \tau \,\{\, a \hookrightarrow \{e_1, e_2 / x_1, x_2\} e \,\} \end{array}\right\}} \tag{A.36a}$$

$$\frac{e_1 = \mathtt{raise}(\,e\,) \quad e \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau \,\{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1\,;a_2\,](\,x_1\,;x_2\,.\,e\,)\,\} \\ \underset{\mathsf{loc}}{\longmapsto} \\ \nu\, a \sim \tau \,\{\, a \hookrightarrow \mathtt{raise}(\,e\,)\,\} \end{array}\right\}} \tag{A.36b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 = \mathtt{raise}(\,e\,) \quad e \ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau \,\{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1\,;a_2\,](\,x_1\,;x_2\,.\,e\,)\,\} \\ \underset{\mathsf{loc}}{\longmapsto} \\ \nu\, a \sim \tau \,\{\, a \hookrightarrow \mathtt{raise}(\,e\,)\,\} \end{array}\right\}} \tag{A.36c}$$

$$\frac{e_1 = \mathtt{raise}(\,e\,) \quad e \ \mathsf{val} \quad e_2 = \mathtt{raise}(\,e'\,) \quad e' \ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau \,\{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1\,;a_2\,](\,x_1\,;x_2\,.\,e\,)\,\} \\ \underset{\mathsf{loc}}{\longmapsto} \\ \nu\, a \sim \tau \,\{\, a \hookrightarrow \mathtt{raise}(\,e\,)\,\} \end{array}\right\}} \tag{A.36d}$$

The sequential ordering of the subtasks specified in the parallel let is used to determine which exception to propagate in the case that both parallel computations raise an exception so as to be consistent with the left-to-right order of the sequential dynamics. No exceptions are propagated until both parallel computations complete to ensure that the **P** machine performs the same work as specified by the structural and cost dynamics.

37.4. The parallel let, $\mathtt{par}(\,e_1\,;e_2\,;x_1\,.\,x_2\,.\,e\,)$ is translated to another parallel let, $\mathtt{par}(\,e_1'\,;e_2'\,;x_1'\,.\,x_2'\,.\,e'\,)$, as follows:

(a) $e_1' \triangleq \mathtt{try}\,\mathtt{l}\cdot e_1\,\mathtt{ow}\,x_1'' \hookrightarrow \mathtt{r}\cdot x_1''$;

(b) $e_2' \triangleq \mathtt{try}\,\mathtt{l}\cdot e_2\,\mathtt{ow}\,x_2'' \hookrightarrow \mathtt{r}\cdot x_2''$;

(c) $e' \triangleq \mathtt{case}\,x_1'\,\{\mathtt{l}\cdot x_1 \hookrightarrow \mathtt{case}\,x_2'\,\{\mathtt{l}\cdot x_2 \hookrightarrow e \mid \mathtt{r}\cdot x_2'' \hookrightarrow \mathtt{raise}(\,x_2''\,)\} \mid \mathtt{r}\cdot x_1'' \hookrightarrow \mathtt{raise}(\,x_1''\,)\}$.

Sums are used to record whether an expression has a normal or exceptional return, and the case analysis represents the join-point logic required for exception propagation consistently with the interpretation developed in the preceding exercises.

Chapter 38

38.1. Simply allocate a future for e_1, and replace uses of x in e_2 by synchronization with that future:

$$\texttt{letfut}\, x\, \texttt{be}\, e_1\, \texttt{in}\, e_2 \triangleq \texttt{let}\, x'\, \texttt{be}\, \texttt{fut}(\, e_1\,)\, \texttt{in}\, \{\texttt{fsyn}(\, x'\,)\,/\,x\}e_2.$$

38.2. Define $\texttt{par}(\, e_1\, ;\, e_2\, ;\, x_1\, .\, x_2\, .\, e\,)$ to stand for

$$\texttt{let}\ x_1'\ \texttt{be}\ \texttt{fut}(e_1)\ \texttt{in}\ \texttt{let}\ x_2\ \texttt{be}\ e_2\ \texttt{in}\ \texttt{let}\ x_1\ \texttt{be}\ \texttt{fsyn}(x_1')\ \texttt{in}\ e.$$

The order of bindings is important to ensure that evaluation of e_2 proceeds in parallel with evaluation of e_1.

Chapter 39

39.1. Let `true` on channel a be represented by the process

$$\$\,?\,a(\,\langle t, f\rangle\,.\,\$\,(\,!!\,(\,t\,;\,\langle\rangle\,;\,\mathbf{1}\,)\,)\,),$$

and let `false` on channel a be represented by the process

$$\$\,?\,a(\,\langle t, f\rangle\,.\,\$\,(\,!!\,(\,f\,;\,\langle\rangle\,;\,\mathbf{1}\,)\,)\,).$$

The conditional branch on the boolean at a between processes P_1 and P_2 may then be represented by

$$\nu\, t\,.\,\nu\, f\,.\,\$(\,!\,a(\,\langle \&\, t, \&\, f\rangle\,;\,\$\,(\,?\,t(\,\_\,.\,P_1\,) + ?\,f(\,\_\,.\,P_2\,)\,)\,)\,).$$

39.2. Define $P \triangleright p$ and $E \triangleright p$ as follows:

$$\mathbf{1} \triangleright p \triangleq\ !\,p(\,\langle\rangle\,)$$

$$(\,P_1 \otimes P_2\,) \triangleright p \triangleq \nu\, p_1\,.\,\nu\, p_2\,.\,(\,(\,P_1 \triangleright p_1\,) \otimes (\,P_2 \triangleright p_2\,) \otimes P_{1,2} \otimes P_{2,1}\,) \text{ where}$$

$$P_{1,2} \triangleq \$\,?\,p_1(\,\_\,.\,\$\,?\,p_2(\,\_\,.\,!\,p(\,\langle\rangle\,)\,)\,) \text{ and}$$

$$P_{2,1} \triangleq \$\,?\,p_2(\,\_\,.\,\$\,?\,p_1(\,\_\,.\,!\,p(\,\langle\rangle\,)\,)\,)$$

$$(\,!\,a(\,e\,)\,) \triangleright p \triangleq\ !\,a(\,e\,)$$

$$(\,\$\,E\,) \triangleright p \triangleq \$\,(\,E \triangleright p\,)$$

$$(\,\nu\, a \sim \tau\,.\,P\,) \triangleright p \triangleq \nu\, a \sim \tau\,.\,(\,P \triangleright p\,) \quad (a \neq p)$$

$$\mathbf{0} \triangleright p \triangleq \mathbf{0}$$

$$(\,E_1 + E_2\,) \triangleright p \triangleq (\,E_1 \triangleright p\,) + (\,E_2 \triangleright p\,)$$

$$(\,?\,a(\,x\,.\,P\,)\,) \triangleright p \triangleq\ ?\,a(\,x\,.\,(\,P \triangleright p\,)\,)$$

Using this we may define $P\,;Q$ by $\nu\, p\,.\,(\,(\,P \triangleright p\,) \otimes \$\,?\,p(\,\_\,.\,Q\,)\,)$, which arranges for the initiation of Q to be deferred until the completion of P. The channel p must be chosen to not already occur in P or in Q so as to avoid unintended interference with the protocol.

39.3. Define $G(i,o)$ to be the process

$$\$\,?\,o(\,\langle q,z\rangle\,.\,\$\,?\,i(\,\langle r,s\rangle\,.\,!\,o(\,\langle r \,\overline{\vee}\, z, q \,\overline{\vee}\, s\rangle\,)\,)\,).$$

This process is the gate array that implements the latch. Then define $L(i,o)$ to be the process

$$*\,G(i,o) \otimes !\,o(\,\langle \texttt{false},\texttt{false}\rangle\,).$$

The companion process to the gate array provides the initial values of Q and Z, which are required to activate the coupled gates.

It is easy to check that $L(i,o) \otimes I_{reset} i$ and $L(i,o) \otimes I_{set}(i)$ behave as specified. The latch is capable of receiving its own sent messages on the o channel, achieving the required feedback.

39.4. Define $P_1 + P_2$ to be the process

$$P \triangleq \nu\, a \sim \texttt{unit}\,.\,(\,\$\,?\,a(\,\_\,.\,P_1\,) \otimes \$\,?\,a(\,\_\,.\,P_2\,) \otimes !\,a(\,\langle\rangle\,)\,),$$

where a is chosen to not occur in either P_1 or P_2. By Rule (39.4d) the process P evolves to either

$$P' \triangleq \nu\, a \sim \texttt{unit}\,.\,(\,P_1 \otimes \$\,?\,a(\,\_\,.\,P_2\,)\,)$$

or

$$P' \triangleq \nu\, a \sim \texttt{unit}\,.\,(\,\$\,?\,a(\,\_\,.\,P_1\,) \otimes P_2\,)$$

Informally these are equivalent to

$$P_1 \otimes \nu\, a \sim \texttt{unit}\,.\,\$\,?\,a(\,\_\,.\,P_2\,)$$

and to

$$\nu\, a \sim \texttt{unit}\,.\,\$\,?\,a(\,\_\,.\,P_1\,) \otimes P_2,$$

respectively, by the choice of the channel a. The accompanying processes are inert because a is private and there is no sender within the scope of its declaration.

39.5. Represent the process P by the process

$$P' \triangleq \nu\, t\,.\,(\,S_t \otimes \$\,?\,a_1(\,x_1\,.\,P_1'\,) \otimes \ldots \otimes \$\,?\,a_k(\,x_k\,.\,P_n'\,)\,),$$

in which, for each $1 \leq i \leq n$,

$$P_i' \triangleq \nu\, s\,.\,\nu\, f\,.\,(\,!\,t(\,s,f\,) \otimes \$\,?\,s(\,\_\,.\,(\,F_t \otimes P_i\,)\,) \otimes \$\,?\,f(\,\_\,.\,(\,F_t \otimes !\,a_i(\,x_i\,)\,)\,)\,).$$

Here $S_t \triangleq \$\,?\,t(\,s,f\,.\,!\,s(\,\langle\rangle\,)\,)$ and $F_t \triangleq \$\,?\,t(\,s,f\,.\,!\,f(\,\langle\rangle\,)\,)$ are the Milner booleans reachable on channel t, which serves as the lock channel. When synchronized the receives check whether the lock is available. If so, it is seized, and the corresponding process is activated; if not, the message is resent for possible synchronization with another receiver executing concurrently.

39.6. The solution is straightforward; simply give up on specifying too accurately the types of values carried on each channel. The polyadic π-calculus is, in this respect, a uni-typed, rather than multi-typed, language.

Chapter 40

40.1. A class declaration is a command with the following syntax:

$$\mathsf{Cmd} \quad \texttt{newcls}[\,\tau\,](\,a\,.\,m\,) \quad \texttt{cls}\,a \sim \tau\,\texttt{in}\,m \quad \text{class declaration}$$

The statics of class declaration is given by the rule

$$\frac{\Gamma \vdash_{\Sigma\,a\sim\tau} m \mathrel{\dot{\sim}} \tau'}{\Gamma \vdash_{\Sigma} \texttt{newcls}[\,\tau\,](\,a\,.\,m\,) \mathrel{\dot{\sim}} \tau'} \tag{A.37}$$

Its dynamics is given by the following execution rule:

$$\frac{}{\texttt{newcls}[\,\tau\,](\,a\,.\,m\,) \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \nu\,a \sim \tau\,\{\,m\,\}} \tag{A.38}$$

The channel reference allocation command may then be defined by the equation

$$\texttt{newch}[\,\tau\,] \triangleq \texttt{newcls}[\,\tau\,](\,a\,.\,\texttt{ret}\,(\,\&\,a\,)\,).$$

40.2. The receive-on-channel-reference event $\texttt{rcvref}(\,e\,)$ is governed by the following statics:

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{cls}(\,\tau\,)}{\Gamma \vdash_{\Sigma} \texttt{rcvref}(\,e\,) : \texttt{event}(\,\tau\,)} \tag{A.39}$$

The dynamics of this construct is given by these rules:

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{rcvref}(\,e\,) \underset{\Sigma}{\longmapsto} \texttt{rcvref}(\,e'\,)} \tag{A.40a}$$

$$\frac{}{\texttt{rcvref}(\,\&\,a\,) \underset{\Sigma,a\sim\tau}{\longmapsto} \texttt{rcv}[\,a\,]} \tag{A.40b}$$

Chapter 41

41.1. A new channel reference may be allocated at any site, which is recorded in its type.

$$\frac{}{\Gamma \vdash_{\Sigma} \texttt{newch}[\,\tau\,] \mathrel{\dot{\sim}} \texttt{chan}[\,\tau\,](\,w\,) \mathbin{@} w} \tag{A.41}$$

Execution allocates a new channel situated at that site, and returns a reference to it.

$$\frac{}{\texttt{newch}[\,\tau\,] \xrightarrow[\Sigma]{\varepsilon @ w} \nu\, a \sim \tau \mathbin{@} w \,\{\, \texttt{ret}(\,\&\, a\,) \otimes \mathbf{1} \,\}} \tag{A.42}$$

The dynamic send command takes a channel reference as argument.

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \texttt{chan}[\,\tau\,](\,w\,) \quad \Gamma \vdash_{\Sigma} e_2 : \tau}{\Gamma \vdash_{\Sigma} \texttt{sndref}(\,e_1\,;e_2\,) \mathrel{\dot{\sim}} \texttt{unit} \mathbin{@} w} \tag{A.43}$$

It evolves to a static send command once the reference is resolved.

$$\frac{e \ \mathsf{val}_{\Sigma}}{\texttt{sndref}(\,\&\, a\,;e\,) \xrightarrow[\Sigma]{\varepsilon @ w} \texttt{snd}[\,a\,](\,e\,)} \tag{A.44}$$

The dynamic receive event takes a channel reference as argument.

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{chan}[\,\tau\,](\,w\,)}{\Gamma \vdash_{\Sigma} \texttt{rcvref}(\,e\,) : \texttt{event}[\,\tau\,](\,w\,)} \tag{A.45}$$

It evolves to a static receive event once the reference is resolved.

$$\frac{}{\texttt{rcvref}(\,\&\, a\,) \underset{\Sigma}{\longmapsto} \texttt{rcv}[\,a\,]} \tag{A.46}$$

41.2. To perform an asynchronous remote send, simply change the locus of control to w' before sending the message.

$$\texttt{at}\ w'\ \texttt{do}\ \texttt{sndref}(\,e\,;e'\,).$$

To perform a synchronous remote send, change the locus of control to w', then allocate a call-back channel on which the result is to be sent along with the payload.

$$\texttt{at}\ w'\ \texttt{do}\ \texttt{bnd}\ r \leftarrow \texttt{cmd}\ \texttt{newch}[\,\tau'\,]\,;\texttt{bnd}\ \_ \leftarrow \texttt{sndref}(\,e\,;\langle e',r\rangle\,)\,;\texttt{sync}(\,\texttt{rcvref}(\,r\,)\,).$$

The reply channel reference r refers to a channel at w', which may be sent along with the payload e' on the channel reference e. Execution then synchronizes on the reply channel to obtain the result.

Chapter 43

43.1. Under the identification of $\kappa_1 \times \kappa_2$ with $\Sigma\, \_ :: \kappa_1 \,.\, \kappa_2$, Rules (43.16a) and (43.16b) are instances of Rules (43.5c) and (43.5d).

The dependent elimination rules are derivable from the non-dependent ones via self-recognition. For suppose that

$$\Delta \vdash c :: \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2.$$

Then by self-recognition

$$\Delta \vdash c :: \Sigma\, u_1 :: \mathtt{S}(\, c \cdot \mathtt{l} :: \kappa_1 \,) \,.\, \kappa_2.$$

Then by sharing propagation

$$\Delta \vdash \Sigma\, u_1 :: \mathtt{S}(\, c \cdot \mathtt{l} :: \kappa_1 \,) \,.\, \kappa_2 \equiv \mathtt{S}(\, c \cdot \mathtt{l} :: \kappa_1 \,) \times \{c \cdot \mathtt{l}/u_1\}\kappa_2.$$

Therefore by Rule (43.16b)

$$\Delta \vdash c \cdot \mathtt{r} :: \{c \cdot \mathtt{l}/u_1\}\kappa_2,$$

as required.

43.2. Under the identification of $\kappa_1 \to \kappa_2$ with $\prod\, \_ :: \kappa_1 \,.\, \kappa_2$, Rule (43.17) is an instance of Rule (43.9c).

Conversely, suppose that

$$\Delta \vdash c :: \textstyle\prod u_1 :: \kappa_1 \,.\, \kappa_2 \quad \text{and} \quad \Delta \vdash c_1 :: \kappa_1.$$

By self-recognition and subsumption it follows that

$$\Delta \vdash c :: \textstyle\prod u_1 :: \mathtt{S}(\, c_1 :: \kappa_1 \,) \,.\, \kappa_2.$$

But then by sharing propagation we have

$$\Delta \vdash c :: \mathtt{S}(\, c_1 :: \kappa_1 \,) \to \{c_1/u_2\}\kappa_2$$

from which the result follows by Rule (43.17).

43.3. The kind $\kappa\{p := c\}$ is defined by induction on p by the following equations:

$$
\begin{aligned}
\Delta \vdash \kappa\{\varepsilon := c\} \ \mathsf{kind} &\triangleq \Delta \vdash \mathtt{S}(\, c :: \kappa \,) \ \mathsf{kind} \\
\Delta \vdash (\Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2)\{\mathtt{l}\, p := c\} \ \mathsf{kind} &\triangleq \Delta \vdash \Sigma\, u_1 :: \kappa_1' \,.\, \kappa_2 \ \mathsf{kind}, \ \text{where} \\
\Delta \vdash \kappa_1' \ \mathsf{kind} &\triangleq \Delta \vdash \kappa_1\{p := c\} \\
\Delta \vdash (\Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2)\{\mathtt{r}\, p := c\} \ \mathsf{kind} &\triangleq \Delta \vdash \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2' \ \mathsf{kind}, \ \text{where} \\
\Delta, u_1 :: \kappa_1 \vdash \kappa_2' \ \mathsf{kind} &\triangleq \Delta, u_1 :: \kappa_1 \vdash \kappa_2\{p := c\} \ \mathsf{kind}
\end{aligned}
$$

The proofs of the required properties proceed by induction on the simple path.

43.4. Any common prefix is traversed, and then kind modification is used to impose the required equation.

$$\Delta \vdash u :: \kappa \;/\; u \cdot \varepsilon \equiv u \cdot \varepsilon \;\mathsf{kind} \triangleq \Delta \vdash \kappa \;\mathsf{kind}$$

$$\Delta \vdash u :: \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2 \;/\; u \cdot \mathtt{l}\, p \equiv u \cdot \mathtt{l}\, q \;\mathsf{kind} \triangleq \Sigma\, u_1 :: \{u \cdot \mathtt{l}/u_1\}\kappa_1' \,.\, \kappa_2, \text{ where}$$

$$\Delta \vdash \kappa_1' \;\mathsf{kind} \triangleq \Delta \vdash u_1 :: \kappa_1 \;/\; u_1 \cdot p \equiv u_1 \cdot q \;\mathsf{kind}$$

$$\Delta \vdash u :: \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2 \;/\; u \cdot \mathtt{r}\, p \equiv u \cdot \mathtt{r}\, q \;\mathsf{kind} \triangleq \Delta \vdash \Sigma\, u_1 :: \kappa_1 \,.\, \{u \cdot \mathtt{r}/u_2\}\kappa_2' \;\mathsf{kind}, \text{ where}$$

$$\Delta, u_1 :: \kappa_1 \vdash \kappa_2' \;\mathsf{kind} \triangleq \Delta, u_1 :: \kappa_1 \vdash u_2 :: \kappa_2 \;/\; u_2 \cdot p \equiv u_2 \cdot q \;\mathsf{kind}$$

$$\Delta \vdash u :: \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2 \;/\; u \cdot \mathtt{l}\, p \equiv u \cdot \mathtt{r}\, q \;\mathsf{kind} \triangleq \Delta \vdash \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2' \;\mathsf{kind}, \text{ where}$$

$$\Delta, u_1 :: \kappa_1 \vdash \kappa_2' \;\mathsf{kind} \triangleq \Delta, u_1 :: \kappa_1 \vdash \kappa_2\{q := u_1 \cdot p\} \;\mathsf{kind}$$

$$\Delta \vdash u :: \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2 \;/\; u \cdot \mathtt{r}\, p \equiv u \cdot \mathtt{l}\, q \;\mathsf{kind} \triangleq \Delta \vdash \Sigma\, u_1 :: \kappa_1 \,.\, \kappa_2' \;\mathsf{kind}, \text{ where}$$

$$\Delta, u_1 :: \kappa_1 \vdash \kappa_2' \;\mathsf{kind} \triangleq \Delta, u_1 :: \kappa_1 \vdash \kappa_2\{p := u_1 \cdot q\} \;\mathsf{kind}$$

Chapter 44

44.1. Taking τ_{key} to be τ_{elt} and τ_{val} to be `bool`, define the module

$$\Gamma, D : \sigma_{\mathsf{dict}} \vdash M_{\mathsf{set}} : \sigma_{\mathsf{set}}$$

by the equations

$$M_{\mathsf{set}} \triangleq [\![D \cdot \mathtt{s} \,;\, \langle \mathtt{emp} \hookrightarrow e_{\mathsf{emp}} \,,\, \mathtt{ins} \hookrightarrow e_{\mathsf{ins}} \,,\, \mathtt{mem} \hookrightarrow e_{\mathsf{mem}} \rangle]\!]$$
$$e_{\mathsf{emp}} \triangleq D \cdot \mathtt{d} \cdot \mathtt{emp}$$
$$e_{\mathsf{ins}} \triangleq \lambda\,(\,\langle x, d\rangle : \tau_{\mathsf{elt}} \times D \cdot \mathtt{s}\,)\, D \cdot \mathtt{d} \cdot \mathtt{ins}(\,\langle\langle x, \mathtt{true}\rangle, d\rangle\,)$$
$$e_{\mathsf{mem}} \triangleq \lambda\,(\,\langle x, d\rangle : \tau_{\mathsf{elt}} \times D \cdot \mathtt{s}\,)\, \mathtt{ifnull}\, D \cdot \mathtt{d} \cdot \mathtt{fnd}(\,\langle x, d\rangle\,)\, \{\mathtt{null} \hookrightarrow \mathtt{false} \mid \mathtt{just}(\,\_\,) \hookrightarrow \mathtt{true}\}.$$

44.2. For $N : \sigma_{\mathsf{ord}}$, define σ_{nodset} to be σ_{set} with τ_{elt} chosen to be $N \cdot \mathtt{s}$, and for $S : \sigma_{\mathsf{nodset}}$, define $\sigma_{\mathsf{nodsetdict}}$ to be σ_{dict} with τ_{key} chosen to be $N \cdot \mathtt{s}$ and τ_{val} to be $S \cdot \mathtt{s}$. Define the module

$$N : \sigma_{\mathsf{ord}}, S : \sigma_{\mathsf{nodset}}, D : \sigma_{\mathsf{nodsetdict}} \vdash M_{\mathsf{grph}} : \sigma_{\mathsf{grph}}$$

by the following equations:

$$M_{\mathsf{grph}} \triangleq [\![D \cdot \mathtt{s} \,;\, [\![\tau_{\mathsf{edg}} \,;\, \langle \mathtt{emp} \hookrightarrow e_{\mathsf{emp}} \,,\, \mathtt{ins} \hookrightarrow e_{\mathsf{ins}} \,,\, \mathtt{mem} \hookrightarrow e_{\mathsf{mem}} \rangle]\!]]\!]$$
$$\tau_{\mathsf{edg}} \triangleq N \cdot \mathtt{s} \times N \cdot \mathtt{s}$$
$$e_{\mathsf{emp}} \triangleq D \cdot \mathtt{d} \hookrightarrow \mathtt{emp}$$
$$e_{\mathsf{ins}} \triangleq \lambda\,(\,\langle e, g\rangle : \tau_{\mathsf{edg}} \times D \cdot \mathtt{s}\,)\, D \cdot \mathtt{d} \cdot \mathtt{ins}(\,\langle e, g\rangle\,)$$
$$e_{\mathsf{mem}} \triangleq \lambda\,(\,\langle\langle s, t\rangle, g\rangle : \tau_{\mathsf{edg}} \times D \cdot \mathtt{s}\,)\, \mathtt{ifnull}\, D \cdot \mathtt{d} \cdot \mathtt{fnd}(\,s\,)\, \{\mathtt{null} \hookrightarrow \mathtt{false} \mid \mathtt{just}(\,a\,) \hookrightarrow S \cdot \mathtt{d} \cdot \mathtt{mem}(\,a\,)\}.$$

To determine whether $\langle s, t\rangle$ is a member of a graph, it suffices to determine whether t is a member of the adjacency set associated to s.

44.3. See Solution **43.3**.

44.4. Straightforward verification, given Solution **44.3**.

Chapter 45

45.1. The required functor may be defined by

$$M_{\mathsf{orddictfun}} \triangleq \lambda\,\langle K\,;V\rangle : \left(\textstyle\sum \_ : \sigma_{\mathsf{ord}} \cdot \sigma_{\mathsf{typ}}\right) \cdot M^{K,V}_{\mathsf{dict}},$$

where the body $M^{K,V}_{\mathsf{dict}}$ is readily adapted from the definition given in Chapter 44, taking τ_{key} to be $K \cdot \mathtt{s}$ and τ_{val} to be $V \cdot \mathtt{s}$.

45.2. The type abstraction σ_{ordset} of finite sets equipped with their ordered elements may be defined as follows:

$$\sigma_{\mathsf{ordset}} \triangleq \textstyle\sum E : \sigma_{\mathsf{ord}} \cdot \left[\!\left[t :: \mathtt{Ty}\,; \tau^{E}_{\mathsf{set}} \right]\!\right]$$
$$\tau^{E}_{\mathsf{set}} \triangleq \langle \mathtt{emp} \hookrightarrow t\,, \mathtt{ins} \hookrightarrow E \cdot \mathtt{s} \times t \to t\,, \mathtt{mem} \hookrightarrow E \cdot \mathtt{s} \times t \to \mathtt{bool}\rangle.$$

The signature of a functor implementing a set abstraction in terms of a given ordered type may be defined as follows:

$$\sigma_{\mathsf{setfun}} \triangleq \textstyle\prod E : \sigma_{\mathsf{ord}} \cdot \sigma_{\mathsf{ordset}}\{\,\cdot\,\mathtt{1} \cdot \mathtt{s} := E \cdot \mathtt{s}\}.$$

This signature may be implemented by the following functor:

$$M_{\mathsf{setfun}} \triangleq \lambda\, E : \sigma_{\mathsf{ord}} \cdot \mathtt{let}\ D\ \mathtt{be}\ M_{\mathsf{dictfun}}\,(\,\langle E\,; [\![\mathtt{bool}\,; \langle\rangle]\!]\rangle\,)\ \mathtt{in}\ M^{E}_{\mathsf{set}},$$

where M^{E}_{set} is readily adapted from M_{set} given in Solution **44.1**.

45.3. The signature $\sigma_{\mathsf{ordgrph}}$ of finite graphs on an ordered type of nodes may be defined as follows:

$$\sigma_{\mathsf{ordgrph}} \triangleq \textstyle\sum N : \sigma_{\mathsf{ord}} \cdot \left[\!\left[t_{\mathsf{grph}} :: \mathtt{Ty}\,; \left[\!\left[t_{\mathsf{edg}} :: \mathtt{S}(\,\tau^{N}_{\mathsf{edg}}\,)\,; \tau^{N}_{\mathsf{grph}} \right]\!\right] \right]\!\right]$$
$$\tau^{N}_{\mathsf{edg}} \triangleq N \cdot \mathtt{s} \times N \cdot \mathtt{s}$$
$$\tau^{N}_{\mathsf{grph}} \triangleq \langle \mathtt{emp} \hookrightarrow t_{\mathsf{grph}}\,, \mathtt{ins} \hookrightarrow \tau^{N}_{\mathsf{edg}} \times t_{\mathsf{grph}} \to t_{\mathsf{grph}}\,, \mathtt{mem} \hookrightarrow \tau^{N}_{\mathsf{edg}} \times t_{\mathsf{grph}} \to \mathtt{bool}\rangle.$$

The signature $\sigma_{\mathsf{grphfun}}$ of a functor implementing graphs in terms of an ordered type may be defined as follows:

$$\sigma_{\mathsf{grphfun}} \triangleq \textstyle\prod N : \sigma_{\mathsf{ord}} \cdot \sigma_{\mathsf{ordgrph}}\{\,\cdot\,\mathtt{1} \cdot \mathtt{s} := N \cdot \mathtt{s}\}.$$

A functor implementing a graph on an ordered set of nodes may be defined as follows:

$$M_{\mathsf{grphfun}} \triangleq \lambda\, N : \sigma_{\mathsf{ord}} \cdot \mathtt{let}\ S\ \mathtt{be}\ M_{\mathsf{setfun}}\,(\,N\,)\ \mathtt{in}\ \mathtt{let}\ D\ \mathtt{be}\ M_{\mathsf{dictfun}}\,(\,\langle N\,; S\rangle\,)\ \mathtt{in}\ M^{N,S,D}_{\mathsf{grph}},$$

where $M^{N,S,D}_{\mathsf{grph}}$ is readily adapted from M_{grph} given in Solution **44.2**.