A Functional Specification of Effects

Wouter Swierstra

February 2009

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy.

Abstract

This dissertation is about effects and type theory.

Functional programming languages such as Haskell illustrate how to encapsulate side effects using monads. Haskell compilers provide a handful of primitive effectful functions. Programmers can construct larger computations using the monadic return and bind operations.

These primitive effectful functions, however, have no associated definition. At best, their semantics are specified separately on paper. This can make it difficult to test, debug, verify, or even predict the behaviour of effectful computations.

This dissertation provides pure, functional specifications in Haskell of several different effects. Using these specifications, programmers can test and debug effectful programs. This is particularly useful in tandem with automatic testing tools such as QuickCheck.

The specifications in Haskell are not total. This makes them unsuitable for the formal verification of effectful functions. This dissertation overcomes this limitation, by presenting total functional specifications in Agda, a programming language with dependent types.

There have been alternative approaches to incorporating effects in a dependently typed programming language. Most notably, recent work on Hoare Type Theory proposes to extend type theory with axioms that postulate the existence of primitive effectful functions. This dissertation shows how the functional specifications implement these axioms, unifying the two approaches.

The results presented in this dissertation may be used to write and verify effectful programs in the framework of type theory.

Contents

Acknowledgements

I would like to thank my supervisor, Thorsten Altenkirch, for his guidance. Thorsten not only provided me with inspiration and ideas, but also gave me the freedom to pursue my own research interests – even those not directly related to my thesis.

I would like to thank my examiners, Graham Hutton and Greg Morrisett, for taking the time to read this dissertation and for providing many suggestions for improvement. Peter Hancock and Iain Lane also provided valuable feedback on parts of this dissertation, for which I am grateful.

I would like to thank my office mates in Nottingham over the years for their company and friendship: James Chapman, Peter Hancock, Mauro Jaskelioff, Conor McBride, Peter Morris, Nicolas Oury, and Rawle Prince.

Finally, I would like to thank Sanne for her love and support.

Chapter 1

Introduction

In the light of the ubiquity of modern information technology, it is rather worrying that large software systems are rarely free of bugs. Developing robust software systems is one of the major challenges of the 21st century.

One promising approach to more reliable software is strong static typing. Mainstream object-oriented languages such as C# (Microsoft, 2001) or Java (Gosling et al., 1996), enforce a more stringent type discipline than their low-level predecessors. This trend is even more evident in functional languages. The Haskell (Peyton Jones, 2003) type system, for example, is constantly being refined in pursuit of increasingly strong static guarantees.

At the very end of the correctness spectrum, theorem provers such as Coq (Bertot and Castéran, 2004), Isabelle/HOL (Nipkow et al., 2002), and PVS (Owre et al., 1992) are used to verify properties of computer programs. These systems typically use an abstract model to represent the software system. Such tools, however, have not been designed for *programming*.

These two avenues of research converge in *dependently typed programming*, where the language of proofs and programs coincide. Martin-Löf type theory (Martin-Löf, 1984) was originally proposed as a foundation of constructive mathematics. In such a type theory, you can view a type as a program's specification; the inhabitants of a type correspond to the programs that satisfy the corresponding specification (Nordström et al., 1990).

Despite its obvious virtues, dependently typed programming is still in its infancy. There are many hurdles that must be overcome before programming languages based on dependent types can be suitable for mainstream adoption. This thesis tackles one of them.

1.1 Effects

Martin-Löf type theory is nothing like a modern programming language. There are no floating point numbers, exceptions, concurrency, GUIs, database bindings, or any kind of Input/Output. The type theory itself is nothing more than a typed lambda calculus.

Yet functional programming languages, such as Haskell or ML (Milner et al., 1997), demonstrate how a functional programming language may be constructed on top of a small, core lambda calculus. If we intend to program in type theory, we would do well to start by studying how modern functional programming languages incorporate such features.

Before we can do so, we need to fix some terminology. Broadly speaking, an *effect* is any consequence of executing an expression besides returning a result. Forking a thread, throwing an exception, writing to a file, or updating mutable state are all examples of effects.

A function is called *pure* if it satisfies the following two properties:

- Executing the function has no observable effects, besides returning its result.
- The result of the function only depends on the value of its arguments. This forbids functions that consult data on the web, measure the temperature of the CPU, etc.

A programming language is called *pure* if all defined functions are pure functions by default.

1.1.1 A brief history

Early functional languages, such as Lisp (McCarthy et al., 1962), have a simple treatment of I/O. Automatic garbage collection precludes many low-level memory errors. Compilers provide a handful of primitive functions, such as *print* and *read* to interact with the user, to perform effects. These functions are *impure*: when they are executed, they have additional side effects besides returning a value. Many modern functional languages, such as ML, have a similar policy regarding effects: they do not distinguish between pure and impure functions.

This approach is less suitable for *non-strict* functional languages, where it is unclear when an expression will be evaluated. Consider the following list in Haskell:

$xs = [\mathit{print}\ \texttt{"Hello"}, \mathit{print}\ \texttt{"Dave"}]$

In a non-strict language it is not clear what text, if any, will get printed. If we only compute the length of the list, for example, neither element of the list need be evaluated further. Consequently, nothing will be printed. On the other hand, pattern matching on value stored in the head of the list will trigger the evaluation of the first print statement. Finally, we could also compare all the elements of the list to the unit value, which would require the entire list to be fully evaluated.

To predict the order that effects occur, you need to know the exact order in which expressions are evaluated. This is relatively easy to predict in a strict language, such as Lisp or ML, but much harder in lazy languages. For that reason, Haskell pursued alternative approaches to dealing with I/O.

The very first installments of Haskell provided side-effecting programs that had the following type:

$\textbf{type}\ \mathit{Dialogue} = [\mathit{Request}] \rightarrow [\mathit{Response}]$

A *Request* might ask the operating system to write a *String* to a file or read input from a handle. In a *Response*, the operating system may, for example, acknowledge the request, report an error, or return the *String* a user entered. The idea of that effectful functional programs can be modelled as stream processors can be traced back to Landin (1965).

While this approach is conceptually quite simple, it does have several serious drawbacks.

- It is easy to make a synchronization error. You have to carefully keep track of those responses in which you are interested. In large programs, the number of responses can be quite substantial: is the information for which you are looking in the 26th or 27th response from the operating system?

- It is quite easy to examine the answer to a question that you have not yet asked. Depending on the exact semantics, this will cause your program to crash or diverge.

- Furthermore, this simple model cannot cope well with concurrency: if two processes each request information from the operating system, there is no way to predict which question the operating system will answer first.
- Most importantly, however, such programs are non-compositional: there is no way to compose two *Dialogue* functions to construct a larger *Dialogue*.

Although solutions have been proposed to overcome some of these restrictions, this approach to I/O was abandoned after Haskell version 1.3.

1.1.2 Monadic I/O

In the late 1980's, Moggi (1989, 1991) observed that the denotational semantics of many effects shared a common structure: all these semantics formed a categorical structure known as a *monad*. A monad is a functor *m* together with a pair of operations:

$$\begin{aligned}&\mathit{return} :: a \to m\ a\\&\mathit{join}\ \ \ \ :: m\ (m\ a) \to m\ a\end{aligned}$$

Initially monads were used to study homological algebra (Mac Lane, 1971) but they soon found other applications, most notably in universal algebra. Moggi was the first to apply monads to the study of programming language semantics.

At the same time in the functional programming community, several people stumbled upon monadic structures. According to Wadler (1992), Spivey (1990) used monads at the same time to structure exception handling in pure functional languages. Thompson (1990) was one of the first to develop a semantics of I/O for lazy languages. In particular, Thompson presents several combinators for composing interactive programs that we would nowadays recognise as being monadic. Gordon (1994) mentions how Cupitt refined Thompson's combinators when developing the Kent Applicative Operating System (Cupitt, 1989; Turner, 1987). Cupitt discovered the following two primitive combinators:

$$\begin{aligned}&\mathit{return} :: a \to \mathit{interact}\ a\\&\mathit{comp}\ \ :: (a \to \mathit{interact}\ b) \to \mathit{interact}\ a \to \mathit{interact}\ b\end{aligned}$$

These two combinators correspond exactly to the unit and Kleisli-composition of a monad.

All these ideas were brought together by Wadler (1990, 1992). Initially, he proposed to use a generalised version of *list comprehensions* to write monadic expressions. Later Peyton Jones and Wadler (1993) showed how monads could be used to structure functional programs in an imperative fashion. These ideas were subsequently implemented in the Glasgow Haskell Compiler. When the final version of the Haskell Report (Peyton Jones, 2003) was written, monadic I/O was the established method of incorporating effects in a non-strict language.

Since then, Haskell's I/O has been augmented with a foreign function interface (Chakravarty, 2002), concurrency (Peyton Jones et al., 1996), software transactional memory (Harris et al., 2005), and many other features. Despite these new features, the underlying concept of monadic I/O has remained unchanged.

1.1.3 The challenge ahead

Although one might be tempted to use Haskell's monadic approach to I/O in a dependently typed programming language, there are good reasons to explore alternatives.

Dependently typed programming languages constitute a single framework for both proofs and programs. Haskell's monadic I/O shows how to safely incorporate effects in a pure functional language, but it does not give effects meaning. If we wish to write verified effectful programs we must know how effects behave. This is the central problem this thesis addresses.

1.2 Related work

There is a large body of research into suitable logics for reasoning about effects. Indeed, one of the earliest challenges for computer science was to find semantics for concepts such as mutable state, concurrency, and interrupts. We will briefly sketch the wider context of this work. We will give a more detailed comparison to closely related work in the latter chapters of this thesis.

1.2.1 Tackling the Awkward Squad

Peyton Jones (2001) has given semantics for several effects in Haskell's IO monad. This semantics is given by a process calculus, leaving the pure part of the language unspecified. Evaluation of pure functions corresponds to a silent transition; executing effects give rise to more interesting transitions. For instance, the transition rule for *putChar* is:

$$\{\mathbb{E}[\mathit{putChar}\ c]\} \xrightarrow{!c} \{\mathbb{E}[\mathit{return}\ ()]\}$$

This rule states that if the next effect to execute is *putChar c*, the program prints c and continues by executing $\mathbb{E}[\mathit{return}\ ()]$. The notation $\mathbb{E}[m]$ is used to denote an evaluation context where the m is the next expression to execute. The shorthand $!c$ is used to denote that the character c is printed to standard output.

Two processes P and Q can be composed in parallel, written $P \mid Q$. This makes it possible to give a straightforward treatment of concurrency. The same operator is used to record information about the computer's memory. For instance, $P \mid \langle M \rangle_r$ expresses that there is a process P running and that the expression M is stored at the memory location r.

Peyton Jones's semantics form an excellent tutorial on monadic I/O in general, but there are several reasons to investigate alternative semantics.

The semantics leaves the pure fragment of the language unspecified. While giving a complete semantics of Haskell is certainly beyond the scope of the paper, it leaves some important questions unanswered. When may a pure function be evaluated silently? What happens when the evaluation of a pure function fails to terminate? When is an expression pure or impure? Such questions must be addressed before the semantics can be used for the formal study of effectful programs.

Furthermore, the treatment of binding and scoping is rather ad hoc. Peyton Jones introduces an explicit 'end of scope' operator to limit the scope of freshly allocated references. There are several rules to shuffle such operators through a process expression. The application of such rules will introduce a substantial overhead to the verification of programs that allocate several references.

It certainly bears repeating that Peyton Jones (2001) presents a clear introduction to the semantics of various effects and monadic I/O in general. The points outlined above, however, make it unsuitable for our purpose.

1.2.2 Hoare Type Theory

Hoare Type Theory (Nanevski and Morrisett, 2005; Nanevski et al., 2006) extends traditional type theory with a new type constructor to decorate computations with preconditions and postconditions. Inhabitants of the Hoare type $\{P\}\, x : A\, \{Q\}$ correspond to computations that, provided the precondition P holds and the computation terminates, will return a value x of type A satisfying the postcondition Q.

Hoare Type Theory has been implemented as an axiomatic extension to Coq (Nanevski et al., 2008). The resulting system, Ynot, consists of a collection of axioms that postulate the existence of the Hoare type and several primitive functions that inhabit this type. Programmers can then use these functions to write effectful expressions, prove properties of these programs in Coq, and extract verified programs to Haskell or ML.

In contrast to Hoare Type Theory, we will focus on a specification of effects that does not extend our type theory with new axioms. As a result we will not have to show that our axioms preserve consistency; on the other hand, we will be more constrained by the theory in which we work – our programs will not be able to use general recursion or higher-order store.

We will discuss the relation between the functional specifications presented in this thesis and Hoare Type Theory in greater depth in Chapters 8 and 9. There we will show how our functional specifications form a model of (the core of) Hoare Type Theory.

1.2.3 Interactive programs in dependent type theory

Hancock and Setzer (2000) have given a type-theoretic description of *interaction structures*. They propose to model interactive interfaces by a four-tuple (S, C, R, n):

- The type S corresponds to the set of all possible states;
- For every state $s : S$, there is a set of valid commands $C\ s$;
- Every command $c : C\ s$ induces a set of valid responses $R\ s\ c$;
- Given an initial state s, command c, and response r, the function n will compute the next state of the system.

Hancock and Setzer show how such interaction structures give rise to predicate transformers. Hancock (2000) has shown that such structures are closed under many familiar operations from the refinement calculus (Back and von Wright, 1998).

Some of the functional specifications we present in this thesis could also be written as such interaction structures. By doing so, we could then use the various combinators to build larger, more complex systems. While this is certainly a promising direction for further research, it is beyond the scope of this thesis.

1.2.4 The essence of multitasking

Harrison (2006) has described how to use the *resumption monad* (Papaspyrou, 2001) to give a specification in Haskell of an operating system kernel. Such a kernel must deal with asynchronous exceptions, preemption, message passing, and other effects associated with operating systems. In this thesis we are more concerned with the effects relevant to programming, rather than the design of operating systems. Indeed, Harrison mentions that 'it is not the intention of the current work to model the awkward squad,' and does not explore this line of research further. This is precisely the goal of this thesis.

1.3 Overview

Synopsis Following this chapter, we will give an introduction to functional specifications of effects in Haskell (Chapter 2). More specifically, we will give functional specifications of mutable state, teletype interactions, and concurrency. We assume a basic familiarity with functional programming and Haskell in particular. Having read any introductory textbook on Haskell (Bird and Wadler, 1988; Hutton, 2007) should be sufficient. Chapter 2 not only serves to introduce the general direction of research, but also highlights the limitations of trying to write such specifications in a language without dependent types.

Chapter 2 gives separate functional specifications of various effects in isolation. Yet many programs rely on several different kinds of effects. To tackle this problem, we will discuss how to combine the specification of different effects in a modular fashion and describe a virtual machine on which different effects may be executed (Chapter 3). We will illustrate how to define new specifications in this framework by giving a functional specification of software transactional memory (Chapter 4).

Before we resolve some of the limitations of our specifications in the earlier chapters, we give a brief introduction to dependently typed programming in Agda (Chapter 5). We will not assume any familiarity with the language.

Using Agda we will give a total functional specification of mutable state (Chapter 6), thereby resolving many of the problems of the corresponding Haskell specification presented in Chapter 2. Besides implementing the specification itself, we will show how smart constructors that automate the weakening of references facilitate programming with our specification.

The specification in Chapter 6 can be extended to incorporate more static information. In Chapter 7 we illustrate how to constrain array access operations to write efficient distributed algorithms.

In Chapter 8 we will describe the Hoare state monad, a general abstraction for writing verified stateful programs. We will illustrate its use by proving that a simple tree relabeling function satisfies its specification. Furthermore, it forms a crucial part of our model of Hoare Type Theory in Chapter 9.

Theses Together these individual chapters support the four main theses borne out by this dissertation:

- Functional specifications can provide a formal semantics of the 'Awkward Squad.'
- To write precise functional specifications, we need to work in a programming language with dependent types.
- These specifications are a viable alternative to the axiomatic approach to side effects that is put forward by Hoare Type Theory and Haskell.
- Such functional specifications may provide the interface to the real world for tomorrow's dependently typed programming languages.

Contributions This dissertation makes several novel contributions, some of which have been published previously:

- This dissertation presents functional specifications of mutable state, teletype I/O, concurrency and transactional memory in Haskell. Portions of Chapter 2 have been published previously (Swierstra and Altenkirch, 2007).
- Chapter 3 describes a general technique for combining data structures and free monads in Haskell. It is loosely based on previous work (Swierstra, 2008). Neither the virtual machine in Chapter 3 nor the functional specification of software transactional memory in Chapter 4 have been published separately.
- Chapter 6 presents a novel, total specification of mutable state.
- Based on existing work (Swierstra and Altenkirch, 2008), Chapter 7 presents a total specification of mutable arrays and distributed arrays. It illustrates how a language with dependent types may be used to enforce domain specific invariants.
- The Hoare state monad in Chapter 8 and its application in the implementation of Hoare Type Theory in Chapter 9 have not been published previously.

The following chapters will deliver these contributions and support the four central theses.

Chapter 2

Functional specifications

2.1 Purely functional data structures

We have a solid mathematical understanding of functions that are both pure and total. As a result, it is relatively easy to test, debug, and formalise the properties of these functions. To illustrate this point we will start this chapter by implementing several well-known data structures in Haskell.

A simple data structure to implement in any functional language is a *stack*. We can represent a stack using a list; the various operations on stacks are straightforward to implement by pattern-matching on the list:

```
type Stack a = [a]

push :: Stack a → a → Stack a
push xs x     = x : xs

top :: Stack a → Maybe a
top [ ]       = Nothing
top (x : xs)  = Just x

pop :: Stack a → Maybe (Stack a)
pop [ ]       = Nothing
pop (x : xs)  = Just xs
```

We can formulate properties that our stacks should satisfy. For instance, the characteristic property of a stack is that the last element to be added to the stack is the first element that will be popped.

```
lifoProp :: Int → Stack Int → Bool
lifoProp x xs = top (push xs x) ≡ Just x
```

By using automated testing tools such as QuickCheck (Claessen and Hughes, 2000), we can check such properties of our implementation of stacks:

```
*Main> quickCheck lifoProp
OK, passed 100 tests.
```

This gives us some degree of certainty that this property does hold.

To convince ourselves further, we can prove such properties using equational reasoning:

$$\begin{array}{ll} & \mathit{top}\ (\mathit{push}\ \mathit{xs}\ x) \\ \equiv & \quad \{\text{definition of } \mathit{push}\} \\ & \mathit{top}\ (x : \mathit{xs}) \\ \equiv & \quad \{\text{definition of } \mathit{top}\} \\ & \mathit{Just}\ x \end{array}$$

These proofs of properties of pure and total functions are especially amenable to mechanised verification in proof assistants such as Coq (Bertot and Castéran, 2004) or Agda (Norell, 2008, 2007).

This example serves to illustrate the wide range of techniques available to debug, test, and reason about pure functional programs.

In a similar vein, we can use lists to implement queues:

```
type Queue a    = [a]
enqueue :: Queue a → a → Queue a
enqueue xs x    = xs ++ [x]
front :: Queue a → Maybe a
front []        = Nothing
front (x : xs)  = Just x
dequeue :: Queue a → Maybe (Queue a)
dequeue []      = Nothing
dequeue (x : xs) = Just xs
```

However, there is a problem with this definition. When we enqueue a new element, we need to traverse the entire queue. For a queue with n elements, *enqueue* is $O(n)$. For simple data structures such as queues, we should really be able to do better.

Okasaki (1998) has shown how to give a more efficient purely functional implementation of queues. Here we will take a slightly different approach. We will give an impure functional implementation of queues using mutable references.

2.2 The IO monad

In Haskell, a value of type *IORef a* is a mutable reference to a value of type *a*. There are three basic functions to manipulate such mutable references:

$$\begin{aligned}
&\mathit{newIORef} &&:: a \to \mathit{IO}\ (\mathit{IORef}\ a)\\
&\mathit{readIORef} &&:: \mathit{IORef}\ a \to \mathit{IO}\ a\\
&\mathit{writeIORef} &&:: \mathit{IORef}\ a \to a \to \mathit{IO}\ ()
\end{aligned}$$

Given an initial value, the *newIORef* function creates a reference storing that value; the *readIORef* function reads the value stored in its argument reference; finally, the *writeIORef* function writes a new value to a reference, overwriting any previous value stored therein. Note that these functions have no pure definition, but are primitives provided by a Haskell compiler.

These three functions return values in the *IO monad*. In the context of Haskell, a *monad* is any type constructor *m* that supports the following two operations, subject to certain coherence conditions:

$$\begin{aligned}
&\mathit{return} :: a \to m\ a\\
&(\gg=) :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}$$

The *return* function lifts a pure value into the monad. The operator $\gg=$, usually pronounced 'bind,' performs the computation associated with its first argument and passes the result to its second argument. As these are the only functions available to manipulate monads, programmers must sequence computations explicitly using the bind operator. In the case of the IO monad, *return* and bind are primitive functions.

As computations are first class values, we can define new control structures. Consider the following examples:

$$\begin{aligned}
&(\gg) :: \mathit{IO}\ a \to \mathit{IO}\ b \to \mathit{IO}\ b\\
&p \gg q = p \gg= \lambda x \to q
\end{aligned}$$

The $\gg$ operator sequences two computations, but discards the result of the first.

$$
\begin{aligned}
&\mathit{sequence} :: [\mathit{IO}\ a] \to \mathit{IO}\ [a] \\
&\mathit{sequence}\ [\,] \qquad\quad = \mathit{return}\ [\,] \\
&\mathit{sequence}\ (\mathit{io} : \mathit{ios}) = \mathit{io} \ggg\!= \lambda x \to \\
&\qquad\qquad\qquad\qquad\ \ \mathit{sequence}\ \mathit{ios} \ggg\!= \lambda \mathit{xs} \to \\
&\qquad\qquad\qquad\qquad\ \ \mathit{return}\ (x : \mathit{xs})
\end{aligned}
$$

The *sequence* combinator takes a list of IO actions, and performs them one by one, returning a list of their results.

Using these combinators, we can write functional programs in an imperative style. The *incr* function, for example, increments its argument reference and returns the reference's previous value:

$$
\begin{aligned}
&\mathit{incr} :: \mathit{IORef}\ \mathit{Int} \to \mathit{IO}\ \mathit{Int} \\
&\mathit{incr}\ \mathit{ref} = \\
&\quad \mathit{readIORef}\ \mathit{ref} \ggg\!= \lambda x \to \\
&\quad \mathit{writeIORef}\ \mathit{ref}\ (x + 1) \gg \\
&\quad \mathit{return}\ x
\end{aligned}
$$

Haskell provides some syntactic sugar to program with monads. For example, our *incr* function could also be written as:

$$
\begin{aligned}
&\mathit{incr} :: \mathit{IORef}\ \mathit{Int} \to \mathit{IO}\ \mathit{Int} \\
&\mathit{incr}\ \mathit{ref} = \mathbf{do} \\
&\quad x \leftarrow \mathit{readIORef}\ \mathit{ref} \\
&\quad \mathit{writeIORef}\ \mathit{ref}\ (x + 1) \\
&\quad \mathit{return}\ x
\end{aligned}
$$

For a complete description of the desugaring from the do-notation and a more thorough treatment of the IO monad, we refer to Peyton Jones's tutorial (Peyton Jones, 2001).

2.3 Implementing queues

The impure implementation of queues uses two mutable references to point to the front and back of the queue. The queue itself is represented by a

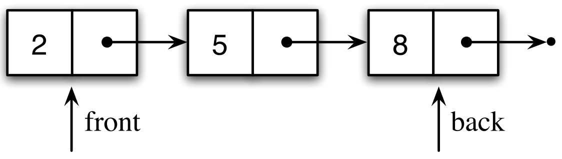

Figure 2.1: An example queue

linked list, where every cell contains an element of the queue and a pointer to the next cell:

$$\begin{aligned}
&\textbf{data}\ \mathit{Data} &&= \mathit{Cell}\ \mathit{Int}\ (\mathit{IORef}\ \mathit{Data}) \mid \mathit{NULL}\\
&\textbf{type}\ \mathit{Queue} &&= (\mathit{IORef}\ \mathit{Data}, \mathit{IORef}\ \mathit{Data})
\end{aligned}$$

Figure 2.1 contains an example queue corresponding to the list $[2,5,8]$; Listing 2.3 contains a standard implementation of functions to dequeue an element, to enqueue an element, and to create an empty queue.

This implementation of *enqueue* is much more efficient than the pure function we implemented previously. Unfortunately, this does come at a cost. Using QuickCheck, we would like to check the following property:

```
enqDeq :: [Int] → IO ([Maybe Int])
enqDeq xs = do
  q ← emptyQueue
  sequence (map (enqueue q) xs)
  sequence (map (λx → dequeue q) xs)


fifoProp :: [Int] → Bool
fifoProp xs = return (map Just xs) ≡ enqDeq xs
```

The *fifoProp* property, however, fails to type check: there is no way to compare if two computations in the IO monad are equal.

As a last resort, we could use the *unsafePerformIO* function that has type *IO a* → *a*:

```
fifoProp :: [Int] → Bool
fifoProp xs =
  map Just xs ≡ unsafePerformIO (enqDeq xs)
```

Listing 1 An implementation of queues

```
emptyQueue :: IO Queue
emptyQueue = do
  front ← newIORef NULL
  back ← newIORef NULL
  return (front, back)

enqueue :: Queue → Int → IO ()
enqueue (front, back) x = do
  newBack ← newIORef NULL
  let cell = Cell x newBack
  c ← readIORef back
  writeIORef back cell
  case c of
    NULL → writeIORef front cell
    Cell y t → writeIORef t cell

dequeue :: Queue → IO (Maybe Int)
dequeue (front, back) = do
  c ← readIORef front
  case c of
    NULL → return Nothing
    (Cell x nextRef) → do
      next ← readIORef nextRef
      writeIORef front next
      return (Just x)
```

As its name suggests, however, *unsafePerformIO* is rather hazardous. By using *unsafePerformIO* we may introduce all kinds of problems: a computation could be inlined by the compiler, causing I/O to be performed more than once; the exact semantics of *unsafePerformIO* depends on the Haskell compiler; most importantly, we lose type safety. This last point is illustrated by the following example:

```
ref :: IORef [a]
ref = unsafePerformIO (newIORef [])
ouch = do
  writeIORef ref [42]
  bang ← readIORef ref
  print (bang :: [Char])
```

This program is well-typed, but does 'go wrong' – in this instance the result is the character with the same binary representation as the integer 42. There is, however, no guarantee that such a character exists for every integer; in that case, this program would crash. Clearly, we should try to avoid functions such as *unsafePerformIO*.

In this chapter, we present a more viable alternative: pure specifications of impure functions.

2.4 Mutable state

The key idea throughout this thesis is to provide *functional specifications* of such effects by defining *pure functions, indistinguishable from their impure counterparts*. In the case of mutable references, this amounts to defining a pure model of a computer's memory.

We begin by modelling memory locations:

```
type Loc = Int
newtype IORef a = IORef Loc
```

A location is simply an integer. Making references and integers synonymous is rather perilous. Users might abuse this specific representation by inventing their own locations, performing pointer arithmetic, or accessing unallocated memory. To hide our internal representation, we introduce a new type for references. The library we propose exports the type *IORef*, but hides its implementation.

The type parameter of the *IORef* data type does not occur on the right hand side: this is called a *phantom type* (Leijen and Meijer, 1999). For the moment, we will only implement references storing the *Data* type we used in our implementation of queues. At the end of this section, however, we will use this phantom type to overcome this limitation.

Now we can define the syntax of our operations by defining the IO_s data type. It has one constructor for every operation on mutable references. Lastly, the *Return* constructor lifts any pure value to a stateful one. Note that IO_s is a monad. To facilitate programming with the IO_s type we define smart constructors, i.e., functions which help build IO_s terms. The smart constructors *newIORef*, *readIORef*, and *writeIORef* provide exactly the same interface as the *IO* monad. Using these smart constructors and the do-notation, we can write imperative programs such as our implementation of queues. The difference is that our definitions correspond to pure terms.

Yet the pure syntactic terms in the IO_s monad do not have any meaning yet. Before we can define the specification, we need to develop our model of the computer's memory further.

We keep track of two pieces of information to represent the state of the memory: the next available memory location and the current contents of the allocated memory. These are both captured by the *Store* data type:

```
data Store = Store {fresh :: Loc, heap :: Heap}
type Heap = Loc → Data

emptyStore :: Store
emptyStore = Store {fresh = 0}
```

Initially, we have no allocated references, so the next available location is zero. We leave the heap undefined: it should not be possible to access unallocated memory. Indeed, provided programmers only use the *newIORef*, *readIORef*, and *writeIORef* functions they will never access unallocated memory.

The complete pure specification of mutable state is in Listing 3. Haskell already has a very convenient library for writing stateful computations that revolves around the state monad:

Listing 2 The syntax of mutable state

```
data IO_s a =
    NewIORef Data (Loc → IO_s a)
  | ReadIORef Loc (Data → IO_s a)
  | WriteIORef Loc Data (IO_s a)
  | Return a

instance Functor IO_s where
  fmap f (NewIORef d io)     = NewIORef d (fmap f ∘ io)
  fmap f (ReadIORef l io)    = ReadIORef l (fmap f ∘ io)
  fmap f (WriteIORef l d io) = WriteIORef l d (fmap f io)
  fmap f (Return x)          = Return (f x)

instance Monad IO_s where
  return                     = Return
  (Return a) ≫= g            = g a
  (NewIORef d f) ≫= g        = NewIORef d (λl → f l ≫= g)
  (ReadIORef l f) ≫= g       = ReadIORef l (λd → f d ≫= g)
  (WriteIORef l d s) ≫= g    = WriteIORef l d (s ≫= g)

newIORef :: Data → IO_s (IORef Data)
newIORef d = NewIORef d (Return ∘ IORef)

readIORef :: IORef Data → IO_s Data
readIORef (IORef l) = ReadIORef l Return

writeIORef :: IORef Data → Data → IO_s ()
writeIORef (IORef l) d = WriteIORef l d (Return ())
```

```
newtype State s a = State {runState :: (s → (a,s))}
```

The state monad has several functions to manipulate the otherwise implicit state. In particular, we will make use the following functions:

```
get       :: State s s
gets      :: (s → a) → State s a
put       :: s → State s ()
evalState :: State s a → s → a
execState :: State s a → s → s
```

To access the hidden state, we use the *get* and *gets* functions that respectively return the hidden state and project value from it. The *put* function updates the state. Finally, the functions *evalState* and *execState* run a stateful computation, and project out the final result and the final state respectively.

In addition to these functions from the Haskell libraries, we use the following functions to modify any particular part of the *Store*:

```
modifyHeap :: (Heap → Heap) → State Store ()
modifyHeap f = do
  s ← get
  put (s {heap = f (heap s)})
modifyFresh :: (Loc → Loc) → State Store ()
modifyFresh f = do
  s ← get
  put (s {fresh = f (fresh s)})
```

Now we can begin defining the function run_s that evaluates the stateful computation described by a value of type IO_s. We begin by constructing a value of type *State Store a*, and subsequently evaluate this computation, starting with an empty store.

The *Return* case ends the stateful computation. Creating a new *IORef* involves allocating memory and extending the heap with the new data. Reading from an *IORef* looks up the data stored at the relevant location. Writing to an *IORef* updates the heap with the new data. Although we require a few auxiliary functions to manipulate the state and the heap, the code in Listing 3 should contain few surprises.

Listing 3 The semantics of mutable state

```
$run_s :: IO_s\ a \to a$
$run_s\ io = evalState\ (runIOState\ io)\ emptyStore$

$runIOState :: IO_s\ a \to State\ Store\ a$
$runIOState\ (Return\ a) = return\ a$
$runIOState\ (NewIORef\ d\ g) =$
  do $loc \leftarrow alloc$
     $extendHeap\ loc\ d$
     $runIOState\ (g\ loc)$
$runIOState\ (ReadIORef\ l\ g) =$
  do $d \leftarrow lookupHeap\ l$
     $runIOState\ (g\ d)$
$runIOState\ (WriteIORef\ l\ d\ p) =$
  do $extendHeap\ l\ d$
     $runIOState\ p$

$alloc :: State\ Store\ Loc$
$alloc =$
  do $loc \leftarrow gets\ fresh$
     $modifyFresh\ ((+)\ 1)$
     $return\ loc$

$lookupHeap :: Loc \to State\ Store\ Data$
$lookupHeap\ l =$
  do $h \leftarrow gets\ heap$
     $return\ (h\ l)$

$extendHeap :: Loc \to Data \to State\ Store\ ()$
$extendHeap\ l\ d = modifyHeap\ (update\ l\ d)$

$update :: Loc \to Data \to Heap \to Heap$
$update\ l\ d\ h\ k$
  $\mid l \equiv k$       $= d$
  $\mid otherwise = h\ k$
```

Applications Now we can actually use these functions to test our implementation of queues. We can formulate and test the following property:

$$\begin{array}{l}
\mathit{fifoProp} :: [\mathit{Int}] \to \mathit{Bool} \\
\mathit{fifoProp}\ \mathit{xs} = \mathit{map}\ \mathit{Just}\ \mathit{xs} \equiv \mathit{run}_s\ (\mathit{enqDeq}\ \mathit{xs})
\end{array}$$

And indeed, QuickCheck fails to find a counterexample for this property.

These specifications can also be used to reason about imperative programs. Bird (2001) has shown how to prove properties of well-known pointer algorithms such as the Schorr-Waite marking algorithm. The proof Bird presents revolves around having a pair of functions with the following types:

$$\begin{array}{l}
\mathit{next} :: \mathit{IORef}\ \mathit{Data} \to \mathit{IO}\ (\mathit{Maybe}\ (\mathit{IORef}\ \mathit{Data})) \\
\mathit{data} :: \mathit{IORef}\ \mathit{Data} \to \mathit{IO}\ (\mathit{Maybe}\ \mathit{Int})
\end{array}$$

Rather than postulate their existence and the properties they satisfy, we can indeed implement these functions. For example, we can define the *next* function as follows:

```
next ref = do
  x ← readIORef ref
  case x of
    NULL → return Nothing
    Cell _ nextRef → return (Just nextRef)
```

We hope that these functional specifications may provide the foundations for the high-level abstractions such as those that Bird has proposed.

Limitations We restricted ourselves to mutable variables storing the *Data* type. A more flexible approach would be to use Haskell's support for dynamic types (Cheney and Hinze, 2002; Baars and Swierstra, 2002) to allow references to different types. Concretely, this would involve the following changes:

- Replacing the occurrences of *Data* in the IO_s type with *Dynamic* and adapting the smart constructors to have the following type:

  $$\mathit{newIORef} :: \mathit{Typeable}\ a \Rightarrow a \to \mathit{IO}\ (\mathit{IORef}\ a)$$

- Representing the heap as a function *Loc* → *Dynamic*;
- Coercing to and from dynamic types explicitly when the heap is updated or read.

This does make reasoning about our programs much more difficult as the implementation of dynamic types relies on compiler specific primitives such as *unsafeCoerce*. For the sake of presentation, we therefore choose to only handle references storing a fixed *Data* type. The price we pay is, of course, having to change this type every time we wish to change the types stored in mutable references. When we move to a richer type theory in Chapter 6, we will show how to overcome this limitation partially without requiring compiler support.

Of course, Haskell already has a monad encapsulating local mutable state, the *ST* monad. The approach we sketched here, however, is not just limited to dealing with mutable references. The same techniques can be used to give specifications of many different kinds of effects.

2.5 Teletype interactions

We will now present a functional specification of simple textual interactions where a process may display text to the user or prompt the user to enter text. Such interactions may be defined using the following two primitive functions:

```
getChar :: IO Char
putChar :: Char → IO ()
```

The *getChar* function reads a character that the user enters from *stdin*; the *putChar* function prints a given character to *stdout*. Using these functions and the combinators we saw from the previous section, we can define more complex interactions:

```
echo :: IO ()
echo = do c ← getChar
          putChar c
          echo
```

The *echo* function simply echoes any character entered in *stdin* to *stdout*.

Listing 4 Teletype IO

```
data IOtt a =
    GetChar (Char → IOtt a)
  | PutChar Char (IOtt a)
  | Return a

instance Functor IOtt where
  fmap f (GetChar io)    = GetChar (fmap f ∘ io)
  fmap f (PutChar c io)  = PutChar c (fmap f io)
  fmap f (Return x)      = Return (f x)

instance Monad IOtt where
  return = Return
  (GetChar f)    ≫= g  = GetChar (λc → f c ≫= g)
  (PutChar c a)  ≫= g  = PutChar c (a ≫= g)
  (Return a)     ≫= g  = g a

getChar    :: IOtt Char
getChar    = GetChar Return

putChar    :: Char → IOtt ()
putChar c = PutChar c (Return ())
```

To provide a pure specification of *putChar* and *getChar*, we begin by defining a data type IO_{tt} that captures the syntax of the primitive interactions that can take place with the teletype in Listing 4. Besides getting and putting a single character, we can end the interaction by returning a value. Once again, we can show that the IO_{tt} type is a monad and define smart constructors for the functions it supports.

Using this data type we can *define* the *getChar* and *putChar* functions as if they were any other functions in our language. Although they will not actually print characters to the teletype, we can use them to specify any interaction.

Given a value of type $IO_{tt}\ a$, we can calculate its behaviour. What should the result of an interaction be? From a user's point of view one of three things may happen: either a value of type *a* is returned, ending the interaction; or the interaction continues after a character is read from

Listing 5 Teletype IO – semantics

```
data Output a =
    Read (Output a)
  | Print Char (Output a)
  | Finish a

data Stream a = Cons {hd :: a, tl :: Stream a}

run_tt :: IO_tt a → (Stream Char → Output a)
run_tt (Return a) cs    = Finish a
run_tt (GetChar f) cs   = Read (run_tt (f (hd cs)) (tl cs))
run_tt (PutChar c p) cs = Print c (run_tt p cs)
```

the teletype or printed to the screen. Our *Output* data type in Listing 5 captures exactly these three cases.

Once we have fixed the type of *Output*, writing the run_{tt} function that models the behaviour of a given interaction is straightforward. We assume that we have a stream of characters that have been entered by the user. Whenever our interaction gets a character, we read the head of the stream and continue the interaction with the tail.

Using the *putChar* and *getChar* functions that we have defined, we can write the same code for teletype interactions as before, but we now have a specification with which we can reason.

Example: echo Using our semantics, we can prove once and for all that *echo* prints out any character entered at the teletype. In particular, we can define the following function that exhibits the behaviour we expect *echo* to have:

```
copy :: Stream Char → Output ()
copy (Cons x xs) = Read (Print x (copy xs))
```

The *copy* function simply copies the stream of characters entered at the teletype to the stream of characters printed to the teletype one at a time. The *Read* constructor is important here: a variation of the echo function that required two characters to be typed before producing any output would not satisfy this specification. We can now prove that running *echo* will behave exactly like the *copy* function.

Using a variation of the take lemma (Bird and Wadler, 1988), we show that *copy cs* and the result of running *echo* on *cs* are identical, for every input stream *cs*. The proof requires us to define an extra *take* function, analogous to the one for lists:

```
take :: Int → Output () → Output ()
take (n + 1) (Print x xs) = Print x (take n xs)
take (n + 1) (Read xs)    = Read (take (n + 1) xs)
take 0       _            = Finish ()
```

We can now prove that:

$$take\ n\ (run_{tt}\ echo\ xs) = take\ n\ (copy\ xs)$$

The proof proceeds by induction on n. The base case is trivial; the induction step is in Listing 6.

Discussion In the specification we have given here, the *Read* constructor explicitly marks when the input stream is consumed. If the input stream is *stdin*, for example, this makes perfect sense. After all, asking the user for input is certainly an observable effect. In general, however, the handle in question may be a private communication channel between two processes. In that case, reading input is not observable. The result should be a stream of characters, rather than the *Output* data type defined above.

There is an interesting question of whether the IO_{tt} data type is inductive or coinductive. If the data type were inductive, we could only produce a finite list of output. If the data type were coinductive, we could write a program that reads its input infinitely, but never produces a value. Calling the run_{tt} function on such a program would cause it to diverge. One solution is to define IO_{tt} as a mixed inductive-coinductive type: it should only consume a finite amount of input before producing potentially infinite output. Ghani et al. (Ghani et al., 2006) have shown how such types can be used to represent continuous functions on streams.

Listing 6 The behaviour of *echo*

$$
\begin{aligned}
& \mathit{take}\ (n+1)\ (\mathit{run}_{tt}\ \mathit{echo}\ (\mathit{Cons}\ x\ \mathit{xs}))\\
\equiv\quad & \{\text{definition of } \mathit{echo}, \mathit{putChar\ and\ getChar}\}\\
& \mathit{take}\ (n+1)\ (\mathit{run}_{tt}\ (\mathit{GetChar}\ \mathit{Return}\\
& \qquad\qquad\qquad\qquad \gg\!= \lambda c \rightarrow \mathit{PutChar}\ c\ (\mathit{Return}\ ())\\
& \qquad\qquad\qquad\qquad \gg \mathit{echo})\\
& \qquad\qquad\qquad (\mathit{Cons}\ x\ \mathit{xs}))\\
\equiv\quad & \{\text{definition of } \mathit{run}_{tt}\ \mathit{and}\ (\gg\!=)\}\\
& \mathit{take}\ (n+1)\\
& \quad (\mathit{Read}\ (\mathit{run}_{tt}\ (\mathit{Return}\ x\\
& \qquad\qquad\qquad\qquad \gg\!= \lambda c \rightarrow \mathit{PutChar}\ c\ (\mathit{Return}\ ())\\
& \qquad\qquad\qquad\qquad \gg \mathit{echo})\\
& \qquad\qquad\qquad \mathit{xs}))\\
\equiv\quad & \{\text{definition of } (\gg\!=)\}\\
& \mathit{take}\ (n+1)\\
& \quad (\mathit{Read}\ (\mathit{run}_{tt}\ (\mathit{PutChar}\ x\ (\mathit{Return}\ () \gg \mathit{echo}))\ \mathit{xs}))\\
\equiv\quad & \{\text{definition of } (\gg)\}\\
& \mathit{take}\ (n+1)\ (\mathit{Read}\ (\mathit{run}_{tt}\ (\mathit{PutChar}\ x\ \mathit{echo})\ \mathit{xs}))\\
\equiv\quad & \{\text{definition of } \mathit{run}_{tt}\}\\
& \mathit{take}\ (n+1)\ (\mathit{Read}\ (\mathit{Print}\ x\ (\mathit{run}_{tt}\ \mathit{echo}\ \mathit{xs})))\\
\equiv\quad & \{\text{definition of } \mathit{take}\}\\
& \mathit{Read}\ (\mathit{Print}\ x\ (\mathit{take}\ n\ (\mathit{run}_{tt}\ \mathit{echo}\ \mathit{xs})))\\
\equiv\quad & \{\text{induction hypothesis}\}\\
& \mathit{Read}\ (\mathit{Print}\ x\ (\mathit{take}\ n\ (\mathit{copy}\ \mathit{xs})))\\
\equiv\quad & \{\text{definition of } \mathit{take}\}\\
& \mathit{take}\ (n+1)\ (\mathit{Read}\ (\mathit{Print}\ x\ (\mathit{copy}\ \mathit{xs})))\\
\equiv\quad & \{\text{definition of } \mathit{copy}\}\\
& \mathit{take}\ (n+1)\ (\mathit{copy}\ (\mathit{Cons}\ x\ \mathit{xs}))
\end{aligned}
$$

2.6 Concurrency

Although the models for mutable state and teletype interactions were relatively straightforward, concurrency poses a more challenging problem. Concurrent Haskell (Peyton Jones et al., 1996) enables programmers to fork off a new thread with the *forkIO* function:

$$forkIO :: IO\ a \rightarrow IO\ ThreadId$$

The new thread that is forked off will be concurrently perform the computation passed as an argument to the *forkIO* function. The programmer can subsequently use the *ThreadId* returned by the *forkIO* function to kill a thread or throw an exception to a specific thread.

Threads can communicate with one another using a synchronised version of an *IORef* called an *MVar*. As with an *IORef* there are three functions to create, write to and read from an *MVar*:

```
newEmptyMVar :: IO (MVar a)
putMVar      :: MVar a → a → IO ()
takeMVar     :: MVar a → IO a
```

Unlike an *IORef*, an *MVar* can be empty. Initially, there is no value stored in an *MVar*. An empty *MVar* can be filled using the function *putMVar*. A filled *MVar* can be emptied using the function *takeMVar*. If a thread tries to fill a non-empty *MVar*, the thread is blocked until another thread empties the *MVar* using *takeMVar*. Dually, when a thread tries to take a value from an empty *MVar*, the thread is blocked until another thread puts a value into the *MVar*.

Although there are several other functions in Haskell's concurrency library, we choose to restrict ourselves to the four functions described above for the moment.

In what should now be a familiar pattern, we begin by defining the data type IO_c for concurrent input/output in Listing 7. Once again, we add a constructor for every primitive function together with an additional *Return* constructor. As is the case in our *IORef* implementation, we model memory addresses and the data stored there as integers. Forked off threads have a unique identifier, or *ThreadId*, which we also model as an integer. The type of *Fork* is interesting as it will take an $IO_c\ b$ as its first argument, regardless of what b is. This corresponds to the parametric polymorphism that the *forkIO* function exhibits – it will fork off a new thread, regardless of the value that the new thread returns.

Once we have defined the data type IO_c, we can show it is a monad just in the same fashion that IO_s and IO_{tt} are monads. We continue by defining the basic functions, corresponding to the constructors.

Running the computation described by a value of type $IO_c\ a$ is not as straightforward as the other models we have seen so far. Our model of concurrency revolves around a scheduler that determines which thread is entitled to run. The *Scheduler* is a function that, given an integer n, returns a number between 0 and $n - 1$, together with a new scheduler. Intuitively, we inform the scheduler how many threads are active and it returns the scheduled thread and a new scheduler. Listing 8 describes how initially to set up the semantics of our concurrency operations.

Besides the scheduler, we also need to keep track of the threads that could potentially be running. The thread *soup* is a finite map taking a *ThreadId* to a *ThreadStatus*. Typically, such a *ThreadStatus* consists of the process associated with a given *ThreadId*. Note, however, that once a thread is finished, there is no value of IO_c that we could associate with its *ThreadId* so we have an additional *Finished* constructor to deal with this situation. Besides the thread soup we also store an integer, *nextTid*, that represents the next unassigned *ThreadId*.

In addition to information required to deal with concurrency, we also need a considerable amount of machinery to cope with mutable state. In particular, we keep track of a *heap* and *fresh* just as we did for our model of mutable state. Unlike an *IORef*, an *MVar* can be empty; hence the *heap* maps locations to *Maybe Data*, using *Nothing* to represent an empty *MVar*. All these ingredients together form the *Store*.

To interpret a value of type $IO_c\ a$, we define a function that will run the concurrent process that it represents. Once again, we use Haskell's state monad to encapsulate the implicit plumbing involved with passing around the *Store*. To run a concurrent process we must tackle two more or less separate issues: how to perform a single step of computation and how to interleave these individual steps. We will begin defining the single steps in Listing 9, leaving the *interleave* function undefined for the moment.

The *step* function closely resembles our semantics for mutable variables,

Listing 7 Concurrency – data type

```
type ThreadId = Int
type Data     = Int
type Loc      = Int

data IO_c a =
     NewEmptyMVar (Loc → IO_c a)
   | TakeMVar Loc (Data → IO_c a)
   | PutMVar Loc Data (IO_c a)
   | ∀ b . Fork (IO_c b) (ThreadId → IO_c a)
   | Return a

newtype MVar = MVar Loc

instance Monad IO_c where
  return = Return
  Return x ≫= g         = g x
  NewEmptyMVar f ≫= g   = NewEmptyMVar (λl → f l ≫= g)
  TakeMVar l f ≫= g     = TakeMVar l (λd → f d ≫= g)
  PutMVar c d f ≫= g    = PutMVar c d (f ≫= g)
  Fork p1 p2 ≫= g       = Fork p1 (λtid → p2 tid ≫= g)

newEmptyMVar        :: IO_c MVar
newEmptyMVar        = NewEmptyMVar (Return ∘ MVar)

takeMVar            :: MVar → IO_c Data
takeMVar (MVar l)   = TakeMVar l Return

putMVar             :: MVar → Data → IO_c ()
putMVar (MVar l) d  = PutMVar l d (Return ())

forkIO              :: IO_c a → IO_c ThreadId
forkIO p            = Fork p Return
```

Listing 8 Concurrency – initialisation

```
newtype Scheduler =
  Scheduler (Int → (Int, Scheduler))
data ThreadStatus =
    ∀ b . Running (IO_c b)
  | Finished
data Store = Store {fresh :: Loc
                   , heap :: Loc → Maybe Data
                   , nextTid :: ThreadId
                   , soup :: ThreadId → ThreadStatus
                   , scheduler :: Scheduler
                   }
initStore :: Scheduler → Store
initStore s  = Store {fresh     = 0
                     , nextTid = 1
                     , scheduler = s
                     }
runIO_c :: IO_c a → (Scheduler → a)
runIO_c io s = evalState (interleave io) (initStore s)
```

with a few minor adjustments. Listing 10 contains a few auxiliary definitions. In contrast to the situation for mutable variables, performing a step may result in three different outcomes.

First of all, a thread might terminate and produce a result. Secondly, a thread might have a side-effect, such as taking the value stored in an *MVar*, and return a new, shorter process. Finally, a thread might be blocked, for instance when it tries to take a value from an empty *MVar*. These three cases together form the *Status* data type that is returned by the *step* function.

Note that we have omitted a few functions that modify a specific part of the state, analogous to *modifyFresh* and *modifyHeap* in Listing 3.

There are a few differences with the model of mutable state. When we return a value, the thread is finished and we wrap our result in a *Stop* constructor. Creating a new *MVar* is almost identical to creating a new *IORef*.

Listing 9 Concurrency – performing a single step

```
data Status a = Stop a | Step (IO_c a) | Blocked

step :: IO_c a → State Store (Status a)
step (Return a) = return (Stop a)
step (NewEmptyMVar f) =
  do loc ← alloc
     modifyHeap (update loc Nothing)
     return (Step (f loc))
step (TakeMVar l f) =
  do var ← lookupHeap l
     case var of
       Nothing → return Blocked
       (Just d) → do emptyMVar l
                     return (Step (f d))
step (PutMVar l d p) =
  do var ← lookupHeap l
     case var of
       Nothing → do fillMVar l d
                    return (Step p)
       (Just d) → return Blocked
step (Fork l r) =
  do tid ← freshThreadId
     extendSoup l tid
     return (Step (r tid))
```

Listing 10 Concurrency – auxiliary definitions

```
lookupHeap :: Loc → State Store (Maybe Data)
lookupHeap l = do h ← gets heap
                  return (h l)
freshThreadId :: State Store ThreadId
freshThreadId = do tid ← gets nextTid
                   modifyTid ((+) 1)
                   return tid
emptyMVar :: Loc → State Store ()
emptyMVar l = modifyHeap (update l Nothing)
fillMVar :: Loc → Data → State Store ()
fillMVar l d = modifyHeap (update l (Just d))
extendSoup :: IO_c a → ThreadId → State Store ()
extendSoup p tid = modifySoup (update tid (Running p))
```

The only difference is that an *MVar* is initially empty, so we extend the heap with *Nothing* at the appropriate location.

The case for *TakeMVar* and *PutMVar* is more interesting. When we read an *MVar* we look up the appropriate information in the heap. If the *MVar* is filled, we empty it and perform a single step. When the *MVar* is empty, the thread is blocked and we cannot make any progress. The situation for writing to an *MVar* is dual.

The final case of the *step* function deals with forking off new threads. We begin by generating a *ThreadId* for the newly created thread. Subsequently, we extend the thread soup with the new thread. Finally, we return the parent thread wrapped in the *Step* constructor as the thread has made progress, but is not yet finished.

Although it was relatively easy to perform a single step, the interleaving of separate threads is more involved. Listing 11 finally defines the *interleave* function. Different threads may return different types. In particular the main thread has type IO_c *a*, but auxiliary threads have type IO_c *b* for some unknown type *b*. To make this distinction, we introduce the *Process* data type.

Listing 11 Concurrency – interleaving

```
data Process a = Main (IO_c a)
  | ∀ b . Aux (IO_c b)

interleave :: IO_c a → State Store a
interleave main =
  do (tid, t) ← schedule main
     case t of
       Main p → do x ← step p
                   case x of
                     Stop r    → return r
                     Step p    → interleave p
                     Blocked   → interleave main
       Aux p → do x ← step p
                  case x of
                    Stop _  → finishThread tid ≫ interleave main
                    Step q  → extendSoup q tid ≫ interleave main
                    Blocked → interleave main

finishThread tid = modifySoup (update tid Finished)

schedule :: IO_c a → State Store (ThreadId, Process a)
schedule main = do tid ← getNextThreadId
                   if tid ≡ 0
                     then return (0, Main main)
                     else do tsoup ← gets soup
                             case tsoup tid of
                               Finished → schedule main
                               Running p → return (tid, Aux p)

getNextThreadId :: State Store ThreadId
getNextThreadId = do Scheduler sch ← gets scheduler
                     n ← gets nextTid
                     let (tid, s) = sch n
                     modifyScheduler (const s)
                     return tid
```

Essentially, to interleave a concurrent process we begin by consulting the scheduler to determine the next active thread. Initially, this will always be the main process. Once the main process forks off child threads, however, such threads may be scheduled instead.

The *schedule* function consults the scheduler for the next *ThreadId*, and returns that *ThreadId* and the process associated with it. We need to pass the main process to the scheduler as it is not in the thread soup, but could still be scheduled. The result of scheduling is a value of type *Process a* together with the *ThreadId* of the thread that has been scheduled.

If we want to use the *Process* returned by the scheduler, we need to be careful. We would like to allow the scheduled process to perform a single step – but what should we do with the result? If the main thread returns a final value, we can wrap things up and return that value. If an auxiliary thread returns a value, we are not particularly interested in its result, but rather want to terminate the thread. As we want to treat the main and auxiliary threads differently, we need to pattern match on the scheduled process.

Regardless of which thread was scheduled, we allow it to perform a single step. There are five possible outcomes of this step, that we cover one by one:

The main thread stops When the main thread terminates, the entire concurrent process is finished. We simply return the value that the step produced. Any auxiliary threads that are still active will never be scheduled.

An auxiliary thread stops If an auxiliary thread finished its computation and returns a value, we discard this value and finish the thread. We update the thread soup to indicate that this thread is finished and continue the interleaving.

The main threads performs a step When the main thread manages to successfully perform a single step, we continue by calling the *interleave* function again. The argument we pass to the *interleave* function is the new main process that was wrapped in a *Step* constructor.

An auxiliary thread performs a step When an auxiliary thread makes progress, we proceed much in the same way as we do for the main

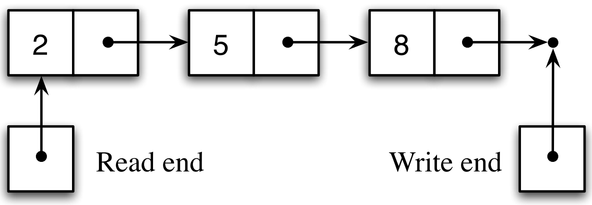

Figure 2.2: An example channel

thread. Instead of passing the new computation to *interleave*, however, we update the thread soup. Once the soup has been updated, we continue by interleaving with the same main thread as we started with.

Blocked If the scheduled thread can make no progress, for instance because it is waiting for an empty *MVar* to be filled, scheduling that thread will return *Blocked*. In that case, we schedule a new thread, until progress is made.

If all threads are blocked, the program reaches a deadlock and the specification diverges.

The semantics for concurrency are more complicated than those for teletype IO and mutable state. Actually using it to test concurrent programs, however, is no more difficult.

Example: channels

When Peyton Jones describes the semantics of concurrency in Haskell (Peyton Jones, 2001), he illustrates how to program with *MVars* by giving an implementation of channels.

Channels enable separate threads to communicate safely. They generalise the queues we have seen previously, as a channel allows multiple processes to read from and write to it. This is accomplished by having a pair of *MVars* storing pointers to the read end and write end of the channel. Whenever a process wants to read from or write to the channel, it must first acquire access to the appropriate end of the queue. Storing these pointers in *MVars* ensures that separate writes or reads can never interfere with one another. One example of a channel is illustrated in Figure 2.2.

Peyton Jones claims that:

Listing 12 Channels

```
type Channel = (MVar, MVar)
data Data =
    Cell Int MVar
  | Ref MVar
  | Res [Int]
newChan :: IO_c Channel
putChan :: Channel → Int → IO_c ()
getChan :: Channel → IO_c Int
```

> . . . each value read will go to exactly one process.

Unfortunately, there is no justification of this claim. Proving such statements can, of course, be rather difficult. In fact, it is already hard to formulate precisely what it means for data to be lost or duplicated.

Listing 12 gives the types of channels and the data stored by references, together with the type signatures of the channel operations. We refer to Peyton Jones's implementation (Peyton Jones, 2001) for a more thorough description of the implementation, and focus here on how to use QuickCheck to demonstrate that certain properties are at least plausible.

Before we can implement the channel operations, we need to fix the data type *Data*, i.e. the type of data stored in an *MVar*. As we can see from Figure 2.2, the data stored in an *MVar* is not always a cell. In particular, the references to the read end and write end of the channel are also stored in an *MVar*. Therefore, we need to add an extra constructor *Ref* to our *Data* data type. Finally, we will later use an *MVar* to store a list of integers in the test we propose to run; therefore, we add a final constructor *Res*.

Listing 13 shows the test we would like to run. The *chanTest* function takes a list of integers, and forks off a thread for each integer that will write that integer to an initially empty channel. It also forks off a thread for each integer that attempts to read from the channel. Once a thread manages to read from the channel, it records the value read in a shared *MVar* called *result*. The main thread then waits until every thread has successfully read from the channel, and concludes by returning the list of all values that have

Listing 13 Testing the implementation of channels

```
chanTest :: [Int] → IO_c [Int]
chanTest ints
 = do ch ← newChan
      result ← newEmptyMVar
      putMVar result (Res [ ])
      forM ints (λi → forkIO (putChan ch i))
      replicateM (length ints) (forkIO (reader ch result))
      wait result ints
reader :: Channel → MVar → IO_c ()
reader channel var
   = do x ← getChan channel
        (Res xs) ← takeMVar var
        putMVar var (Res (x : xs))
wait :: MVar → [Int] → IO_c [Int]
wait var xs
 = do (Res r) ← takeMVar var
      if length r ≡ length xs
        then return r
        else do putMVar var (Res r)
                wait var xs
```

been read. This final result should, of course, be a permutation of our original list.

The semantics of concurrency we have presented abstracts over the scheduling algorithm. Before we can run the test we have in mind, we must therefore decide what scheduler to use. As we are already using QuickCheck, we implement a random scheduling algorithm in an attempt to maximise the number of interleavings. Listing 14 gives one possible implementation of such a scheduler.

The *streamSch* function defines a scheduler, given a stream of integers. The definition of the *Stream* data type can be found in Listing 5. Whenever it is asked to schedule a thread, it uses the appropriate modulus on the

Listing 14 Random scheduling

```
streamSch :: Stream Int → Scheduler
streamSch xs =
  Scheduler (λk → (hd xs ‘mod‘ k, streamSch (tl xs)))
instance Arbitrary a ⇒ Arbitrary (Stream a) where
  arbitrary = do x ← arbitrary
                 xs ← arbitrary
                 return (Cons x xs)
```

head of the stream and continues scheduling with its tail. As we can use QuickCheck to generate a random stream of integers, we use the *streamSch* to produce a random scheduler.

The following property should hold:

```
chanProp ints stream =
  sort (runIO_c (chanTest ints) (streamSch stream))
    ≡ sort ints
```

Once again, QuickCheck informs us that the above property holds for 100 test runs. When we classify the input lists according to their length, it is reassuring to see that this property even holds for lists of more than 90 elements – which corresponds to 200 randomly scheduled pseudothreads vying for access to a single channel.

Clearly, this property is insufficient to verify Peyton Jones’s claim. We should also check that the resulting channel is empty and all the threads are finished. Even then, we have only checked one kind of scenario, where every thread either writes or reads a single value. Yet our semantics are capable of providing *some* form of sanity check. It is not clear how such a check could be realised using Peyton Jones’s semantics.

It may not be a surprise that the implementation of channels using *MVars* is correct. Running this test, however, found a very subtle bug in our scheduling function. Recall that the *schedule* function returns the *ThreadId* and process of the scheduled thread. If we schedule a finished thread, we call the *schedule* function again, in search of a thread that is not yet finished.

In a faulty version of our specification, if we encountered a finished thread, we called the schedule function again, but returned the *ThreadId* of the finished thread. This caused quite some chaos in the thread soup, as threads were lost and duplicated.

As the entire state of concurrent computations is a pure value, we can access otherwise inaccessible data, such as the size of the heap or the number of threads that have finished. In particular, abstracting over the scheduler allows us to check certain algorithms with specific schedulers or check a large number of interleavings using a random scheduler as we see fit.

2.7 Discussion

Related work Similar functional specifications to those presented have appeared previously. Reynolds (1998), for example, gives an untyped version of the functional specification of mutable state. Gordon has described teletype IO in his thesis (Gordon, 1994) and cites related work dating back more than twenty years (Holmström, 1983; Karlsson, 1981). Claessen (1999) has presented an alternative pure specification of concurrency, based on continuations. Our specification, explicitly parametrised by a scheduler, makes it easier to experiment with different scheduling strategies.

Further work We have not given functional specifications of the entire 'Awkward Squad.' There are several functions from Haskell's concurrency libraries that we have not covered. It should not be too difficult to do so – we do not need to extend the code that deals with the interleaving and scheduling, but can restrict ourselves to adapting the IO_c data type and the *step* function. For instance, it is fairly straightforward to give a specification of functions such as:

$$\begin{aligned}&\mathit{killThread} :: \mathit{ThreadId} \to \mathit{IO}\ ()\\&\mathit{yield} :: \mathit{IO}\ ()\end{aligned}$$

The *killThread* function removes a certain thread from the thread soup; the *yield* function passes control to some other thread.

We have chosen to define a separate data type to represent the syntactical terms, before assigning these terms semantics. Sometimes it may be easier to define the semantics directly. For example, we could have the *newIORef* constructor return a computation in the state monad, rather than the IO_s data type. This extra level of indirection, however, is crucial when combining different effects, as we shall see shortly.

This work presented in this chapter may be improved in several different ways:

Modularity We have treated the specifications of mutable state, teletype interactions, and concurrency in isolation. The next chapter will describe how to structure functional specifications modularly.

Totality The specifications we have given are not total: the initial heap is undefined; the conversion to and from Haskell's *Dynamic* types may fail; the *interleave* function may diverge if all threads are blocked. We will address most of these points in Chapter 6, where we present a total functional specification of mutable state.

Efficiency The specifications we have presented in this chapter have been designed to be as simple as possible. We could drastically improve the efficiency of our specifications, for example, by using (balanced) binary trees to model the heap. While this may make it more difficult to reason with our specifications, it makes viable the automatic testing of large programs that require many, many threads and references. Furthermore, we have chosen to distinguish between the syntactic data types and the monadic specification. While this may introduce substantial overhead (Voigtländer, 2008), it forms the basis of our approach to modular specifications discussed in the next chapter.

Correctness In this chapter we have given functional specifications. It is up to compiler implementors to prove that an implementation satisfies these specifications. Verifying a complete compiler is no easy task(Leroy, 2006). Instead of completely verifying a compiler, we could already gain a greater amount of faith in our specifications by proving them equivalent to the semantics set forth by Peyton Jones (2001).

Chapter 3

Structuring Functional Specifications

In the previous chapter we presented the functional specification of different effects in isolation. Many programs however, do not use a single kind of effect but some mix of mutable state, interactions, and concurrency. To allow such larger programs to be written, we need to investigate how functional specifications may be combined.

Based on previous work on defining Haskell data types and functions in a modular fashion (Swierstra, 2008), we show how to structure functional specifications.

3.1 The Expression Problem

Implementing an evaluator for simple arithmetic expressions in Haskell is entirely straightforward.

```
data Expr = Val Int
    | Add Expr Expr
eval :: Expr → Int
eval (Val x)     = x
eval (Add x y) = eval x + eval y
```

Once we have chosen the data type of our expression language, we are free to define new functions over the *Expr* data type. For instance, we might want to define the following function to render an expression as a string:

```
render :: Expr → String
render (Val x)     = show x
render (Add x y) = "(" ++ render x ++ " + " ++ render y ++ ")"
```

If we want to add new operators to our expression language, such as multiplication, we run into trouble. While we could extend our data type for expressions, this will require additional cases for the functions we have defined so far. Wadler (1998) has dubbed this the *Expression Problem*:

> The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

As the above example illustrates, Haskell can cope quite nicely with new function definitions; adding new constructors, however, forces us to modify existing code.

In what follows we will outline an alternative, yet equivalent, definition of the *Expr* data type that supports both the addition of new constructors and the definition of new functions over expressions.

So what should the data type for expressions be? If we fix the constructors in advance, we will run into the same problems as before. Rather than choose any particular constructors, we parameterise the expression data type as follows:

```
data Expr f = In (f (Expr f))
```

One can think of the type parameter *f* as the *signature* of the constructors. Intuitively, the type constructor *f* takes a type parameter corresponding to the expressions that occur as the subtrees of constructors. The *Expr* data type then ties the recursive knot, replacing the argument of *f* with *Expr f*.

The *Expr* data type is best understood by studying some examples. For instance, if we wanted expressions that only consisted of integers, we could write:

```
data Val e = Val Int
type IntExpr = Expr Val
```

The only valid expressions would then have the form *In* (*Val x*) for some integer *x*. The *Val* data type does not use its type parameter *e*, as the constructor does not have any expressions as subtrees.

Similarly, we might be interested in expressions consisting only of addition:

```
data Add e = Add e e
type AddExpr = Expr Add
```

In contrast to the *Val* constructor, the *Add* constructor does use its type parameter. Addition is a binary operation; correspondingly, the *Add* constructor takes two arguments of type *e*. Values of type *AddExpr* must be of the form *In* (*Add x y*), where *x* and *y* are themselves of type *AddExpr*. As there is no base case, such values must be infinitely deep binary trees.

Neither values nor addition are particularly interesting in isolation. The big challenge, of course, is to combine the *ValExpr* and *AddExpr* types somehow.

The key idea is to combine expressions by taking the coproduct of their signatures.

The coproduct of two signatures is straightforward to define in Haskell. It is very similar to the *Either* data type; the only difference is that it does not combine two *base types*, but two *type constructors*.

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

An expression of type *Expr* (*Val* :+: *Add*) is either a value or the sum of two such expressions; it is isomorphic to the original *Expr* data type in the introduction.

Combining data types using the coproduct of their signatures comes at a price. It becomes much more cumbersome to write expressions. Even a simple addition of two numbers becomes an unwieldy jumble of constructors:

```
addExample :: Expr (Val :+: Add)
addExample = In (Inr (Add (In (Inl (Val 118))) (In (Inl (Val 1219)))))
```

Obviously, writing such expressions by hand is simply not an option for anything other than small examples. Furthermore, if we choose to extend our expression language even further by constructing larger coproducts, we will need to update any values we have written: the injections *Inl* and *Inr* may no longer be the right injection into the coproduct. Before we deal with these problems, however, we consider the more pressing issue of how to evaluate such expressions.

3.1.1 Evaluation

The first observation we make, is that the types we defined to form the signatures of an *Expr* are both functors.

```
instance Functor Val where
  fmap f (Val x) = Val x
instance Functor Add where
  fmap f (Add e1 e2) = Add (f e1) (f e2)
```

Furthermore, the coproduct of two functors, is itself a functor.

```
instance (Functor g, Functor h) ⇒ Functor (g :+: h) where
  fmap f (Inl e1) = Inl (fmap f e1)
  fmap f (Inr e2) = Inr (fmap f e2)
```

These are crucial observations. If *f* is a functor, we can fold over any value of type *Expr f* as follows:

```
foldExpr :: Functor f ⇒ (f a → a) → Expr f → a
foldExpr f (In t) = f (fmap (foldExpr f) t)
```

The first argument of the fold is called an *algebra*. An algebra of type $f\ a \to a$ determines how the different constructors of a data type affect the final outcome: it specifies one step of recursion, turning a value of type $f\ a$ into the desired result a. The fold itself uniformly applies these operations to an entire expression. This definition of the generic fold over any data type arising as a fixed-point of a functor can be traced back to Meijer et al. (1991).

Using Haskell's type class system, we can define and assemble algebras in a modular fashion. We begin by introducing a separate class corresponding to the algebra we aim to define.

```
class Functor f ⇒ Eval f where
  evalAlgebra :: f Int → Int
```

The result of evaluation should be an integer; this is reflected in our choice of algebra. As we want to evaluate expressions consisting of values and addition, we need to define the following two instances:

```
instance Eval Val where
  evalAlgebra (Val x) = x
instance Eval Add where
  evalAlgebra (Add x y) = x + y
```

These instances correspond exactly to the cases from our original definition of evaluation in the introduction. In the case for addition, the variables *x* and *y* are not expressions, but the result of a recursive call.

Last of all, we also need to evaluate composite functors built from coproducts. Defining an algebra for the coproduct *f* :+: *g* boils down to defining an algebra for the individual functors *f* and *g*.

```
instance (Eval f, Eval g) ⇒ Eval (f :+: g) where
  evalAlgebra (Inl x) = evalAlgebra x
  evalAlgebra (Inr y) = evalAlgebra y
```

With all these ingredients in place, we can finally define evaluation by folding over an expression with the algebra we have defined above.

```
eval :: Eval f ⇒ Expr f → Int
eval expr = foldExpr evalAlgebra expr
```

Using *eval* we can indeed evaluate simple expressions.

```
Main⟩ eval addExample
1337
```

Although we can now define functions over expressions using folds, actually writing expressions such as *addExample*, is still rather impractical to say the least. Fortunately, we can automate most of the overhead introduced by coproducts.

3.1.2 Automating injections

The definition of *addExample* illustrates how messy expressions can easily become. In this section, we remedy the situation by introducing smart constructors for addition and values.

As a first attempt, we might try writing:

```
val :: Int → Expr Val
val x = In (Val x)
(⊕) :: Expr Add → Expr Add → Expr Add
x ⊕ y = In (Add x y)
```

While this is certainly a step in the right direction, writing *val* 1 ⊕ *val* 3 will result in a type error. The smart constructor ⊕ expects two expressions that must themselves solely consist of additions, rather than values.

We need our smart constructors to be more general. We will define smart constructors with the following types:

```
(⊕) :: (Add :≺: f) ⇒ Expr f → Expr f → Expr f
val :: (Val :≺: f) ⇒ Int → Expr f
```

You may want to read the type constraint *Add* :≺: *f* as 'any signature *f* that supports addition.'

The constraint *sub* :≺: *sup* should only be satisfied if there is some injection from *sub a* to *sup a*. Rather than write the injections using *Inr* and *Inl* by hand, the injections will be inferred using this type class.

```
class (Functor sub, Functor sup) ⇒ sub :≺: sup where
  inj :: sub a → sup a
```

The (:≺:) class only has three instances. These instances are not valid according to the Haskell 98 standard (Peyton Jones, 2003), as there is some overlap between the second and third instance definition. Later on, we will see why this should not result in any unexpected behaviour.

```
instance Functor f ⇒ f :≺: f where
    inj = id
instance (Functor f, Functor g) ⇒ f :≺: (f :+: g) where
    inj = Inl
instance (Functor f, Functor g, Functor h, f :≺: g) ⇒
  f :≺: (h :+: g) where
    inj = Inr ∘ inj
```

The first instance states that (:≺:) is reflexive. The second instance explains how to inject any value of type *f a* to a value of type (*f* :+: *g*) *a*, regardless of g. The third instance asserts that provided we can inject a value of type $f\ a$ into one of type $g\ a$, we can also inject $f\ a$ into a larger type $(h :+: g)\ a$ by composing the first injection with an additional *Inr*.

We use coproducts in a list-like fashion: the third instance only searches through the right-hand side of coproduct. Although this simplifies the search and we never perform any backtracking, it may fail to find an injection, even if one does exists. For example, the following constraint will not be satisfied:

$$f :\prec: ((f :+: g) :+: h)$$

Yet clearly $\mathit{Inl} \circ \mathit{Inl}$ would be a suitable candidate injection. Users should never encounter these limitations, provided their coproducts are not explicitly nested. By declaring the type constructor $(:+:)$ to associate to the right, types such as $f :+: g :+: h$ are parsed in a suitable fashion.

Using this type class, we define our smart constructors as follows:

$$\begin{aligned}
&\mathit{inject} :: (g :\prec: f) \Rightarrow g\ (\mathit{Expr}\ f) \rightarrow \mathit{Expr}\ f\\
&\mathit{inject} = \mathit{In} \circ \mathit{inj}\\
&\mathit{val} :: (\mathit{Val} :\prec: f) \Rightarrow \mathit{Int} \rightarrow \mathit{Expr}\ f\\
&\mathit{val}\ x\ = \mathit{inject}\ (\mathit{Val}\ x)\\
&(\oplus) :: (\mathit{Add} :\prec: f) \Rightarrow \mathit{Expr}\ f \rightarrow \mathit{Expr}\ f \rightarrow \mathit{Expr}\ f\\
&x \oplus y = \mathit{inject}\ (\mathit{Add}\ x\ y)
\end{aligned}$$

Now we can easily construct and evaluate expressions:

```
Main⟩ let x :: Expr (Add :+: Val) = val 30000 ⊕ val 1330 ⊕ val 7
Main⟩ eval x
31337
```

The type signature of x is very important. We exploit the type signature to figure out the injection into a coproduct: if we fail to provide the type signature, a compiler cannot infer the right injection.

As we mentioned previously, there is some overlap between the instances of the $(:\prec:)$ class. Consider the following example:

$$\begin{aligned}
&\mathit{inVal} :: \mathit{Int} \rightarrow \mathit{Expr}\ (\mathit{Val} :+: \mathit{Val})\\
&\mathit{inVal}\ i = \mathit{inject}\ (\mathit{Val}\ i)
\end{aligned}$$

Which injection should be inferred, *Inl* or *Inr*? There is no reason to prefer one over the other—both choices are justified by the above instance definitions. The functions we present here, however, do not inspect *where* something occurs in a coproduct. Indeed, we can readily check that calling *eval* on $(In\ (Inl\ (Val\ x)))$ and $(In\ (Inr\ (Val\ x)))$ yields the same result for all integers x as the instance of the *Eval* class for coproducts does not distinguish between *Inl* and *Inr*. In other words, the result of *eval* will never depend on the choice of injection. Although we need to allow overlapping instances to compile this class, it should only result in unpredictable behaviour if you abuse the information you have about the order of the constructors of an expression.

3.1.3 Examples

So far we have done quite some work to write code equivalent to the evaluation function defined in introduction. It is now time to reap the rewards of our investment. How much effort is it to add multiplication to our little expression language? We begin by defining a new type and its corresponding functor instance.

```
data Mul x = Mul x x
instance Functor Mul where
  fmap f (Mul x y) = Mul (f x) (f y)
```

Next, we define how to evaluate multiplication and add a smart constructor.

```
instance Eval Mul where
  evalAlgebra (Mul x y) = x ∗ y
infixl 7 ⊗
(⊗) :: (Mul :≺: f) ⇒ Expr f → Expr f → Expr f
x ⊗ y = inject (Mul x y)
```

With these pieces in place, we can evaluate expressions:

```
Main⟩ let x :: Expr (Val :+: Add :+: Mul) = val 80 ⊗ val 5 ⊕ val 4
Main⟩ eval x
404
```

```
Main⟩ let y :: Expr (Val :+: Mul) = val 6 ⊗ val 7
Main⟩ eval y
42
```

As the second example illustrates, we can also write and evaluate expressions of type *Expr* (*Val* :+: *Mul*), thereby leaving out addition. In fact, once we have a menu of expression building blocks, we can assemble our own data types *à la carte*. This is not even possible with proposed language extensions for open data types (Löh and Hinze, 2006).

Adding new functions is not much more difficult. As a second example, we show how to render an expression as a string. Instead of writing this as a fold, we give an example of how to write open-ended functions using recursion directly.

We begin by introducing a class, corresponding to the function we want to write. An obvious candidate for this class is:

```
class Render f where
  render :: f (Expr f) → String
```

The type of *render*, however, is not general enough. To see this, consider the instance definition for *Add*. We would like to make recursive calls to the subtrees, which themselves might be values, for instance. The above type for *render*, however, requires that all subtrees of *Add* are themselves additions. Clearly this is undesirable. A better choice for the type of render is:

```
class Render f where
  render :: Render g ⇒ f (Expr g) → String
```

This more general type allows us to make recursive calls to any subexpressions of an addition, even if these subexpressions are not additions themselves.

Assuming we have defined instances of the *Render* class, we can write a function that calls *render* to pretty print an expression.

```
pretty :: Render f ⇒ Expr f → String
pretty (In t) = render t
```

All that remains, is to define the desired instances of the *Render* class. These instances closely resemble the original *render* function defined in the introduction; there should be no surprises here.

```
instance Render Val where
  render (Val i) = show i
instance Render Add where
  render (Add x y) = "(" ++ pretty x ++ " + " ++ pretty y ++ ")"
instance Render Mul where
  render (Mul x y) = "(" ++ pretty x ++ " * " ++ pretty y ++ ")"
instance (Render f, Render g) => Render (f :+: g) where
  render (Inl x) = render x
  render (Inr y) = render y
```

Sure enough, we can now pretty-print our expressions:

```
Main⟩ let x :: Expr (Val :+: Add :+: Mul) = val 80 ⊗ val 5 ⊕ val 4
Main⟩ pretty x
"((80 * 5) + 4)"
```

3.2 Structuring syntax

In the previous section we showed how to combine simple Haskell data types. How can we apply these techniques to combine our functional specifications?

In general, the coproduct of two monads is fairly complicated (Lüth and Ghani, 2002). Fortunately, all our syntactical data types from the previous chapter have the same shape: a constructor for every effectful operation together with a single *Return* constructor. Each of these data types can be defined as a particular instance of the following *Term* data type:

```
data Term f a =
    Pure a
  | Impure (f (Term f a))
```

The type argument *f* abstracts over the constructors corresponding to the effectful operations, just as the *Expr* data type in the previous section. These terms consist of either pure values or an impure effect, constructed using *f*.

We can express all the data types representing the syntax of effectful computations using the *Term* data type. For example, we could define the syntax of computations using mutable state as follows:

```
data IOₛ t = NewIORef Dynamic (Loc → t)
    | ReadIORef Loc (Dynamic → t)
    | WriteIORef Dynamic Loc t
```

Note that we have replaced the *Data* type with Haskell's built-in *Dynamic* type. We discussed ramifications of this choice in Section 2.4.

When *f* is a functor, *Term f* is a monad. This is illustrated by the following two instance definitions.

```
instance Functor f ⇒ Functor (Term f) where
  fmap f (Pure x)   = Pure (f x)
  fmap f (Impure t) = Impure (fmap (fmap f) t)
instance Functor f ⇒ Monad (Term f) where
  return x          = Pure x
  (Pure x) ≫= f     = f x
  (Impure t) ≫= f   = Impure (fmap (≫= f) t)
```

These monads are known as *free monads* (Awodey, 2006). In general, a structure is called *free* when it is left-adjoint to a forgetful functor. In this specific instance, the *Term* data type is a higher-order functor that maps a functor *f* to the monad *Term f*; this is illustrated by the above two instance definitions. This *Term* functor is left-adjoint to the forgetful functor from monads to their underlying functors.

All left-adjoint functors preserve coproducts. In particular, computing the coproduct of two free monads reduces to computing the coproduct of their underlying functors, which is exactly what we achieved in the previous section.

Just as we did previously, we can define smart constructors for the different effectful operations. These smart constructors in Listing 15 use the *inject* function to build terms, inserting the necessary injections automatically.

Listing 15 uses the following pair of functions to convert to and from dynamic types:

```
fromDynamic :: Typeable a ⇒ Dynamic → Maybe a
toDyn       :: Typeable a ⇒ a → Dynamic
```

The *fromJust* function in the definition of *readIORef* may fail. However, the phantom types carried in the *IORef* data type, guarantees that the coercion

Listing 15 Smart constructors for mutable state

```
data IORef a = IORef Loc
inject :: (g :≺: f) ⇒ g (Term f a) → Term f a
inject = Impure ∘ inj
newIORef :: (IO_s :≺: f, Typeable a) ⇒ a → Term f (IORef a)
newIORef d =
  inject (NewIORef (toDyn d) (Pure ∘ IORef))
readIORef :: (IO_s :≺: f, Typeable a) ⇒ IORef a → Term f a
readIORef (IORef l) =
  inject (ReadIORef l (Pure ∘ fromJust ∘ fromDynamic))
writeIORef :: (IO_s :≺: f, Typeable a) ⇒ a → IORef a → Term f ()
writeIORef d (IORef l) =
  inject (WriteIORef (toDyn d) l (Pure ()))
```

will succeed, provided the user only uses the smart constructors to write effectful programs.

Of course we can define similar smart constructors for other specifications from the previous chapter. Using these smart constructors, we can then write terms that use different specifications. For example, the following function reads a character from *stdin* and stores it in a mutable reference:

```
readToRef :: Term (IO_s :+: IO_tt) (IORef Char)
readToRef = do
  c ← getChar
  newIORef c
```

Note that we could equally well have given *readToRef* the following, more general type:

```
(IO_s :≺: f, IO_tt :≺: f) ⇒ Term f (IORef Char)
```

There is a clear choice here. We could choose to let *readToRef* work in any *Term* that supports these two operations; or we could want to explicitly state that *readToRef* should only work in the *Term* (*Recall* :+: *Incr*) monad.

Now all that remains is to define the specification of these operations.

3.3 Virtual machine

To write functions over terms, we define the following fold:

```
foldTerm :: Functor f ⇒ (a → b) → (f b → b) → Term f a → b
foldTerm pure impure (Pure x)    = pure x
foldTerm pure impure (Impure t) =
  impure (fmap (foldTerm pure impure) t)
```

The first argument, *pure*, is applied to pure values; the case for impure terms closely resembles the fold over expressions.

Although combining the syntactic structure of programs is fairly easy, modularly combining the semantics is much more difficult (Jaskelioff et al., 2008). Instead of trying to combine the specifications themselves, we take a more pragmatic approach: we define a small, fixed virtual machine and show how all our specifications are interpreted on this machine. The disadvantage of this approach is that if we need to modify the virtual machine if we wish to define new specifications of effects that it cannot handle. Fixing the virtual machine finds a middle ground between modularity and complexity: the syntax is perfectly modular, but we do not need to implement the complex coproduct of two specifications.

Essentially, the virtual machine, *VM*, we define consists of the state monad transformer layered on top of some primitive effects.

```
type VM a = StateT Store Effect a
data Effect a =
    Done a
  | ReadChar (Char → Effect a)
  | Print Char (Effect a)
  | Fail String
```

We have chosen a small number of fixed effects. As a result, we would need to extend the *Effect* data type if we want to add new specifications. For example, to add interactions with the file system, we would need to adapt the *Effect* data type accordingly.

The *Store* data type encapsulates the state of the virtual machine. We defer the definition of the *Store* data type for the moment.

Now to execute our syntactical terms on the virtual machine, we need to define suitable instances of the following class:

```
class Functor f ⇒ Executable f where
  step :: f a → VM (Step a)
data Step a = Step a | Block
```

For every specification, we should describe how to make an atomic step. A scheduler will then interleave the steps, in the same fashion we accomplished in the specification of concurrency from the previous chapter.

The definition of the *Store* data type is in Listing 16. It closely follows the store of the concurrency specification. Note that the *ThreadSoup* consists of executable terms: when the scheduler chooses a particular thread we allow that thread to perform a single atomic action. This *Store* data type also keeps track of the blocked threads and finished threads for efficiency reasons.

The code to execute any term in the *VM* monad is in Listing 17. Here *schedule* consults the scheduler to determine the next thread. We have made a minor refinement compared with the previous definition: we keep track of two lists of those threads that we know to be blocked or finished. We no longer schedule threads whose *ThreadId* occurs in one of these lists. If all the threads are blocked or finished, we have reached a deadlock. Instead of diverging as we did in the previous chapter, we can now fail with a more informative error message.

Every time a thread makes progress, we reset the list of blocked threads; every time a thread fails to make progress, we add it to the list of blocked threads and do not attempt to schedule it again until it becomes unblocked. Besides this minor refinement, the code is extremely similar to the *interleave* function from the previous chapter.

It should be fairly obvious how to adapt the functional specifications from the previous chapter to target the *VM* monad. Once we have defined suitable instances of the *Executable* class for all our syntactic terms, we can simulate the execution of any combination of effects using this virtual machine.

Listing 16 The store of our virtual machine

```
type Data            = Dynamic
type Loc             = Int
type Heap            = Loc → Maybe Data
newtype ThreadId = ThreadId Int
data ThreadStatus =
    ∀f b . Executable f ⇒ Running (Term f b)
  | Finished
type ThreadSoup = ThreadId → ThreadStatus
newtype Scheduler =
  Scheduler (Int → (Int, Scheduler))
data Store =
  Store {fresh :: Loc
        , heap :: Heap
        , nextTid :: ThreadId
        , scheduler :: Scheduler
        , threadSoup :: ThreadSoup
        , blockedThreads :: [ThreadId]
        , finishedThreads :: [ThreadId]
        }
```

Listing 17 Executing a term on the virtual machine

```
data Process a =
    ∀f . Executable f ⇒ Main (Term f a)
  | ∀f b . Executable f ⇒ Aux (Term f b)
execVM :: Executable f ⇒ Term f a → VM a
execVM main = do
  (tid, t) ← schedule main
  case t of
    (Main (Pure x)) → return x
    (Main (Impure p)) →
      do x ← step p
         case x of
           Step y → resetBlockedThreads ≫ execVM y
           Block → blockThread mainTid ≫ execVM main
    (Aux (Pure _)) → finishThread tid ≫
                     execVM main
    (Aux (Impure p)) → do x ← step p
                          case x of
                            Step y → resetBlockedThreads ≫
                                     updateSoup tid y ≫
                                     execVM main
                            Block → blockThread tid ≫
                                    execVM main
```

Conclusion This chapter shows how the functional specifications from the previous chapter may be structured in Haskell. In the next chapter, we will put our virtual machine to the test to see just how hard it is to define new specifications.

Chapter 4

Software transactional memory

How general is the virtual machine presented in the previous chapter? To evaluate the design of the virtual machine, we show how to give a specification for software transactional memory (Harris et al., 2005). Before we give our specification, we will give a brief introduction to software transactional memory by studying the following problem, taken from Peyton Jones (2007):

> Write a procedure to transfer money from one bank account to another. To keep things simple, both accounts are held in memory: no interaction with databases is required. The procedure must operate correctly in a concurrent program, in which many threads may transfer money simultaneously. No thread should be able to observe a state in which the money has left one account, but not arrived in the other (or vice versa).

The problem is actually quite a bit harder than you may think. A traditional solution would use locks to guarantee that transactions are atomic. As Peyton Jones points out, such lock-based programs are not modular: it is all too easy to take too few locks, take too many locks, take the wrong locks, or take locks in the wrong order.

Software transactional memory provides a controlled form of shared memory. Programmers may group sequences of read and write operations into *atomic blocks*. The intermediate states that arise during the execution of an atomic block are never exposed to other threads that have access to the same shared memory.

While this atomicity makes programming with shared memory easier, efficient implementations of transactional memory are quite complex. Executing atomic blocks one by one fails to utilise much of the potential speedup that modern multi-core computers provide. Instead, an atomic block is speculatively executed, maintaining a log of all the read and writes to shared memory. At the end of an atomic block, the resulting changes are committed to the shared memory *provided that they do not break atomicity*. Validating when such changes are safe to commit is still a subject of active research (Hu and Hutton, 2008).

In Haskell, transactional memory is implemented by introducing a separate monad, distinct from the IO monad, in which memory transactions can be constructed. There are three primitive operations to manipulate transactional memory:

```
newTVar   :: a → STM (TVar a)
readTVar  :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
```

A transactional variable, or *TVar*, is created and manipulated in the same way as a an *IORef* or *MVar*. The crucial difference is that these operations live in the *STM* monad, and do not perform any I/O. To execute a transaction atomically, there is one other operation:

```
atomically :: STM a → IO a
```

For any transaction *t* in the *STM* monad, *atomically t* executes *t* in one atomic step.

Using these functions, we could now implement the *transfer* procedure as follows:

```
transfer :: TVar Int → TVar Int → Int → IO ()
transfer from to amount = atomically transaction
  where
  transaction = do
    fromBalance ← readTVar from
    toBalance ← readTVar to
    writeTVar from (fromBalance − amount)
    writeTVar to (toBalance + amount)
```

The central idea is to separate the construction of atomic transactions in the *STM* monad from their execution in the *IO* monad.

Many concurrent programs require two other operations: blocking and choice. There are situations in which a transaction may not be possible: you may attempt to read from an empty channel; debit an already overdrawn bank account; or write to an overflowing buffer. To deal with these cases, there is a special operation *retry*:

```
retry :: STM a
```

When *retry* is called, the current transaction is abandoned. As the name suggests, it will be retried at some later time.

Finally, the *orElse* operation lets you 'handle' a retry.

```
orElse :: STM a → STM a → STM a
```

Given two transactions, *orElse* tries to execute its first argument. If the first argument calls *retry* – either explicitly, or implicitly as a result of failing to commit – it will try to execute its second argument. If the second transaction also retries, the whole computation retries.

For example, when withdrawing money from an account we may want to ensure it is never overdrawn. To do so, we call *retry* if the transaction would result in an overdrawn account:

```
withdraw :: TVar Int → Int → STM ()
withdraw account amount = do
  balance ← readTVar account
  if amount > 0 ∧ amount > balance
    then retry
    else writeTVar account (balance − amount)
```

These six operations constitute the core of software transactional memory in Haskell. We will now present a functional specification of these operations that makes the informal description rigorous.

4.1 Syntax

We begin by fixing the syntax for transactions. Once again we define a data type, *STM*, with constructors for all the transactional operations and corresponding smart constructors.

```
data STM a =
    STMReturn a
  | NewTVar Data (Loc → STM a)
  | ReadTVar Loc (Data → STM a)
  | WriteTVar Loc Data (STM a)
  | Retry
  | OrElse (STM a) (STM a)
newTVar :: Typeable a ⇒ a → STM (TVar a)
newTVar d = NewTVar (toDyn d) (STMReturn ∘ TVar)
readTVar :: Typeable a ⇒ TVar a → STM a
readTVar (TVar l) =
  ReadTVar l (STMReturn ∘ fromJust ∘ fromDynamic)
writeTVar :: Typeable a ⇒ TVar a → a → STM ()
writeTVar (TVar l) d = WriteTVar l (toDyn d) (STMReturn ())
retry :: STM a
retry = Retry
orElse :: STM a → STM a → STM a
orElse p q = OrElse p q
```

It is easy to show that the *STM* type constructor is a monad.

Choosing how to represent the *atomically* operation is a bit more interesting:

```
data Transaction t = ∀b . Atomically (STM b) (b → t)
atomically :: (Transaction :<: f) ⇒ STM a → Term f a
atomically stm = inject (Atomically stm (return))
```

Clearly, the *Atomically* constructor should take a transaction as its first argument. What type should this transaction return? Here we have chosen to allow the transaction to return *any* type. Consequently, we universally quantify over the type the transaction returns. The second argument of the *Atomically* constructor corresponds to the rest of the computation, that may use the result of the atomic action. We can use the *inject* function from the previous chapter to define a smart constructor.

4.2 Specification

Now that we have fixed the syntax of our computations, we can describe how to execute transactions on our virtual machine. We will begin by defining an *executeSTM* function that runs any transaction on our virtual machine. Using this function, we can then give a specification of *atomically*.

The *executeSTM* function in Listing 18 takes a transaction of type *STM a* and runs it on the virtual machine to produce a result of type *Maybe a*. Here we will use *Nothing* to represent a transaction that calls *retry*; on the other hand, *Just x* will correspond to a transaction that returns the value *x* without retrying.

Listing 18 Executing transactions on the virtual machine

```
executeSTM :: STM a → VM (Maybe a)
executeSTM (STMReturn x) = return (Just x)
executeSTM (NewTVar d io) =
  do loc ← alloc
     updateHeap loc d
     executeSTM (io loc)
executeSTM (ReadTVar l io) =
  do (Just d) ← lookupHeap l
     executeSTM (io d)
executeSTM (WriteTVar l d io) =
  do updateHeap l d
     executeSTM io
executeSTM Retry = return Nothing
executeSTM (OrElse p q) =
  do state ← get
     case runStateT (executeSTM p) state of
       Done (Nothing, _) → executeSTM q
       Done (Just x, s)  → put s ≫ return (Just x)
```

The cases of the *executeSTM* function that deal with transactional variables should be familiar. They closely follow the specification of mutable state from the previous two chapters. The cases for *Retry* and *OrElse* are more interesting.

If a transaction calls *Retry*, we simply return *Nothing*. The whole computation must be retried at some later point. In the case for *OrElse*, we begin by requesting a copy of the current store. We proceed by executing the first argument of *OrElse*, starting from the current store. If this transaction retries and returns *Nothing*, we continue by executing the second transaction and discard any changes to the store the first transaction may have made. If the transaction succeeds, on the other hand, it will return a result together with a new store that is the result of committing the transaction. In that case, we overwrite the old state with this new store and return the result of the first transaction.

Finally, we give the specification of *Atomically*. We begin by creating a copy of the current store and attempt to run the argument transaction against that store.

```
instance Executable Transaction where
  step (Atomically stm io) = do
    state ← get
    case runStateT (executeSTM stm) state of
      Done (Nothing, _)         → return Block
      Done (Just x, finalState) → put finalState ≫ return (Step (io x))
```

If the transaction fails, we return the *Block* constructor of the *Step* data type to indicate that this thread cannot currently make progress. If the transaction succeeds, we overwrite the current store; the result of our transaction is then fed to the remaining computation *io* and wrapped in a *Step* constructor to indicate that progress has been made.

Note that this specification models the simple 'stop-the-world' semantics of transactional memory. It does not speculatively evaluate threads in parallel in the hope that their results may be reconciled, but rather executes individual atomic blocks in one step.

This specification makes quite heavy use of the functionality provided by the virtual machine. Instead of rewriting all the code to interleave concurrent processes, we only need to specify the behaviour of a single atomic step. Implementing the behaviour of a *TVar* shares a lot of functions with the specifications of *IORefs* and *MVars*. This gives us some degree of confidence that the virtual machine provides useful functionality that can be used to implement the specification of other effects.

Testing transfers money Just as we have seen previously, we can now employ QuickCheck to test Haskell functions that use software transactional memory. One crucial property of the *transfer* function, for example, is that it should never change the total amount of money in the two accounts.

To test that this property is maintained, we can define the following test:

```
$transferTest\ init_1\ init_2\ trans_1\ trans_2 = \mathbf{do}$
  $var_1 \leftarrow atomically\ (newTVar\ init_1)$
  $var_2 \leftarrow atomically\ (newTVar\ init_2)$
  $forM\ trans_1\ (forkIO \circ transfer\ var_1\ var_2)$
  $forM\ trans_2\ (forkIO \circ transfer\ var_2\ var_1)$
  $atomically\ (liftM2\ (,)\ (readTVar\ var_1)\ (readTVar\ var_2))$
```

The *transferTest* takes four arguments: two integers and two lists of integers, $trans_1$ and $trans_2$. The function begins by initialising two transactional variables storing the two integer arguments. It then proceeds by forking off a threads to transfer money between these two variables. For every integer i in $trans_1$, there will be a thread that transfers i from one account to the other; the integers in $trans_2$ give rise to transfers in the other direction. The *transferTest* concludes by atomically reading the two transactional variables and returning this result. Note that not all transactions will necessarily be completed when the *transferTest* function returns its result.

We can now test that the transfer function does satisfies the following property:

```
$transferProp :: Int \rightarrow Int \rightarrow [Int] \rightarrow [Int] \rightarrow Scheduler \rightarrow Bool$
$transferProp\ i_1\ i_2\ trans_1\ trans_2\ s =$
  $\mathbf{let}\ Done\ (res_1, res_2) = evalVM\ (transferTest\ i_1\ i_2\ trans_1\ trans_2)\ s$
  $\mathbf{in}\ i_1 + i_2 \equiv res_1 + res_2$
```

Put in words, no number of transactions between the two accounts should change the sum of money in the two accounts. We use the *evalVM* function to run a computation in the *VM* monad that we saw in the previous chapter. Reassuringly, QuickCheck fails to find a counterexample after one hundred tests.

Conclusion In the first chapters of this dissertation, we have shown how functional specifications may be implemented and structured in Haskell. We still need to address one of the major shortcomings of these specifications: they do not correspond to total functions. In order to accomplish this, however, we need to shift to a different language.

Chapter 5

Dependently typed programming in Agda

At the end of Chapter 2, we identified several problems with the functional specifications we have seen so far. Crucially, the Haskell specifications presented in Chapter 2 are partial. In the remainder of this dissertation we will show how such specifications can be made total in a richer type theory.

Before we do so, we will explore the dependently typed programming language Agda. This chapter will only cover the features of Agda used in the rest of this thesis. There are several other tutorials (Bove and Dybjer, 2008; Norell, 2008) that give a more complete overview of the language. Norell's thesis (Norell, 2007) documents many of the more technical issues.

5.1 Agda basics

Agda is both a pure programming language and a consistent type theory. In other words, Agda functions must be pure and total and may not contain undefined values; any well-typed Agda term corresponds to a proof that its type is inhabited.

To enforce totality, Agda has a *coverage checker* and a *termination checker*. The coverage checker ensures that there are no missing cases when pattern matching. The termination checker ensures that all functions are obviously structurally recursive. Both these problems are undecidable in general, so it is up to the programmer to define functions in such a manner that they do pass the required checks.

Data types in Agda can be defined using a similar syntax to that for Generalized Algebraic Data Types, or GADTs, in Haskell (Peyton Jones et al., 2006). For example, consider the following definition of the natural numbers.

```
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat
```

There is one important difference with Haskell. We must explicitly state the *kind* of the data type that we are introducing; in particular, the declaration *Nat* : *Set* states that *Nat* is a base type.

We can define functions by pattern matching and recursion, just as in any other functional language. To define addition of natural numbers, for instance, we could write:

```
_+_ : Nat → Nat → Nat
Zero   + m = m
Succ n + m = Succ (n + m)
```

Note that Agda uses underscores to denote the positions of arguments when defining new operators. Using this notation you can also define prefix, postfix, or even mix-fix operators.

Polymorphic lists are slightly more interesting than natural numbers:

```
data List (a : Set) : Set where
  Nil : List a
  Cons : a → List a → List a
```

To uniformly parameterise a data type, we can write additional arguments to the left of the colon. In this case, we add (*a* : *Set*) to our data type declaration to state that lists are type *constructors*, parameterised over a type variable *a* of kind *Set*. By convention, we will capitalise data constructors throughout this thesis, although this is not required by Agda.

Just as we defined addition for natural numbers, we can define a function that appends one list to another:

```
append : (a : Set) → List a → List a → List a
append a Nil         ys = ys
append a (Cons x xs) ys = Cons x (append a xs ys)
```

The *append* function is polymorphic. In Agda, such polymorphism can be introduced via the *dependent function space*, written $(x : a) \rightarrow y$, where the variable x may occur in the type y. This particular example of the dependent function space is not terribly interesting: it corresponds to parametric polymorphism. Later we will encounter more interesting examples, where *types* depend on *values*.

5.1.1 Implicit arguments

One drawback of using the dependent function space for such parametric polymorphism, is that we must explicitly instantiate polymorphic functions. For example, the recursive call to *append* in the *Cons* case takes a type as its first argument. Fortunately, Agda allows us to mark certain arguments as *implicit*. Using implicit arguments, we could also define *append* as in any other functional language:

$$\begin{aligned}
&append : \{a : Set\} \rightarrow List\ a \rightarrow List\ a \rightarrow List\ a\\
&append\ Nil \qquad\qquad ys = ys\\
&append\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ xs\ ys)
\end{aligned}$$

Arguments enclosed in curly brackets, such as $\{a : Set\}$, are implicit: we do not write a to the left of the equals sign and do not pass a type argument when we call the make a recursive call. Provided the value of an implicit argument is determined by the other arguments of a function, the Agda type checker will automatically instantiate this function whenever we call it, much in the same way as type variables are automatically instantiated in Haskell. Note that we can explicitly pass implicit arguments by enclosing a function's arguments in curly brackets. For example, the *appendNats* function specialises the *append* function to take two lists of natural numbers as its arguments:

$$\begin{aligned}
&appendNats : List\ Nat \rightarrow List\ Nat \rightarrow List\ Nat\\
&appendNats = append\ \{Nat\}
\end{aligned}$$

This notation for implicit arguments can be rather cumbersome. For example, if we wish to define the S combinator in Haskell, we would write:

$$\begin{aligned}
&s : (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c\\
&s\ f\ g\ x = f\ x\ (g\ x)
\end{aligned}$$

In Agda we would need to explicitly quantify over all three type variables:

$$s : \{a : Set\} \to \{b : Set\} \to \{c : Set\} \to ...$$

Agda provides a shorthand for grouping together several implicit arguments. We could also write:

$$s : \textbf{forall}\ \{a\ b\ c\} \to ...$$

The type checker will try to infer the types of a, b, and c.

Similarly, we will sometimes group several variables of the same type in a single set of parentheses, for example writing $(n\ \ m : Nat) \to P\ n\ m$ rather than the more verbose $(n : Nat) \to (m : Nat) \to P\ n\ m$.

5.1.2 Indexed families

Besides polymorphic data types, Agda also supports *indexed families*, similar to Haskell's GADTs. In Haskell, you can define a GADT to represent a small expression language as follows:

```
data Expr a where
  Lit :: Int → Expr Int
  Add :: Expr Int → Expr Int → Expr Int
  IsZero :: Expr Int → Expr Bool
  If :: Expr Bool → Expr a → Expr a → Expr a
```

Here the data type *Expr* is parameterised by a type variable *a*. When defining conventional Haskell data types, such as lists, the result type of all the constructors must be the type constructor applied to the type parameters of the data type. For example, both the *Nil* and *Cons* constructor return a value of type *List a*. The constructors of a GADT, on the other hand, are free to specialise the arguments to the type constructor. In the above GADT, for example, the *Lit* constructor takes any integer literal and returns a value of type *Expr Int*.

We can now define an evaluation function for our expression language *without any dynamic checks*. By pattern matching on the *Expr* argument, we learn the required type of the case branch. Bearing this in mind, we can define the evaluation function as follows:

$$
\begin{array}{ll}
\mathit{eval} : \mathit{Expr}\ a \rightarrow a & \\
\mathit{eval}\ (\mathit{Lit}\ x) & = x \\
\mathit{eval}\ (\mathit{Add}\ l\ r) & = \mathit{eval}\ l + \mathit{eval}\ r \\
\mathit{eval}\ (\mathit{IsZero}\ e) & = \mathit{eval}\ e \equiv 0 \\
\mathit{eval}\ (\mathit{If}\ c\ t\ e) & = \mathbf{if}\ \mathit{eval}\ c\ \mathbf{then}\ \mathit{eval}\ t\ \mathbf{else}\ \mathit{eval}\ e
\end{array}
$$

Note that the *Lit* branch, for example, has type *Int* because the *Lit* constructor returns a value of type *Expr Int*. The *IsZero* branch, on the other hand, returns a boolean.

Indexed families in Agda differ from GADTs as they may indexed by *any* value and not just types. For example, we can define the family of finite types:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Fin} : \mathit{Nat} \rightarrow \mathit{Set}\ \mathbf{where} \\
\quad \mathit{Fz} : \mathbf{forall}\ \{n\} \rightarrow \mathit{Fin}\ (\mathit{Succ}\ n) \\
\quad \mathit{Fs} : \mathbf{forall}\ \{n\} \rightarrow \mathit{Fin}\ n \rightarrow \mathit{Fin}\ (\mathit{Succ}\ n)
\end{array}
$$

The type *Fin n* corresponds to a finite type with *n* distinct values. For example, *Fin* 1 is isomorphic to the unit type; *Fin* 2 is isomorphic to *Bool*. Note that the argument *n* is left implicit in both the constructors of *Fin*. It is easy to see that *Fin n* does indeed have *n* elements by a simple inductive argument. From the types of these constructors, it is clear that *Fin Zero* is uninhabited. For every *n*, the *Fs* constructor embeds *Fin n* into *Fin* $(\mathit{Succ}\ n)$; the *Fz* constructor, on the other hand, adds a single new element to *Fin* $(\mathit{Succ}\ n)$ that was not in *Fin n*. Hence for every natural number *n*, the type *Fin n* has exactly *n* inhabitants.

We can freely mix parameterised and indexed data types. The classic example is that of vectors, or list of some fixed length:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Vec}\ (a : \mathit{Set}) : \mathit{Nat} \rightarrow \mathit{Set}\ \mathbf{where} \\
\quad \mathit{Nil} : \mathit{Vec}\ a\ \mathit{Zero} \\
\quad \mathit{Cons} : \mathbf{forall}\ \{n\} \rightarrow a \rightarrow \mathit{Vec}\ a\ n \rightarrow \mathit{Vec}\ a\ (\mathit{Succ}\ n)
\end{array}
$$

Vectors are parameterised by a type variable $a : \mathit{Set}$, representing the elements of the vector, and indexed by a natural number, corresponding to the vector's length.

5.2 Pattern matching

In the presence of inductive families, pattern matching becomes rather subtle. This section illustrates various aspects of pattern matching in Agda by means of several examples.

First of all, we will occasionally pattern match on implicit arguments. To do so, we simply enclose the pattern of the implicit argument in curly brackets. For example, we may define the largest element of *Fin n* by induction on an implicit argument *n* as follows:

$$\begin{aligned}
&\mathit{fmax} : \{n : \mathit{Nat}\} \to \mathit{Fin}\ (\mathit{Succ}\ n)\\
&\mathit{fmax}\ \{\mathit{Zero}\} \quad\ = \mathit{Fz}\\
&\mathit{fmax}\ \{\mathit{Succ}\ k\} = \mathit{Fs}\ \mathit{fmax}
\end{aligned}$$

With the definition of finite types and vectors from the previous section, we can define the following operation that takes a vector of length *n* and an element of *Fin n* as its arguments. It returns the element of the vector at index *Fin n*:

$$\begin{aligned}
&\_!\_ : \mathbf{forall}\ \{n\ a\} \to \mathit{Vec}\ a\ n \to \mathit{Fin}\ n \to a\\
&\mathit{Nil} \mathbin{!} ()\\
&(\mathit{Cons}\ x\ \mathit{xs}) \mathbin{!} \mathit{Fz} = x\\
&(\mathit{Cons}\ x\ \mathit{xs}) \mathbin{!} (\mathit{Fs}\ i) = \mathit{xs} \mathbin{!} i
\end{aligned}$$

The two latter cases for a non-empty vector are relatively straightforward: *Fz* returns the first element of the list; *Fs i* continues recursively. In the case for the empty vector, however, there is no element to return – but if the vector is empty, the second argument must be of type *Fin Zero*, which is uninhabited. Agda allows us to write the special pattern () to 'kill off' impossible case branches. When we match on an uninhabited type, we have no obligation to provide a right-hand side of the function definition – after all, there is no way this function can be called with that argument.

The definition of the !-operator illustrates an important point: when we pattern matched on the empty vector, we learned that the second argument must be uninhabited: pattern matching on dependent types may introduce equations between values. This crucial point is probably best illustrated with another example.

Suppose we define the following data type:

$$
\begin{array}{l}
\textbf{data}\ \_\equiv\_\ \{a : \mathit{Set}\}\ (x : a) : a \to \mathit{Prop}\ \textbf{where} \\
\quad \mathit{Refl} : x \equiv x
\end{array}
$$

A value of type $(x \equiv y)$ corresponds to a proof that x and y are equal. The $\equiv$-type is parameterised by an implicit type a and a value x of type a, but is indexed by a second argument of type a. It has a single constructor, *Refl*, that corresponds to a proof that any x is equal to itself. This type plays a fundamental role in intensional type theory (Nordström et al., 1990).

Note that the $x \equiv y$ is a value of type *Prop* and not *Set*. Although there is a subtle difference, you may want to think of *Prop* as a synonym for *Set* for the moment. We will come back to this point in Section 5.4.

Whenever we pattern match on such a proof, we learn how two values are related. For example, suppose we want to write the following function:

$$
\begin{array}{l}
\mathit{subst} : \{a : \mathit{Set}\} \to (P : a \to \mathit{Prop}) \to \\
\quad (x : a) \to (y : a) \to x \equiv y \to P\ x \to P\ y
\end{array}
$$

What patterns should we write on the left-hand side of the definition? Clearly, any argument of type $x \equiv y$ must be *Refl*. As soon as we match on that argument, however, we learn something about x and y, i.e., they must be the same. We will write this as follows:

$$\mathit{subst}\ P\ x\ \lfloor x \rfloor\ \mathit{Refl}\ \mathit{px} = ...$$

The pattern $\lfloor x \rfloor$ means 'the value of this argument can only be equal to x.'

In Agda you must currently write out how the different patterns relate by hand – that is, you must explicitly mark such forced patterns. Epigram (McBride and McKinna, 2004), on the other hand, demonstrated that this process can be automated.

Occasionally, we may not be interested in the information we learn, in which case we will use the underscore as a wildcard pattern:

$$f\ x\ \_\ \mathit{Refl} = ...$$

While not necessary, we believe that writing out patterns such as $\lfloor x \rfloor$ explicitly serves as important, machine-checked documentation of what we learn during pattern matching.

5.3 Proofs

As we have mentioned previously, dependently typed programming languages such as Agda provide a unified framework for proofs and programs. We briefly give a taste of how to write such proofs in Agda, but refer to existing literature (Nordström et al., 1990) for a more thorough treatment.

Suppose we want to prove that $n \equiv n + 0$ for all natural numbers n. One way to prove this is by using the following property of equality:

$$\begin{array}{l} \mathit{cong} : \textbf{forall}\ \{a\ b\ x\ y\}\ (f : a \to b) \to x \equiv y \to f\ x \equiv f\ y \\ \mathit{cong}\ f\ \mathit{Refl} = \mathit{Refl} \end{array}$$

That is to say, if we have a proof that $x \equiv y$ then $f\ x \equiv f\ y$ for all functions $f : a \to b$.

Using *cong* we can now prove our lemma as follows:

$$\begin{array}{l} \mathit{plusZero} : (n : \mathit{Nat}) \to n \equiv (n + 0) \\ \mathit{plusZero}\ \mathit{Zero} \qquad = \mathit{Refl} \\ \mathit{plusZero}\ (\mathit{Succ}\ k) = \mathit{cong}\ \mathit{Succ}\ (\mathit{plusZero}\ k) \end{array}$$

In the base case we must show $0 \equiv 0 + 0$. As Agda's type checker automatically unfolds the definition of addition, this reduces to showing $0 \equiv 0$, which itself holds by reflexivity.

To show $(\mathit{Succ}\ k) \equiv (\mathit{Succ}\ k + 0)$, we make a recursive call to *plusZero* to produce a proof that $k \equiv k + 0$. Now applying *cong Succ* to this proof yields a proof that $\mathit{Succ}\ k \equiv \mathit{Succ}\ (k + 0)$, as required.

The proof of non-trivial theorems can become rather large. Sometimes it is a bit more convenient to write out such proofs differently. Alternatively we could have proven the *plusZero* theorem as follows:

$$\begin{array}{l} \mathit{plusZero} : (n : \mathit{Nat}) \to n \equiv (n + 0) \\ \mathit{plusZero}\ \mathit{Zero} = \mathit{Refl} \\ \mathit{plusZero}\ (\mathit{Succ}\ k)\ \textbf{with}\ k + 0 \mid \mathit{plusZero}\ k \\ \mathit{plusZero}\ (\mathit{Succ}\ k) \mid \lfloor k \rfloor \mid \mathit{Refl} = \mathit{Refl} \end{array}$$

Although the base case is unchanged, the inductive step is somewhat different. The with-rule introduces a local pattern match, similar to Haskell's case-expressions. Here we pattern match on two expressions: $k + 0$ and *plusZero k*. On the next line we repeat the left-hand side of the function definition, followed by the new patterns $\lfloor k \rfloor$ and *Refl*. We separate the new patterns by a vertical bar. Once we match on *plusZero k* we learn that $k \equiv k + 0$. Correspondingly, the proof that $\mathit{Succ}\ k \equiv \mathit{Succ}\ (k + 0)$ becomes trivial.

The with-rule does much more than introduce a simple case expression. It generates an auxiliary function with a type that abstracts over all occurrences of the expression on which we wish to pattern match. In order for this type to be well-formed we may need to abstract over additional expressions. In our example, we matched on both $k + 0$ and *plusZero k*. If we had matched on just *plusZero k*, the auxiliary function that Agda generates would not generalise over $k + 0$ and correspondingly not recognise *Refl* as a proof that $k \equiv k + 0$. For a more thorough description of the with-rule we refer to Norell's thesis (2007).

5.4 Dependent pairs

Besides the dependent function space, there is a second dependent type that will play an important part throughout this thesis. We can define a pair of two types as follows:

```
data Pair (A : Set) (B : Set) : Set where
  _,_ : A → B → Pair A B
```

A *Pair* of two types *A* and *B* consists of a value of type *A* and a value of type *B*.

In a theory with dependent types, we can generalise this as follows:

```
data Sigma (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Sigma A B
```

In such a dependent pair, or *Sigma* type, the type of the second component of the pair may depend on the value of the first component. This is reflected in the two type arguments of the *Sigma* type: a set *A* and function *B* from *A* to *Set*.

Such dependent pairs arise quite naturally. For example, we may be interested in all the natural numbers larger than 3. One way to represent such numbers in Agda would be as a dependent pair of a number *n* and a proof that $n > 3$. This pattern appears frequently enough to warrant a variation of the above definition:

```
data Spec (A : Set) (P : A → Prop) : Set where
  Satisfies : (x : A) → P x → Spec A P
```

Rather than a pair of two values, the inhabitants of this type consist of a value x and a proof that x satisfies some property P. Here we distinguish between *propositions* and *types*. In contrast to types in *Set*, a proposition of type *Prop* has no computational content: we are not interested in the exact value of the proof, but only care about whether a proposition is true or not. We will use notation reminiscent of set comprehensions and write $\{n : Nat \mid n > 3\}$ rather than the more verbose *Spec Nat* $(\lambda n \to n > 3)$.

The idea of tightly coupling code and proof has emerged in many different settings (McKinna, 1992; Necula, 1997; Sozeau, 2007). In this dissertation, such dependent pairs will play a central role in the Chapters 8 and 9 when we discuss the relation between our functional specifications and Hoare Type Theory.

5.5 Universes

Universes are a fundamental concept in type theory (Martin-Löf, 1984). We explain what a universe is using a concrete example that should be familiar to Haskell programmers.

Agda does not have type classes. Yet years of experience with Haskell has underlined the importance of ad hoc polymorphism. How might we achieve the same in a dependently typed programming language?

Type classes are used to describe the collection of types that support certain operations, such as a decidable equality. The same issue also arises in type theory, where you may be interested in a certain collection of types that share some property, such as having a finite number of inhabitants. It is unsurprising that the techniques from type theory for describing such collections of types can be used to model type classes.

A *universe* consists of a pair of a type U and a function $el : U \to Set$. Intuitively, the data type U contains 'codes' for typs; the function *el* maps these codes to their corresponding type. This is best illustrated with an example.

Consider the following type *U*:

```
data U : Set where
  CHAR : U
  NAT : U
  VEC : U → Nat → U
```

Every data constructor of *U* corresponds to some type constructor. In dependently typed programming languages such as Agda, we can define the decoding function *el* that makes this relationship precise.

```
el : U → Set
el CHAR      = Char
el NAT       = Nat
el (VEC u n) = Vec (el u) n
```

We can now define operations on the types in this universe by induction on *U*. For example, every type represented by *U* can be rendered as a *String*:

```
show : {u : U} → el u → String
show {CHAR} c          = charToString c
show {NAT} Zero        = "Zero"
show {NAT} (Succ k)    = "Succ " ++ parens (show k)
show {VEC u Zero} Nil  = "Nil"
show {VEC u (Succ k)} (Cons x xs)
  = parens (show x) ++ " : " ++ parens (show xs)
parens : String → String
parens str = "(" ++ str ++ ")"
```

The *show* function takes two arguments: an implicit argument *u* of type *U*; and a second argument of type *el u*. Pattern matching on *u* determines the type of the data that must be rendered. For instance, if *u* is the code for natural numbers *NAT*, the second argument must be *Zero* or *Succ k*. This definition overloads the *show* function – enabling the same function to work on different types, just as Haskell's type classes.

When we call *show*, Agda will fill in the implicit argument of type *U* for us. This allows us to use the same name, *show*, for a function that operates on different types. This is illustrated by the following interaction with the Agda prompt:

```
Main> show 2
"Succ (Succ (Zero))"
Main> show 'c'
"c"
```

Note that, in contrast to Haskell's type classes, the data type *U* is closed. We cannot add new types to the universe without extending the data type *U* and the function *el*.

Such universes can been used for generic programming (Altenkirch and McBride, 2003; Morris et al., 2007; Morris, 2007) or interfacing to foreign data (Oury and Swierstra, 2008). We will encounter another application in the next chapter.

Chapter 6

A Total Specification of Mutable State

There are two important problems with the specification of mutable state in Chapter 2:

- All references must store data of a single, fixed type. We can circumvent this issue to some extent by using Haskell's support for dynamic typing, but by doing so we sacrifice the ability to do any kind of equational reasoning about effectful programs. Furthermore we have not provided any justification for why the phantom types associated with memory locations preclude failing downcasts.
- The specification relies on using partial functions. For example, the initial heap is left undefined. As a result, any proof using the specification may be unsound.

In this chapter we show how both these issues can readily be fixed by moving to Agda's richer type theory. The key idea is to assign the specifications from Chapter 2 more precise types, thereby demonstrating these specifications are indeed total functions.

6.1 Syntax and semantics of mutable state

We begin by choosing the types that may be stored in a mutable reference. Rather than fix any particular set of types, we can parameterise the module we are defining by some universe:

module *Refs* $(U : Set)$ $(el : U \to Set)$ **where**

When programmers import this module, they will need to instantiate *U* and *el* to a universe that suits their purpose. For example, you might define the universe of Gödel's System T as follows:

```
data U : Set where
  NAT : U
  _ ⇒ _ : U → U → U
el : U → Set
el NAT = Nat
el (u ⇒ v) = el u → el v
```

For example, with this choice of universe you may store natural numbers or functions of type $Nat \to Nat$ in references. Note that the universe is not limited to these two types. For instance, $(NAT \Rightarrow NAT) \Rightarrow NAT$ or $NAT \Rightarrow NAT \Rightarrow NAT$ are also inhabitants of *U*.

Heaps and references How should we implement the heap? If we allow values of different types to be stored on the heap a simple list will no longer suffice. To handle heterogeneous values stored on the heap, we begin by defining the following type:

```
Shape : Set
Shape = List U
```

The *Shape* of a heap determines the type of the data stored at every memory location.

The heap itself is indexed by a *Shape*. We define it here as a heterogeneous list of values, in accordance with the types prescribed by its shape:

```
data Heap : Shape → Set where
  Empty : Heap Nil
  Alloc : forall {u s} → el u → Heap s → Heap (Cons u s)
```

The *Empty* constructor corresponds to an empty heap; the *Alloc* constructor extends a heap of shape *s* with a value of type *el u* to construct a heap of shape *Cons u s*.

Next, we model references that denote locations in the heap. A value of type *Ref u s* corresponds to a reference to a value of type *el u* in a heap of shape *s*.

```
data Ref : U → Shape → Set where
  Top : forall {u s} → Ref u (Cons u s)
  Pop : forall {u v s} → Ref u s → Ref u (Cons v s)
```

These references share a great deal of structure with the *Fin* type we saw in the previous chapter. For every non-empty heap, there is a reference to the first element *Top*; we can weaken any reference to denote a location in a larger heap using the *Pop* constructor.

Syntax With these data types in place, we define a data type capturing the syntax of the permissible operations that manipulate the heap. Crucially, the *IO* type is indexed by *two* shapes: a value of type *IO a s t* denotes a computation that takes a heap of shape *s* to a heap of shape *t* and returns a result of type *a*. This pattern of indexing operations by an initial and final state is a common pattern in dependently typed programming (McKinna and Wright, 2006; Altenkirch and Reus, 1999).

```
data IO (a : Set) : Shape → Shape → Set where
  Return : forall {s} → a → IO a s s
  Write : forall {s t u} → Ref u s → el u → IO a s t → IO a s t
  Read : forall {s t u} → Ref u s → (el u → IO a s t) → IO a s t
  New : forall {s t u} →
    el u → (Ref u (Cons u s) → IO a (Cons u s) t) → IO a s t
```

The *IO* type has four constructors. The *Return* constructor returns a pure value of type *a* without modifying the heap. The *Write* constructor takes three arguments: a reference to a value of type *el u*; the value to write to that reference; and the rest of the computation. The *Read* constructor also takes a reference as its first argument. The result of reading the value stored in this reference may be used in the remaining computation. Finally, the *New* constructor actually changes the size of the heap. Given a value of type *el u*, it extends the heap with this value; the second argument of *New* may then use this fresh reference to continue the computation in a larger heap.

The *IO* data type is a *parameterised monad* (Atkey, 2006) – a monad with *return* and bind operators that satisfy certain coherence conditions with respect to the *Shape* indices.

$$
\begin{array}{l}
\mathit{return} : \textbf{forall}\ \{s\ a\} \to a \to \mathit{IO}\ a\ s\ s \\
\mathit{return} = \mathit{Return} \\
\_ \gg\!= \_ : \textbf{forall}\ \{s\ t\ u\ a\ b\} \to \mathit{IO}\ a\ s\ t \to (a \to \mathit{IO}\ b\ t\ u) \to \mathit{IO}\ b\ s\ u \\
\mathit{Return}\ x \gg\!= f = f\ x \\
\mathit{Write}\ l\ d\ \mathit{wr} \gg\!= f = \mathit{Write}\ l\ d\ (\mathit{wr} \gg\!= f) \\
\mathit{Read}\ l\ \mathit{rd} \gg\!= f = \mathit{Read}\ l\ (\lambda d \to \mathit{rd}\ d \gg\!= f) \\
\mathit{New}\ d\ \mathit{io} \gg\!= f = \mathit{New}\ d\ (\lambda l \to \mathit{io}\ l \gg\!= f)
\end{array}
$$

The return of the *IO* data type lifts a pure value into a computation that can run on a heap of any size. Furthermore, *return* does not modify the shape of the heap. The bind operator, $\gg\!=$, can be used to compose monadic computations. To sequence two computations, the heap resulting from the first computation must be a suitable starting point for the second computation. This condition is enforced by the type of the bind operator.

Semantics We have described the syntax of array computations using the *IO* data type above, but we have not specified how these computation *behave*. Before we do so, we will define two auxiliary functions to update and lookup mutable references.

$$
\begin{array}{l}
\_\,!\,\_ : \textbf{forall}\ \{s\ u\} \to \mathit{Heap}\ s \to \mathit{Ref}\ u\ s \to \mathit{el}\ u \\
\_\,!\,\_\ \mathit{Empty}\ () \\
\mathit{Alloc}\ x\ \_\ !\ \mathit{Top} = x \\
\mathit{Alloc}\ \_\ h\ !\ \mathit{Pop}\ k = h\ !\ k \\
\mathit{update} : \textbf{forall}\ \{s\ u\} \to \mathit{Heap}\ s \to \mathit{Ref}\ u\ s \to \mathit{el}\ u \to \mathit{Heap}\ s \\
\mathit{update}\ \mathit{Empty}\ ()\ d \\
\mathit{update}\ (\mathit{Alloc}\ \_\ \mathit{heap})\ \mathit{Top}\ d = \mathit{Alloc}\ d\ \mathit{heap} \\
\mathit{update}\ (\mathit{Alloc}\ x\ \mathit{heap})\ (\mathit{Pop}\ i)\ d = \mathit{Alloc}\ x\ (\mathit{update}\ \mathit{heap}\ i\ d)
\end{array}
$$

The !-operator dereferences a *Ref* by traversing the heap and reference simultaneously until we reach a *Top* constructor. Note that if the heap is empty, there are no valid references that can be passed as arguments to the !-operator. We omit the right-hand side of the definition accordingly. The *update* function takes a heap, a reference, and a value as its three arguments. It returns a new heap, identical to its first argument, except that for the data stored at the argument reference is overwritten. Just as we saw for the definition of the dereferencing operator, we can safely omit the case for the empty heap.

We now have all the pieces in place to assign semantics to *IO* computations. The *run* function below takes a computation of type *IO a s t* and an initial heap of shape *s* as arguments, and returns a pair consisting of the result of the computation and the final heap of shape *t*.

```
run : forall {a s t} → IO a s t → Heap s → Pair a (Heap t)
run (Return x) h       = (x, h)
run (Write loc d io) h = run io (update h loc d)
run (Read loc io) h    = run (io (h ! loc)) h
run (New d io) h       = run (io Top) (Alloc d h)
```

The code here closely follows the partial semantics from Chapter 2. The *Return* constructor simply pairs the result and heap. For the *Write* constructor, we recurse with an appropriately updated heap. In the *Read* case, we lookup the data from the heap and recurse with the same heap. Finally, when a new array is created, we extend the heap with a new value and continue recursively.

The *run* function is total. By parameterising the *IO* type by the shape of the initial and final heap, we can be explicit about the shape of the heaps that the *run* function expects and returns. The corresponding Haskell specification in Chapter 2, on the other hand, did not make this distinction and treated all heaps equally. Furthermore, by parametrising the specification by a universe we can store different types of data on the heap. To accomplish this in Haskell, we needed to use the built-in compiler support for dynamic types.

There is a problem with this specification. We always allocate new values on the top of the heap. By convention, the *Top* constructor always refers to the most recently created reference. This convention creates some problems, as we shall see in the next section.

6.2 Automatic weakening

Previously, we introduced smart constructors to make it easier to write syntactic terms. Suppose, however, we were to define the following smart constructors:

$$\begin{aligned}
&\mathit{newRef} : \textbf{forall}\ \{u\ s\} \rightarrow \mathit{el}\ u \rightarrow \mathit{IO}\ (\mathit{Ref}\ u\ (\mathit{Cons}\ u\ s))\ s\ (\mathit{Cons}\ u\ s)\\
&\mathit{newRef}\ x = \mathit{New}\ x\ \mathit{Return}\\
&\mathit{writeRef} : \textbf{forall}\ \{u\ s\} \rightarrow \mathit{Ref}\ u\ s \rightarrow \mathit{el}\ u \rightarrow \mathit{IO}\ \mathit{Unit}\ s\ s\\
&\mathit{writeRef}\ \mathit{ref}\ x = \mathit{Write}\ \mathit{ref}\ x\ (\mathit{Return}\ \mathit{void})\\
&\mathit{readRef} : \textbf{forall}\ \{u\ s\} \rightarrow \mathit{Ref}\ u\ s \rightarrow \mathit{IO}\ (\mathit{el}\ u)\ s\ s\\
&\mathit{readRef}\ \mathit{ref} = \mathit{Read}\ \mathit{ref}\ \mathit{Return}
\end{aligned}$$

While these definitions are correct, using them can be somewhat problematic. This is best illustrated with an example.

Suppose we have chosen the following universe U as the code for the types we wish to store in memory:

$$\begin{aligned}
&\textbf{data}\ U : \mathit{Set}\ \textbf{where}\\
&\quad \mathit{NAT} : U\\
&\quad \_ \Rightarrow \_ : U \rightarrow U \rightarrow U\\
&\mathit{el} : U \rightarrow \mathit{Set}\\
&\mathit{el}\ \mathit{NAT} = \mathit{Nat}\\
&\mathit{el}\ (u \Rightarrow v) = \mathit{el}\ u \rightarrow \mathit{el}\ v
\end{aligned}$$

Now the following function fails to type check:

$$\begin{aligned}
&f : \mathit{IO}\ \mathit{Unit}\ \mathit{Nil}\ (\mathit{Cons}\ \mathit{NAT}\ (\mathit{Cons}\ \mathit{NAT}\ \mathit{Nil}))\\
&f = \mathit{newRef}\ 1 \gg\!= \lambda \mathit{ref}_1 \rightarrow\\
&\qquad \mathit{newRef}\ 2 \gg\!= \lambda \mathit{ref}_2 \rightarrow\\
&\qquad \mathit{writeRef}\ \mathit{ref}_1\ 3
\end{aligned}$$

The problem lies with the call to *writeRef*. The *WriteRef* constructor requires a reference to a value in a heap of shape *Cons NAT* (*Cons NAT Nil*). Yet the first reference we created, ref_1, is a reference to a value in a heap of shape *Cons NAT Nil*. To fix this, we need to wrap an additional *Pop* constructor around ref_1:

```
f = newRef 1 >>= λref₁ →
      newRef 2 >>= λref₂ →
      writeRef (Pop ref₁) 3
```

While this solution is correct, it can be rather cumbersome to weaken references manually every time new memory is allocated. Fortunately this process can be automated.

6.2.1 Manual weakening

Before we revisit our smart constructors, we need to develop a bit of machinery. We need to decide how many *Pop* constructors are necessary to weaken a reference to denote a position in a larger heap. This corresponds to proving that one shape is a suffix of a larger shape. One way to represent such a proof is using an inductive data type:

```
data IsSuffix : Shape → Shape → Set where
  Base : forall {s} → IsSuffix s s
  Step : forall {u s t} → IsSuffix s t → IsSuffix s (Cons u t)
```

Every proof that s is a suffix of t is built from two constructors *Base* and *Step*. The base case corresponds to stating that every list is a suffix of itself; the step case states that if s is a suffix of t then s is also a suffix of *Cons* u t for any u.

If we have such a proof that s is a suffix of t, we can weaken any reference of type *Ref* u s to make a reference of type *Ref* u t. The weaken function does exactly this:

```
weaken : forall {s t u} → IsSuffix s t → Ref u s → Ref u t
weaken Base ref      = ref
weaken (Step i) ref = Pop (weaken i ref)
```

It proceeds by induction over the proof argument, adding a *Pop* constructor for every step.

While the *weaken* function adds the necessary *Pop* constructors, there is still no way to compute its proof argument automatically. There is an alternative representation of such proofs that does facilitate such automation.

6.2.2 An alternative proof

Suppose that our universe supports the following operation:

$$\mathit{decEqU} : (u : U) \to (v : U) \to \mathit{Either}\ (u \equiv v)\ ((u \equiv v) \to \bot)$$

That is, for every two codes u and v we either prove that u and v are equal or we show that no such proof can exist. Here $\bot$ corresponds to the uninhabited type.

Note that we do not require that the equality on the inhabitants of this universe is decidable. For instance, we can easily decide when two codes of type U in the example universe with natural numbers and the function space are equal; however, there is no there is no algorithm that decides when two functions of type $\mathit{Nat} \to \mathit{Nat}$ are equal.

Using our assumption about our universe of types U, we can define the function with the following type:

$$\begin{aligned}&\mathit{decEqShape} : (s : \mathit{Shape}) \to (t : \mathit{Shape})\\ &\quad \to \mathit{Either}\ (s \equiv t)\ ((s \equiv t) \to \bot)\end{aligned}$$

The definition is fairly unremarkable. We traverse the lists s and t, comparing every element. If the elements of both lists are equal at every position, the entire lists are equal; if the lists have different lengths or store different elements, the two shapes are distinct.

Using this function, we can define the following function that checks whether or not one list is a suffix of another:

$$\begin{aligned}&\_ \leqslant \_ : \mathit{Shape} \to \mathit{Shape} \to \mathit{Bool}\\ &\mathit{Nil} \leqslant s \qquad\qquad = \mathit{True}\\ &\mathit{Cons}\ u\ s \leqslant \mathit{Nil} = \mathit{False}\\ &\mathit{Cons}\ u\ s \leqslant \mathit{Cons}\ v\ t\ \textbf{with}\ \mathit{decEqShape}\ (\mathit{Cons}\ u\ s)\ (\mathit{Cons}\ v\ t)\\ &\mathit{Cons}\ u\ s \leqslant \mathit{Cons}\ \lfloor u \rfloor\ \lfloor s \rfloor \mid \mathit{Inl}\ \mathit{Refl} = \mathit{True}\\ &\mathit{Cons}\ u\ s \leqslant \mathit{Cons}\ v\ t \qquad \mid \mathit{Inr}\ q = (\mathit{Cons}\ u\ s) \leqslant t\end{aligned}$$

Now we can reflect any *Bool* into *Set* as follows:

$$\begin{aligned}&\mathit{So} : \mathit{Bool} \to \mathit{Prop}\\ &\mathit{So}\ \mathit{True}\ = \mathit{Unit}\\ &\mathit{So}\ \mathit{False} = \bot\end{aligned}$$

Finally, we give an alternative definition of the *IsSuffix* predicate: a shape s is a suffix of a shape t if and only if $\mathit{So}\ (s \leqslant t)$ is inhabited. This alternative definition is equivalent to the definition using *IsSuffix*. We can prove the equivalence in one direction as follows:

```
equiv : (s : Shape) → (t : Shape) → So (s ⩽ t) → IsSuffix s t
equiv Nil Nil p = Base
equiv (Cons u s) Nil ()
equiv Nil (Cons v t) p = Step (equiv Nil t p)
equiv (Cons u s) (Cons v t) p with decEqShape (Cons u s) (Cons v t)
equiv (Cons u s) (Cons ⌊u⌋ ⌊s⌋) p | Inl Refl = Base
equiv (Cons u s) (Cons v t) p | Inr q = Step (equiv (Cons u s) t p)
```

The equivalence in the other direction is similar. We have omitted it as we will not need it.

So why go through all this effort to write an equivalent representation of the inductive *IsSuffix* data type? The two forms of proof are useful for different reasons:

IsSuffix s t By defining the proof as an inductive data type, we can pattern match on proofs. We use this to define the *weaken* function. Unfortunately, we need to write an inhabitant of *IsSuffix* by hand to pass to the weaken function.

So (s ⩽ t) By defining the $\leqslant$-operator we have written a function that decides when one list is a suffix of another. In particular, for any pair of *closed* shapes $s \leqslant t$ reduces to either *True* or *False*. Correspondingly, the type $\mathit{So}\ (s \leqslant t)$ is either trivial or uninhabited.

The central idea behind our smarter constructors is to weaken using the *inductive representation*, but to require a *trivial implicit witness*.

6.2.3 Smarter constructors

With this in mind, we revise our smart constructors for reading from and writing to references. These smarter constructors now require an implicit proof that s is a suffix of t. Using our *equiv* function, we can compute the inductive representation of such a proof. Using this inductive representation, we can weaken references as necessary.

$writeRef : \{s : Shape\} \to \{t : Shape\} \to \{p : So\ (s \leqslant t)\} \to \{u : U\} \to$
$\quad Ref\ u\ s \to el\ u \to IO\ Unit\ t\ t$
$writeRef\ \{s\}\ \{t\}\ \{p\}\ \{u\}\ ref\ d = Write\ wkRef\ d\ (Return\ void)$
$\quad$ **where**
$\quad wkRef = weaken\ (equiv\ s\ t\ p)\ ref$
$readRef : \{s : Shape\} \to \{t : Shape\} \to \{p : So\ (s \leqslant t)\} \to \{u : U\} \to$
$\quad Ref\ u\ s \to IO\ (el\ u)\ t\ t$
$readRef\ \{s\}\ \{t\}\ \{p\}\ \{u\}\ ref = Read\ wkRef\ Return$
$\quad$ **where**
$\quad wkRef = weaken\ (equiv\ s\ t\ p)\ ref$

The beauty of this solution is that Agda will automatically instantiate implicit arguments of type *Unit*. In other words, for any closed *IO* term a programmer need not worry about passing proof arguments. For example, we can now write:

$f : IO\ Unit\ Nil\ (Cons\ NAT\ (Cons\ NAT\ Nil))$
$f = newRef\ 1 \gg\!= \lambda ref_1 \to$
$\qquad newRef\ 2 \gg\!= \lambda ref_2 \to$
$\qquad writeRef\ ref_1\ 3$

The smarter constructors will now automatically weaken ref_1 after ref_2 has been allocated.

Unfortunately, not all in the garden is rosy. If we have an open term, Agda will warn you that it cannot find a suitable proof argument. For example, consider the following function that increments the value stored in a reference:

$inc : \{s : Shape\} \to Ref\ NAT\ s \to IO\ Unit\ s\ s$
$inc\ ref = read\ ref \gg\!= \lambda x \to$
$\qquad\qquad write\ ref\ (Succ\ x)$

While any particular call to *inc* is safe, Agda fails to automatically deduce that $So\ (s \leqslant s)$ reduces to the unit type. Even though it is clearly true for any particular choice of s and we can fill in the proof argument manually, the techniques we have outlined here fail to provide the required proof automatically.

Without more language support it is unlikely that we can avoid this restriction. There is still much work to be done in designing some means to customise the inference process of implicit arguments.

6.3 Discussion

Related work There have been several proposals for region-based type systems that strive to provide programmers type safe control of memory management (Tofte and Talpin, 1997; Fluet and Morrisett, 2004). The *Shape* data type presented here closely resembles such regions. More recently, Kiselyov and Shan have used similar techniques to those described in this chapter to embed such region-based type systems in Haskell (Kiselyov and chieh Shan, 2008).

Further work The specification we have sketched here does have its limitations. First of all, although we have parameterised the specification by the universe that represents the types stored in references, we cannot store effectful computations in a reference. We will discuss this point in greater detail in Chapter 10.

Furthermore, the automatic weakening of references requires a decidable equality on our universe. This excludes references storing dependent types, such as dependent pairs or dependent functions. It would be interesting to investigate how to remove this restriction and better support the automatic weakening of functions that are polymorphic with respect to the shape of the heap, such as the *inc* function above.

The interface that our functions provide expose a great deal of information about the structure of the heap. In the future we would like to explore how to hide some of this type information from the user. Haskell has already shown that existentially quantifying over the state safely encapsulates stateful computations (Launchbury and Peyton Jones, 1994). We would like to extend Launchbury and Peyton Jones's parametricity result to our specification.

This chapter has focused on mutable state. Yet we have already seen Haskell specifications in the earlier chapters of this dissertation for other effects, most notably concurrency and teletype I/O. It should certainly be possible to develop similar total specifications for these effects, even if there are some technical problems that must be overcome.

In the case of concurrency, we need to provide a formal account for why our specification terminates. As we saw in the Haskell specification of concurrency in Chapter 3, every time we schedule an active thread it will either be blocked or make progress. If the thread makes progress, we make a recursive call to a structurally smaller computation. If the thread is blocked, however, we make a recursive call to a smaller pool of active threads. Formalising this informal argument in such a way that the specification passes Agda's termination checker is still quite some work.

The specification of teletype I/O gives rise to different problems. We could quite easily give a specification of finite interactions, i.e., those teletype interactions that will only produce finite output and consume finite input. Many teletype interactions, however, are designed to indefinitely produce output. Giving a total specification of such processes would require a better understanding of how to incorporate coinductive data types in type theory which is still a field of active research (Coquand, 1993).

Finally, the specification is fairly low-level. There are many high-level logical systems for reasoning about imperative languages. In Chapter 8 we will show how this specification forms a model of Hoare Type Theory. Before we do so, however, we extend our specification in an entirely different direction.

Chapter 7

Distributed Arrays

Computer processors are not becoming significantly faster. To satisfy the demand for more and more computational power, manufacturers are now assembling computers with multiple microprocessors. It is hard to exaggerate the impact this will have on software development: tomorrow's programming languages must embrace parallel programming on multicore machines (Sutter, 2005).

Researchers have proposed several new languages to maximise the potential speedup that multicore processors offer (Allen et al., 2005; Chamberlain et al., 2005, 2000; Charles et al., 2005; Scholz, 2003). Although all these languages are different, they share the central notion of a *distributed array*, where the elements of an array may be distributed over separate processors or even over separate machines. To write efficient code, programmers must ensure that processors only access *local* parts of a distributed array – it is much faster to access data stored locally than remote data on another core.

When writing such locality-aware algorithms it is all too easy to make subtle mistakes. Programming languages such as X10 require all arrays operations to be local (Charles et al., 2005). Any attempt to access non-local data results in an exception. To preclude such errors, Grothoff et al. (2007) have designed a type system, based on a dependently typed lambda calculus, for a small core language resembling X10 that is specifically designed to guarantee that programs only access local parts of a distributed array. Their proposed system is rather intricate and consists of a substantial number of intricate type rules that keep track of locality information.

In this chapter, we explore an alternative avenue of research. Designing and implementing a type system from scratch is a lot of work. New type systems typically require extensive proofs of various soundness, completeness, principle typing, and decidability theorems. Instead, we show how to tailor a general purpose dependently typed programming language to enforce certain domain-specific properties – resulting in a *domain-specific embedded type system*. We no longer need to prove any meta-theoretical results, but immediately inherit all the desirable properties of our dependently typed host type system, such as decidable type checking and subject reduction. This illustrates how our functional specifications may be extended with domain-specific properties, such as locality constraints.

7.1 Mutable arrays

Before we study distributed arrays, we will briefly describe how the specification from the previous chapter can be adapted to cover mutable arrays. Just as before we will give the specification of three operations: the creation of new arrays; reading from an array; and updating a value stored in an array.

To keep things simple, we will only work with flat arrays storing natural numbers. This is, of course, an oversimplification. Using the universe construction we presented in the previous chapter, however, we could easily give specifications of arrays storing different types. We have chosen to work in this simple setting to accentuate the locality constraints in the next section.

To avoid confusion between numbers denoting the size of an array and the data stored in an array, we introduce the *Data* type synonym. Throughout the rest of this chapter, we will use *Data* to refer to the data stored in arrays; the *Nat* type will always refer to the size of an array.

$$
\begin{aligned}
&\mathit{Data} : \mathit{Set}\\
&\mathit{Data} = \mathit{Nat}
\end{aligned}
$$

Using the *Fin* type, we can give a functional specification of arrays of a fixed size by mapping every index to the corresponding value.

$$
\begin{aligned}
&\mathit{Array} : \mathit{Nat} \rightarrow \mathit{Set}\\
&\mathit{Array}\ n = \mathit{Fin}\ n \rightarrow \mathit{Data}
\end{aligned}
$$

As the heap will store arrays of different sizes, its type should explicitly state how many arrays it stores and how large each array is. To accomplish this, we represent the shape of the heap as a list of numbers:

```
Shape : Set
Shape = List Nat
```

The *Shape* of the heap represents the size of the arrays stored in memory.

Next, we define the *Heap* and *Ref* data types in the same vein as we saw previously:

```
data Heap : Shape → Set where
  Empty : Heap Nil
  Alloc : {n : Nat} → {ns : Shape} →
    Array n → Heap ns → Heap (Cons n ns)
data Ref : Nat → Shape → Set where
  Top : {n : Nat} → {ns : Shape} → Ref n (Cons n ns)
  Pop : forall {n k ns} → Ref n ns → Ref n (Cons k ns)
```

The only differences with the previous chapter are an immediate consequence of our choice of *Shape*. By choosing to store only arrays on the heap, we effectively restrict ourself to the universe where the codes are *Nat* and the decoding function is *Array*.

The constructors for the *IO* data type should be familiar.

```
data IO (a : Set) : Shape → Shape → Set where
  Return : {ns : Shape} → a → IO a ns ns
  Write : forall {n ns ms} →
    Ref n ns → Fin n → Data → IO a ns ms → IO a ns ms
  Read : forall {n ns ms} →
    Ref n ns → Fin n → (Data → IO a ns ms) → IO a ns ms
  New : forall {ns ms} →
    (n : Nat) → (Ref n (Cons n ns) → IO a (Cons n ns) ms) →
    IO a ns ms
```

Note that we require an additional argument to the *Read* and *Write* constructors corresponding to the index in the array to read from or write to respectively. The *New* constructor requires a natural number *n*. It allocates an array of size *n* with all its indices initially set to zero.

Once again, this *IO* data type forms a parameterised monad. We can define *return* and bind operators just as we did in the previous chapter.

Semantics We have described the syntax of array computations using the *IO* data type; now we shall give a functional specification of their behaviour. Recall that we can model arrays as functions from indices to natural numbers:

```
Array : Nat → Set
Array n = Fin n → Data
```

Before specifying the behaviour of *IO* computations, we define several auxiliary functions to update an array and lookup a value stored in an array.

```
lookup : forall {n ns} → Ref n ns → Fin n → Heap ns → Data
lookup Top      i (Alloc a _) = a i
lookup (Pop k) i (Alloc _ h) = lookup k i h
```

The *lookup* function takes a reference to an array *l*, an index *i* in the array at location *l*, and a heap, and returns the value stored in the array at index *i*. It dereferences *l*, resulting in a function of type *Fin n → Data*; the value stored at index *i* is the result of applying this function to *i*.

Next, we define a pair of functions to update the contents of an array.

```
updateArray : {n : Nat} → Fin n → Data → Array n → Array n
updateArray i d a = λj → if i ≡ j then d else a j
updateHeap : forall {n ns} →
  Ref n ns → Fin n → Data → Heap ns → Heap ns
updateHeap Top      i x (Alloc a h) = Alloc (updateArray i x a) h
updateHeap (Pop k) i x (Alloc a h) = Alloc a (updateHeap k i x h)
```

The *updateArray* function overwrites the data stored at a single index. The function *updateHeap* updates a single index of an array stored in the heap. It proceeds by dereferencing the location on the heap where the desired array is stored and updates it accordingly, leaving the rest of the heap unchanged.

Now we can define the *run* function that takes a computation of type *IO a ns ms* and an initial heap of shape *ns* as arguments, and returns a pair consisting of the result of the computation and the final heap of shape *ms*. This specification closely follows the one we have seen in the previous chapter. The only real difference is in the allocation of new arrays. When a new array is created, we extend the heap with a new array that stores *Zero* at every index, and continue recursively.

```
run : forall {a ns ms} → IO a ns ms → Heap ns → Pair a (Heap ms)
run (Return x) h       = (x, h)
run (Read a i rd) h    = run (rd (lookup a i h)) h
run (Write a i x wr) h = run wr (updateHeap a i x h)
run (New n io) h       = run (io Top) (Alloc (λi → Zero) h)
```

Just as we did previously, the *Top* reference refers to the most recently allocated memory. We can define smart constructors to wrap additional *Pop* constructors around existing references.

7.2 Distributed arrays

Arrays are usually represented by a continuous block of memory. *Distributed arrays*, however, can be distributed over different *places* – where every place may correspond to a different core on a multiprocessor machine, a different machine on the same network, or any other configuration of interconnected computers.

We begin by determining the type of places where data is stored and code is executed. Obviously, we do not want to fix the type of all possible places prematurely: you may want to execute the same program in different environments. Yet regardless of the exact number of places, there are certain operations you will always want to perform, such as iterating over all places, or checking when two places are equal.

We therefore choose to abstract over the *number* of places in the module we will define in the coming section. With this in mind, we parameterise over the module we are defining as follows:

```
module DistrArray (placeCount : Nat) where
```

When programmers import the *DistrArray* module, they are obliged to choose the number of places. Typically, there will be one place for every available processor. From this number, we can define a data type corresponding to the available places:

$$\begin{aligned}&\mathit{Place} : \mathit{Set}\\&\mathit{Place} = \mathit{Fin}\ \mathit{placeCount}\end{aligned}$$

The key idea underlying our model of locality-aware algorithms is to index computations by the place where they are executed. The new type declaration for the *IO* monad corresponding to operations on *distributed* arrays will become:

$$\textbf{data}\ \mathit{DIO}\ (a : \mathit{Set}) : \mathit{Shape} \to \mathit{Place} \to \mathit{Shape} \to \mathit{Set}\ \textbf{where}$$

You may want to think of a value of type $\mathit{DIO}\ a\ ns\ p\ ms$ as a computation that can be executed at place p and will take a heap of shape ns to a heap of shape ms, yielding a final value of type a.

We strive to ensure that any well-typed program written in the *DIO* monad will never access data that is not local. The specification of distributed arrays now poses a twofold problem: we want to ensure that the array manipulations from the previous section are 'locality-aware,' that is, we must somehow restrict the array indices that can be accessed from a certain place; furthermore, X10 facilitates several *place-shifting* operations that change the place where certain chunks of code are executed.

Regions and Points Before we define the *DIO* monad, we need to introduce several new concepts. In what follows, we will try to stick closely to X10's terminology for distributed arrays. Every array is said to have a *region* associated with it. A region is a set of valid index points. A *distribution* specifies a place for every index point in a region.

Once again, we will only treat flat arrays storing natural numbers and defer any discussion about how to deal with more complicated data structures for the moment. In this simple case, a region merely determines the size of the array.

$$\begin{aligned}&\mathit{Region} : \mathit{Set}\\&\mathit{Region} = \mathit{Nat}\end{aligned}$$

As we have seen in the previous section, we can model array indices using the *Fin* data type:

$$\begin{aligned}&\mathit{Point} : \mathit{Region} \to \mathit{Set}\\&\mathit{Point}\ n = \mathit{Fin}\ n\end{aligned}$$

To model distributed arrays, we now need to consider the distribution that specifies *where* this data is stored. In line with existing work (Grothoff et al., 2007), we assume the existence of a fixed distribution.

```
postulate
  distr : forall {n ns} → Ref n ns → Point n → Place
```

X10 provides several combinators for defining such distributions. Rather than covering them here, we postulate that some distribution exists for the sake of simplicity.

Syntax We proceed by defining the *DIO* monad:

```
data DIO (a : Set) : Shape → Place → Shape → Set where
```

As it is a bit more complex than the data types we have seen so far, we will discuss every constructor individually.

The *Return* constructor is analogous to one we have seen previously: it lifts any pure value into the *DIO* monad.

```
Return : {p : Place} → {ns : Shape} → a → DIO a ns p ns
```

The *Read* and *Write* operations are more interesting. Although they correspond closely to the operations we have seen in the previous section, their type now keeps track of the place where they are executed. Any read or write operation to point *pt* of an array *l* can *only* be executed at the place specified by the distribution. This invariant is enforced by the types of our constructors:

```
Read : forall {n ns ms} →
  (l : Ref n ns) → (pt : Point n) →
  (Data → DIO a ns (distr l pt) ms) →
  DIO a ns (distr l pt) ms
Write : forall {n ns ms} →
  (l : Ref n ns) → (pt : Point n) → Data →
  DIO a ns (distr l pt) ms →
  DIO a ns (distr l pt) ms
```

In contrast to *Read* and *Write*, new arrays can be allocated at any place.

$$
\begin{array}{l}
\mathit{New} : \textbf{forall}\ \{p\ \mathit{ns}\ \mathit{ms}\} \to \\
\quad (n : \mathit{Nat}) \to (\mathit{Ref}\ n\ (\mathit{Cons}\ n\ \mathit{ns}) \to \mathit{DIO}\ a\ (\mathit{Cons}\ n\ \mathit{ns})\ p\ \mathit{ms}) \to \\
\quad \mathit{DIO}\ a\ \mathit{ns}\ p\ \mathit{ms}
\end{array}
$$

Finally, we add a constructor for the place-shifting operator *At*:

$$
\begin{array}{l}
\mathit{At} : \textbf{forall}\ \{p\ \mathit{ns}\ \mathit{ms}\ \mathit{ps}\} \to \\
\quad (q : \mathit{Place}) \to \mathit{DIO}\ \mathit{Unit}\ \mathit{ns}\ q\ \mathit{ms} \to \mathit{DIO}\ a\ \mathit{ms}\ p\ \mathit{ps} \to \mathit{DIO}\ a\ \mathit{ns}\ p\ \mathit{ps}
\end{array}
$$

The *At* operator lets us execute a computation at another place. As we will discard the result of this computation, we require it to return an element of the unit type.

We can now define appropriate *return* and bind operations for the *DIO* monad.

$$
\begin{array}{l}
\mathit{return} : \{\mathit{ns} : \mathit{Shape}\}\ \{a : \mathit{Set}\}\ \{p : \mathit{Place}\} \to a \to \mathit{DIO}\ a\ \mathit{ns}\ p\ \mathit{ns} \\
\mathit{return}\ x = \mathit{Return}\ x \\
\_ \gg\!= \_ : \{\mathit{ns}\ \mathit{ms}\ \mathit{ks} : \mathit{Shape}\}\ \{a\ b : \mathit{Set}\}\ \{p : \mathit{Place}\} \to \\
\quad \mathit{DIO}\ a\ \mathit{ns}\ p\ \mathit{ms} \to (a \to \mathit{DIO}\ b\ \mathit{ms}\ p\ \mathit{ks}) \to \mathit{DIO}\ b\ \mathit{ns}\ p\ \mathit{ks} \\
(\mathit{Return}\ x) \gg\!= f = f\ x \\
(\mathit{Read}\ l\ i\ \mathit{io}) \gg\!= f = \mathit{Read}\ l\ i\ (\lambda x \to \mathit{io}\ x \gg\!= f) \\
(\mathit{Write}\ l\ i\ x\ \mathit{io}) \gg\!= f = \mathit{Write}\ l\ i\ x\ (\mathit{io} \gg\!= f) \\
(\mathit{New}\ n\ \mathit{io}) \gg\!= f = \mathit{New}\ n\ (\lambda l \to \mathit{io}\ l \gg\!= f) \\
(\mathit{At}\ q\ \mathit{there}\ \mathit{here}) \gg\!= f = \mathit{At}\ q\ \mathit{there}\ (\mathit{here} \gg\!= f)
\end{array}
$$

It is worth noting that the bind operator $\gg\!=$ can only be used to sequence operations at the same place.

Semantics To run a computation in the *DIO* monad, we follow the *run* function defined in the previous section closely.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{run} : \textbf{forall}\ \{a\ \mathit{ns}\ \mathit{ms}\} \to} \\
\multicolumn{2}{l}{\quad (p : \mathit{Place}) \to \mathit{DIO}\ a\ \mathit{ns}\ p\ \mathit{ms} \to \mathit{Heap}\ \mathit{ns} \to \mathit{Pair}\ a\ (\mathit{Heap}\ \mathit{ms})} \\
\mathit{run}\ p\ (\mathit{Return}\ x)\ h & = (x, h) \\
\mathit{run}\ \lfloor \mathit{distr}\ l\ i \rfloor\ (\mathit{Read}\ l\ i\ \mathit{rd})\ h & = \mathit{run}\ (\mathit{distr}\ l\ i)\ (\mathit{rd}\ (\mathit{lookup}\ l\ i\ h))\ h \\
\mathit{run}\ \lfloor \mathit{distr}\ l\ i \rfloor\ (\mathit{Write}\ l\ i\ x\ \mathit{wr})\ h & = \textbf{let}\ h' = \mathit{updateHeap}\ l\ i\ x\ h \\
 & \quad\ \textbf{in}\ \mathit{run}\ (\mathit{distr}\ l\ i)\ \mathit{wr}\ h' \\
\mathit{run}\ p\ (\mathit{New}\ n\ \mathit{io})\ h & = \mathit{run}\ p\ (\mathit{io}\ \mathit{Top})\ (\mathit{Alloc}\ (\lambda i \to \mathit{Zero})\ h) \\
\mathit{run}\ p\ (\mathit{At}\ q\ \mathit{io}_1\ \mathit{io}_2)\ h & = \mathit{run}\ p\ \mathit{io}_2\ (\mathit{snd}\ (\mathit{run}\ q\ \mathit{io}_1\ h))
\end{array}
$$

Our new *run* function, however, must be locality-aware. Therefore, we parameterise the *run* function explicitly by the place where the computation is executed.

Now we can see that the *Read* and *Write* operations may not be executed at *any* place. Recall that the *Read* and *Write* constructors both return computations at the place *distr l i*. When we pattern match on a *Read* or *Write*, we know exactly what the place argument of the *run* function must be. Correspondingly, we do not pattern match on the place argument – we know that the place can only be *distr l i*.

The other difference with respect to the previous *run* function, is the new case for the *At* constructor. In that case, we sequence the two computations io_1 and io_2. To do so, we first execute the io_1 at q, but discard its result; we continue executing the second computation io_2 with the heap resulting from the execution of io_1 at the location p. In line with existing research (Grothoff et al., 2007), we have assumed that io_1 and io_2 are performed synchronously – executing io_1 before continuing with the rest of the computation. Using techniques to model concurrency that we have presented in Chapter 2, we believe we could give a more refined treatment of the X10's *globally asynchronous/locally synchronous* semantics and provide specifications for X10's clocks, *finish*, and *force* constructs.

Locality-aware combinators Using the place-shifting operator *at*, we can define several locality-aware control structures. With our first-class distribution and definition of *Place*, we believe there is no need to define more primitive operations.

We can begin by defining it an auxiliary function, *for*, that iterates over all the indices of an array:

$$\begin{aligned}
&\mathit{for} : \textbf{forall}\ \{n\ ns\ p\} \rightarrow \\
&\quad (\mathit{Point}\ n \rightarrow \mathit{DIO}\ \mathit{Unit}\ ns\ p\ ns) \rightarrow \mathit{DIO}\ \mathit{Unit}\ ns\ p\ ns \\
&\mathit{for}\ \{\mathit{Succ}\ k\}\ \mathit{dio} = \mathit{dio}\ \mathit{Fz} \gg (\mathit{for}\ \{k\}\ (\mathit{dio}\ \circ\ \mathit{Fs})) \\
&\mathit{for}\ \{\mathit{Zero}\}\ \mathit{dio} = \mathit{return}\ \mathit{void}
\end{aligned}$$

Using the *for* function we define a distributed map, that applies a function to all the elements of a distributed array at the place where they are stored:

$$
\begin{array}{l}
\mathit{dmap} : \textbf{forall}\ \{n\ \mathit{ns}\ p\} \rightarrow \\
\quad (\mathit{Data} \rightarrow \mathit{Data}) \rightarrow \mathit{Ref}\ n\ \mathit{ns} \rightarrow \mathit{DIO}\ \mathit{Unit}\ \mathit{ns}\ p\ \mathit{ns} \\
\mathit{dmap}\ f\ l = \mathit{for}\ (\lambda i \rightarrow \mathit{at}\ (\mathit{distr}\ l\ i)\ (\mathit{readArray}\ l\ i \gg\!= \lambda x \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{writeArray}\ l\ i\ (f\ x)))
\end{array}
$$

Besides *dmap*, we implement two other combinators: *forallplaces* and *ateach*. The *forallplaces* operation executes its argument computation at all available places. We define it using the *for* function to iterate over all places. The *ateach* function, on the other hand, is a generalisation of the distributed map operation. It iterates over an array, executing its argument operation once for every index of the array, at the place where that index is stored.

$$
\begin{array}{l}
\mathit{forallplaces} : \textbf{forall}\ \{p\ \mathit{ns}\} \rightarrow \\
\quad ((q : \mathit{Place}) \rightarrow \mathit{DIO}\ \mathit{Unit}\ \mathit{ns}\ q\ \mathit{ns}) \rightarrow \mathit{DIO}\ \mathit{Unit}\ \mathit{ns}\ p\ \mathit{ns} \\
\mathit{forallplaces}\ \mathit{io} = \mathit{for}\ (\lambda i \rightarrow \mathit{at}\ i\ (\mathit{io}\ i)) \\
\mathit{ateach} : \textbf{forall}\ \{n\ \mathit{ns}\ p\} \rightarrow \\
\quad (l : \mathit{Ref}\ n\ \mathit{ns}) \rightarrow ((\mathit{pt} : \mathit{Point}\ n) \rightarrow \mathit{DIO}\ \mathit{Unit}\ \mathit{ns}\ (\mathit{distr}\ l\ \mathit{pt})\ \mathit{ns}) \rightarrow \\
\quad \mathit{DIO}\ \mathit{Unit}\ \mathit{ns}\ p\ \mathit{ns} \\
\mathit{ateach}\ l\ \mathit{io} = \mathit{for}\ (\lambda i \rightarrow \mathit{at}\ (\mathit{distr}\ l\ i)\ (\mathit{io}\ i))
\end{array}
$$

Example: distributed sum We will now show how to write a simple algorithm that sums all the elements of a distributed array. To do so efficiently, we first locally sum all the values at every place. To compute the total sum of all the elements of the array, we add together all these local sums. In what follows, we will need the following auxiliary function, *increment*:

$$
\begin{array}{l}
\mathit{increment} : \textbf{forall}\ \{n\ \mathit{ns}\ p\} \rightarrow \\
\quad (l : \mathit{Ref}\ n\ \mathit{ns}) \rightarrow (i : \mathit{Fin}\ n) \rightarrow \mathit{Nat} \rightarrow (\mathit{distr}\ l\ i \equiv p) \rightarrow \\
\quad \mathit{DIO}\ \mathit{Unit}\ \mathit{ns}\ p\ \mathit{ns} \\
\mathit{increment}\ l\ i\ x\ \mathit{Refl} = \mathit{readArray}\ l\ i \gg\!= \lambda y \rightarrow \mathit{writeArray}\ l\ i\ (x + y)
\end{array}
$$

Note that *increment* is a bit more general than strictly necessary. We could return a computation at *distr l i*, but instead we choose to be a little more general: *increment* can be executed at any place, as long as we have a proof that this place is equal to *distr l i*.

We can use the *increment* function to define a simple sequential *sum* function:

```
sum : forall {n ns p} → Ref n ns → Ref 1 ns → DIO Unit ns p ns
sum l out = ateach l (λi → readArray l i ≫= λn →
                              at (distr out Fz) (increment out Fz n Refl))
```

The *sum* function takes an array as its argument, together with a reference to a single-celled array, *out*. It reads every element of the array, and increments *out* accordingly.

Finally, we can use both these functions to define a parallel sum. The *psum* function takes four arguments: the array *l* whose elements you would like to sum; an array *localSums* that will store the intermediate sums; an assumption regarding the distribution of this array; and finally, the single-celled array to which we write the result.

```
psum : forall {n ns} →
  (l : Ref n ns) → (localSums : Ref placeCount ns) →
  ((i : Place) → distr localSums i ≡ i) →
  (out : Ref 1 ns) → DIO Nat ns (distr out Fz) ns
psum l localSums ldistr out =
  ateach l (λi → (readArray l i ≫= λn →
                  increment localSums (distr l i) n (ldistr (distr l i))))
  ≫ sum localSums out
  ≫ readArray out Fz
```

For every index *i* of the array *l*, we read the value stored at index *i*, and increment the corresponding local sum. We then add together the local sums using our previous sequential *sum* function, and return the final result. We use our assumption about the distribution of the *localSums* array when calling the increment function. Without this assumption, we would have to use the place-shifting operation *at* to update a (potentially) non-local array index.

There are several interesting issues that these examples highlight. First of all, as our *at* function only works on computations returning a unit type, the results of intermediate computations must be collected in intermediate arrays.

More importantly, however, whenever we want to rely on properties of the global distribution, we need to make explicit assumptions in the form of proof arguments. This is rather unfortunate: it would be interesting to research how a specific distribution can be associated with an array when it is created. This would hopefully allow for a more fine-grained treatment of distributions and eliminate the need for explicit proof arguments.

7.3 Discussion

There are clearly several serious limitations of this work as it stands. We have had to make several simplifying assumptions. First and foremost, we have assumed that every array only stores natural numbers, disallowing more complex structures such as multi-dimensional arrays. This can be easily fixed by defining a more elaborate *Shape* data type – as we have seen in the previous chapter.

We could extend our model of the heap even further by having every place maintain its own local heap. As our example in the previous section illustrated, assuming the presence of a global distribution does not scale well. Decorating every array with a distribution upon its creation should help provide locality-information when it is needed.

We have not discussed how code in the *IO* or *DIO* monad is actually compiled. At the moment, Agda can only be compiled to Haskell. Agda does provide several pragmas to customise how Agda functions are translated to their Haskell counterparts. The ongoing effort to support data parallelism in Haskell (Chakravarty et al., 2001, 2007) may therefore provide us with a most welcome foothold.

There are many features of X10 that we have not discussed here at all. Most notably, we have refrained from modelling many of X10's constructs that enable asynchronous communication between locations, even though we would like to do so in the future.

Chapter 8

The Hoare state monad

In the previous chapters, we proposed to use functional specifications to program with and reason about mutable state. This is not the whole story.

There are a great many logics designed to facilitate reasoning about mutable state (Floyd, 1967; Hoare, 1969; Ishtiaq and O'Hearn, 2001; Reynolds, 2002). Compared with such logics, our functional specifications are rather low-level. Inspired by work on Hoare Type Theory (Nanevski and Morrisett, 2005; Nanevski et al., 2006, 2008), we will explore how to reason with Hoare logic in Agda in this chapter. The next chapter will discuss the precise relation with Hoare Type Theory in greater detail.

8.1 A problem of relabelling

Before we demonstrate how to reason with Hoare logic in type theory, we will introduce a verification challenge posed by Hutton and Fulger (2008).

Consider the following Haskell data type:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

We will now show how to use the state monad is to traverse such a tree and assign a unique integer to every leaf. Recall that the state monad has type:

$$\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: (s \to (a, s))\}$$

The return of the state monad leaves the state unchanged; the bind threads the state from the first computation to the next.

Using the state monad, we can define a function *relabel* as follows:

$$
\begin{array}{l}
\mathit{relabel} :: \mathit{Tree}\ a \rightarrow \mathit{State}\ \mathit{Int}\ (\mathit{Tree}\ \mathit{Int}) \\
\mathit{relabel}\ (\mathit{Leaf}\ \_) = \mathbf{do}\ x \leftarrow \mathit{get} \\
\qquad\qquad\qquad\qquad\quad\ \mathit{put}\ (x+1) \\
\qquad\qquad\qquad\qquad\quad\ \mathit{return}\ (\mathit{Leaf}\ x) \\
\mathit{relabel}\ (\mathit{Node}\ l\ r) = \mathbf{do}\ l' \leftarrow \mathit{relabel}\ l \\
\qquad\qquad\qquad\qquad\qquad\ r' \leftarrow \mathit{relabel}\ r \\
\qquad\qquad\qquad\qquad\qquad\ \mathit{return}\ (\mathit{Node}\ l'\ r')
\end{array}
$$

If we encounter a *Leaf*, we use the current state as the new label and increment the state accordingly. In the case for a *Node*, we relabel both subtrees. The bind of thet state monad ensures that the state arising from the first recursive call is passed to the second recursive call.

How can we prove that this function is correct? Before we can talk about correctness, we need to establish the specification that we expect the *relabel* function to satisfy. One way of formulating the desired specification is by defining the following auxiliary functions:

$$
\begin{array}{l}
\mathit{flatten} :: \mathit{Tree}\ a \rightarrow [a] \\
\mathit{flatten}\ (\mathit{Leaf}\ x) \quad = [x] \\
\mathit{flatten}\ (\mathit{Node}\ l\ r) = \mathit{flatten}\ l +\!\!+ \mathit{flatten}\ r \\
\mathit{size} :: \mathit{Tree}\ a \rightarrow \mathit{Int} \\
\mathit{size}\ (\mathit{Leaf}\ x) \qquad = 1 \\
\mathit{size}\ (\mathit{Node}\ l\ r) \quad = \mathit{size}\ l + \mathit{size}\ r \\
\mathit{sequence} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow [\mathit{Int}] \\
\mathit{sequence}\ x\ n = \mathbf{if}\ n \leqslant 0\ \mathbf{then}\ [\,] \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ x : \mathit{sequence}\ (x+1)\ (n-1)
\end{array}
$$

We can define the desired property of our *relabel* function as follows:

$$
\mathit{flatten}\ (\mathit{evalState}\ (\mathit{relabel}\ t)\ i) = \mathit{sequence}\ i\ (\mathit{size}\ t)
$$

That is, flattening a tree labelled with the initial integer i should produce the sequence $[i \ldots i + \mathit{size}\ t - 1]$. The usual way to reason about functions in the state monad is to desugar the do-notation and expand the definitions of return and bind. Hutton and Fulger (2008) propose this problem as a benchmark of high-level reasoning techniques about state and give an equational proof. Their proof, however, revolves around defining an intermediate function:

$$label' :: Tree\ a \to State\ [b]\ (Tree\ b)$$

The *label′* function carries around an (infinite) list of fresh labels that are used to relabel the leaves of the argument tree. To prove that *label* meets the above specification, Hutton and Fulger need to prove various lemmas relating *label* and *label′*. It is not clear how to extend their technique to other functions in the state monad, in contrast to the approach we will take in this chapter.

8.2 A direct proof

Before we can attempt to prove any property of the *relabel* function, we must define both it and its specification in Agda. For the most part, this simply consists of transcribing the functions *size*, *flatten*, and *sequence* from the previous section in Agda.

Now rather than write a simply-typed *relabel* function that happens to satisfy the desired specification, we will engineer a dependently typed definition that satisfies the specification by construction. The *type* of the relabel function carries information about its *behaviour*.

We choose the following type for the relabelling function:

$$\begin{aligned}
&relabel : \mathbf{forall}\ \{a\} \to Tree\ a \to (i : Nat) \to \\
&\quad \{\,(t,f) : Pair\ (Tree\ Nat)\ Nat \mid \\
&\qquad f \equiv i + size\ t \wedge flatten\ t \equiv sequence\ i\ (size\ t)\,\}
\end{aligned}$$

Recall that we use the notation $\{\,x : a \mid P\ x\,\}$ to denote a dependent pair of a value x of type a together with a proof that x satisfies some property P.

Given a tree and initial natural number i, our relabelling function returns a relabeled tree and new number f. The final tree and output number should satisfy a post-condition. At this point, we are free to choose the post-condition as we see fit: when defining the function we must then provide the proof that our definition satisfies our choice of post-condition.

Besides the obvious condition that $flatten\ t \equiv sequence\ i\ (size\ t)$, we also require a second condition: the final state f should be exactly $size\ t$ larger than the initial state i. The importance of this tighter specification will become apparent once we try to prove that the relabelling function satisfies the first part of the specification.

The code for the *relabel* is in Listing 21; several auxiliary definitions and lemmas can be found in Listing 19 and Listing 20.

Before we cover it in greater detail, it is worth looking at the computational fragment. Recall that *Satisfies*, the constructor of a dependent pair, is defined in Section 5.4. If we omit the proofs – that is, the second component of the dependent pairs – we have this simple definition left:

$$\begin{array}{l} \mathit{relabel}\ (\mathit{Leaf}\ \_)\ s = (\mathit{Leaf}\ s, \mathit{Succ}\ s) \\ \mathit{relabel}\ (\mathit{Node}\ l\ r)\ s_1\ \textbf{with}\ \mathit{relabel}\ l\ s_1 \\ \mathit{relabel}\ (\mathit{Node}\ l\ r)\ s_1\ |\ (l', s_2)\ \textbf{with}\ \mathit{relabel}\ r\ s_2 \\ \mathit{relabel}\ (\mathit{Node}\ l\ r)\ s_1\ |\ (l', s_2)\ |\ (r', s_3) = (\mathit{Node}\ l'\ r', s_3) \end{array}$$

This definition closely follows what you might write in Haskell without using the state monad. Instead of repeating the exact same left-hand side of a function definition when using the with-rule, Agda allows us to replace these left-hand sides with an ellipsis which we do in Listing 21.

The proof that this definition satisfies the desired post-condition is relatively straightforward. In the base case, we need to prove the following statement:

$$\mathit{Succ}\ s \equiv s + 1 \land \mathit{Cons}\ s\ \mathit{Nil} \equiv \mathit{Cons}\ s\ \mathit{Nil}$$

The first conjunct follows from the commutativity of addition; the proof of the second conjunct is trivial.

The case for nodes is a bit trickier. We need to provide a proof of the following proposition:

$$\begin{array}{l} s_3 \equiv s_1 + \mathit{size}\ l' + \mathit{size}\ r' \\ \quad \land \mathit{flatten}\ l' +\!\!+ \mathit{flatten}\ r' \equiv \mathit{sequence}\ s_1\ (\mathit{size}\ l' + \mathit{size}\ r') \end{array}$$

After applying our induction hypotheses, the first conjunct reduces to:

$$(s_1 + \mathit{size}\ l') + \mathit{size}\ r' \equiv s_1 + (\mathit{size}\ l' + \mathit{size}\ r')$$

Which is follows immediately from the associativity of addition. Similarly the second conjunct reduces to:

$$\begin{array}{l} \mathit{sequence}\ s_1\ (\mathit{size}\ l') +\!\!+ \mathit{sequence}\ s_2\ (\mathit{size}\ r') \\ \quad \equiv \mathit{sequence}\ s_1\ (\mathit{size}\ l' + r') \end{array}$$

Listing 19 Auxiliary definitions

```
subst : {a : Set} → {x y : a} → (P : a → Prop) → x ≡ y → P x → P y
subst P Refl px = px
cong : {a b : Set} → {x y : a} → (f : a → b) → x ≡ y → f x ≡ f y
cong f Refl = Refl
trans : {a : Set} → {x y z : a} → x ≡ y → y ≡ z → x ≡ z
trans Refl Refl = Refl
sym : {a : Set} → {x y : a} → x ≡ y → y ≡ x
sym Refl = Refl
data _∧_ (p : Prop) (q : Prop) : Prop where
  _,_ : p → q → p ∧ q
infix 10 _∧_
assocAdd : (x y z : Nat) → (x + y) + z ≡ x + (y + z)
assocAdd Zero y z = Refl
assocAdd (Succ k) y z = cong Succ (assocAdd k y z)
succLemma : (n m : Nat) → n + Succ m ≡ Succ (n + m)
succLemma Zero m = Refl
succLemma (Succ k) m = cong Succ (succLemma k m)
plusZero : (n : Nat) → n ≡ n + 0
plusZero Zero = Refl
plusZero (Succ k) = cong Succ (plusZero k)
commAdd : (x y : Nat) → x + y ≡ y + x
commAdd m Zero = sym (plusZero m)
commAdd x (Succ y) =
  trans (succLemma x y) (cong Succ (commAdd x y))
```

Listing 20 The *seqLemma*

```
seqLemma : (x y z : Nat) →
  sequence x y ++ sequence (x + y) z ≡ sequence x (y + z)
seqLemma x Zero z with (x + Zero) | plusZero x
seqLemma x Zero z | ⌊x⌋ | Refl = Refl
seqLemma x (Succ k) z = cong (λxs → Cons x xs) ih
  where
  ih : sequence (Succ x) k ++ sequence (x + Succ k) z
      ≡ sequence (Succ x) (k + z)
  ih with x + Succ k | succLemma x k
  ih | ⌊Succ x + k⌋ | Refl = seqLemma (Succ x) k z
```

We now need to use our assumption that $s_2 \equiv$ *size* l'. Without this stronger post-condition it is impossible to complete the proof. After this last step, we complete the proof using the auxiliary lemma about *sequence* in Listing 20.

This, however, does not complete our original goal: a verified monadic relabelling function that does not expand the definitions of return and bind.

8.3 The Hoare state monad

One might define the state monad in Haskell as follows:

```
type State s a = s → (a, s)
```

This does not suit our purposes: we want to write a verified program by construction. We need to refine this definition slightly, and define a slightly different monad *HoareState*, directly inspired by the work on Hoare Type Theory (Nanevski and Morrisett, 2005; Nanevski et al., 2006, 2008).

A computation of type *HoareState* P a Q does not accept any state of type s, but instead requires the initial state to satisfy the precondition P. Given an initial state i that satisfies the precondition, the computation will return a value x of type a and final state f such that the predicate $Q\ i\ x\ f$ holds. Bearing these two points in mind, we arrive at the following definition of our augmented state monad over some state of type S:

Listing 21 The relabelling function

```
relabel : forall {a} → Tree a → (i : Nat) →
      {(t,f) : (Pair (Tree Nat) Nat) |
        f ≡ i + size t ∧ flatten t ≡ sequence i (size t)}
relabel (Leaf _) s = Satisfies (Leaf s, Succ s) prfLeaf
  where
  prfLeaf : Succ s ≡ s + 1 ∧ Cons s Nil ≡ Cons s Nil
  prfLeaf =  (subst (λn → Succ s ≡ n) Refl (commAdd 1 s), Refl)
relabel (Node l r) s1 with relabel l s1
... | Satisfies (l′, s2) (sizeL, treeL) with relabel r s2
... | Satisfies (r′, s3) (sizeR, treeR) = Satisfies (Node l′ r′, s3) prfNode
  where
  prfSize : s3 ≡ s1 + size l′ + size r′
  prfSize with s3 | s2 | sizeR | sizeL
  prfSize | ⌊(s1 + size l′) + size r′⌋ | ⌊s1 + size l′⌋ | Refl | Refl
     = assocAdd s1 (size l′) (size r′)
  prfTree : flatten l′ ++ flatten r′ ≡ sequence s1 (size l′ + size r′)
  prfTree with flatten l′ | flatten r′ | treeL | treeR
  prfTree | ⌊sequence s1 (size l′)⌋ | ⌊sequence s2 (size r′)⌋ | Refl | Refl
    with s2 | sizeL
  ... | ⌊s1 + size l′⌋ | Refl = seqLemma s1 (size l′) (size r′)
  prfNode = (prfSize, prfTree)
```

$$\begin{aligned}
&\mathit{Pre} : \mathit{Set1}\\
&\mathit{Pre} = S \to \mathit{Prop}\\
&\mathit{Post} : \mathit{Set} \to \mathit{Set1}\\
&\mathit{Post}\ a = S \to a \to S \to \mathit{Prop}\\
&\mathit{HoareState} : \mathit{Pre} \to (a : \mathit{Set}) \to \mathit{Post}\ a \to \mathit{Set}\\
&\mathit{HoareState}\ \mathit{pre}\ a\ \mathit{post} = \ \{ i : S \mid \mathit{pre}\ i \} \ \to \{ (x,f) : \mathit{Pair}\ a\ S \mid \mathit{post}\ i\ x\ f \}
\end{aligned}$$

Note that a postcondition is a property that may refer to the input state, result, and output state.

We refer to this as the Hoare state monad as it enables reasoning about computations in the state monad using Hoare logic.

We still need to define the *return* and *bind* functions for the Hoare state monad. The *return* function does not place any restriction on the input state, it simply returns its argument leaving the state intact:

$$\begin{aligned}
&\mathit{top} : \mathit{Pre}\\
&\mathit{top} = \lambda s \to \mathit{Unit}\\
&\mathit{return} : \{ a : \mathit{Set} \} \to (y : a) \to \mathit{HoareState}\ \mathit{top}\ a\ (\lambda i\ x\ f \to i \equiv f \wedge x \equiv y)\\
&\mathit{return}\ y\ (\mathit{Satisfies}\ s\ \_) = \mathit{Satisfies}\ (y, s)\ (\mathit{Refl}, \mathit{Refl})
\end{aligned}$$

Here the precondition *top* places no constraints on the input state. Note that the definition of the *return* of the Hoare state monad is identical to the corresponding definition of the state monad: we have only made its behaviour evident from its type.

The *bind* of the Hoare state monad is a bit more subtle. Recall that the bind of a monad *m* has the following type:

$$m\ a \to (a \to m\ b) \to m\ b$$

We would expect the definition of the bind of the Hoare state monad to have the type:

$$\mathit{HoareState}\ P_1\ a\ Q_1 \to (a \to \mathit{HoareState}\ P_2\ b\ Q_2) \to \mathit{HoareState} \ldots b \ldots$$

Before we consider the precondition and postcondition of the resulting computation, we note that we can generalise this slightly. In the above type signature, the second argument of *bind* is not dependent. We can be slightly more general and parameterise P_2 and Q_2 by the result of the first computation:

$$\begin{aligned}
&\mathit{HoareState}\ P_1\ a\ Q_1\\
&\quad \to ((x : a) \to \mathit{HoareState}\ (P_2\ x)\ b\ (Q_2\ x))\\
&\quad \to \mathit{HoareState} \ldots b \ldots
\end{aligned}$$

Now we need to choose suitable preconditions and postconditions for the composite computation returned by the *bind* function.

The bind of the state monad is defined as follows:

$$\begin{aligned}
c_1 \ggg\!= c_2 = \lambda s_1 \to\ &\mathbf{case}\ c_1\ s_1\ \mathbf{of}\\
&\quad (x, s_2) \to c_2\ x\ s_2
\end{aligned}$$

The bind of the state monad starts by running the first computation, and subsequently feeds its result to the second computation. So clearly the precondition of the composite computation should imply the precondition of the first computation c_1, otherwise we could not justify running c_1 with the initial state s. Furthermore the postcondition of the first computation should imply the precondition of the second computation. If this wasn't the case, the call to c_2 would be unjustifiable. These considerations lead to the following choice of precondition for the composite computation:

$$(\lambda s_1 \to P_1\ s_1 \wedge ((x : a) \to (s_2 : S) \to Q_1\ s_1\ x\ s_2 \to P_2\ x\ s_2))$$

In words, the precondition of the composite computation is the precondition P_1 of the first computation together with a proof that any state s_2 and result x satisfying the postcondition of the first computation, Q_1, also satisfy the precondition P_2 of the second computation.

What about the postcondition? Recall that a postcondition is a predicate on the initial state, resulting value, and the final state. We would expect the postcondition of the second argument computation to hold after executing the composite computation. However, we also know that the postcondition of the first computation holds for the intermediate state, after the first computation has been completed. To record this information in the postcondition of the composite computation, we existentially quantify over the results of first computation, yielding the following postcondition for the *bind* operation:

$$(\lambda s_1\ x\ s_3 \to \{(y, s_2) : \mathit{Pair}\ a\ S \mid Q_1\ s_1\ y\ s_2 \wedge Q_2\ y\ s_2\ x\ s_3\}))$$

In words, the postcondition of the composite computation states that there is an intermediate state s_2 and a value y resulting from the first computation, such that these satisfy the postcondition of the first computation Q_1.

Listing 22 The bind of the Hoare state monad

$$
\begin{array}{l}
\mathit{bind} : \textbf{forall}\ \{a\ b\ P_1\ Q_1\} \rightarrow \{P_2 : a \rightarrow \mathit{Pre}\} \rightarrow \{Q_2 : a \rightarrow \mathit{Post}\ b\} \rightarrow \\
\quad \mathit{HoareState}\ P_1\ a\ Q_1 \rightarrow \\
\quad ((x : a) \rightarrow \mathit{HoareState}\ (P_2\ x)\ b\ (Q_2\ x)) \rightarrow \\
\quad \mathit{HoareState} \\
\qquad (\lambda s_1 \rightarrow P_1\ s_1 \wedge ((x : a) \rightarrow (s_2 : S) \rightarrow Q_1\ s_1\ x\ s_2 \rightarrow P_2\ x\ s_2)) \\
\qquad b \\
\qquad (\lambda s_1\ x\ s_3 \rightarrow \{(y, s_2) : \mathit{Pair}\ a\ S \mid Q_1\ s_1\ y\ s_2 \wedge Q_2\ y\ s_2\ x\ s_3\}) \\
\mathit{bind}\ c_1\ c_2\ (\mathit{Satisfies}\ s_1\ \mathit{pre})\ \textbf{with}\ c_1\ (\mathit{Satisfies}\ s_1\ (\mathit{fst\ pre})) \\
...\mid (\mathit{Satisfies}\ (x, s_2)\ p)\ \textbf{with}\ c_2\ x\ (\mathit{Satisfies}\ s_2\ (\mathit{snd\ pre}\ x\ s_2\ p)) \\
...\mid (\mathit{Satisfies}\ (y, s_3)\ q) = \mathit{Satisfies}\ (y, s_3)\ (\mathit{Satisfies}\ (x, s_2)\ (p, q))
\end{array}
$$

Furthermore, the postcondition of the second computation Q_2 relates these intermediate results and the final state s_3 and final value x.

Once we have chosen the desired precondition and postcondition of *bind* we can define the *bind* in Listing 22. If we ignore all the propositional information, however, this definition is identical to the bind of the state monad. The definition of bind runs the first computation c_1 against the initial state s_1. The resulting value x and state s_2 are passed to the second computation c_2. The overhead compared with the usual definition of bind is largely a consequence of the pre- and postconditions.

In a similar vein, we can define the operations *get* and *put* that respectively read and overwrite the current state:

$$
\begin{array}{l}
\mathit{get} : \mathit{HoareState\ top}\ S\ (\lambda i\ x\ f \rightarrow i \equiv f \wedge x \equiv i) \\
\mathit{get}\ (\mathit{Satisfies}\ s\ \_) = \mathit{Satisfies}\ (s, s)\ (\mathit{Refl}, \mathit{Refl}) \\
\mathit{put} : (s : S) \rightarrow \mathit{HoareState\ top\ Unit}\ (\lambda i\ x\ f \rightarrow f \equiv s) \\
\mathit{put}\ s\ \_ = \mathit{Satisfies}\ (\mathit{void}, s)\ \mathit{Refl}
\end{array}
$$

The *get* and *put* functions have trivial preconditions. The postcondition of the *get* function guarantees to leave the state intact and return the current state. The postcondition of the *put* function on the other hand states that the final state is equal to its first argument.

8.4 Relabelling revisited

Having defined the Hoare state monad, we can now write a different version of the relabelling function that more closely resembles the original Haskell definition. We begin by defining the postcondition that our relabelling function must satisfy:

$$
\begin{aligned}
&\mathit{relabelPost} : \mathit{Post}\ (\mathit{Tree}\ \mathit{Int})\\
&\mathit{relabelPost} = \lambda i\ t\ f \to f \equiv i + \mathit{size}\ t\\
&\qquad\qquad\qquad\qquad\qquad \wedge \mathit{flatten}\ t \equiv \mathit{sequence}\ i\ (\mathit{size}\ t)
\end{aligned}
$$

Using the Hoare state monad, the definition of our relabel function now becomes:

$$
\begin{aligned}
&\mathit{relabel} : \textbf{forall}\ \{a\} \to \mathit{Tree}\ a \to \mathit{HoareState}\ \mathit{top}\ (\mathit{Tree}\ \mathit{Nat})\ \mathit{relabelPost}\\
&\mathit{relabel}\ (\mathit{Leaf}\ \_) =\\
&\quad \mathit{get} \ggg\!= \lambda \mathit{fresh} \to\\
&\quad \mathit{put}\ (\mathit{Succ}\ \mathit{fresh}) \gg\\
&\quad \mathit{return}\ (\mathit{Leaf}\ \mathit{fresh})\\
&\mathit{relabel}\ (\mathit{Node}\ l\ r) =\\
&\quad \mathit{relabel}\ l \ggg\!= \lambda l' \to\\
&\quad \mathit{relabel}\ r \ggg\!= \lambda r' \to\\
&\quad \mathit{return}\ (\mathit{Node}\ l'\ r')
\end{aligned}
$$

Unfortunately, this definition is still flawed. There are two reasons that Agda does not accept this definition.

First of all, the preconditions and postconditions of our definition do not match the precondition and postcondition that we have assigned our *relabel* function. For example, the precondition of the *Leaf* branch is:

$$
\begin{aligned}
&(\mathit{Pair}\ \mathit{Unit}\ ((n : \mathit{Nat})\ (h : \mathit{Nat}) \to i \equiv h \wedge n \equiv i \to\\
&\quad \mathit{Pair}\ \mathit{Unit}\ ((n' : \mathit{Unit})\ (h' : \mathit{Nat}) \to (h' \equiv \mathit{Succ}\ n) \to \mathit{Unit})))
\end{aligned}
$$

As we have used the bind operator to construct composite computations, this is reflected in the type of the conditions Agda infers. Fortunately, we can solve this problem by explicitly strengthening the precondition and weakening the postcondition. In line with Hoare Type Theory, we name this operation *do*:

```
do : forall {a P1 P2 Q1 Q2} →
  ((i : S) → P2 i → P1 i) →
  ((i : S) → (x : a) → (f : S) → P2 i → Q1 i x f → Q2 i x f) →
  HoareState P1 a Q1 → HoareState P2 a Q2
do str wkn c (Satisfies i p) with c (Satisfies i (str i p))
... | (Satisfies (a,f) q) = Satisfies (a,f) (wkn i a f p q)
```

The *do* function takes three arguments: a proof that the precondition P_2 is stronger than the precondition P_1; a proof that the postcondition Q_2 is weaker than the postcondition Q_1; and a computation of type *HoareState* P_1 *a* Q_2. The *do* function uses these proof arguments to construct a computation of type *HoareState* P_2 *a* Q_2 that has the same computational behaviour as the original computation.

Although it has no computational content, the *do* function is necessary to ensure that the inferred pre- and postconditions match the ones stated in the type signature of the *relabel* function.

The second problem with this definition is more subtle and harder to resolve. Let's have a closer look at the arguments to the bind operator:

```
bind : forall {a b P1 Q1} →
  {P2 : a → Pre} → {Q2 : a → Post b} →
  HoareState P1 a Q1 →
  ((x : a) → HoareState (P2 x) b (Q2 x)) →
  ...
```

When we sequence two computations using the bind operator, it must infer how to instantiate the implicit arguments. For some arguments, such as P_1 and Q_1, this is not a problem: they are uniquely determined by the type of the first computation. Unfortunately, this is not the case for P_2 and Q_2. For example, consider the following code fragment:

```
get ≫= λfresh →
put (Succ fresh)
```

Although Agda knows the types of *get* and *put* it cannot infer how to generalise the pre- and postcondition of *put* to instantiate the appropriate arguments of *bind*. Filling these arguments in manually however pollutes our code with implicit arguments of little relevance. This is a rather unfortunate limitation of Agda's inference mechanism for implicit arguments.

8.5 Discussion

The two issues outlined in the previous section make it rather difficult to complete the version of the relabelling function that uses the Hoare state monad. This is a shortcoming of Agda, that has not been designed for this style of programming. In comparison, the entire proof is less than 120 lines in Coq using Sozeau's Program tactic (Sozeau, 2007). We have refrained from presenting that proof here, as it would require the reader to learn yet another proof assistant.

We believe that the Hoare state monad presented in this chapter has much wider applications: it seems to be a useful abstraction to facilitate the writing of verified stateful programs.

Related work Outside of Hoare Type Theory, similar techniques have been used by others to incorporate some form of pre- and postcondition reasoning in a proof assistant. Leroy (2006) has developed similar techniques in the Compcert project. His solution revolves around defining an auxiliary data type:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Res}\ \{s : \mathit{Set}\}\ (a : \mathit{Set})\ (x : s) : \mathit{Set} := \\
\quad \mathit{Error} : \mathit{Res}\ a\ x \\
\quad \mathit{OK} : a \to \textbf{forall}\ (y : s), R\ x\ y \to \mathit{Res}\ a\ x.
\end{array}
$$

Here *R* is some relation between states. Unfortunately, the bind of this monad yields less efficient extracted code, as it requires an additional pattern match on the *Res* resulting from the first computation. Furthermore, the Hoare state monad presented here is slightly more general as its postcondition may also refer to the result of the computation.

Cock et al. (2008) have used a similar monad in the verification of the seL4 microkernel. There are a few differences between their monad and the one presented here. Firstly, we have chosen the postconditions to be ternary relations between the initial state, result, and final state. As a result, we do not need to introduce 'ghost variables' to relate intermediate results. Furthermore, their rules are presented as predicate transformers, using Isabelle/HOL's verification condition generator to infer the weakest precondition of a computation.

Chapter 9

Implementing Hoare Type Theory

The functional specifications we have seen so far are not the only way to incorporate effects in type theory. Hoare Type Theory (Nanevski and Morrisett, 2005; Nanevski et al., 2006, 2008) takes a different approach from the functional specifications we have seen in the previous chapters. In particular, Hoare Type Theory adds new axioms to the type theory in which we work. These axioms postulate the existence of primitive effectful functions – much in the same way primitive functions are introduced in Haskell. For example, there are three axioms that postulate that the three functions for manipulating mutable state from Chapter 6 exist. As we shall see, the types of these functions carry all the information about how they behave. This still makes it possible to reason with such primitive functions, even if they have no definition in the type theory.

In this chapter we will give a pure implementation of some of the functions postulated by Hoare Type Theory. By implementing these functions we no longer need to worry about whether postulating their existence results in an inconsistent type theory, a point we will discuss in greater detail in Section 9.3.

9.1 An overview of Hoare Type Theory

Hoare Type Theory extends a 'vanilla' type theory, such as the Calculus of Inductive Constructions (Coquand and Huet, 1988) or Martin-Löf's Theory

Listing 23 The *ST* type of Hoare Type Theory

```
postulate
  Loc : Set
data Dyn : Set where
  dyn : (a : Set) → a → Dyn
Heap : Set
Heap = Loc → Maybe Dyn
Pre : Set
Pre = Heap → Prop
Post : Set → Set
Post a = Heap → a → Heap → Prop
postulate
  ST : Pre → (a : Set) → Post a → Set
```

of Types (Martin-Löf, 1984), with several new constructs. Listing 23 gives an overview of the new types and memory model; Listing 24 contains the functions of Hoare Type Theory that we will implement in this chapter. Before discussing our implementation, however, we will briefly explain the types and functions in these two listings.

First of all, Hoare Type Theory constructs a simple memory model. Rather than fixing the type of locations in memory to be natural numbers, it is kept abstract. The heap itself is a function from such locations to *Maybe Dyn*, where *Dyn* is a pair of a type *a* and a value of type *a*. These dynamic types replace the explicit universe we have seen previously.

Besides postulating some type of locations, Hoare Type Theory assumes the existence of a type constructor *ST*. You may want to think of a value of type *ST P a Q* as a computation that, provided the heap satisfies the precondition *P*, will produce a value of type *a* and a heap satisfying the postcondition *Q*. The definitions of *Pre* and *Post* are identical to those we have seen in the previous chapter where the state has been instantiated to the *Heap* type.

There are several functions that can be used to yield programs with this *ST* type. Their type signatures, together with several auxiliary definitions

Listing 24 The core definitions and postulates of Hoare Type Theory.

```
data Sigma (a : Set) (b : a → Set) : Set where
  Exists : (x : a) → b x → Sigma a b

Refs : Loc → (a : Set) → a → Pre
Refs l a x = λh → h l ≡ Just (dyn a x)

RefsType : Loc → Set → Pre
RefsType l a = λh → Sigma a (λx → Refs l a x h)

IsRef : Loc → Pre
IsRef l = λh → Sigma Dyn (λd → h l ≡ Just d)

update : forall {a} → Heap → Loc → a → Heap
update {a} h l x = λl′ → if l ≡ l′ then Just (dyn a x) else h l′

postulate
  return : {a : Set} → (x : a) → ST top a (λi y f → f ≡ i ∧ y ≡ x)
  bind : forall {a b P₁ Q₁} {P₂ : a → Pre} {Q₂ : a → Post b} →
    ST P₁ a Q₁ → ((x : a) → ST (P₂ x) b (Q₂ x)) →
    ST (λs₁ → P₁ s₁ ∧ (forall x s₂ → Q₁ s₁ x s₂ → P₂ x s₂))
       b
       (λs₁ x s₃ → {(y, s₂) : Pair a Heap | Q₁ s₁ y s₂ ∧ Q₂ y s₂ x s₃})
  new : {a : Set} → (x : a) →
    ST top Loc (λi l f → i l ≡ Nothing ∧ f ≡ update i l x)
  read : (a : Set) → (l : Loc) →
    ST (RefsType l a) a
       (λi x f → f ≡ i ∧ ((y : a) → Refs l a y i → y ≡ x))
  write : (a : Set) → (l : Loc) → (x : a) →
    ST (IsRef l) Unit (λi x f → f ≡ update i l x)
```

are in Listing 24. Besides the monadic functions we have seen previously, there are three functions to read from, write to, and create mutable references. Each of these functions returns a value in the *ST*, with pre- and postconditions explaining how they effect the heap.

New references are created using the *new* function. It has a trivial precondition and returns a fresh location. Its postcondition states that the memory location returned was previously unallocated and the new heap is identical to the previous heap, except that the freshly allocated location.

The *read* and *write* functions have more interesting preconditions. The *read* function requires a location that references a value of type *a* and returns the value stored at that location, while leaving the heap intact. The *write* function takes a valid reference of any type and a value of any type as arguments. It overwrites its argument reference with its second argument, returning the unit type. Note that the value written to the reference need not have the same type as the value currently stored there, a property sometimes referred to as *strong updates*.

Hoare Type Theory has been implemented as an axiomatic extension to Coq (Nanevski et al., 2008). There are also similar functions to free memory, throw and handle exceptions, and a fixpoint combinators. We will not deal with these just yet, and defer the discussion to the end of this chapter. Besides the version we present here, there is also a richer version designed to support separation logic (Nanevski et al., 2006).

9.2 An implementation of Hoare Type Theory

One way to prove that Hoare Type Theory is consistent is by providing a definition for all the functions in Listings 23 and 24 in Agda, thereby constructing a model in Agda's underlying type theory.

First of all, note that the Hoare state monad from the previous chapter provides the *ST* type and the monadic functions *return* and *bind*, together with the *do* operator to strengthen preconditions and weaken postconditions. All we need to do is choose a suitable state to use for the Hoare state monad and implement the remaining functions.

In our model we diverge a bit from Hoare Type Theory in our choice of heap. Instead of choosing a function $Loc \rightarrow Maybe\ Dyn$, we choose a simpler *List Dyn*. By doing so, we will avoid running into the limitations of the intensional type theory underlying Agda and Coq.

Furthermore, we do not use the same definition of *Dyn*. The definition of *Dyn* used in Hoare Type Theory is justified by a pen and paper model construction (Petersen et al., 2008). Simply transcribing the definition from Hoare Type Theory to Agda will introduce a technical problem. There is type theoretic equivalent of Russell's paradox known as Girard's paradox (Girard, 1972; Coquand, 1986). It revolves around constructing a 'type of all types,' much as Russell's paradox constructs a set of all sets. To preserve a consistency many type theories maintain that the type of all types should be itself not be a type. For this reason, the type of *Set* in Agda is a new type Set_1, which itself has type Set_2, and so forth.

To accommodate different types in our references, we therefore parameterise our implementation of Hoare Type Theory by a universe *U*:

$$\textbf{module}\ \mathit{HTT}\ (U : \mathit{Set})\ (\mathit{el} : U \rightarrow \mathit{Set})\ \textbf{where}$$

Using this universe, we can define *Dyn*, the heap, and locations as follows:

$$\begin{aligned}
&\mathit{Dyn} : \mathit{Set}\\
&\mathit{Dyn} = \mathit{Sigma}\ U\ \mathit{el}\\
&\mathit{Heap} : \mathit{Set}\\
&\mathit{Heap} = \mathit{List}\ \mathit{Dyn}\\
&\mathit{Loc} : \mathit{Set}\\
&\mathit{Loc} = \mathit{Nat}
\end{aligned}$$

In line with the functional specification we have seen in Chapter 2 we use natural numbers to represent locations in the heap.

As we have chosen to model the heap slightly differently using a list rather than a function, we will implement a slight adaptation of the functions and types of Hoare Type Theory that we saw in the previous section. To begin with, a location is a valid reference if it is less than the length of the heap:

$$\begin{aligned}
&\mathit{IsRef} : \mathit{Loc} \rightarrow \mathit{Heap} \rightarrow \mathit{Set}\\
&\mathit{IsRef}\ l\ h = \mathit{So}\ (l < \mathit{length}\ h)
\end{aligned}$$

Recall that $\mathit{So} : \mathit{Bool} \rightarrow \mathit{Set}$ from Chapter 5 reflects booleans as types, mapping *True* to the unit type and *False* to the empty type.

Before we give the definition of the other predicates from Listing 24, we need to define the following *lookup* function:

$$
\begin{aligned}
&\mathit{lookup} : (l : \mathit{Loc}) \to (h : \mathit{Heap}) \to \mathit{IsRef}\ l\ h \to \mathit{Dyn}\\
&\mathit{lookup}\ \mathit{Zero}\ \mathit{Nil}\ ()\\
&\mathit{lookup}\ \mathit{Zero}\ (\mathit{Cons}\ x\ \mathit{xs})\ p = x\\
&\mathit{lookup}\ (\mathit{Succ}\ \_)\ \mathit{Nil}\ ()\\
&\mathit{lookup}\ (\mathit{Succ}\ k)\ (\mathit{Cons}\ x\ \mathit{xs})\ p = \mathit{lookup}\ k\ \mathit{xs}\ p
\end{aligned}
$$

Here we use the proof *IsRef l h* to kill off the unreachable branches.

Using this function, we can define the *RefsType* predicate that states a location *l* is a valid reference of type *u* in the heap *h*:

$$
\begin{aligned}
&\mathit{RefsType} : \mathit{Loc} \to U \to \mathit{Heap} \to \mathit{Set}\\
&\mathit{RefsType}\ l\ u\ h = \mathit{Sigma}\ (\mathit{IsRef}\ l\ h)\ (\lambda p \to \mathit{fst}\ (\mathit{lookup}\ l\ h\ p) \equiv u)
\end{aligned}
$$

Here we use the proof that the location is a valid reference in order to perform the *lookup* on the heap to find its type.

Besides looking up values, we can also update values on the heap:

$$
\begin{aligned}
&\mathit{update} : (l : \mathit{Loc}) \to (h : \mathit{Heap}) \to (d : \mathit{Dyn}) \to \mathit{IsRef}\ l\ h \to \mathit{Heap}\\
&\mathit{update}\ \mathit{Zero}\ \mathit{Nil}\ d\ ()\\
&\mathit{update}\ \mathit{Zero}\ (\mathit{Cons}\ x\ \mathit{xs})\ d\ p = \mathit{Cons}\ d\ \mathit{xs}\\
&\mathit{update}\ (\mathit{Succ}\ k)\ \mathit{Nil}\ d\ ()\\
&\mathit{update}\ (\mathit{Succ}\ k)\ (\mathit{Cons}\ x\ \mathit{xs})\ d\ p = \mathit{Cons}\ x\ (\mathit{update}\ k\ \mathit{xs}\ d\ p)
\end{aligned}
$$

A call to *update l h d p* proceeds by traversing the heap, replacing the value stored at location *l* with the dynamic value *d*. Note that there is no relation between the type of the previous and new value on the heap at location *l*.

We now define *read*, *write*, and *new* functions in Listing 26. To complete these definition we require several small lemmas in Listing 25. The proofs of all these lemmas are reassuringly simple: they all proceed by straightforward induction. The definitions in Listing 26 require a bit more explanation.

Compared to the functions postulated in Listing 24, the types of the functions in Listing 26 are rather verbose. This is largely a result of our choice to model the heap as a list rather than a function. Instead of a simple application, looking up the value stored on the heap requires a proof

Listing 25 Auxiliary lemmas necessary for the definitions in Listing 26.

```
lengthLemma : {a : Set} → (xs : List a) → (y : a)
   → So (length xs < length (xs ++ Cons y Nil))
lengthLemma Nil y = void
lengthLemma (Cons x xs) y = lengthLemma xs y

leqIrref : (n : Nat) → So (n < n) → ⊥
leqIrref Zero ()
leqIrref (Succ k) p = leqIrref k p

surjectivePairing : forall {a b} →
  (x : Sigma a b) → (Exists (fst x) (snd x)) ≡ x
surjectivePairing (Exists a b) = Refl

leqLt : (n m : Nat) → So (n < m) → So (n ⩽ m)
leqLt Zero Zero p = void
leqLt Zero (Succ _) p = void
leqLt (Succ k) Zero ()
leqLt (Succ k) (Succ y) p = leqLt k y p

lookupLast : forall h x →
  lookup (length h) (h ++ Cons x Nil) (lengthLemma h x) ≡ x
lookupLast Nil x = Refl
lookupLast (Cons y ys) x = lookupLast ys x

lookupInit : forall h d l p q →
  lookup l (h ++ Cons d Nil) p ≡ lookup l h q
lookupInit Nil d Zero p ()
lookupInit (Cons y ys) d Zero p q = Refl
lookupInit Nil d (Succ y) p ()
lookupInit (Cons y ys) d (Succ k) p q = lookupInit ys d k p q
```

Listing 26 Implementing *read*, *write*, and *new*

```
read : forall {u} → (l : Loc) →
  HoareState (RefsType l u) (el u)
    (λi x f → i ≡ f
              ∧ (Sigma (IsRef l i) (λp → Exists u x ≡ lookup l i p)))
read l (Satisfies heap (Exists p Refl)) =
  Satisfies (snd dyn, heap) (Refl, Exists p (surjectivePairing dyn))
  where
  dyn = lookup l heap p
write : forall {u} → (l : Loc) → (x : el u) →
  HoareState (λi → IsRef l i) Unit
    (λi _ f → Sigma (IsRef l i) (λp → f ≡ update l i (Exists u x) p))
write {u} l x (Satisfies h p) =
  Satisfies (void, update l h (Exists u x) p) (Exists p Refl)
new : forall {u} → (x : el u) →
  HoareState top Loc
    (λi l f → (IsRef l i → ⊥)
      ∧ (forall l p q → lookup l f p ≡ lookup l i q)
      ∧ (Sigma (So (l < length f)) (λp → lookup l f p ≡ Exists u x)))
new {u} x (Satisfies h p) =
  Satisfies (length h, h ++ (Cons dyn Nil)) prfs
  where
  dyn = Exists u x
  isFreshPrf = leqIrref (length h)
  initPrf = lookupInit h dyn
  lastPrf = Exists (lengthLemma h dyn) (lookupLast h dyn)
  prfs = (isFreshPrf, initPrf, lastPrf)
```

that the location being dereferenced is indeed valid. Passing these proofs around introduces a bit of overhead.

The *read* function calls the *lookup* function we defined previously. This returns a pair of a dynamic value. We project out the second component and return it together with the unchanged heap. The postcondition requires a proof that the heap is untouched and that the value returned is the result of performing a lookup with the argument location. Proving both these statements is trivial.

Implementing the *write* function is similar. We compute a new heap by calling the *update* function. The only interesting part of the postcondition states that the new heap is the result of appropriately updating the initial heap. As our implementation closely follows this specification, this proof is simply *Refl*.

The *new* function is a bit more complicated. We extend the heap by adding a new element to the end. Besides the extended heap, we need to return a location, i.e., a natural number, that indicates where to find the latest addition to the heap. As we have added a new element to the end of the heap *h*, the number we return is *length h*.

The postcondition of the *new* function consists of three conjuncts: the new location must be fresh; existing locations must not be overwritten; dereferencing the new location must yield the appropriate value. Each of these individual propositions requires a lemma that can be found in Listing 25. The proof of all three lemmas proceeds by straightforward induction.

9.3 Comparison

There are several important differences between the functional specifications presented in this thesis and the axiomatic approach to effects as set forth by Hoare Type Theory.

The functional specifications in this dissertation are defined in the type theory. This is a mixed blessing. On the one hand, we are bound by theory in which we work; on the other, the specifications are no different from any other function in our theory.

Hoare Type Theory takes a different approach. It postulates the existence of new type formers and functions, thereby extending the type theory.

Hence some form of justification is required to guarantee that the extended theory is still consistent. The flip side is that it is usually much easier to formulate new axioms than to implement them in the type theory.

There are clear advantages to working with functional specifications. As they have a definition in the type theory, they can be used to test code automatically, as we have seen in the earlier chapters of this dissertation. This is particularly valuable when you are unsure about which properties should hold. Such automatic testing can provide a sanity check before you decide to invest a great deal of effort into proving a property, that may turn out to be false.

There is a second advantage to our functional specifications that is a bit more subtle. Proofs in type theory can be very hard. The only thing you ever get 'for free' is that definitions automatically expand. Unfortunately, the Hoare Type Theory postulates have no associated definition. Despite their richer logic, this may make some programs and proofs more difficult. Furthermore, once you postulate the existence of the *ST* type, the only way to prove new properties of this type is by adding new postulates. There is no definition you can work with; inhabitants of the *ST* type do not compute to values.

Hoare Type Theory, on the other hand, is much more readily extensible. For example, it postulates a fixpoint combinator with the following type:

$$\begin{aligned}&(((x : a) \to ST\ (p\ x)\ (b\ x)\ (q\ x)) \to ((x : a) \to ST\ (p\ x)\ (b\ x)\ (q\ x)))\\ &\quad \to (x : a) \to ST\ (p\ x)\ (b\ x)\ (q\ x)\end{aligned}$$

In general, such fixed point combinators lead to an inconsistent type theory. In this instance, however, because the combinator is limited to the *ST* type it does not introduce such an inconsistency.

This an important drawback of the functional specifications. It is quite difficult to define a fixed point combinator within the type theory. Although there are recent proposals to specify fixed point combinators using coinduction (Capretta, 2005), these proposals entail much more effort than formulating a postulate.

When Hoare Type Theory introduces new postulates, there must be some justification. After all, the postulates could lead to an inconsistent type theory. One way to prove that the postulates preserve consistency is to define the *ST* type and its associated primitive functions in our type theory. If you are only interested in proving consistency of your axioms, you could define the *ST* type as follows:

$$\begin{array}{l} \mathit{ST} : \mathit{Pre} \rightarrow (a : \mathit{Set}) \rightarrow \mathit{Post}\ a \rightarrow \mathit{Set} \\ \mathit{ST}\ \mathit{pre}\ a\ \mathit{post} = \mathit{Unit} \end{array}$$

Now it is easy to give a trivial implementation of a fixed point combinator and all the other functions we have seen in this chapter.

Such a simple choice for the *ST* type does have its disadvantages. While it does show that the postulated functions do not give rise to an inconsistent type theory, it creates a semantic gap between the actual definition of *ST* and its intended meaning. For instance, we could now define an operation:

$$\begin{array}{l} \mathit{oops} : \mathit{ST}\ \mathit{top}\ \mathit{Unit}\ (\lambda i\ x\ f \rightarrow \bot) \\ \mathit{oops} = \mathit{void} \end{array}$$

Using this function we can show that a computation in the *ST* type satisfies any postcondition. Although the type theory remains consistent, the predicates attached to the *ST* type have become meaningless. To make matters worse, we can prove that any pair of computations that inhabit the *ST* type are equal. Clearly, such trivial implementations of the *ST* type have enormous drawbacks.

By starting from a definition in the type theory, as our functional specifications do, it becomes much harder to introduce undesirable primitives. It is impossible to provide an implementation for the *oops* function in the functional specification we have seen in this chapter.

As an alternative to such a trivial model, there have been recent pen-and-paper proofs of consistency (Petersen et al., 2008). Such proofs require substantial effort. If you decide to add a new axiom at a later date, you are required to redo the entire proof. Such a proof is far removed from the programming language. If nothing else, this seems to go against the spirit of type theory: should the language of proofs and programs not coincide?

These points are all quite subtle. We need more experience with the two approaches to find some middle ground, combining the benefits of both Hoare Type Theory and the functional specifications.

Chapter 10

Discussion

10.1 Overview

What have we achieved in this thesis? Before discussing any further work and conclusions, I will briefly summarise the individual chapters.

Chapter 2 presents functional specifications of several different effects in Haskell. These functional specifications are useful for debugging and testing impure code. In particular, we have shown how such functional specifications may be used in tandem with tools such as QuickCheck to check properties of effectful functions.

The individual functional specifications in Chapter 2 are described in isolation. We have shown how the syntax of different effects may be combined (Chapter 3). By interpreting the syntax in terms of a fixed virtual machine, we avoid having to consider combining the semantics of different effects.

Using this approach, it can be straightforward to define new specifications that run on this virtual machine. We have demonstrated this in Chapter 4, where we present a functional specification of software transactional memory.

Although these functional specifications presented in Haskell are certainly useful for debugging and testing, there is a problem. The specifications we have presented are not *total*: the programmer may access unallocated memory and type casts may fail. This makes them inappropriate for formal reasoning.

We show how the functional specification of mutable state can be made total in the dependently typed programming language Agda. Chapter 5 introduces Agda; the specification is given in Chapter 6. This is total functional specification of mutable state is one of the central results of this dissertation.

Besides ensuring totality, dependent types may be used to enforce domain specific invariants. This is illustrated in Chapter 7, where we show how to constrain array access operations in order to write efficient distributed algorithms.

The last chapters of this dissertation relate these results to recent work on Hoare Type Theory. Chapter 8 shows how the return and bind of the Hoare Type can be implemented in Agda by decorating the state monad with propositional information. We use this to complete our implementation of Hoare Type Theory in Chapter 9.

10.2 Further work

This thesis is by no means the final word on effects in type theory. There are plenty of directions for further research. Although some individual chapters discussed limitations and further work, there are a few issues I would like to mention separately.

Higher-order store Our functional specification of mutable state in Chapter 6 is parameterised by a universe representing the types of those values that may be stored in references. It is, however, not possible to store computations themselves in references.

The reason for this is rather subtle. Recall that the *Read* constructor of the *IO* type in Chapter 6 had the following type:

$$\textbf{forall}\ \{s\ t\ u\} \rightarrow \mathit{Ref}\ u\ s \rightarrow (\mathit{el}\ u \rightarrow \mathit{IO}\ a\ s\ t) \rightarrow \mathit{IO}\ a\ s\ t$$

Now if the decoding function *el* were allowed to produce values of type *IO*, this would introduce a negative occurrence of the data type we were defining. Such negative occurrences can be used to write arbitrary fixed-point combinators, leading to an inconsistent type theory. Negative data types are forbidden in Agda. This complication should not come as a surprise: Landin observed that if you can store procedures in the store, you can write generally recursive functions.

To allow procedures to be stored in references, we need to develop a stratified model of the *IO* type. In such a model, we would construct a sequence of *IO* types. The first *IO* type stores values on the heap; the second type may stores procedures of the first *IO* type on the heap; the third *IO* type may store procedures of the second *IO* type on the second heap; etc. By doing this, we avoid the negative occurrence in the *Read* constructor we saw above. This solution is reminiscent of the stratified universe of types that many type theories support, e.g., in Agda *Set* has type Set_1, Set_1 has type Set_2, and so forth. It would be interesting to investigate how to reduce the overhead that such a model would introduce, making programming with this advanced model no more difficult than with the specification we have seen already.

Case studies Although this thesis has developed the framework in which we can reason about effectful programs, we have yet to apply them to substantial examples. There are plenty of well-established pointer algorithms, such as the Schorr-Waite marking algorithm (Schorr and Waite, 1967) or the implementation of queues from Chapter 2, that would make interesting case studies.

Other effects The second half of this dissertation focused on the semantics of mutable state. The earlier chapters established functional specifications in Haskell of various other effects, including concurrency and software transactional memory. Solidifying these specifications in a dependently typed language and proving that they can be made total is an important next step.

Verifying specifications Although we have defined semantics for teletype IO, mutable state, and concurrency, we do not know if our specifications are a faithful representation of the real side-effects in Haskell. We need to guarantee that that the semantics we have presented here can actually be trusted.

We could try prove that our semantics are equivalent to those presented by Peyton Jones (Peyton Jones, 2001). This would certainly reaffirm our conviction that these functional semantics are correct. This still does not guarantee that our specifications are semantically equivalent to the code produced by a Haskell compiler, but merely proves the two sets of semantics are equivalent.

An alternative approach would be to describe how Haskell compiles to code for some low-level machine code. We could then compare the behaviour of the primitive *readIORef* with the *readIORef* we have defined. If these two are semantically equivalent on the machine level, we know that it is safe to reason using the functions we have defined. Hutton and Wright take a very similar approach to proving the correctness of a compiler for a simple language with exceptions (Hutton and Wright, 2004).

Combining specifications In Chapter 3 we discussed techniques that can be used to combine the syntax of effectful operations. These compound syntactical terms could then be executed on a fixed virtual machine. We did not, however, discuss how to combine the specifications themselves. Combining monadic semantics is a notoriously hard problem that is still subject to active research (King and Wadler, 1992; Liang et al., 1995; Lüth and Ghani, 2002). Yet many programs rely on a combination of several different effects. Further research is warranted, both in the study of modular semantics and in the design of dependently typed programming languages, before it is possible to assemble more complex specifications from those presented in this dissertation.

10.3 Conclusions

In the first chapter we formulated the four theses that this dissertation defends:

- Functional specifications can provide a formal semantics of the 'Awkward Squad.'
- To write precise functional specifications, we need to work in a programming language with dependent types.
- These specifications are a viable alternative to the axiomatic approach to side effects that is put forward by Hoare Type Theory and Haskell.
- Such functional specifications may provide the interface to the real world for tomorrow's dependently typed programming languages.

In this final section, I will reflect on these four points one last time.

There are many different functional specifications presented in this thesis: mutable state, concurrency, teletype I/O, and software transactional memory (Chapter 2 and Chapter 4). Although there are some members of the 'Awkward Squad' that we have not covered, such as asynchronous exceptions or the foreign function interface, the functional specifications presented here capture a wide spectrum of different effects. This provides evidence supporting our first claim: functional specifications can provide a formal semantics of effects.

The functional specifications written in Haskell are not total. Therefore they are unsuitable for the formal verification of impure programs. As this dissertation shows, this partiality is not an inherent shortcoming but merely highlights the limitations of Haskell's type system. This dissertation demonstrates that the specification of mutable state can be made total in Agda: by programming in a language with dependent types we can make the totality of our specifications manifest.

Hoare Type Theory extends the ambient type theory with new postulates. The implementation of Hoare Type Theory presented in the latter chapters of this thesis shows that this is not always necessary. Although it is more work to write a functional specification than formulate a postulate, the result is more rewarding: the corresponding functional specification is executable and guaranteed to preserve consistency.

There is much work to be done before the functional specifications presented in this dissertation may be used by laymen to verify complex effectful programs. These specifications do, however, provide the foundations on which richer logics can be built. Such functional specifications are a first step towards effectful dependently typed programming, but certainly not the last.

Bibliography

Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification. Technical report, Sun Microsystems, Inc., 2005.

Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, 2003.

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453–468, 1999.

Robert Atkey. Parameterised notions of computation. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, 2006.

Steve Awodey. *Category Theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006.

Arthur I. Baars and S. Doaitse Swierstra. Typing Dynamic Typing. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 2002.

Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

Richard Bird. Functional Pearl: Unfolding pointer algorithms. *Journal of Functional Programming*, 11(3):347–358, 2001.

Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.

Ana Bove and Peter Dybjer. Dependent types at work. In *Lecture Notes for the LerNET Summer School*, 2008.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

Manuel Chakravarty, editor. *The Haskell 98 Foreign Function Interface 1.0*, 2002. An Addendum to the Haskell 98 Report.

Manuel M.T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal – Nested Data-Parallelism in Haskell. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference*, volume LNCS 2150, 2001.

Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, 2007.

Brad Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. Chapel. Technical report, Cray Inc., 2005.

Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3), 2000.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005.

James Cheney and Ralf Hinze. A Lightweight Implementation of Generics and Dynamics. In Manuel Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM-Press, October 2002.

Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3):313–323, May 1999.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

Thierry Coquand. Infinite objects in type theory. In *Types for Proofs and Programs (TYPES '93)*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

Thierry Coquand. An analysis of Girard's Paradox. In *In Symposium on Logic in Computer Science*, pages 227–236, 1986.

Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

John Cupitt. A brief walk through KAOS. Technical Report 58, Computing Laboratory, University of Kent, 1989.

Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19, 1967.

Matthew Fluet and Greg Morrisett. Monadic regions. In *ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 103–114, New York, NY, USA, 2004. ACM. ISBN 1-58113-905-5.

Neil Ghani, Peter Hancock, and Dirk Pattinson. Continuous Functions on Final Coalgebras. *Electronic Notes in Theoretical Computer Science*, 164(1): 141–155, 2006.

J.Y. Girard. *Interpretation fonctionnelle et elimination des coupures de l'arithmetique d'orde superieur*. PhD thesis, Paris VII, 1972.

Andrew D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Prentice Hall, 1996.

Christian Grothoff, Jens Palsberg, and Vijay Saraswat. Safe arrays via regions and dependent types. Submitted for publication, 2007.

Peter Hancock. *Ordinals and Interactive Programs*. PhD thesis, School of Informatics at the University of Edinburgh, 2000.

Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *Springer Lecture Notes in Computer Science*, pages 317–331, 2000.

Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

William L. Harrison. Cheap (but functional) threads. Submitted to Journal of Functional Programming, 2005.

William L. Harrison. The Essence of Multitasking. In Michael Johnson and Varmo Vene, editors, *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2006.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Sören Holmström. PFL: A Functional Language for Parallel Programming. In *Declarative Programming Workshop*, pages 114–139, 1983.

Liyang Hu and Graham Hutton. Towards a Verified Implementation of Software Transactional Memory. In *Proceedings of the Symposium on Trends in Functional Programming*, 2008.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

Graham Hutton and Diana Fulger. Reasoning about effects: Seeing the wood through the trees. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*, 2008.

Graham Hutton and Joel Wright. Compiling Exceptions Correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*. Springer, 2004.

S.S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. *ACM SIGPLAN Notices*, 36(3):14–26, 2001.

Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. In *Proceedings of the 2nd Workshop on Mathematicall Structured Functional Programming*, 2008.

Kent Karlsson. Nebula: A Functional Operating System. Technical report, Chalmers University of Technology, 1981.

David J. King and Philip Wadler. Combining monads. In John Launchbury and Patrick M. Sansom, editors, *Proceedings of the Glasgow Workshop on Functional Programming*, pages 134–143, Glasgow, 1992. Springer.

Oleg Kiselyov and Chung chieh Shan. Lightweight monadic regions. In *Haskell '08: Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 1–12, 2008. ISBN 978-1-60558-064-7.

P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3), 1965.

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. *SIGPLAN Not.*, 29(6):24–35, 1994.

Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, 1999.

Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL '06: 33rd Symposium on Principles of Programming Languages*, pages 42–54, 2006.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, 1995.

Andres Löh and Ralf Hinze. Open data types and open functions. In *Principles and Practice of Declarative Programming*, 2006.

Christoph Lüth and Neil Ghani. Composing monads using coproducts. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 2002.

Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Leving. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

James McKinna. *Deliverables: a categorical approach to program development in type theory*. PhD thesis, School of Informatics at the University of Edinburgh, 1992.

James McKinna and Joel Wright. A type-correct, stack-safe, provably correct, expression compiler in Epigram. Accepted for publication in the Journal of Functional Programming, 2006.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture*, 1991.

Microsoft. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

Eugenio Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, 1989.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007.

Peter Morris, Thorsten Altenkirch, and Neil Ghani. A universe of strictly positive families. *Theory of Computation*, 2007.

Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICPF '06: Proceedings of the Eleventh ACM SIGPLAN Internation Conference on Functional Programming*, 2006.

Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the Twelfth ACM SIGPLAN Internation Conference on Functional Programming*, 2008.

George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

Ulf Norell. Dependently typed programming in Agda. In *6th International School on Advanced Functional Programming*, 2008.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Nicolas Oury and Wouter Swierstra. The Power of Pi. In *ICFP '08: Proceedings of the Twelfth ACM SIGPLAN Internation Conference on Functional Programming*, 2008.

S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

Nikolaos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *In Proceedings of the 3rd Panhellenic Logic Symposium*, 2001.

R.L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A Realizability Model of Impredicative Hoare Type Theory. In *Proceedings of the European Symposion on Programming*, 2008.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*. IOS Press, 2001.

Simon Peyton Jones. Beautiful concurrency. In Andy Oram and Greg Wilson, editors, *Beautiful Code*. O'Reilly, 2007.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, 1996.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN Internation Conference on Functional Programming*, 2006.

Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 71–84, 1993.

John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, 2002.

Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 1967.

Matthieu Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.

Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

Wouter Swierstra and Thorsten Altenkirch. Beauty in the Beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, 2007.

Wouter Swierstra and Thorsten Altenkirch. Dependent types for distributed arrays. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*, 2008.

Simon Thompson. Interactive functional programs. In *Research topics in functional programming*, pages 249–285, 1990.

Mads Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb 1997.

David Turner. Functional programming and communicating processes. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 54–74. Springer-Verlag, 1987.

Janis Voigtländer. Asymptotic improvement of computations over free monads. In Christine Paulin-Mohring and Philippe Audebaud, editors, *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, pages 388–403. Springer-Verlag, 2008.

Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990.

Philip Wadler. The essence of functional programming. In *POPL '92: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1992.

Philip Wadler. The Expression Problem. E-mail message available from `http://homepages.inf.ed.ac.uk/wadler/papers/expression/`, 1998.