High-Level Abstractions for Low-Level Programming

Iavor Sotirov Diatchki
M.Sc. Computer Science, OGI School of Science & Engineering
at Oregon Health & Science University (2007)
B.Sc. Mathematics and Computer Science,
University of Cape Town (1999)

A dissertation presented to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science

May 2007

The dissertation "High-Level Abstractions for Low-Level Programming" by Iavor Sotirov Diatchki has been examined and approved by the following Examination Committee:

Dr. Mark P. Jones, Associate Professor
Dept. of Computer Science and Engineering
Thesis Research Adviser

Dr. Andrew Tolmach, Associate Professor
Dept. of Computer Science
Portland State University

Dr. Greg Morrisett, Professor
Division of Engineering and Applied Science
Harvard University

Dr. Peter A. Heeman, Assistant Professor
Dept. of Computer Science and Engineering

Acknowledgments

The writing of this dissertation would not have been possible without the help of many people, and I would like to thank all of them for their continuous support. In particular:

- Many thanks to my family—my mum, my dad, and my sister—for always believing in me, and for exposing me to the world.

- Special thanks to my advisor, Mark P. Jones, whose help and guidance have been invaluable, and are really appreciated. I am looking forward to many future collaborations!

- Thanks to all my friends in Portland for their support, and for reminding me to have fun.

- Last but not least, many thanks to the faculty and staff at OGI for creating an excellent research environment, and for always being ready to help.

This work was supported, in part, by the National Science Foundation award number 0205737, "ITR: Advanced Programming Languages for Embedded Systems."

Contents

List of Figures

Abstract

High-Level Abstractions for Low-Level Programming
Iavor Sotirov Diatchki

Ph.D., OGI School of Science & Engineering
at Oregon Health & Science University
May 2007

Thesis Advisor: Dr. Mark P. Jones

Computers are ubiquitous in modern society. They come in all shapes and sizes: from standard household appliances and personal computers, to safety and security critical applications such as vehicle navigation and control systems, bank ATMs, defense applications, and medical devices. Modern programming languages offer many features that help developers to increase their productivity and to produce more reliable and flexible systems. It is therefore somewhat surprising that the programs that control many computers are written in older, less robust languages, or even in a lower-level assembly language. This situation is the result of many factors, some entirely non-technical. However, at least in part, the problem has to do with genuine difficulties in matching the results and focus of programming language research to the challenges and context of developing systems software.

This dissertation shows how to extend a modern statically-typed functional programming language with features that make it suitable for solving problems that are common in systems programming. Of particular interest is the problem of manipulating data with rigid representation requirements in a safe manner. Typically, the constraints on the representation of the data are imposed by an external specification such as an operating system binary interface or the datasheet for a hardware device.

The design provides support for two classes of datatypes whose representation is under programmer control. The first class consists of datatypes that are stored in bit fields and accessed as part of a single machine word. Standard examples can be found in operating system APIs, in the control register formats that are used by device drivers, in multimedia and compres-

sion codecs, and in programs like assemblers and debuggers that work with machine code instruction encodings. The second class consists of datatypes that require a fixed representation in regions of memory. Often these 'memory areas' are specific to a particular processor architecture, hardware device, or OS kernel.

The approach builds upon well established programming language technology, such as type inference, polymorphism, qualified types, and the use of monads to control the scope of effects.

Chapter 1

Introduction

1.1 Overview

Computers are ubiquitous in modern society. They come in all shapes and sizes: from standard household appliances and personal computers, to safety and security critical applications such as vehicle navigation and control systems, bank ATMs, defense applications, and medical devices. Modern programming languages offer many features that could potentially help developers to increase their productivity and to produce more reliable and flexible systems. For example, module systems help to manage the complexity of large projects; type systems can be used to detect bugs at compile-time; and automatic storage management techniques eliminate a common source of errors. It is therefore somewhat surprising that the programs that control many computers are written in older, less robust languages, or even in a lower-level assembly language.

This situation is the result of many factors, some entirely non-technical. However, we believe that at least part of the problem has to do with genuine difficulties in matching the results and focus of programming language research to the challenges and context of developing systems software. Other projects have already explored the potential for using higher-level languages for lower-level programming: a small sample includes the Fox Project [40], Ensemble [61], Cyclone [46], and Timber [52]. Based on these, and on our own experience using Haskell [76] to develop device drivers and an operating system kernel [39], we have noticed that high-level language designs sometimes omit important functionality that is needed to program at the level

of hardware interfaces, kernel data structures, and operating system APIs. We have therefore been working to identify the gaps in functionality more precisely, and to investigate the design of language features that might fill them.

1.2 Systems Programming

So, what do we mean by *systems software*? To answer this question, we need to examine the structure of a typical software system. Usually, such systems are designed using a layered approach. Each layer contains functionality that abstracts from the details of the layers below it, thus providing a nicer environment for the layers above it. In this way, the layered approach to software development enables us to split complex systems into simpler components (for an example, see Figure 1.1). If, in addition, we also have well defined interfaces between the layers, then we gain the further advantage that we can reuse layers among different systems.

There are many concrete examples of layered systems, and we can view the layers at different degrees of granularity. For example, if we take a coarse view, we may split most software systems into two parts: (i) an operating system (e.g., Linux), which abstracts hardware details and provides an 'idealized' view of the machine; and (ii) application software (e.g., an Internet browser), which performs specific user tasks, and is usually hardware independent. Each of these layers itself consists of a number of layers. For example, file systems abstract from the details of storage devices, while window managers enable us to multiplex many displays on a single physical screen.

The different layers in the system have rather different goals. The lower layers typically interact with various hardware devices, while the upper layers tend to deal with more abstract tasks, such as performing complex computations, or interacting with people. To reflect these differences, it is common to use the term *systems programming* to describe writing software for the lower layers in a system, while the term *application programming* describes the implementation of the higher layers.

At present, there is a difference in the tools that are used to write applications and systems software. Advanced programming languages are gaining popularity in the implementation of applications software but systems software is still written in fairly low-level languages that do not utilize modern language technology. At least in part, this is because advanced program-

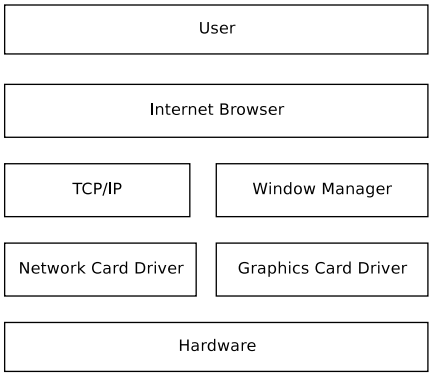


Figure 1.1: Example of a layered system.

ming languages often lack good support for writing systems software. This is unfortunate, because the correctness of the systems programs is of critical importance to the operation of the entire system. We should therefore use all available tools, including advanced programming languages, to make the construction of correct systems software simpler and less error prone.

Because of its low-level nature, systems software is often written in a style that differs from application software. An important difference between the two is that application software often has a lot of freedom in choosing the representation of the data that it manipulates, while systems software usually has to follow fairly rigid constraints dictated by the design of the hardware, or by the format of low-level communication protocols. In the following sections, we will describe a number of concrete examples that illustrate the different kinds of data that typical systems programs manipulate. We do not assume any specific details of these structures. Our only purpose in describing them is to highlight the kinds of issues that arise when we write systems programs.

1.2.1 Bitdata

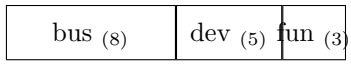
A common aspect of many system programs is that they need to manipulate data that is stored in bit fields and accessed as part of a single machine word. Standard examples can be found in operating system APIs; in the control register formats that are used by device drivers; in multimedia and compression codecs; and in programs like assemblers and debuggers that work with machine code instruction encodings. We will refer to examples like this

collectively as *bitdata*.

Much of the time, the specific bit patterns that are used in bitdata encodings are determined by external specifications and standards to which the systems programmer must conform. For example, an operating system standard will often fix a particular encoding for the set of flags that are passed to a system call, while the datasheet for a particular hardware device will specify the layout of the fields in a particular control register. In the general case, bit-level encodings may use *tag bits*—that is, specific patterns of 0s and 1s in certain positions—to distinguish between different types of value, leaving the bits that remain to store the actual data. For example, some bits in the encoding of a machine code instruction set might be used to identify a particular type of instruction, while others are used to specify operands.

We will now describe a small collection of examples that can be used to illustrate some of the challenges of dealing with bitdata.

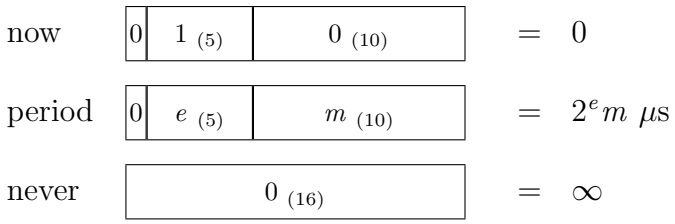
PCI Device Addresses. PCI [74] is a high performance bus standard that is widely used on modern PCs for interconnecting chips, expansion boards, and processor/memory subsystems. Individual devices in a given system can be identified by a 16 bit address that consists of three different fields: an eight bit `bus` identifier, a five bit `device` code, and a three bit `function` number. We can represent the layout of these fields in a simple block diagram that specifies the name and width (as a subscript) of each field, and from which we can then infer the corresponding positions of each field.



By convention, we draw diagrams like this with the most significant bit on the left and the least significant bit on the right. With this encoding, function 3 of device 6 on bus 1 is represented by the 16 bit value that is written `0x0133` in hexadecimal notation or as `00000001 00110 011` in binary, using spaces to show field boundaries.

Timeouts in the L4 Micro-kernel. L4 is a second generation micro-kernel design that was developed to show that it is possible to obtain a minimal but flexible operating system kernel without compromising on performance [60]. One of the versions of the L4 reference manual includes a detailed ABI (application binary interface) that specifies the format that is

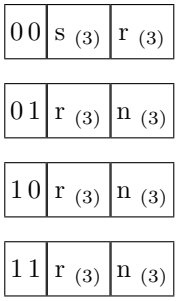
used for system call arguments and results [54]. For example, one of the parameters in the interprocess communication (IPC) system call is a timeout period, which specifies how long the sender of a message should be prepared to wait for a corresponding receive request in another process. Simplifying the details just a little, there are three possible timeout values, as shown in the following diagrams:



There are two special values here: A timeout of 'now' specifies that a send operation should abort immediately if no recipient is already waiting, while a timeout of 'never' specifies that the sender should wait indefinitely. All other time periods are expressed using a simple kind of (unnormalized) floating point representation that can encode time periods, at different levels of granularity, from 1μs up to $(2^{10} - 1)2^{31}\mu$s, which is a period slightly exceeding 610 hours. There is clearly some redundancy in this encoding; a period of 2μs can be represented with $m = 2$ and $e = 0$ or with $m = 1$ and $e = 1$. Moreover, the representations for 'now' and 'never' overlap with the representations for general time periods; for example, a sixteen bit period with $e = 0$ and $m = 0$ must be interpreted as 'never' and not as the 0μs time that we might calculate from the formula $2^e m\mu$s. While this detail of the encoding may seem counter-intuitive, it was likely chosen because many programs use only 'never' timeouts, and most machines can test for this special case—a zero word—very quickly with a single machine instruction. One final point to note is that the most significant bit in all of these encodings is zero. In fact, the L4 ABI also provides an interpretation for time values which have one in the most significant bit. Such values indicate an absolute rather than a relative time. Because there are places in the ABI where only relative times are permitted, we prefer to treat these using different types, but we still need to retain the full 16-bit representation for compatibility.

Instruction Set Encodings for the Z80. The Zilog Z80 is an 8 bit microprocessor with a 16 bit address bus that was first released in 1976, and continues to find many uses today as a low-cost micro-controller [103]. The

Z80 instruction set has a language of 252 root instructions, each of which is represented by a single byte opcode. There are, of course, 256 possible byte values, and the four bytes that do not correspond to instructions are used instead as prefixes to access an additional 308 instructions. For example, one of these prefixes is the byte `0xCB`, which signals that the next byte in the instruction stream should be interpreted as a particular bit twiddling instruction using one of the four formats illustrated by the following diagrams:



`SHIFT` s, r

`BIT` r, n

`RES` r, n

`SET` r, n

n	r	s
000	B	RLC
001	C	RRC
010	D	RL
011	E	RR
100	H	SLA
101	L	SRA
110	(HL)	—
111	A	SRL

The most significant two bits in each case are tag bits that serve to distinguish between shift, bit testing, bit setting, and bit resetting instructions, respectively. The remaining portion of each byte is split into two three-bit fields, each of which specifies either a bit number n, a register/operand r, or a shift type s as shown by the table on the right. Details of what each of these operands means can be found in documentation elsewhere [103]; for the purposes of this document it suffices to note only that there are three distinct types for n, r, and s, all of which are encoded in just three bits, and that the appropriate interpretation of the lower six bits in each byte is determined by the value of the two tag bits. (Technically speaking, the s type contains only seven distinct values rather than the maximum of eight that is permitted by a three bit encoding because it does not use the bit pattern `110`.)

1.2.2 Memory Areas

In our explorations of systems-level programming we have also encountered a wide range of uses of *memory areas*. These are data structures that are stored in memory, but they have a markedly different character to the familiar list- and tree-like data structures that are common in functional programming, or the abstract objects common in object-oriented languages. Often these

memory areas are specific to a particular processor architecture, hardware device, or OS kernel. Next we review some examples of such data structures.

Examples from the Intel IA32. The Intel IA32 processor family [45] contains numerous examples of both bitdata and memory areas. In particular, the operation of the machine is controlled by manipulating a number of tables that reside in memory and have a specific format that is determined by the hardware. For example, *page directories* and *page tables* are used to control the translation between virtual and physical addresses (see Figure 1.2). *Interrupt descriptor tables* specify how to handle exceptions and interrupts. *Segment descriptor tables* are used to partition the virtual memory of a machine into segments, which may have different protection properties. *Task state segments* are records that contain data used to support hardware-based multitasking. Besides the different configuration tables for the machine, memory areas are also used when we need to manipulate memory mapped devices (e.g., the text console), and when the hardware needs to save some of its state, for example when transferring control to the handler for a particular exception.

As a more detailed example, Figure 1.2 illustrates one of the ways in which the IA32 hardware uses the page directory and page tables to decode virtual addresses. A 32-bit virtual address is viewed as a piece of bitdata that contains three indexes of sizes 10, 10 and 12 bits respectively. The first index identifies an entry in the page directory, whose physical address is stored in register CR3. This entry is used to locate an appropriate page table to use for decoding. The second index identifies an entry in the page table, which contains the address of a physical page. Finally the last index is the location in the physical page that contains the data for the virtual address. Pages are all of size $2^{12} = 4096$ bytes. This data structure exhibits a number of interesting properties. For example, both page tables and page directories contain 1024 entries, which are indexed with 10 bits. This ensures that we will never mistakenly access data that is outside the table. Another interesting property is that the entries in the page directory/table are also a special form of bitdata, which uses only 20 bits to identify the address of a page-table/physical page, while the remaining bits are used to specify configuration options, such as access permissions. The fact that we only have 20 bits to identify a memory area places constraints on its locations in memory. For example, with 20 bits we can only specify page-table addresses

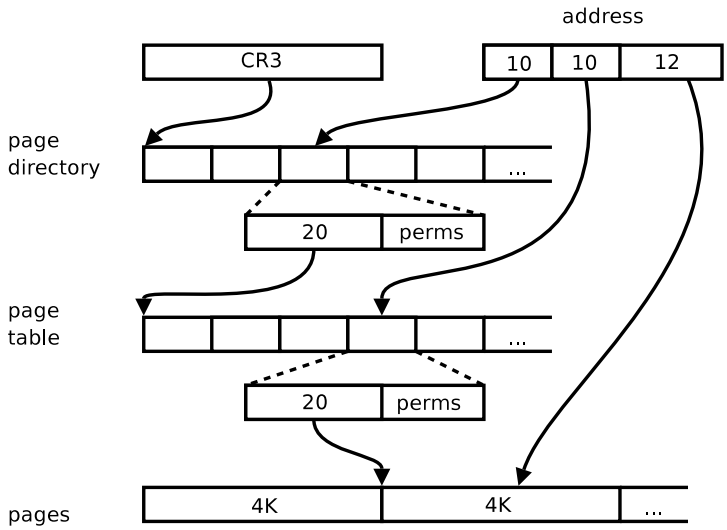


Figure 1.2: Decoding virtual addresses.

that are aligned on 4K boundaries.

Example from the L4 micro-kernel. Not all constraints on data in systems programming are due to hardware requirements. We get similar constraints when we specify a binary interface between software systems, for example the interface between an OS kernel and the processes that use it. The specification of the L4 micro-kernel [54] includes a number of data structures with rigid memory representations. The kernel information page (KIP) is a record that is mapped into every address space and contains data about the current configuration of the kernel. Another example from the L4 microkernel is the user-space thread control blocks (UTCBs), which are memory areas that are used to communicate values between the kernel and user threads. There are also numerous examples of uses of memory areas in concrete L4 implementations that are not dictated by the L4 specification, but instead are engineering decisions made by the kernel implementers.

Similar examples can be found for other processor architectures, devices, or operating systems, both inside the kernel implementation, and outside in the interfaces that the kernel presents to user processes. These memory areas exhibit a number of features that distinguish them from ordinary abstract

data found in high-level languages. In addition, these memory area structures often have fixed sizes, rigidly defined formats or representations, and may be subject to restrictions on the addresses at which they are stored. As we have already discussed, an IA32 page table is always 4K bytes long and must begin at an address that is a multiple of 4K. In this case, the alignment of the structure on a 4K boundary is necessary to ensure that each page table can be uniquely identified by a 20 bit number. In other cases, alignment constraints are used for performance reasons or because of cache line considerations. Storage allocation for memory areas is often entirely static (for example, an OS may allocate a single interrupt descriptor table that remains in effect for as long as the system is running), or otherwise managed explicitly (e.g., by implementing a custom allocation/garbage collection scheme).

1.3 Working with Low-Level Data

In this section we examine various existing approaches to working with the low-level data that is common in systems programming.

1.3.1 Bit Twiddling

From a high-level, it is clear that bitdata structures have quite a lot in common with the 'sum-of-product' algebraic datatypes that are used in modern functional languages: where necessary, each encoding uses some parts of the data to distinguish between different kinds of value (the sum), each of which may contain zero or more data fields (the product).

In practice, however, programmers usually learn to manipulate bitdata using so-called *bit twiddling* techniques that involve combinations of shifts, bitwise logical operators, and carefully chosen numeric constants. Some of the more common idioms of this approach include clearing the `i`th bit in a word `x` using `x &= ~(1 << i)`, or extracting the most significant byte from a 32 bit word `w` using `(w >> 24) & 0xff`. With experience, examples like these can become quite easy for programmers to recognize and understand. However, in general, bit-twiddling leads to code that is hard to read, debug, and modify. One reason for this is that bit-twiddling code can over-specify and obfuscate the semantics of the operation that it implements. Our two examples show how a conceptually simple operation, such as clearing a single bit, can be obscured behind a sequence of arguably more complex steps. As a

result, human readers must work harder to read and understand the effect of this code. Compilers must also rely on more sophisticated optimization and instruction selection schemes to recover the intended semantics and, where possible, substitute more direct implementations. Like the Z80, many machines include a bit reset instruction that can be used to clear a single bit in a register or memory operand. However, it will take special steps for a compiler to recognize when this instruction can be used to implement the earlier, bit twiddling code fragment. Bit twiddling idioms can also result in a loss of type information, and hence reduce the benefits of strong typing in detecting certain kinds of program error at compile-time. The bit pattern that is used to program a device register may, for example, consist of several different fields that contain conceptually different types of value. Bit twiddling, however, usually bypasses this structure, treating all data homogeneously as some kind of machine word, with few safeguards to ensure that individual fields are accessed at the correct offset, with the correct mask, or with appropriately typed contents.

Some of the most widely used systems programming languages, notably C/C++ and Ada, provide special syntax for describing and accessing bit fields, and these go some way to addressing the problems of raw bit twiddling. In C/C++, however, the primary purpose of bit fields is to allow multiple data values to be packed into a single machine word, and specific details of data layout, including alignment and ordering, can vary from one implementation to the next. As a result, different C/C++ compilers will, in general, require different renderings of the same bitdata structure to achieve the correct layout. Ada improves on this by allowing programmers to provide explicit and more portable representation specifications for user-defined datatypes. These languages, however, do not typically provide direct mechanisms for dealing with tag bits, or for using them to support pattern-matching constructs that automate the task of distinguishing between different forms of data.

In practice, programs that involve significant manipulation of bitdata often define a collection of symbolic constants (representing field offsets and masks, for example) and basic functions or macros that present a higher-level, and possibly more strongly typed interface to the operations that are needed in a given application. This approach also isolates portability concerns in a software layer that can potentially be rewritten to target a different compiler or platform. In effect, this amounts to defining a simple, domain-specific language for each application, which would not be such a bad thing if it weren't

for the duplication of effort that is involved in identifying, implementing, and learning to use the set of basic abstractions that it provides. One of the goals of this work is to provide general and flexible constructs for working with bitdata so that we can avoid the need to invent a new domain-specific language for each application that we work on.

1.3.2 Low-level Representations for Bitdata

In this section, we consider how we might write Haskell programs that manipulate PCI addresses of the form described in Section 1.2.1. If the language had already been extended with an appropriate collection of sized integer types (e.g., `Int3`, `Int8`, etc.), then it would actually be possible to represent PCI addresses as elements of a standard Haskell data type:

```
data PCIAddr = PCIAddr { bus :: Int8, dev :: Int5, fun :: Int3 }
```

Ideally, we might hope that a 'smart enough' Haskell compiler could generate code using 16 bit values to represent values of type `PCIAddr` with exactly the same layout that was suggested by the earlier diagram. For Haskell, at least, this is impossible because the language uses lazy evaluation. This means that every type—including `PCIAddr` as well as each of its component types `Int8`, `Int5`, and `Int3`—has to contain a value corresponding to 'suspended' computations. It follows, therefore, that the semantics of `PCIAddr` has more values than can be represented in 16 bits. This specific problem can (almost) be addressed by inserting strictness annotations in front of each of the component types, as in the following variation:

```
data PCIAddr = PCIAddr { bus :: !Int8, dev :: !Int5, fun :: !Int3 }
```

Given this definition, it is conceivable that a Haskell compiler might be able to infer that it is safe to use our preferred sixteen bit representation for PCI addresses. (Technically, this would require a lifted semantic domain to account for the remaining bottom element in this modified `PCIAddr` type.) However, there is nothing in the semantics of Haskell to guarantee this choice of representation, and, to the best of our knowledge, no existing Haskell (or ML) compiler even attempts it.

For most purposes, the representation that a compiler chooses for a given data type is only important when values of that type must be communicated with the outside world. Using either of the previous Haskell representations

for `PCIAddr`, we could define functions like the following to marshal back and forwards between external and internal representations of PCI addresses (we take some liberties with Haskell syntax here, using `>>` and `<<` for the corresponding shift operators of C and `&` and `|` for bitwise *and* and *or* operators, respectively):

```
toPCIAddr      :: Int16 → PCIAddr
toPCIAddr addr  = PCIAddr { bus = (addr >> 8) & 0xff,
                            dev = (addr >> 3) & 0x1f,
                            fun = addr & 7 }

fromPCIAddr    :: PCIAddr → Int16
fromPCIAddr pci = (pci.bus << 8) | (pci.dev << 3) | pci.fun
```

In effect, functions like these might be used to package bit twiddling code in a single place so that the higher-level `PCIAddr` representation can be used exclusively in the remaining portions of the code. In theory, at least, if we follow this approach, then the details of the representation that a compiler chooses for `PCIAddr` would not be significant.

In practical terms, however, there are some serious consequences. For example, it would be unfortunate if we ended up using these functions to import PCI addresses into the functional world, and then export them out again, without ever having made any use of their structure. In that case, all the resources that are spent in decoding, storing, and then re-encoding would be wasted. (Note that lazy evaluation is no help here, not just because of the costs involved in creating a suspended computation, but also because it would simply delay these costs, not eliminate them.) This is one example of the overhead that we inevitably incur if we try to maintain multiple representations of a single type.

Furthermore, while it is not particularly difficult to write functions like `toPCIAddr` and `fromPCIAddr`, it is tedious work, especially when dealing with more complex examples. It is also somewhat error prone because there is nothing to ensure that the functions we define are mutual inverses. Clearly, it would be better if we could arrange to have the code for these functions generated automatically by a carefully designed tool that would guarantee the required behavior. In some cases, we could use Haskell's `deriving` mechanism to get bit representations for values automatically. However, in many other situations this is not possible because, standard Haskell datatypes do not provide the necessary information about layout.

1.3.3 External Specifications

Another possibility is to use an 'interface description language' (IDL) that describes the format of the data that we need to manipulate. IDLs are small languages that describe various data representations. Usually IDLs come together with tools that can turn IDL specifications into code that can encode and decode data. IDLs have been widely used to provide programmer support for remote procedure call (RPC) mechanisms. Invoking a remote procedure requires a programmer to encode the data for transmission over the network, and then later decode the result of the remote procedure. The same idea has been used for communication between programs written in different languages, as they often use different representations for values.

There are a number of different specification languages that can be used to work with data in many different encodings and sources. Some, such as HaskellDirect [26] and CamlIDL [58], are designed as part of the foreign function interface to high-level languages. Their goal is to generate functions that translate between the external representations of data (described in the specification) and the internal representation used by a particular implementation of the high-level language. Others, such as ASN.1 [23], DataScript [6] or PADS [27], support the description of fairly complex data formats. Their functionality is comparable to that of parser generators because they can generate functions that process large amounts of data conforming to a complex specification. Last, but not least, there are a number of domain specific IDLs that are useful when solving a specific task. Examples of these include Devil [62], which can be used to specify the interface to hardware devices (e.g., a mouse, or a network card), and SLED [81], which is aimed at describing the formats of machine instructions (e.g., the encoding of the instructions on the IA32 architecture).

While some IDLs, such as Devil, for example, can be quite useful in systems programming, in general, using a separate IDL to work with the kinds of examples we described at the beginning of this section has a number of drawbacks. One problem is that it requires programmers to learn one or more new languages, which typically have a number of common features, but are presented in different ways. This imposes unnecessary burden on the programmer, but also can make reasoning about programs more complex because different parts of the code are written in different languages. Another problem is that the tools that generate code from IDLs often use marshalling to import and export data. This has a negative impact on the performance

of programs. While marshalling could in principle be avoided, this often requires close cooperation between the IDL tools, and the implementation of the general purpose language, which blurs the distinction between the two languages. When we use a separate IDL, we also get a fairly weak integration with the general purpose language. For example, often the two use different notions of types, and we cannot use specialized language notations (e.g., pattern matching) to manipulate the data that was described with the IDL.

In conclusion, external specifications are not ideally suited to the kind of problem that arise in systems programming, and a much more attractive alternative is to use a programming language that has direct support for working with low-level data.

1.4 This Dissertation

In this work, our goal is to show how to extend a modern statically-typed functional programming language with features that make it suitable for solving problems that are common in systems programming. In particular, we target the problem of safely manipulating data with rigid representation requirements, such as *bitdata* and *memory areas*. Our approach builds upon well established programming language technology, such as type inference, polymorphism, qualified types, and the use of monads to control the scope of effects.

Using a high-level programming language is a first step towards the development of reliable high-assurance systems software because such languages enable programmers to express their ideas in a direct style, automating many error prone low-level tasks. This makes programs easier to write, understand, analyze, and modify. In addition, functional languages are built upon a well-understood semantic framework, which makes programs written in such languages more amenable to formal reasoning and verification.

To illustrates the kind of programs that can be written using the ideas in this dissertation, we present a program fragment that specifies the memory layout for the text console on PCs. This example is worked out more fully in Section 11.1.

```
area screen in VideoRam :: Ref Screen

type Screen   = Array Rows Row
type Row      = Array Cols (LE SChar)
```

```
type Rows     = 25
type Cols     = 80

bitdata SChar = SChar { attr :: Attr, char :: Bit 8 }

bitdata Attr
  = Attr { flash  :: Bool,   -- flashing?
           bg     :: Color, -- background color
           bright :: Bool,  -- foreground bright?
           fg     :: Color  -- foreground color
         }

bitdata Color
  = Black as 0 | Blue as 1 | Green as 2  | Cyan as 3
  | Red as 4 | Magenta as 5 | Yellow as 6 | White as 7 :: Bit 3
```

The overall structure of the dissertation is illustrated in Figure 1.3. In Part I, we present different aspects of the programming language technology that we shall use throughout this dissertation. We start with an overview of related work in Chapter 2. In Chapters 3–5, we provide details about the type system:

- Chapter 3 provides an overview of the basics. It introduces some notation and contains references to additional information about the relevant topics.
- In Chapter 4, we describe how we work with natural numbers in the type system, as they play an important role in our language design.
- In Chapter 5, we describe an intuitive notation for working with type functions. While this is not essential to our design, as in theory we could do without it, in practice it leads to more readable programs.

Chapter 6 introduces a calculus of patterns and definitions. We use this notation to specify the meanings of various pattern matching constructs introduced in later chapters.

Our main language design is presented in Part II of the dissertation, which contains Chapters 7–10:

- In Chapter 7, we show how to extend a functional language—such as Haskell or ML—with support for bitdata types. In Chapter 8, we

identify certain bitdata types that require special care because they violate useful algebraic properties. We present algorithms to detect such types and to help programmers work with them.

- In Chapters 9 and 10, we describe how to add support for working with memory areas. Chapter 9 introduces the basic components of the design, while Chapter 10 describes operations for working with structured areas such as arrays and structures.

Part III concludes the dissertation. In Chapter 11, we present a number of examples to demonstrate our design in action. Our design evolved together with an implementation and, in Chapter 12, we present some of the details of the back-end of our implementation. Finally, in Chapter 13 we outline some ideas for future work and summarize our findings.

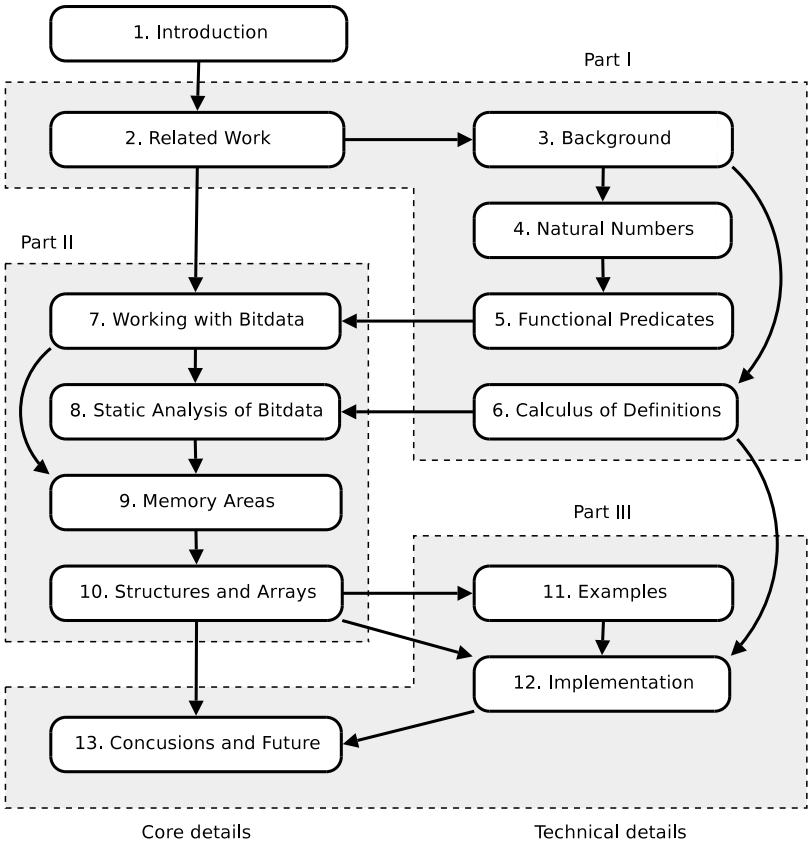


Figure 1.3: Structure of the dissertation. The gray areas group chapters that belong to the same part. The arrows indicate key dependencies between chapters. The chapters on the left contain the core of our design, while the chapters on the right contain technical details and examples.

Part I

Background and Related Work

Chapter 2

Related Work

There has been a lot of research on various approaches to writing systems software. In this chapter we provide a broad survey of the different approaches that have been used, studied, or proposed in previous work.

2.1 Imperative Languages

In this section we examine some well-known 'imperative' languages that were designed and traditionally have been used for systems programming. For our purposes, we consider a language to be 'imperative' if it is a first-order language (i.e., functions are not values), and if the main way to write programs is by sequencing statements, rather than evaluating expressions.

C At present, the most popular language for writing systems software is probably C [53]. It was designed in the early 1970s and, since then, it has been used for the development of many systems (e.g., UNIX [10]). It is a fairly simple language, whose design favors pragmatics over safety.

C is a statically typed language, but the type system is fairly simple, and it is common to cast values between seemingly incompatible types. As a result, it is possible to write C programs that could not have been written in a type-safe manner in the languages that were used when C was designed. Unfortunately, this flexibility comes at a cost: it is quite easy to write programs that contain errors that are difficult to detect and correct.

Typically, implementations do not perform any run-time checks, and working with malformed values usually results in unspecified behavior. The

basic types in C include various numeric types, enumerations, structures, unions, arrays and pointers. The types determine how to manipulate data, but ensuring the correctness of the data (and determining if it is there at all) is left entirely to the programmer. For example, given a pointer to an array `p`, we can access its n^{th} element with the expression `p[n-1]`. It is the job of the programmer to ensure that `p` is really a pointer to an array that has at least `n` elements.

The types in C are often used to describe the layout of memory areas. Because the language allows arbitrary casts between values, it is easy to change the 'view' on low level data, which is quite useful in some situations. For example, we can cast a pointer to an array of bytes, into a pointer to structured data, which then enables us to use the structured interface to manipulate the data.

One pitfall is that the C language leaves a lot of the representation details up to implementations. In some cases this is reasonable, because many system-level data structures are hardware specific. It is unfortunate, however, that different implementations on the same platform could potentially use different data representations. As a result of this, programs that simply cast bytes into structured data have very implementation specific behavior. For example, the C language specification states that the fields of structures should be placed in increasing memory locations, but it also allows arbitrary padding between them [17] (Section 6.7.2.1, bullet 12). Such casting is also dangerous if the representation of multi-byte values is important, as they can be stored using either little- or big- endian representation.

The C language also has some support for working with bitdata. Programmers may specify the number of bits that should be occupied by fields in structures and unions. This feature is convenient when programmers need to pack more data into less space, but it is not commonly used when working with hardware devices because, again, it is up to each implementation to determine how exactly to organize the bit-fields in a machine word. As a consequence, programmers have less control over the representation of the data.

From this discussion, it is apparent that there is some tension in the design of C: on the one hand, implementations have a lot of freedom to choose the representation for data, on the other, systems programmers need to specify explicit layouts for bitdata and memory areas. In practice, this tension is resolved by using a single language implementation. For example, the `gcc` compiler [89] is the commonly used C implementation on Linux.

Using a single implementation also addresses the issue of implementation specific language extensions. For example, an important issue when writing systems software is the alignment of data, but there is little support for this in standard C. To work around this, `gcc` supports an extension that can be used to specify alignment when declaring data. For example, this is how we can declare an array that is aligned on 4096 byte boundary:

```
int page[1024] __attribute__((aligned(4096)));
```

This state of affairs is unfortunate because, even with a single language implementation, there may be differences between the versions of the implementation (one of the reasons for the proverbial 'bit-rot'), and the lack of an precise specification makes it difficult to write robust software. Indeed, similar concerns have motivated the development of a theory for describing the implementation-dependent assumptions that are present in programs written in C-like languages [72].

Modula-3. Another imperative language that was designed for writing systems software is Modula-3 [18]. It is was designed in the 1980s, as a simplification and improvement over the previous languages in the Modula family. It is particularly interesting because it was designed to be a *safe* language for systems programming. It supports automatic memory management via garbage collection, and it is a strongly typed language (i.e., it disallows arbitrary casting between different types of values). It has good support for abstraction through modules, which have well-defined interfaces.

Modula-3 has a number of types, including ordinal types, references, arrays, and records. The representations of the types are not specified by the Modula-3 report, although the language contains operators that can determine the number of bits, bytes, and addressable locations occupied by values. There is some support for working with bits using packed types or sets. Programmers may use packed types to specify how many bits should be occupied by the values of a type when they are stored inside other structures (e.g., arrays). This is mostly useful for reducing the amount of space occupied by data because there is no way to specify the bit-patterns that should be used to represent the values. Sets are bit-vectors that contain a bit for every value in the type of the set elements. The bit for a value is 1 when the value is present in the set, and 0 otherwise. The bit-vectors in this case are used as an efficient representation for finite sets.

Another interesting property of the types in Modula-3 is that arrays may be *fixed*, in which case their size is known statically; otherwise they are *open*. Programmers can use range types (e.g., `{0 .. 7}`) to index safely into arrays. There are operations that manipulate range types (performing checks), but it is nice that, for fixed arrays, programmers may use `FOR` loops to iterate safely through an entire array without performing unnecessary bounds checks. Unfortunately, because the language does not have support for polymorphism, programmers often have to resort to using open arrays, for example, in functions that can manipulate arrays of different sizes.

With respect to references, Modula-3 is mostly safe, in that it does not allow arbitrary operations on references in safe modules. However, references may be `NULL`, in which case dereferencing results in a checked run-time error. In addition, Modula-3 allows arbitrary reference arithmetic, but only in modules that are marked as unsafe, a property that is automatically tracked and propagated by the system.

SPIN. Modula-3 was used for the development of an extensible operating system kernel, called SPIN [11, 84]. The kernel is designed in such a way that its functionality can be extended by adding modules that share the same address space as the kernel. This gains efficiency because it avoids costly context switches. To ensure the safety of the resulting system, SPIN relies on the fact that modules are written in a safe subset of Modula-3, which ensures that they cannot accidentally corrupt the kernel's private data. In the process of implementing the kernel, the SPIN team identified the need for some extensions to Modula-3 [43]. In particular, they identified the need for a `VIEW` construct that allows casting between arrays of bytes and structured data, such as records, without copying. To implement such a construct, the types in the language would need fixed representations.

Cyclone. While the simplicity of C has its benefits, the language provides little to help programmers write safe and correct code. A number of the safety gaps in C are addressed in the design of Cyclone [46], which is a safe dialect of C. Cyclone programs resemble C programs, but the language utilizes modern programming language technology to help programmers. For example, besides the usual C types, Cyclone provides support for working with algebraic data types, which provide a convenient and safe way to work with tree-like data structures. In addition, Cyclone has a much stronger type

system than C, and this is used in a variety of ways to enforce important safety properties of programs. Examples include the use of polymorphism to support functions that safely manipulate different types of data; distinguishing between nullable and non-nullable pointers types; tracking the sizes of arrays; and keeping track of the regions occupied by various pieces of data. Where the type system cannot ensure the correctness of programs statically, the Cyclone implementation inserts run-time checks, which raise exceptions if they fail. In summary, while it resembles C, Cyclone is a modern systems programming language in its own right.

2.2 Functional Languages

In this section we examine different aspects of writing systems programs in functional languages. The distinguishing characteristic of functional languages is that programs are typically written in a style that evaluates expressions, rather than sequencing statements, which gives them a more declarative feel. In addition, functional languages typically have good support for working with function values. Like any other value, functions can be passed or returned from other functions, and also stored in data structures. This simple idea is very powerful and can be used to encode a number of programming patterns directly in the language. In general, functional languages tend to operate at a fairly high level of abstraction, often utilizing automatic memory management and advanced type systems. (The use of advanced type systems is in no way specific to functional languages. For example, TAL [69] is an assembly language with an advanced type system, while Scheme [1] and Erlang [3] are functional languages that are dynamically typed.)

Effects. Writing systems software requires programs to interact with their environment (i.e., the various devices controlled by the software), to manipulate state, and to handle exceptions. Features such as these are commonly known as *computational effects*. In functional languages, working with computational effects poses some challenges. The reason is that computational effects often have an 'imperative' semantics (e.g., the order in which they are performed is important), which is different from the expression-centric evaluation of functional languages. Fortunately, there are a number of approaches that have been developed to deal with this problem.

One option is to use expressions with *side-effects*. Evaluation of such

expressions not only produces a result, but may also have a computational effect, such as interacting with a device, for example. Working with such expressions requires special attention by programmers (and implementations) because they violate the principle of referential transparency (i.e., the same expression may produce different values if evaluated multiple times). This approach is used in a number of functional languages that have strict evaluation semantics (e.g., SML [64]), but it is not suitable for languages with 'lazy' (i.e., on demand) evaluation semantics because, in such languages, it is difficult to determine the order in which expressions are evaluated, and hence the order in which effects are performed.

Even in strict languages, expressions with side-effects complicate reasoning about programs, and it is therefore desirable to distinguish between pure and effectful parts of a program. This can be done using effect systems [87], or monadic encapsulation [66, 67, 78], which are closely related [25, 96]. The idea behind monadic encapsulation is to introduce special values that *describe* what effects need to be performed rather than *performing* them directly. Such 'computation' values can be manipulated like any other value in the language and, in fact, closely resemble functional values. In this setting, the execution of functional programs has two 'modes': (i) pure evaluation of expressions that produce values; and (ii) execution of the side effects described by computation values. This approach is used in Haskell [76, 75].

For the specific effect of interacting with the environment, there is another (non-monadic) approach that has been used. The idea is to view programs as transforming a (lazy) stream of requests from the environment into a (lazy) stream of responses [55, 42, 38]. The different approaches to performing input and output in functional languages are studied in Gordon's dissertation [34].

Embedded Gofer. An interesting example of writing systems software in a functional language is Malcolm Wallace's work on Embedded Gofer [97]. It describes extensions to Haskell that make it more suitable for writing software for embedded systems. In particular, it adds support for communicating processes with type checked message passing, interrupt handling, register access and real-time garbage collection. The language implementation has a very small run-time system and was used to write and test some real embedded software [98]. While Embedded Gofer does not have support for working with low-level data, it addresses a number of other issues that are important to the systems programmer.

The Fox Project. A large amount of research related to writing systems software in functional languages was conducted as part of the Fox Project at CMU [40]. One part of the Fox project was the development of FoxNet [14], which is a network protocol stack that is entirely implemented in ML. It makes use of the abstraction mechanisms of the language (in particular, its powerful module system), resulting in a modular implementation. The FoxNet implementation requires efficient manipulation of arrays, which is achieved with a new abstraction called *word arrays* [14]. There is also a general framework that provides a common interface for manipulation of sequence types with different implementations [13]. While FoxNet presents an example of a piece of system software written in a functional language, the Fox project also has a number of general contributions to functional languages, which had an indirect but important impact on their suitability for systems programming. For example, advances in compiler technology [68, 88], make it possible to execute functional programs with an efficiency comparable to that of imperative code.

Operating Systems. Another area of interest is the use of functional languages for operating system development. An early attempt in this area was the development of the 'Hello' operating system [30], which is implemented in SML. The idea with this line of research is to adjust the run-time system of a programming language in such a way that it can operate directly on the hardware, without the support of an underlying OS. A more recent development in the same spirit is the 'House' kernel [39], which is implemented in Haskell. The 'House' kernel replaces Haskell's IO monad with a different monad (called `H`) that provides a low-level interface to working with the machine. The `H` layer of the system is implemented in a mixture of C and Haskell using Haskell's foreign function interface. Another line of research in a similar direction is the development of the Coyotos operating system [83], which is to be implemented in BitC, a functional programming language developed in parallel with the system. There is also a project called Singularity [44], currently in development at Microsoft, which aims to develop a new operating system design by exploiting advanced programming language technology provided in C#.

Experience in Industry. Functional languages have also been used for the development of systems software in industry. A prominent example in

this area is Erlang [3], developed and used by Ericsson to program telephony switches. Erlang is a dynamically typed language that has good support for concurrency and distributed computation, and has been used to implement large commercial systems. It has some support for working with binary data by using special values called *binaries*, which can be manipulated with its bit syntax [37].

Another interesting functional language developed in industry is BlueSpec [5], which is aimed at hardware design (e.g., programming FPGAs). BlueSpec's design is based on Haskell and so it inherited a host of useful features, including Haskell's powerful type system and its clean syntax (although later versions of BlueSpec have adopted a different notation more familiar to hardware designers). BlueSpec is particularly related to our work because it has good support for working with bitdata. This is necessary because, in order to program the hardware, data has to have rigid bit representations.

Foreign Function Interfaces. Because functional languages operate at a fairly high level of abstraction, they often come equipped with a foreign function interface (FFI) that enables them to interact with code written in other languages (often C). Data representation issues are important at the boundary between languages because different languages may represent data in different formats. There are two main approaches to implementing an FFI: one approach uses a small external specification language to generate marshalling code automatically [26, 58], while the other allows programs to manipulate external data directly. This second approach is called *data level interoperability* [29] and has been used, for example, to integrate new functional code with existing legacy software [91].

The C FFI of Standard ML of New Jersey (SML/NJ) [15] is based on data level interoperability. To manipulate C data, programmers use a library that provides an ML encoding of the C type system (assuming the representation used by a particular C implementation). In this way, programmers can manipulate C data structures directly from ML without having to marshal large amounts of data. The ML types corresponding to the C types are implemented using a library of unsafe operations that are not intended to be used outside the FFI implementation. The library provides support for basic types and arrays, while support for structure and union types is provided by generating specialized definitions using an external tool. The library provides an encoding of numbers in ML's type system and makes an extensive

use of phantom types to distinguish different C types (e.g., arrays of different sizes). In some situations, for example when performing pointer arithmetic, programmers have to pass explicit type information specifying the sizes of the C types involved in the computation.

Haskell has a well-defined FFI [21], which is an official addendum to the language report [76]. Haskell's FFI is very flexible and it has been used in a number of applications. To deal with bitdata, it provides operations for bit-twiddling, while to work with memory areas, it provides a variety of different pointer types with associated operations. Haskell's FFI operates at a fairly low level of abstraction and, in general, it is unsafe because it supports arbitrary pointer arithmetic. However, it is expressive enough to allow most marshalling code to be written directly in Haskell, which is useful because it can be used to build safer interfaces on top of the low-level primitives.

2.3 Domain Specific Languages

In this section, we review some domain specific languages (DSLs) that have support for working with low level data.

Perhaps the most closely related domain specific language is Devil [62], which is designed to describe interfaces to hardware devices. To interact with a device, programmers write a Devil specification describing the device, which is then processed by a tool to generate C code that can send and receive data from the device. The Devil DSL identifies three important concepts: *ports*, *registers*, and *device variables*. Ports are used to specify points of interaction with a device. Registers specify the granularity of interaction with the device. Programmers use ports to specify how to access and manipulate various registers. Finally, the logical values that are to be communicated to a device are specified with device variables, which can be stored in a part of a register, or spread across multiple registers. In addition to these basic concepts, Devil provides more advanced features that are useful when working with some of the more exotic devices. Devil specifications are concise and can be verified by the Devil implementation to detect specification errors.

Another DSL with a purpose similar to Devil is NDL [22], which describes device interfaces, but also uses state machines to describe higher level protocols for interacting with a device.

In Chapter 1, we presented some examples of bitdata that are common in systems programming. Another form of bitdata, which we shall not target

in this dissertation, is the manipulation of streams of bits. Such bitdata is often used when working with encryption or compression algorithms and multimedia streams. A domain specific language that has good support for working with such data is Cryptol [31], developed at Galois Connections, and designed for working with cryptographic protocols. Another approach for manipulating bit streams is described by Wallace and Runciman [99], who use a file-like interface to manipulate bit-streams.

The concept of working with bit-streams is also used in the Specification Language for Encoding and Decoding (SLED) [81], but in a more general form. SLED is a domain specific language for working with streams of machine language instructions, which is useful for developing tools such as assemblers, linkers, and debuggers. Because, in general, instructions may require multiple machine words, SLED processes a stream of *tokens*, which can be of any size. SLED specifications partition tokens into a number of *fields*, specified by a range of bits. Fields contain the different components of machine instructions (e.g., the op-code). To recognize instructions, programmers define *patterns*, which can be combined using conjunction, disjunction, or concatenation, while new instructions may be created using *constructors*. SLED has a rich language for manipulating bitdata, and it also provides special syntax to enable the concise specification of large instruction sets.

Another DSL for working with binary data is DataScript [6]. DataScript specifications describe binary file formats as (dependent) types. These specifications are processed by a language binding, which produces a library that can be used to read and write files in the given format. The types supported by DataScript include basic types, arrays, structures, and (disjoint) unions. The specification language has support for specifying the byte ordering of multi-byte values. Fields in composite types may be annotated with constraints, which are used to validate the contents of fields, or to distinguish between the alternatives in a union type. DataScript has been used to specify the formats for a variety of binary data, including Java class files, network packets, and fairly complex file formats, such as ELF.

Yet another DSL for working with data in predefined formats is PADS [27], which is a DSL designed to process ad-hoc data. This includes both binary data and textual data using various encodings. As such, it can handle punctuation and delimiters, and it can deal with malformed data. In this respect, the PADS tools resemble parser generators such Lex and Yacc, except that they can both process incoming data and also produce outgoing data. Its basic specification mechanism is similar to that of DataScript and other

data description languages. This commonality was noticed and captured in a dependently typed calculus [28], which can be used to provide a solid semantic basis for data description languages.

2.4 Summary

We have examined a wide variety of high level programming languages that have been studied, proposed, or used in the context of systems programming. Our survey was biased towards issues of data representation: the mechanisms that can be used by programmers to work with data with rigid constraints on representation. Most languages abstract away from the representation of data, and as a result make it difficult to write robust and correct systems software: even veterans of systems programming, such as C, have to rely on implementation specific behavior and extensions.

The goal of our work is to design a language mechanism that enables programmers to work safely with data that has rigid representation constraints. At the same time, such data should coexist with the usual abstract data that is common in most high level languages. Such a language mechanism is particularly useful in the development of systems software.

Chapter 3

Background: Type Systems

In this chapter, we provide a brief introduction to the style of type system that we shall use extensively in this dissertation. The ideas in this chapter are well known, and we present them here only for completeness. Readers familiar with type systems based on qualified types can safely skip the chapter, although it may be useful to skim through the content to ensure familiarity with our notation.

We shall present the ideas incrementally, by describing a series of simple languages. In Section 3.1, we present the simply typed *λ-calculus*, which is at the core of the system. In Section 3.2, we add support for Hindley-Milner style *polymorphism*. In Section 3.3, we show how to restrict polymorphism using *qualified types*. In Section 3.4, we describe the idea of *improvement*. And, in Section 3.5, we show how to use *kinds* to check the correctness of types.

3.1 The λ-calculus

The core of higher order functional languages is the (pure) λ-calculus [8], which has the following syntax:

$e =$	x	Variable
	$\mid e\ e$	Function application
	$\mid \lambda x.e$	Function value

This is a simple calculus that allows us to define and apply functions to perform computation. The λ-calculus is expressive enough to encode Turing

machines and so, in principle, we could use it to write any program we want. In practice, functional languages also contain a number of constants, as well as various mechanisms to define values and introduce new constants. For example, most functional languages provide built-in types for characters, integers, and floating point values, as well as mechanisms for defining algebraic datatypes.

In the pure λ-calculus the only type of value we have is the function, and the only operation is application. In practical functional languages, however, we have many different types of values and associated operations. Naturally, not all operations make sense for all values. For example, it does not make sense to try to use the operation that adds integers on function values. We use *type systems* to detect such malformed programs. Programs in *statically* typed languages are analyzed for type errors before they are executed, while programs in *dynamically* typed languages detect type errors at run-time. In this work, we concentrate on statically typed languages.

Types. A type system associates types with the terms of a language (the λ calculus in this case). Terms that do not have types are invalid and should be rejected. The syntax of the types that we use is quite simple:

$$\begin{array}{rll} \tau = & \tau\ \tau & \text{Type application} \\ \mid & c & \text{Type constant} \end{array}$$

A type expressions is essentially a tree of type constants. The constants correspond to built-in types. We assume the presence of a type constant ($\rightarrow$), used to write function types. We shall also use some type constants for basic types: `Int` and `Char` for integers and characters respectively. For example, a function that maps integers to characters has the type: `(→) Int Char`. We adopt the convention that type (and expression) application is left associative (e.g., `F a b` is the same as `(F a) b`). To improve readability we also write ($\rightarrow$) using an infix notation like this: `Int → Char`, which, when written as an infix operator, is right associative.

Explicitly Typed Calculus. The above syntax for the λ-calculus does not mention types—all typing information is implicit. There is an alternative formulation, which requires that we annotate function arguments with their types:

$$e' = \ldots \mid \lambda x :: \tau.\ e'$$

The implicit formulation is more convenient when we write expressions because we do not need to clutter the program with types, while the explicit formulation is more convenient when we analyze programs because we can easily compute the types of expressions.

Typing Environments. The type of an expression depends on the types of the free variables that it mentions. For example, the type of the body of a function depends on the types of the function arguments. To keep track of the types of free variables we use a *typing environment*. It is common to use the Greek letter Γ to refer to typing environments. A typing environment is a partial map from variable names to types. We write $\Gamma(x) = \tau$ to assert that the typing environment Γ associates the type τ with the variable x. We write $\Gamma, x :: \tau$ for the environment that is like Γ, except that it maps x to τ.

Type System. We specify a type system for a language with a relation that associates expressions with their types. In our presentation, we shall also associate expressions in the implicitly typed calculus with expressions in the explicit calculus. To define the typing relation, we use *typing judgments* of the form:

$$\Gamma \vdash e \rightsquigarrow e' :: \tau$$

Intuitively, a judgment of this form asserts that, if the the free variables of an expression e have types as defined by the typing environment Γ, then the expression e has type τ (and that e' is a fully annotated version of e).

There are three rules for constructing typing judgments, corresponding to the different ways in which we can define expressions. The names of the rules follow the convention used in systems based on natural deduction: elimination rules are marked with E, while introduction rules are marked with I.

$$[\text{var}]\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x \rightsquigarrow x :: \tau}$$

$$[\rightarrow E]\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' :: \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 \rightsquigarrow e_2' :: \tau_1}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' :: \tau_2}$$

$$[\rightarrow I]\ \frac{\Gamma, x :: \tau_1 \vdash e \rightsquigarrow e' :: \tau_2}{\Gamma \vdash \lambda x.\ e \rightsquigarrow \lambda x :: \tau_1.\ e' :: \tau_1 \rightarrow \tau_2}$$

3.2 Hindley-Milner Polymorphism

When we use the implicit λ-calculus, there are many expressions that have more than one type. For example, the expression $\lambda x.x$ is a function whose result is the same as its argument and so, for any type τ, it has the type $\tau \to \tau$. We say that such expressions are *polymorphic*.

The Hindley-Milner type system [63] enables us to work with such expressions. The idea is to allow typing environments that associate free variables with *type schemes* (σ) rather than just simple types.

$$\begin{aligned} \sigma &= \forall\alpha.\sigma \mid \tau \\ \tau &= \ldots \mid \alpha \end{aligned}$$

The variables in types (α) are place-holders for concrete types. When we use a polymorphic value in a particular context, we *instantiate* it to concrete types. The type of the resulting expression is obtained by replacing the type variables in a type scheme with the concrete types. For example, a value like $\lambda x.x$ will be associated with a scheme of the form $(\forall\alpha.\ \alpha \to \alpha)$, and if we use it in a context where we need to apply it to an integer argument, then we would instantiate it with the type `Int` to get `Int` $\to$ `Int`.

To define polymorphic values we use an additional new form of expression:

$$e \equiv \ldots \mid \textbf{let } x = e \textbf{ in } e$$

Using **let**, we can define polymorphic local variables: if the type of the expression defining x contains type variables that are not mentioned in the environment (i.e., they can be anything at all), then we can *generalize* over them to obtain a type scheme for x. As in the previous section, it is convenient to have an explicit version of the language, where all local variable are annotated with their schemes, and we make instantiation and generalization explicit in the terms:

$e' \equiv \ldots$	
$\mid \textbf{let } x :: \sigma = e' \textbf{ in } e'$	Local variable
$\mid e'_\tau$	Instantiate
$\mid \Lambda\alpha.e'$	Generalize

Here is an example of an expression in the explicit calculus:

$$\textbf{let } id :: \alpha \to \alpha \ = \ \Lambda\alpha.\lambda x :: \alpha.x \textbf{ in } id_{Int}\ 1$$

Notice that in the example we did not write the quantifier in the scheme for *id*. This is just syntactic sugar, 'free' type variable are implicitly quantified.

The typing judgments for applying functions and creating function values remain the same as in the previous section. The remaining rules are like this:

$$[\text{lkp}]\ \frac{\Gamma(x) = \sigma}{\Gamma \vdash x \rightsquigarrow x :: \sigma} \qquad [\text{let}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' :: \sigma \quad \Gamma, x :: \sigma \vdash e_2 \rightsquigarrow e_2' :: \tau}{\Gamma \vdash \begin{array}{l}\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rightsquigarrow \\ \textbf{let}\ x :: \sigma = e_1'\ \textbf{in}\ e_2' :: \tau\end{array}}$$

$$[\forall E]\ \frac{\Gamma \vdash e \rightsquigarrow e' :: \forall\alpha.\sigma}{\Gamma \vdash e \rightsquigarrow e_\tau' :: \sigma[\tau/\alpha]} \qquad [\forall I]\ \frac{\Gamma \vdash e \rightsquigarrow e' :: \sigma}{\Gamma \vdash e \rightsquigarrow \Lambda\alpha.e' :: \forall\alpha.\sigma}\ (\alpha \notin \mathit{fvs}\ \Gamma)$$

Notice that these rules relate expressions to type schemes instead of simple types. Perhaps the most interesting rule is $\forall I$, which creates a polymorphic value. The side condition on this rule is crucial because, if a type variable is mentioned in the environment, then it is part of the type of a free variable. Therefore, this is not a truly generic variable, and hence, we are not free to quantify over it.

3.3 Qualified Types

The Hindley-Milner type system allows us to instantiate a polymorphic value at *any* type. This form of polymorphism is sometimes called *parametric* polymorphism. In some situations, what we need is a way to restrict the types that may be used to instantiate schemes. Consider, for example, adding equality to the λ-calculus. One way to do this would be to add equality functions as new constants (one for each type):

```
eqInt :: Int → Int → Bool
eqChar :: Char → Char → Bool
...
```

This approach works to a degree, but it is clumsy. For example, if we were to write a function that is polymorphic in its arguments, except that it needs to compare them for equality, then we would have to pass the equality operator explicitly as an extra argument. Functional programs tend to involve a large number of small functions that call each other, and so, if we were to use this approach, then we would have to pass the extra equality operators manually to all functions. Furthermore, because we cannot use the same equality

operator to compare values of different types, we may have to pass multiple equality operators, which clutters polymorphic programs even more.

It would be nicer to use the same operator following the usual mathematical convention. We could try to do this with a polymorphic equality operator:

```
(==) :: a → a → Bool
```

Unfortunately, this is not quite correct because it suggests that we can test values of *any* type for equality, but values of some types may not be easily comparable for equality. For example, in general, equality on function values is undecidable.

To solve this problem, we can use the theory of qualified types [48]. The key idea in this approach is to introduce *predicates* on types (π), and to augment schemes with predicates:

$$\sigma = \ldots \mid \pi \Rightarrow \sigma$$

Now a scheme may be instantiated only at types that satisfy its predicates. For example, using qualified types we can give `(==)` a more accurate type:

```
(==) :: (Eq a) ⇒ a → a → Bool
```

In this case `(Eq a)` is a predicate that can be discharged only for types that support equality. In general, we specify how to discharge predicates, using an *entailment* relation $\vdash\!\!\!\vdash$:

$$\Pi \vdash\!\!\!\vdash p : \pi$$

Here Π is a set of assumptions, π is a predicate, and p is the evidence that we can use to convince ourselves that π holds. For example, the evidence that `Eq` τ holds might be a function of type $\tau \to \tau \to$ `Bool` that we can use to compare values of type τ. To add support for predicates to our type system, we add a set of assumptions, Π, as an extra argument to the typing judgments:

$$\Pi; \Gamma \vdash e \rightsquigarrow e' :: \tau$$

To support qualified types we augment the explicit calculus with operations for evidence application and abstraction, which are analogous to the instantiation and generalization operations that we added in the previous

section. This leads us to the final version of the explicit language:

$e' = x$	Variable
$\mid \textbf{let } x :: \sigma = e' \textbf{ in } e'$	Local variables
$\mid e'\ e'$	Function application
$\mid \lambda x :: \tau . e'$	Function value
$\mid e'_\tau$	Type application
$\mid \Lambda\alpha.\ e'$	Type abstraction
$\mid e' \cdot p$	Evidence application
$\mid \bar{\lambda} x :: \pi.\ e'$	Evidence abstraction

The full typing rules for the system are shown in Figure 3.1. There are two rules that deal with qualified types. The first one is $(\Rightarrow E)$, which is part of the instantiation judgments. We use it to instantiate variables that have qualified schemes. The other rule is $(\Rightarrow I)$, and it is part of the generalization judgments. We use it to introduce values with qualified schemes.

Notice that, in the presentation of the type system, we left the predicates and the entailment relation unspecified. We did this because they are parameters to the system. As we present our language design for systems programming, we shall gradually introduce a number of predicates that will complete the specification of the concrete type system for our language.

Implicit Arguments. The predicates introduced by the theory of qualified types restrict the instantiation of ordinary Hindley-Milner schemes. We obtain another useful way to understand the system, if we examine it through the Curry-Howard isomorphism [90]. This establishes a correspondence between logic and programming that relates the predicates and proofs of a logic, to the types and terms of a programming language (and vice versa). With this in mind, we can think of the predicates in qualified schemes as the types of the language defined by the evidence for the predicates. Thus, qualified schemes resemble ordinary Hindley-Milner schemes with extra arguments. Consider, for example, the type of the overloaded equality operator again:

```
(==) :: Eq a ⇒ a → a → Bool
```

Using the predicates-as-parameters interpretation, we can understand the equality operator as a function that has three arguments instead of two: the first (predicate) argument tells us how to compare the remaining two arguments.

$$[\text{var}]\ \frac{\Gamma(x) = \sigma}{\Pi; \Gamma \vdash_i x \rightsquigarrow x :: \sigma}$$

$$[\text{let}]\ \frac{\Pi; \Gamma \vdash_g e_1 \rightsquigarrow e_1' :: \sigma \quad \Pi; \Gamma, x :: \sigma \vdash e_2 \rightsquigarrow e_2' :: \tau}{\Pi; \Gamma \vdash \begin{array}{l}\textbf{let } x = e_1 \textbf{ in } e_2 \rightsquigarrow \\ \textbf{let } x :: \sigma = e_1' \textbf{ in } e_2' :: \tau\end{array}}$$

$$[\rightarrow E]\ \frac{\Pi; \Gamma \vdash e_1 \rightsquigarrow e_1' :: \tau_1 \rightarrow \tau_2 \quad \Pi; \Gamma \vdash e_2 \rightsquigarrow e_2' :: \tau_1}{\Pi; \Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' :: \tau_2}$$

$$[\rightarrow I]\ \frac{\Pi; \Gamma, x :: \tau_1 \vdash e \rightsquigarrow e' :: \tau_2}{\Pi; \Gamma \vdash \lambda x.\ e \rightsquigarrow \lambda x :: \tau_1.\ e' :: \tau_1 \rightarrow \tau_2}$$

$$[\forall E]\ \frac{\Pi; \Gamma \vdash e \rightsquigarrow e' :: \forall \alpha.\sigma}{\Pi; \Gamma \vdash e \rightsquigarrow e_\tau' :: \sigma[\tau/\alpha]}$$

$$[\forall I]\ \frac{\Pi; \Gamma \vdash e \rightsquigarrow e' :: \sigma}{\Pi; \Gamma \vdash e \rightsquigarrow \Lambda\alpha.e' :: \forall \alpha.\sigma}\ (\alpha \notin \mathit{fvs}\ (\Pi, \Gamma))$$

$$[\Rightarrow E]\ \frac{\Pi; \Gamma \vdash e \rightsquigarrow e' :: \pi \Rightarrow \sigma \quad \Pi \Vdash p : \pi}{\Pi; \Gamma \vdash e \rightsquigarrow e' \cdot p :: \sigma}$$

$$[\Rightarrow I]\ \frac{\Pi, x :: \pi; \Gamma \vdash e \rightsquigarrow e' :: \sigma}{\Pi; \Gamma \vdash e \rightsquigarrow \bar{\lambda} x :: \pi.\ e' :: \pi \Rightarrow \sigma}$$

Figure 3.1: Type system for Hindley-Milner with qualified types.

3.3.1 Overloading in Haskell

Haskell's class system [95, 76] provides a mechanism for defining overloaded functions, and can be formalized using the theory of qualified types [48]. The design of Haskell uses **class** declarations to introduce new predicates and **instance** declarations to extend the entailment relation.

In addition to introducing a new predicate, a class declaration defines a number of *methods*, which are functions whose schemes are implicitly qualified by the predicate introduced with the class. We can think of the methods as the names for the evidence that is required to discharge the class predicate. For example, the equality operation is defined in Haskell with the `Eq` class:

```
class Eq a where
  (==) :: a → a → Bool
```

The first line introduces a new predicate `Eq` that has one parameter `a`, and the second line indicates that the class has only a single method named `(==)`. There are instances for various Haskell types, for example:

```
instance Eq Int where
  x == y  = eqInt x y

instance Eq Char where
  x == y  = eqChar x y
```

The two instances state that we can discharge the predicate `Eq` for the types `Int` and `Char` respectively. The method definitions provide the required evidence.

We shall sometimes use Haskell's notation for classes and instances as a concise way to introduce and define predicates, functions, and the entailment relation. Note that the entailment relation for the Haskell class system is *open*, (i.e., it can be arbitrarily extended with new instances by programmers). In some parts of our design we use qualified types to enforce certain properties of values, and so we do not allow programmers to add their own instances: instead the implementation automatically provides instances when they are safe and needed.

3.4 Improvement

Improvement [49] is a technique that uses information about the satisfiability of predicates to infer more accurate types for expressions, and also to detect expressions that are associated with schemes that can never be used.

Consider, for example, a predicate P that can be satisfied for just one type τ. Then, if a variable x is associated with a scheme $\forall\alpha.\ P\ \alpha \Rightarrow \tau_1$, then we can safely replace the universally quantified variable α with τ, because this is the only type that satisfies P. This is beneficial to programmers, because they get to work with simpler types. It is also beneficial to compilers because they can exploit the extra type information to generate better code.

We can also use improvement for more accurate error reporting. Suppose, for example, that we infer that some function f is of type $\forall\beta.\ P\ \tau_2 \Rightarrow \tau_3$, with $\tau_2 \neq \tau$ for any β. Technically, there is nothing wrong with f, but any attempt to use it will result in a type error, because we will not be able to discharge the predicate. By using improvement, we could detect that f

cannot be instantiated and report an error, or at least warn the programmer about f's unusual type. Issuing a warning would be useful in a system where the entailment relation is defined in different parts of the program, and the type checker cannot be sure if a predicate really cannot be satisfied without examining the entire program.

To formalize this idea, we need to consider the set of satisfiable instances for a predicate:

$$\lfloor p \rfloor = \{q \mid \exists \theta.\ (q = \theta p)\ \wedge\ (\nvdash q)\}$$

In this definition, p is a predicate, θ is a substitution, and θp is the predicate that we obtain when we apply the substitution to the predicate. Thus, $\lfloor p \rfloor$ is the set of all instantiations of p that can be discharged with the given entailment relation. For example, if $\lfloor p \rfloor = \{\}$, then we know that p cannot be discharged, no matter how we instantiate its variables.

Of particular interest are substitutions that do not change the set of satisfiable instances for a predicate. We say that such substitutions *improve* the predicate:

$$\mathit{improves}\ \theta\ p \equiv \lfloor p \rfloor = \lfloor \theta p \rfloor$$

Improving substitutions can be used to eliminate variables from a typing judgment without changing its meaning. This is formalized with the following rule:

$$[\text{improve}]\ \frac{\Pi; \Gamma \vdash e \rightsquigarrow e' : \tau \quad \mathit{improves}\ \theta\ \Pi}{\theta\Pi; \theta\Gamma \vdash e \rightsquigarrow \theta e' : \theta\tau}$$

In general, it may be difficult for a type checker to compute improving substitutions from the entailment relation for a language. To help out, a language designer (or even the programmer) can specify *improvement rules*, which we shall write using the following notation, following the original paper on improvement [49]:

$$\textbf{improve}\ \Pi\ \textbf{using}\ \{\tau_1 = \tau_1'; \ldots; \tau_n = \tau_n'\}$$

The variables in Π are implicitly universally quantified, and we can instantiate improvement rules in different contexts. The resulting improving substitution is the combination of the most general unifiers of the equations in the **using** clause. For example, we can can formalize the example from the beginning of this section with a rule like this:

$$\textbf{improve}\ P\ \alpha\ \textbf{using}\ \alpha = \tau$$

From a logical point of view, we can think of improvement rules as an implication, stating that the equalities in the **using** clause are *necessary* for the predicates Π to hold. In our language design we shall use a mixture of axioms and improvement rules to specify the entailment relation for our particular language.

It is important that the various improvement rules present in a system are consistent (i.e., that they do not contradict each other). Here is a simple example of two inconsistent rules:

```
improve P a using a = Int
improve P a using a = Char
```

While detecting conflicts of this form is not hard, checking for consistency of improvement rules in general may require arbitrary logical reasoning, and so is undecidable. For this reason, adding programmer defined improvement rules to a language requires care. However, it is quite possible for an implementation to use improvement internally. For example, as we have already mentioned, in Haskell the entailment relation is always open, in the sense that implementations assume that there may always be new instances for a class in modules that have not yet been examined. However, in some situations this is not case: for example, if a class is not exported from the module where it is defined, then the only instances for the class would be in the same module where the class was defined. An implementation could use this to compute improvement rules that would enable it to infer more accurate types, and to report errors early.

3.4.1 Functional Dependencies

Functional dependencies [51] provide a concise notation for introducing improvement rules. The idea of using functional dependencies originated in relational algebras, which are used as a foundation for describing and querying databases. It is always pleasing when well-established ideas in one field of research turn out to be useful in seemingly unrelated areas!

There are many examples where functional dependencies come in handy, which is why they have become a fairly popular extension to Haskell. To see how they are typically used, consider defining a type class that enables us to overload the numeric operations. In Haskell this is done with the `Num` class:

```
class Num a where
  (+) :: a → a → a
```

```
  ...
instance Num Int where ...
instance Num Float where ...
```

This works well, but notice that addition (and other operations) require their arguments and result to be all of the same type. Thus we can use `(+)` to add integers to integers, and floats to floats, but adding floats to integers requires explicit coercions. We could make the insertion of these coercions automatic by using a multi-parameter class:

```
class Nums a b c where
  add :: a → b → c
  ...

instance Nums Int   Int   Int   where ...
instance Nums Int   Float Float where ...
instance Nums Float Int   Float where ...
instance Nums Float Float Float where ...
```

The class `Nums` has three arguments: two for the arguments, and one for the result of the functions. In this way we can add floating point numbers to integers. However, there is a problem. Suppose that we have two variables `x::Int`, and `y::Float`. What is the type of `add x y`? At first, we might think that the answer is `Float` because of the second instance. Unfortunately, the expressions has a more general type:

```
add x y :: Nums Int Float c ⇒ c
```

While this extra generality may be useful in some situations, we quickly run into a problem. Consider, for example, that we want to check if the sum of `x` and `y` is `0`. Because literals in Haskell are also overloaded (i.e., we can use the same notation for `0::Int` and `0::Float`, which is very handy) we would get the following strange type:

```
add x y == 0 :: (Nums Int Float c, Eq c) ⇒ Bool
```

Types like these are usually rejected by the system because they are ambiguous: notice that the type variable `c` does not appear on the right hand side of the ⇒ symbol, and so there is no way to determine from the context what type should be substituted for `c`. The system cannot pick an arbitrary type, because picking different types may result in completely different answers,

which would be very confusing for the programmer and the system would not have a well-defined semantics.

We can work around this problem by using a functional dependency that states that the types of the argument of the class *uniquely* determine the type of the result:

```
class Nums a b c | a b ⇝ c where
  add :: a → b → c
  ..

instance ...
```

The new component is the functional dependency which is written after the vertical bar in the class declaration. Specifying a functional dependency amounts to adding a new improvement rule to the system. In this particular example, the new rule is:

```
improve (Nums a b c1, Nums a b c2) using c1 = c2
```

In general, the improvement rule for a functional dependency $\bar{\alpha} \rightsquigarrow \bar{\beta}$ on a class `C` has the form:

$$\textbf{improve}\ (C\ \bar{\alpha}\ \bar{\beta}_1, C\ \bar{\alpha}\ \bar{\beta}_2)\ \textbf{using}\ \bar{\beta}_1 = \bar{\beta}_2$$

To see how improvement is used by the system, consider again adding `x` and `y` as in the previous example. Initially, the system infers the same type as before:

```
add x y :: Nums Int Float c ⇒ c
```

However, from the instance rules we also know that `Nums Int Float Float` holds. Therefore, we can apply the improvement rule to determine that `c` must be `Float`, and thus (with an extra simplification step) obtain a nicer type:

```
add x y :: Nums Int Float c ⇒ c
(improve)
add x y :: Nums Int Float Float ⇒ Float
(simplify)
add x y :: Float
```

3.5 Using Kinds

Another useful aspect of a type system is the *kind system* [7], which provides a way to validate and classify types (i.e., the kind system is like a type system for the language of types). Just as a type system ensures that the terms (programs) written by a programmer are valid, a kind system ensures the validity of types written by programmers.

Depending on the language in question, programmers may have to write types for different reasons. For example, many popular programming languages (e.g., C and Java) do not support type inference, and instead require programmers to specify the types of all function parameters, variables, and class attributes. Even in statically typed functional languages—such as Haskell or ML—that have good support for type inference, programmers need to write types. Examples include datatype definitions and explicit type signatures.

Even in a language where the programmers do not need to write any types, the kind system is quite useful as a check for implementations: for example inferring a type that does not have a valid kind may indicate that there is a bug in the implementation, which, without a kind system, might have gone unnoticed.

Richer type systems often benefit from correspondingly richer kind systems. In languages with complicated type systems, one may go even further and define a type system for the kind system itself. There are many variations on the theme: there are languages where the type system is the same as the kind system; there are also languages that have infinitely many 'levels' of type systems, each level (after the first) ensuring the validity of the previous one. In our work, we adopt a fairly simple kind system, similar to the kind system of Haskell:

$$\kappa = * \mid \ldots \mid \kappa \to \kappa$$

The only difference from Haskell's kind system are some constants that we shall discuss in later chapters. The kind `*` is used to classify 'ordinary' types that contain values, while function kinds (of the form $\kappa \to \kappa$) are used to classify type constructors. For example, the kind of the type `Int` is `*`, because it is an ordinary type that contains values. On the other hand, to represent polymorphic lists we need to use a type constructor, `List :: * → *`. The kind for `List` tells us that this constructor expects an ordinary type, such as `Int`, as an argument, and that it will produce an ordinary type (i.e., the

lists containing elements of the argument type). Thus the type `List Int` is a valid type, while `List List` is not, and will be rejected by the kind system.

Another interesting example is the kind of the function space type constructor: [1]

```
(→) :: * → * → *
```

This type constructor has two arguments, one for the domain of the function and one for the codomain. It is interesting to note that, from the kind of `(→)`, we can see that we are working with a higher-order language: because the result is of kind `*`, function types may appear as both arguments and results to functions. As an example of how to use kinds, we show how we might set up the kinds in a language that has first order functions that may have multiple arguments. We can do this with two type constructors:

```
Arg :: * → Fun → Fun
Res :: * → Fun
```

Here, `Fun` is a new kind that classifies first order functions. Note that we did not use `*` because functions are not ordinary values in this language. The two constructors essentially describe non-empty lists of types, the last element (tagged with `Res`) contains the result of the function, while the preceding elements are the arguments to the function. For example, the type of a first-order addition function on integers would be:

```
primAdd :: Arg Int (Arg Int (Res Int))
```

In our work we shall use two new kinds: `Nat` and `Area`. The kind `Nat` classifies natural numbers in the type system, and it is discussed in Chapter 4. The kind `Area` classifies types that describe explicit memory areas, and it is discussed in Chapter 9.

3.6 Summary

In this Chapter, we briefly described the basic language technology that is used throughout this dissertation. We started with the λ-*calculus*, which lies at the core of functional languages. We added a *type system* which was used

[1] Note that the arrow on the left of `::` is quite different from the arrows on the right, because it is a type constructor, while the arrows on the right are at the kind level.

to detect malformed programs, and also a *kind system* which was used to detect malformed types. We also described two generalizations to the simple type system. The first one introduced Hindley-Milner style polymorphism, which was used to assign *type schemes* to expressions that have multiple simple types. The second extension allowed us to *qualify* type schemes with predicates, in this way restricting the types that could be used to instantiate the schemes. We also briefly outlined how these ideas are used in the type system of Haskell, which is a pure functional language.

Chapter 4

Natural Number Types

Our design for working with data that has rigid representation constraints makes an extensive use of natural numbers in the type system. These 'types' do not classify any values, but instead we shall use them as parameters to other types to record statically known information (e.g., various sizes). For example, we shall work with types such as `Bit 8` that classify bit vectors of size 8.

Using natural numbers at the type level is not a new idea and has been used in many research systems [73, 101], although the use of type level natural numbers is quite restricted in mainstream programming languages. For example, languages like C and Pascal have some built-in types that contain numbers (e.g., arrays), but no support for polymorphism or user defined datatypes that mention numbers. On the other end of the spectrum are languages, such as Coq [12] or Cayenne [4], that support dependent types. Such languages blur the distinction between types and values and as a result programmers may use numbers in the types system. This increased expressiveness comes as the cost of a more complicated static analysis phase—in particular, type inference is not well supported in many languages that utilize dependent types.

In this chapter, we explore a different point the design space by showing that the Hindley-Milner type system, augmented with qualified types and support for improvement, provides a good framework for working with numbers in the type system. The starting point for our design is the type system that we summarized in Chapter 3. We extend the system with a set of constants, as follows:

```
0,1,2,...     :: Nat
```

```
(_ + _ = _)   :: Nat → Nat → Nat → Pred
(_ * _ = )_   :: Nat → Nat → Nat → Pred
(GCD _ _ = _) :: Nat → Nat → Nat → Pred
(2 ^ _ = _)   :: Nat → Nat → Pred
```

First we add a new kind, `Nat`, whose inhabitants are the natural numbers. The remaining declarations specify the kinds of four new predicates that are used to manipulate natural numbers. The kind `Pred` indicates that these constants are predicates, and not ordinary types (we may think of the signatures as **class** declarations that have no methods).

Each predicate corresponds to the usual mathematical operation (`GCD` stands for *greatest common divisor*). For example, a predicate of the form `x + y = z` is an assertion that the sum of `x` and `y` is equal to `z`. Thus, the system can discharge the predicate `2 + 3 = 5`, but will fail if it encounters `2 + 3 = 6`.

4.1 Basic Axioms

The simplest way to discharge predicates on the natural numbers is when all arguments are statically known constants. In that case, all we have to do is check that the appropriate relation holds. We may think of these basic axioms as an infinite family of instances:

```
instance 1 + 1 = 2
instance 1 + 2 = 3
...
instance 42 + 1 = 43
instance 42 + 2 = 44
...
```

We also need some axioms that deal with the various algebraic identities, and allow us to solve predicates, even if some of the operands are variables. Here are the rules that we used in our implementation, written as Haskell instances:

```
instance 0 + a = a
instance a + 0 = a
```

```
instance 0 * a = 0
instance a * 0 = 0
instance 1 * a = a
instance a * 1 = a

instance GCD 0 a = a
instance GCD a 0 = a
instance GCD 1 a = 1
instance GCD a 1 = 1
instance GCD a a = a
```

Note that we have assumed that `GCD 0 0 = 0` (which is quite common), otherwise the rule `GCD 0 a = a` would not be valid.

4.2 Computation With Improvement

Of course, the system also needs to perform some computation. Using just the basic axioms we cannot solve simple predicates like `2 + 5 = a`. Why is that? This predicate does not match any of the algebraic identities from the previous section, and the only other predicates that we can discharge require that there are no variables in the predicate.

We introduce computation to the system in the form of improvement rules. The idea is that, when we encounter a predicate of the form `2 + 5 = a`, we can improve it to `2 + 5 = 7`, which can then be discharged using the basic axioms. It is interesting to note that, because we are working with relations, we can also perform 'backwards' computation in the style of Prolog. For example, we can improve `2 + a = 5` to `2 + 3 = 5`. Next we present the computational improvement rules that we used in our system (they are not very surprising!). Because we have families of improvement rules (one for every natural number really) in the rules below we adopt the convention that `x` and `y` are used for concrete natural number types, while `a` and `b` are used for type variables. For example, when we know the first two arguments of an addition predicate we can improve the result:

```
improve 2 + 5 = a using a = 7
```

We shall write the family of rules of this form like this:

```
improve x + y = a using a = (x + y)
```

Note that the operator + is used in two different ways in the previous rule: on the left-hand-side of the keyword **using** it is part of the name of the predicate that is being improved, while on the right-hand-side it refers to the algebraic operation on the natural numbers. Here are the remaining improvement rules that define the operations:

```
improve x + a = y using a = (y - x)  -- if x ≤ y
improve a + x = y using a = (y - x)  -- if x ≤ y
improve a + b = 0 using a = 0; b = 0

improve x * y = a using a = (x * y)
improve x * a = y using a = (y / x)  -- if x divides y
improve a * x = y using a = (y / x)  -- if x divides y
improve 0 * a = b using b = 0
improve a * 0 = b using b = 0
improve a * b = 1 using a = 1; b = 1

improve GCD x y a using a = (gcd x y)
improve GCD 1 a b using b = 1
improve GCD a 1 b using b = 1

improve 2 ^ x = a using a = (2^x)
improve 2 ^ a = x using a = (log x)  -- if log x is natural
```

4.3 Rules for Equality

The computational rules from the previous section are useful to turn 'unknown' type variables to concrete natural numbers. It is also useful to have some rules that establish equalities between different natural numbers. For example, any two of the three variables in an addition predicate will uniquely determine the third. We can represent this with a collection of functional dependencies, { (a b ⇝c), (b c ⇝a), (c a ⇝b) } [51] and then use improvement [49] to instantiate and simplify many of the addition predicates that are generated during type inference. For example, if we encounter the following pair of predicates (a + b = c, a + b = d) during type inference, then we can conclude that c = d, and then use simplification to reduce the set to just a single predicate (a + b = c). This is exactly what functional dependencies enable us to do.

We can also do similar things with multiplication `a * b = c`, greatest common divisor `GCD a b = c` and power of two `2 ^ a = b` predicates. Because all of them are functions, we have the 'forward' functional dependency `a b` $\rightsquigarrow$`c` (or `a` $\rightsquigarrow$ `b` for exponentiation). Unfortunately, not all operators are injective functions, so we do not always have the 'backward' functional dependencies. In particular, knowing the result and one of the arguments to `GCD` tells us little about the other argument: `GCD 3 5 = 1`, but also `GCD 7 5 = 1`. The situation is not quite as bad with multiplication, but multiplication by 0 destroys the backward functional dependency: `1 * 0 = 0`, but also `2 * 0 = 0`. We can still do the backward improvement if we know that the argument is not 0 (e.g., if it is a non 0 constant):

```
improve (a1 * k = c, a2 * k = c) using a1 = a2  -- if k not 0
improve (k * b1 = c, k * b2 = c) using b1 = b2  -- if k not 0
```

The exponentiation operator is injective, and so there we have both the forward and backward dependencies.

In addition to the functional dependencies, we also have some other improvement rules that enable us to infer equalities between types. We list the ones that we used in our implementation below:

```
improve 0 + a = b using a = b
improve a + 0 = b using a = b

improve a * 1 = b using a = b
improve 1 * a = b using a = b

improve GCD 0 a = b using a = b
improve GCD a 0 = b using a = b
improve GCD a a = b using a = b
```

4.3.1 Expressiveness of the System

The rules for equality between natural numbers are by no means complete, but work well for the constraints that arise in many practical programs. There are a couple of reasons to explain why this happens. The first is the application domain where we use the type level natural numbers: typical programs do not require complex proofs because often programs tend to be fairly monomorphic, and we can discharge predicates by evaluating them. The other reason is that, in many situations, using the improvement rules is

optional: by using improvement we can infer more concise types, or detect type errors a little sooner, so it is clearly advantageous, but it is not necessary. For example, inferring a type like `(2 + 3 = x)` ⇒ `Bit x` is not as concise as inferring `Bit 5`, but the program will not fail, or be less useful. Another way to put this is that we can delay predicates that we cannot solve immediately by placing them in inferred schemas. Later, when the schema is instantiated at concrete types, we can attempt to solve the predicates again.

The only situation where we cannot delay the predicates is when we work with type signatures. For example, suppose a programmer specifies a signature:

```
f :: P ⇒ T
```

Now, if we infer the type `T` for `f`, but under the assumptions Q, then we would have to prove that $P \Vdash Q$, before accepting the program. This requires a form of automated theorem proving that may, in some cases, be too difficult for the type inference engine.

In many cases, if the system cannot prove a certain goal, it is not hard to alter the program so that it is accepted by the system. Typically, a programmer might have to add an explicit type signature, or alter an existing type signature.

More practical experience with the system will be required to determine if we need to add more rules to the system. There certainly are more options. For example, we could encode the commutativity and associativity of addition using rules like these:

```
improve (a + b = c, b + a = d) using c = d
improve (a1 + a2 = a12, a12 + a3 = a123,
             a2 + a3 = a23, a1 + a23 = a123')
  using a123 = a123'
```

Other rules that are presently missing from the system are rules that relate the various operations to each other. For example, if we have the predicates `(2 * a = b, a + a = c)`, the system could conclude that `b = c`. Here is how we could encode a distributivity rule:

```
improve (b + c = bc, a * bc = d,
             a * b = ab, a * c = ac, ab + ac = d')
  using d = d'
```

Note that some of the more complex rules become a little complex to write using just the basic, three place predicates. In the next chapter, we present a notation that allows us to write such formulas using the usual notation, and to have the system translate it automatically to this primitive notation.

At present, it is not clear if some of the more complex rules are useful in practice. If practical experience shows that we do need a more sophisticated predicate solver, then there are a few topics that we would like to explore in future work. One obvious direction is to try to integrate the type checker of the programming language with an external tool that is specifically designed for solving problems with natural numbers. There are well-known decision procedures that can deal with Presburger arithmetic (essentially equations that just have addition). Another interesting direction of research would be to devise an efficient algorithm for finding and applying improvement rules. The simple algorithm that we used in our implementation is quite inefficient and is not likely to scale well if it has to deal with many rules, especially of the more complex variety with many predicates in the assumption part.

4.4 Other Operations

The operations on the natural numbers that we have presented so far are sufficient for our design of a language extension for systems programming. Of course, we could add some operations that could be potentially useful to programmers. However, adding a new predicate is not a trivial task because we have to add enough rules to the system so that the new predicate can be useful.

In this section we show how we can obtain some more operations 'for free' (i.e., without having to add more rules to the system). The main observation is that we can use the natural number predicates in 'reverse' to get relations for subtraction, division, and logarithm with base two. For example, we can express the constraint that `x - y = z`, by writing `z + y = x` instead.

4.4.1 Predicate Synonyms

To make such definitions a little easier to work with, we introduce a small generalization to Haskell. In Haskell, programmers can use **type** declarations to introduce shorter names for complex type expressions. For example, we can declare:

```
type Parser a = String → Maybe (a, String)
```

Then, writing `Parser Int` in a type is exactly the same as using the function type on the right-hand side of the `=` except that the resulting program is shorter and probably easier to read.

Type synonyms are not restricted to types of kind `*`, but can have any kind (i.e., we can have type synonyms that expand to type constructors). In the same spirit, we allow type synonyms of kind `Pred`, which essentially introduce different names for predicates. This is quite a small change, which does not introduce major technical difficulties. Using this extension (and a bit of syntactic sugar) we can define some more operations in terms of the ones we have already presented:

```
type (x - y = z)   = (z + y = x)
type (x / y = z)   = (z * y = x)
type (Log2 x = y)  = (2 ^ y = x)
```

The new predicates work mostly as one might expect, serving as the inverses of the functions that we have already defined. When one of these predicate synonyms appears in a type, the system can simply replace it by its definition and then use the old rules to handle the resulting constraints.

One interesting point about these definitions is that division is not quite a function. As we have already discussed in Section 4.3, multiplication is not injective and so its inverse is a proper relation rather than a function. This shows up in just one (rather unusual) case, when we divide 0 by 0. Following the above definition `0 / 0 = z` is the same as `z * 0 = 0`, which can be solved trivially. Thus, `0/0` can be any number at all.

Chapter 5

Notation for Functional Predicates

In this chapter, we introduce syntactic sugar that is useful when working with predicates constrained by functional dependencies. In Section 5.1, we present the basic idea, together with some examples of why it is useful. In Sections 5.2, 5.3, 5.4, and 5.5, we describe the effect of the syntactic sugar on the different parts of a Haskell program that contain types. In Section 5.6, we show how our idea is related to the proposal for associated type synonyms. In Section 5.7, we summarize the key ideas of this chapter.

5.1 Overview

In Chapter 4, we described the kind `Nat` of natural numbers and we formalized the operations on `Nat` as relations between arguments and results. We noted, however, that the relations were all *functional*, which was captured with functional dependencies stating that the arguments uniquely determine the result.

While the relational notation has its uses, it can get a bit unwieldy in some situations. We already saw examples of this when we specified a rule to formalize the associativity of addition:

```
improve (a1 + a2 = a12, a12 + a3 = a123,
             a2 + a3 = a23, a1 + a23 = a123')
  using a123 = a123'
```

This rule is a lot less readable than the usual statement of associativity:

```
improve using (a + b) + c = a + (b + c)
```

We can use the more readable notation if we introduce an addition function to the type language. Attractive as this might appear, the introduction of an associative, commutative function into the type language makes type inference much more complicated. Furthermore, we cannot completely replace the three-place addition predicate with the two-place addition function because the latter cannot express constraints like `x+y=3`.

Fortunately, there is a way to have our cake and eat it too! We can treat the second improvement rule above as a *purely syntactic abbreviation* for the first: any expression of the form `a+b` in a type can be replaced with a new variable, `c`, so long as we also add a predicate `(a+b=c)` to our set of assumptions. For example, in the following chapters we shall introduce a type for bit vectors called `Bit`. This type is parameterized by the width of the bit vectors. Then, using this translation, we can write `Bit n` → `Bit (n+2)` as a shorthand for `(n+2=m)` ⇒ `Bit n` → `Bit m`. The same notation can also be used with the other predicates described previously. As another example, here is how we can completely desugar the nice looking associativity rule into the expanded form.

```
improve
  using (a + b) + c = a + (b + c)
⇒
improve (a + b) + c = v1
  using v1 = a + (b + c)
⇒
improve (a + b = v2, v2 + c = v1)
  using v1 = a + (b + c)
⇒
improve (a + b = v2, v2 + c = v1, a + (b + c) = v3)
  using v1 = v3
⇒
improve (a + b = v2, v2 + c = v1, a + v4 = v3, b + c = v4)
  using v1 = v3
```

In fact, we can generalize this idea to arbitrary predicates with functional dependencies. If $P\ t_1\ \ldots\ t_n\ t_{n+1}$ is an $(n+1)$-place predicate in which the last argument is uniquely determined by the initial arguments, then we will allow an expression of the form $P\ t_1\ \ldots\ t_n$ in a type as an abbreviation for some new variable, a, subject to the constraint $P\ t_1\ \ldots\ t_n\ a$. Note that,

because of the functional dependency, this translation does not introduce any ambiguity. For example, if the expression $P\ t_1\ \ldots\ t_n$ appears twice in a given type, then we will initially generate two constraints $P\ t_1\ \ldots\ t_n\ a$ and $P\ t_1\ \ldots\ t_n\ b$. However, from this point, we can use improvement to infer that $a = b$ and then to eliminate the duplicated predicate.

This simple technique gives us the conciseness of a functional notation in many situations, without losing the expressiveness of the underlying relational predicate. We will use this abbreviation notation quite frequently in the following chapters, and we expect that it will prove to be useful in other applications beyond the scope of this work. Our approach is similar to the 'functional notation for functional dependencies' suggested by Neubauer *et al.* [71], but it is more general (because it admits multiple dependencies) and it does not require any changes to the concrete syntax of Haskell other than support for functional dependencies.

The notation for functional predicates introduces new qualifiers to types. In the following sections we examine the various Haskell declarations that may contain types and describe how these new assumptions are handled in each case.

5.2 Type Signatures and Instances

The assumptions arising from the abbreviations in the type part of a type signature or an instance are added to the context for the signature or instance declaration. Because signatures and instance declarations all have contexts, there are no technical difficulties here. Consider, for example, the following instance:

```
instance C (Bit (1 + n)) where ...
```

This is simply an abbreviation for the instance:

```
instance (1 + n = x) ⇒ C (Bit x) where ...
```

The Importance of Contexts. Despite the fact that there are no technical difficulties, our translation may transform a type signature that appears to be fully polymorphic into one that is qualified by a context. For example:

```
f :: Bit n → Bit (n+2)
```

appears to be a fully polymorphic function, but after removing the functional predicate notation we obtain:

```
f :: (n + 2 = m) ⇒ Bit n → Bit m
```

In some situations this is important, in particular in implementations that use dictionary passing [95] to implement qualified types. In such implementations, a predicate context is translated into extra function arguments. As a result, overloaded values are represented with functions that produce different results depending on their evidence arguments. For example, the following definition introduces an overloaded numeric value.

```
two :: (Num a) ⇒ a
two = 2
```

A dictionary passing implementation would represent `two` as a function, and when it is used in different contexts this function would be applied automatically to (possibly) different arguments. Thus, in an integer context, the implementation would supply the evidence for `Num Int`, while in a floating point context it would supply the evidence for `Num Float`. This may affect the performance of a program because an overloaded value may be computed multiple times instead of being computed just once and then saved. Indeed, concerns like this motivated the (somewhat controversial) introduction of the *monomorphism restriction* to Haskell [76], which requires programmers to write explicit type signatures on overloaded values that do not syntactically appear to be functions.

Note that not all implementations of overloading suffer from this problem. For example, our implementation eliminates all overloading at compile time by generating specialized versions of polymorphic functions. In particular, overloaded values such as `two` from the previous example are evaluated only once for each different context in which they appear. It is also possible to achieve similar behavior in dictionary passing implementation by using optimizations (e.g., multiple instantiations with the same evidence may be eliminated by a form of common sub-expression elimination).

5.3 Contexts

It is quite possible that a syntactic abbreviation is used in the argument of a predicate. In such situations we add the extra assumptions to the context

that contains the predicate. This works well for the contexts in type signatures and instances, as well as the improvement rules that we have been using. In Haskell 98, predicates in the context of a class declaration may contain only type variables, which disallows the use of our abbreviation notation in such situations.

Types in Class Contexts Several Haskell implementations support an extension to the type system that allows arbitrary types in the contexts of class declarations. This works in a similar fashion to ordinary (Haskell 98) super classes. For example, programmers may write:

```
class C [a] ⇒ D a where ...
```

This declaration states that, whenever we can discharge the predicate $D\ \tau$, then we should also be able to discharge $C\ [\tau]$. More formally, we get the following new axiom:[1]

```
∀a. D a ⇒ C [a]
```

Usually, implementations enforce this by proving the goal $C\ [\tau]$, for every instance $D\ \tau$ in the program.

The super-class axiom is used to simplify the contexts in qualified types, as well as in situations when we have to check if a user specified type matches the inferred type for a declaration. For example, we may simplify the context `(D a, C [a])` to just `(D a)`, because the predicate `C [a]` is implied by the super-class axiom.

With this extension, the context of a class declaration may contain types that make use of the abbreviation notation. For example, we might write the following class declaration:

```
class P (x + y) ⇒ C x y where ...
```

Then, if we eliminate the functional predicate notation we would get the following declaration:

```
class (x + y = z, P z) ⇒ C x y where ...
```

[1] Notice that if we think of ⇒ as an implication symbol, then Haskell's notation for super classes is backwards: the logical implication is in the opposite direction from the **class** declaration.

Notice that now the context contains a new variable, `z`, which is not mentioned in the class head (i.e, to the right of the symbol ⇒). In current Haskell implementations such declarations are rejected, and so we may not use the abbreviation notation in the contexts of class declarations. The problem is the interaction of functional dependencies and class contexts, and next we examine the issues and some possible design choices.

Super Classes and Functional Dependencies. Consider the following declarations:

```
class F a b | a ⇝ b
class F a b ⇒ C a

instance F Int Bool
```

The second class declaration here is interesting because the context of `C` contains a variable `b` that is not in the class head. Should this be allowed, and if so, what does it mean? There are several possible answers.

One option would be to extrapolate directly from the current behavior of class contexts in Haskell. This results in adding the following axiom:

```
∀a,b. C a ⇒ F a b
```

While there is nothing inherently wrong with the axiom, it disallows us from defining any instance for `C`. Consider, for example, an instance like the following:

```
instance C Int
```

Before we can accept this declaration, we would have to check that it does not violate the super-class axiom, and so we would have to prove that:

```
∀b. F Int b
```

We cannot do this because the functional dependency on `F` states that `F Int b` holds for exactly *one* type (in this case `b=Bool`). Our goal, however, is to show that `F Int b` holds for *all* types, and so we fail. By a similar argument, we would have to reject *any* attempt to provide an instance of `C`, and so we could never discharge a predicate involving `C`. Notice that this has to be the case because an instance for `C` would render the system unsound in the presence of the stated super-class axiom. Consider, for example, a function of the following type:

```
f :: (F a b, C a) ⇒ a → b
```

Now, using the super-class axiom, we may simplify the type of `f` to:

```
f :: (C a) ⇒ a → b
```

These types look quite different: in the first type we had a relation between `a` and `b` that was captured by `F`, but in the second type `b` is completely unconstrained. The only way to justify such a transformation would be if there were no instances for `C` (i.e., in logical terms we have assumed *false*). From this discussion we see that, with the proposed super-class axiom, we may never define instances for the class and for this reason we should probably report an error, thus rejecting the declaration.

An alternative design would be to use the following super-class axiom:

```
∀a.∃b. C a ⇒ F a b
```

The difference here is that we use an existential quantifier for variables that are mentioned in the class context but that do not appear in the class head. This axiom permits us to define instances for `C`. For example, we may accept an instance for `C Int`, because we can solve `∃b.F Int b`, using the instance `F Int Bool`. On the other hand, we would reject an instance like `C Char` because there is no way to solve `∃b. F Char b`. An interesting observation about this form of the super-class axiom is that we cannot use it for context simplification. For example, we cannot simplify `(F a b, C a)` to `(C a)` because we cannot be sure that the `b` in the context is the same as the one 'hidden' by the existential.

In conclusion, the second design choice is more useful because it enables programmers to use super-classes to control the valid instances for a class.

5.4 Type Synonyms

Haskell's type synonyms provide a convenient mechanism for introducing names for type expressions. Haskell 98 type synonyms do not support contexts, and so we would not be able to use the functional notation in the definitions of type synonyms. This is unfortunate, because we have found many examples where it is convenient to use the functional notation in a type synonym. In this section, we show that this limitation can be avoided by generalizing the type synonym construct of Haskell to support contexts.

Our generalization is based on an observation that relates type synonyms to Haskell's class system. First, note that we can think of a type synonym as a simple (first order) function on types. However, the functional dependency extension to Haskell also provides a way to define functions on types. It is then natural to ask if we can encode type synonyms using functional dependencies. It turns out that this is fairly easy to do. Consider the following type synonym declaration:

type `T a =` τ

Here, `a` is a parameter to the type synonym[2], and τ is a type expression that may mention `a`. As a function on types, `T` maps the argument `a` to the value τ. We can define the same function by using functional dependencies with the following two declarations:

class `T a b | a` $\rightsquigarrow$ `b`
instance `T a` τ

The class declaration specifies that `T` is a type function of the appropriate arity, while the instance declaration provides the definition for the type function. Now we can use the functional predicate notation to write `T A` in a type (where `A` is some type expression). This will be expanded to a new variable `b`, subject to the constraint `T A b`, which can then be discharged by using the above instance, thus substituting τ`[A/a]` for `b`. Essentially what happens is that we have expanded the type synonym. This shows that that the two definitions are equivalent. We are not suggesting that implementations should implement type synonyms by using the class system (although they could!), because implementations often try to delay expanding type synonyms as much as possible so that the synonyms can appear in types that are visible to the programmer (e.g., inferred types or type errors).

One technical point on this topic is that Haskell has a rather ad-hoc way of handling polymorphic kinds: they are simply monomorphised by replacing kind variables with `*`. Because of this, splitting a type synonym into a class declaration and an instance might result in the inference of the wrong kind if declarations are processed one at a time. We could avoid this either by generalizing Haskell's type system to support polymorphic kinds, or simply by delaying the kind monomorphization until we have checked the instance defining the type synonym.

[2]We use a single parameter to avoid clutter, but the same development works for multiple parameters as well.

5.4.1 Type Synonyms with Contexts

The observation that type synonyms can be encoded in the type system is useful because it shows us how to safely generalize type synonyms in several ways. In particular, we get a simple and consistent semantics for type synonyms with contexts, which are needed to support the functional predicate notation:

type `C` $\Rightarrow$ `T a =` τ

is equivalent to:

class `T a b | a` $\rightsquigarrow$ `b`
instance `C` $\Rightarrow$ `T a` τ

More concretely, consider the following example:

```
type Eq a ⇒ EqList a = [a]
```

This declaration defines the name `EqList` to refer to the types of lists whose elements belong to the `Eq` class (i.e., the elements can be compared for equality). The class system translation of this declaration is:

```
class EqList a b | a ⇝ b
instance Eq a ⇒ EqList a [a]
```

Now consider how we can use this type synonym:

```
elem :: a → EqList a → Bool
elem x xs = ...
```

The type signature would be desugared using the following steps:

```
elem :: a → EqList a → Bool
-- expand notation
elem :: EqList a b ⇒ a → b → Bool
-- improvement with instance
elem :: (Eq a, EqList a [a]) ⇒ a → [a] → Bool
-- context reduction using instance
elem :: Eq a ⇒ a → [a] → Bool
```

Essentially what happens is that using a type synonym with a context appends the instantiated context of the synonym to the appropriate set of assumptions (following the same rules that we use for functional predicates).

5.5 Data Types

Haskell's **data** declarations are used by programmers to define new types. Each data declaration specifies a list of constructors, and each constructor may contain a number of fields. If we allow the functional predicate notation to be used in data declarations, then we need a context that can store the extra assumptions that are introduced by the notation. Haskell 98 already supports contexts on data types, and so we will start by examining how these work.

Consider the following datatype declaration that has a context:

```
data Eq a ⇒ EqList a = Nil | Cons a (EqList a)
```

In Haskell 98 [76], this declaration introduces a new type `EqList` (of kind $* \rightarrow *$), and the constructors `Nil` and `Cons` with the following types:

```
Nil  :: EqList a
Cons :: Eq a ⇒ a → EqList a → EqList a
```

Notice that some of the constructors inherit (parts of) the context. The rule is that the context of a constructor contains only the predicates that restrict the types of the constructor's fields. This is why `Cons` is qualified by the `Eq` constraint, while `Nil` is not.

In the Haskell 98 semantics for overloaded constructors, the evidence supplied to overloaded constructors is not stored with the newly constructed value. This is why overloaded constructors generate extra constraints corresponding to their context when they are used in patterns. Here is an example that illustrates this behavior:

```
elem :: (Eq a) ⇒ a → EqList a → Bool
elem _ Nil          = False
elem x (Cons y ys)  = x == y || elem x ys
```

The Haskell 98 **data** declarations cannot support the functional predicate notation because the only variables that may appear in their context and fields are the datatype parameters, whereas the translation of our notation may introduce new type variables. Fortunately, it is not hard to extend Haskell 98 to avoid this problem. We use the fact that the new type variables are determined by the datatype parameters and the functional dependencies on the predicates in the context. This is important because it ensures that the datatype constructors do not have an ambiguous type. As an example, consider the following declarations:

```
class F a b | a ⇝ b

data T a = C (F a)
```

In the definition of `T`, we used the abbreviation `F a`, which would be desugared to the following code:

```
data (F a b) ⇒ T a = C b
```

Notice that we have introduced a new variable `b`, but it is determined from the parameter `a` via the functional dependency on `F`. The constructor `C` has the following type:

```
C :: (F a b) ⇒ b → T a
```

The type variable `b` appears in the arguments to `C` but not in the result so it is essentially existentially quantified. Apart from that, the qualified constructor behaves as it would in a Haskell 98 **data** declaration with context:

```
getField :: (F a b) ⇒ T a → b
getField (C x) = x
```

5.5.1 Constructor Contexts

Using a single context for the abbreviations arising from all constructors leads to some constructors getting types that are more complicated than is perhaps necessary. Consider, for example, a declaration like the following:

```
class F a b | a ⇝ b
class G a b | a ⇝ b

data T a  = C1 (F a) | C2 (G a)
```

Following the rules that we have so far, this would result in the following types for the constructors:

```
data (F a b1, G a b2) ⇒ T a = C1 b1 | C2 b2

C1 :: (F a b1, G a b2) ⇒ b1 → T a
C2 :: (F a b1, G a b2) ⇒ b2 → T a
```

Notice that both constructors get both predicates from the context, which makes the types more complicated than they need to be. This happens because we are using a single context for the entire datatype. To avoid the mixing of predicates relating to the different constructors we could make another small change to Haskell 98 so that each constructor in a datatype can have its own context. This is useful because it gives more control to the programmer to specify exactly how types should be constrained. Furthermore, if we use the functional predicate notation, then we can constrain each constructor by only the predicates that arise from abbreviations used in its own fields. For example, this is what would happen if we were to use this approach on the previous example:

```
data T a = (F a b1) ⇒ C1 b1 | (G a b2) ⇒ C2 b2

C1 :: (F a b1) ⇒ b1 → T a
C2 :: (G a b2) ⇒ b2 → T a
```

Of course, if we pattern match with multiple constructors, then we still get the union of the predicates on the constructors in the patterns.

5.5.2 Storing Evidence

So far, we have followed Haskell 98's design choice for contexts on data types, in that constructors do not store the evidence for their contexts with the newly constructed value. The immediate effect of this is that using constructors in patterns adds additional constraints to the type of the function that performs the pattern matching. Another design choice would be to store the evidence with the value, which is similar to what happens with constraints on existentially quantified variables [56, 65, 19]. This would eliminate the need to add extra constraints to functions that perform pattern matching because the evidence will be provided as a part of the the argument that is being examined, which results in simpler looking types. For example, recall the type `EqList`:

```
data Eq a ⇒ EqList a = Nil | Cons a (EqList a)

elem x Nil          = False
elem x (Cons y ys)  = x == y || elem x ys
```

If we adopt the semantics where constructors store evidence with the constructed values, then we get the following types:

```
Nil  :: (Eq a) ⇒ EqList a
Cons :: (Eq a) ⇒ a → EqList a → EqList a

elem :: a → EqList a → Bool
```

There is no extra `Eq` constraint on the function `elem` because the evidence that `a` belongs to class `Eq` is packaged together with the list value, and we can access it when we pattern match on the argument. It is interesting to note that even if we package values with their evidence, there are situations where we may require identical evidence to be passed separately to a function. This may occur if we need to use the evidence *before* pattern matching on a value. For an example, consider the following function:

```
knownName x y env
  | x == y    = True
  | otherwise = elem y env
```

This function uses the equality operator *before* examining the `EqList` argument `env`. It cannot use the evidence stored in the `env` argument because there is no guarantee that this argument has been evaluated. Therefore, we infer the following type for `knownName`:

```
knownName :: (Eq a) ⇒ a → a → EqList a → Bool
```

If we use this design, then there is also an interesting interaction between constructors and predicates with functional dependencies. We can illustrate the problem with an example:

```
class F a b | a ⇝ b
data (F a b) ⇒ T a = C a b
-- C :: (F a b) ⇒ a → b → T a

f (C _ b) = b
```

What is the type of `f`? If we use the semantics where constructors do not store evidence, then we get:

```
f :: (F a b) ⇒ T a → b
```

However, if we use the semantics where constructors store the evidence, we do not have the extra predicate to relate the type of the parameter to `T` to the type of the result of the function.

The paper that introduced the idea of using functional dependencies in the Haskell class system [51] suggests that it may be useful to give names to functional dependencies. It seems that such a feature may help with the above problem. If the name of the functional dependency on `F` is `u`, then we could give `f` the following type:

```
f :: T a → u a
```

Then we would need to extend the system further so that it can check equalities between types including this new form of type. It seems likely that such a system would use some sort of constraint capturing equality between types, and indeed there is some recent research in this direction [86]. The idea of named functional dependencies is also closely related to the idea of *associated type synonyms* [20], which we shall discuss in more detail in the next section.

5.6 Associated Type Synonyms

Recently, there was a proposal to allow type synonyms to be associated with Haskell's type classes [20]. For example, this is how we could define a class that captures some general operations on graphs using associated type synonyms:

```
type Edge g = (Node g, Node g)
class Graph g where
  type Node g
  outEdges :: Node g → g → [Edge g]
```

In this example, `g` is a type that represents graphs, `Node g` is the type of the nodes in the graph, and `outEdges` is a function that computes the outgoing edges from a given node in the graph. The novel component here is the associated type synonym `Node g`. When programmers define instances of the class `Graph`, they need to define the type `Node g` as well as the method in the class.

```
type AdjMat = Array (Int,Int) Bool
instance Graph AdjMat where
  type Node AdjMat = Int
  outEdges = ...
```

The essence of the idea is that `Node` is a function on types whose domain is restricted to types that are in the `Graph` class. Every instance of `Graph` provides another equation for the type function `Node`.

Using Abbreviations. We can achieve something very similar by using ordinary functional dependencies and the notation for functional predicates. We will illustrate this by showing how the graph example can be recoded this way. First, we define a type function called `Node` that has graphs as its domain:

```
class Graph g ⇒ Node g n | g ⇝ n
```

Next, we define the `Graph` class pretty much as before:

```
class Graph g where
  outEdges :: Node g → g → [Edge g]
```

Note that using the abbreviation `Node g` makes the type of `outEdges` look just like the type that we get when we use associated type synonyms. The desugared type of `outEdges` is like this:

```
outEdges :: Node g n ⇒ n → g → [Edge g]
```

Defining instances for the `Graph` class is also very similar:

```
instance Node AdjMat Int
instance Graph AdjMat where
  ...
```

The encoding used here is entirely mechanical: given a class with some associated type synonyms, we define the class in the same way as we would if we had associated type synonyms, except that we replace each associated type with a new class that has a functional dependency and the original class as a super class.

This representation allows us to attach constraints to the associated type synonyms if we need them. For example, suppose that we wanted to state that all graphs should have nodes that are in the `Eq` class. We can modify the `Node` function to capture such a constraint:

```
class (Graph g, Eq n) ⇒ Node g n | g ⇝ n
```

The only difference from before is the `Eq` constraint on `n`. The proposal for associated type synonyms [20] does not provide this feature, although it could probably be extended along the same lines that we have shown here.

5.7 Summary

In this chapter, we have described some syntactic sugar and mild extensions to Haskell 98 that is useful in programs that use predicates with functional dependencies. The idea uses the fact that classes with functional dependencies introduce new type functions, which are defined by the instances of the class. With this in mind, we allow programmers to write partially applied predicates directly in types, and then leave it to an implementation to replace these with appropriately constrained type variables.

In the process of describing how this idea interacts with various Haskell constructs that contain types, we proposed several small extensions to Haskell 98, including contexts on type synonyms and class contexts that contain variables, which are not in the class head.

Finally, we showed how our notation—together with functional dependencies—can achieve expressiveness similar to associated type synonyms. All of these proposals would be subsumed by a mechanism for defining (perhaps constrained) functions at the type level.

Chapter 6

A Calculus for Definitions

In this chapter, we present a calculus for reasoning about programs that contain a rich definitional language, including patterns, pattern bindings, and functions defined with multiple equations. We use this calculus in two ways: (i) to define the semantics for the new patterns that we shall introduce in later chapters; and (ii) as an intermediate language for our compiler. While some of the components of the calculus are novel (e.g., guarded patterns), it resembles many of the intermediate languages found in the literature and in existing functional language implementations.

6.1 Overview

Modern functional languages typically use a rich definitional language that enables programmers to present fairly complex decision trees in a concise and readable form. Functions are commonly defined using a sequence of definitional equations that involve pattern matching. The left hand sides of these equations use patterns to specify when we should use a particular equation, while the right hand sides specify the result to be computed once an equation has been chosen. For example, consider the following definition from Haskell's Prelude:

```
filter p [] = []
filter p (x:xs)
  | p x       = x : rest
  | otherwise = rest
```

```
where
rest = filter p xs
```

This definition defines a function called `filter` that has two arguments. The first argument is a predicate called `p`, and the second argument is a list. The function `filter` traverses the list and constructs a new list that contains only those elements of its argument, that satisfy the predicate `p`. The first equation deals with the case where the argument list (and hence the result) is empty, written here as `[]`. The second equation deals with lists that have an element, `x`, at the front, followed by another list, `xs`. For such lists we need to consider two cases, here encoded with guards, which start with a vertical bar. Both guards may refer to the local variable `rest`, which recursively processes the rest of the list. The first guard is used when `x` satisfies the predicate `p`, and the second guard is used when it does not.

We can translate this definition into our calculus without too much difficulty:

```
filter = { p → [] → return []
         ▯ p → (x:xs) → let rest = filter p xs;
                        ( if p x; return (x:rest)
                        ▯ if otherwise; return rest
                        )
         }
```

Note that the two definitions are fairly similar and, indeed, this was our intention. As we shall see in the rest of this chapter, the calculus satisfies a number of laws that can be used to transform terms systematically. In this way, we can give semantics to rather complex definitional constructs, and also encode transformations, such as pattern matching compilation.

As a starting point for our calculus, we assume a language that has expressions and simple declarations (i.e., declarations that just give names to expressions). As usual we shall use a type system to weed out syntactically correct, but otherwise malformed constructs in the language. For expressions, we use judgments of the form:

$$\Gamma \vdash_{expr} e : \tau$$

As we have discussed in Chapter 3, the typing judgments for a language with qualified types need an additional environment, Π, for the set of assumptions that we can use to discharge predicates. To avoid clutter we omit Π from

the judgments in this chapter. For declarations we shall assume judgments of the form:

$$\Gamma \vdash_{decl} d : \Delta$$

This asserts that, in a typing environment Γ, d is a well-formed declaration that defines the variables described by the typing environment Δ.

In addition to expressions and declarations our calculus contains three more syntactic categories:

- *Matches* (discussed in Section 6.2) are like expressions that support pattern-matching failure. They have typing judgments of the form:

$$\Gamma \vdash_{mat} m : \bar{\tau} \to \tau$$

- *Qualifiers* (discussed in Section 6.3) are like declarations that are aware of pattern-matching failure. They have typing judgments of the form:

$$\Gamma \vdash_{qual} q : \Delta$$

- *Patterns* (discussed in Section 6.4) are used to examine and name values. They have typing judgments of the form:

$$\Gamma \vdash_{pat} p : \tau \to \Delta$$

After we discuss these constructs, Section 6.5 shows how to integrate the new constructs into the expressions and declarations of the basic language.

6.2 Matches

We use *matches* to describe function definitions and pattern bindings. Pattern bindings provide another way to define values. A pattern binding is like a normal value definition except that the left hand side of the definition may contain a pattern. If the pattern matches the value of the right hand side of the definition, then the variables in the pattern are introduced to the run-time environment, otherwise the program terminates prematurely (in a lazy language, the termination would be delayed until one of the values is actually used). The syntax for matches is as follows:

```
mat = pat → mat      -- argument
    | qual ; mat     -- guard
    | mat [] mat     -- alternatives
    | return expr    -- success
```

From the grammar, we can see that a match is a decision tree with forks at the [] symbol. The paths through the tree contain patterns (`pat`) and qualifiers `qual`. Each path is terminated with a match of the form **return** `expr`. The patterns on a branch indicate arguments to the function that is being defined, and so all paths should contain the same number of patterns. The branching construct, [], is left-biased, which means that we explore alternatives from left to right. Decisions in the tree are made either by patterns, which examine function arguments, or by qualifiers, which can perform arbitrary decisions based on variables that are in scope.

As we have already mentioned, not all syntactically valid matches are well-formed. The main well-formedness constraints are that the different alternatives in a match should all have the same number of arguments (of the same type), and that all alternatives should choose an expression of the same type. This is why the types for matches have the form $\bar{\tau} \to \tau$. As usual, the list of types before the arrow keeps track of the arguments to the match, while the type after the arrow is for the result of the match (note that this arrow is different from the function type in the language). With this in mind, the typing rules for matches are straightforward (see Figure 6.1).

$$\frac{\Gamma \vdash_{pat} p : \tau \to \Delta \quad \Gamma, \Delta \vdash_{mat} : \bar{\tau}_1 \to \tau_2}{\Gamma \vdash_{mat} p \to m : (\tau, \bar{\tau}_1) \to \tau_2}$$

$$\frac{\Gamma \vdash_{qual} q : \Delta \quad \Gamma, \Delta \vdash_{mat} m : \bar{\tau}_1 \to \tau_2}{\Gamma \vdash_{mat} q; m : \bar{\tau}_1 \to \tau_2}$$

$$\frac{\Gamma \vdash_{mat} m_1 : \bar{\tau}_1 \to \tau_2 \quad \Gamma \vdash_{mat} m_2 : \bar{\tau}_1 \to \tau_2}{\Gamma \vdash_{mat} m_1 [] m_2 : \bar{\tau}_1 \to \tau_2}$$

$$\frac{\Gamma \vdash_{expr} e : \tau}{\Gamma \vdash_{mat} \textbf{return}\ e : () \to \tau}$$

Figure 6.1: Typing rules for matches.

Denotationally, the meaning of a match $\Gamma \vdash_{mat} m : \bar{\tau_1} \to \tau_2$ is a function of type $[\![\Gamma]\!] \times [\![\bar{\tau_1}]\!] \to M\ [\![\tau_2]\!]$. Here, M is some monad that supports raising and handling exceptions. A simple example would be Haskell's `Maybe` monad, which would be sufficient for a pure language. Because, in general, matches may evaluate expressions, the choice of the monad in the semantics of matches is ultimately dependent on the monad that is used to give semantics to the rest of the language. We may capture this by using a monad transformer [59]: if the expression language is given semantics in a monad N, then we could use the monad `ExceptT N` to give semantics to the matches of the language. Computations in `ExceptT N` differ from computations in `N` in that a computation may fail to produce a result, indicating that pattern matching failed.

Another interesting observation about the semantics of matches is that the monad appears in only one place in the type of the semantic function. This indicates that a match examines all the arguments at the same time, before making a decision. This is in contrast to a type like `a → M (b → M (c → M d))`, which would examine the arguments one at a time.

In general, we expect matches to satisfy a number of laws. For example, the operation $[]$ should be associative. Also, once we make a successful choice, we never try any other alternatives, which is captured by the equation:

$$(\mathbf{return}\ e)\ []\ m\ =\ \mathbf{return}\ e$$

Note that, because $[]$ is left-biased, the symmetric version of this rule is not valid. There are other useful equations about the behavior of matches in our framework, including the following distributivity properties:

$$\begin{aligned} q; (m_1 [] m_2) &= (q; m_1) [] (q; m_2) \\ p \to (m_1 [] m_2) &= (p \to m_1) [] (p \to m_2) \end{aligned}$$

Such equations (if viewed as rewrite rules from right to left) are quite useful for pattern-matching compilation. Note that these equations would not hold in a language with side effects whose duplication may be observed, because on the left-hand-side we have a single occurrence of q and p, while on the right-hand side we have two such occurrences.

6.3 Qualifiers

We use qualifiers to make decisions and also to introduce new names to the environment. Qualifiers are described by the following grammar, and the typing rules for qualifiers are in Figure 6.2.

```
qual = pat ← expr    -- pattern guard
     | if expr       -- guard
     | let decl      -- declaration
     | qual ; qual   -- sequencing
```

The most interesting qualifier is the *pattern guard* [24], which has the form `p ← e`. A pattern guard succeeds if the expression `e` evaluates to a value that matches the pattern `p`. If this is the case, the variables defined by `p` are introduced to the environment, otherwise the pattern guard fails. For example, if `e` is some expression that evaluates to the value `Just 2`, then the pattern guard `Just x ← e` succeeds and binds `x` to 2. Alternatively, if `e` evaluates to `Nothing`, then the pattern guard fails.

Pattern guards enable us to make arbitrary decisions. Decisions can also be made with ordinary guards of the form **if** `e` that take a Boolean expression, and succeed if it evaluates to `True`, or fail otherwise. There is a close relation between guards and pattern guards. If the language supports a pattern to recognize the value `True`, then we can define guards as a special form of a pattern guard:

if `e` $\equiv$ `True ← e`

Qualifiers marked with the keyword **let** cannot fail. They are used to introduce new names to the current scope. We may also compose qualifiers, using `;` (semicolon).

Denotationally, we may think of a qualifier of type Δ as a function of type $[\![\Gamma]\!] \to M[\![\Delta]\!]$ where M is the same monad that we used for the semantics of matches.

The reader may have noticed that we used the same notation for sequencing of qualifiers, as we used for separating qualifiers and matches. This is justified by the following rule relating qualifiers and matches.

$$(q_1; q_2); m \;=\; q_1; (q_2; m)$$

The fact that **let** qualifiers cannot fail justifies a rule that enables us to move **let** qualifiers in and out of expressions:

$$\textbf{let}\ d; \textbf{return}\ e \;=\; \textbf{return}\ (\textbf{let}\ d\ \textbf{in}\ e)$$

$$\frac{\Gamma \vdash_{pat} p : \tau \to \Delta \quad \Gamma \vdash_{expr} e : \tau}{\Gamma \vdash_{qual} p \leftarrow e : \Delta}$$

$$\frac{\Gamma \vdash_{decl} d : \Delta}{\Gamma \vdash_{qual} \textbf{let}\ d : \Delta}$$

$$\frac{\Gamma \vdash_{expr} e : \mathit{Bool}}{\Gamma \vdash_{qual} \textbf{if}\ e : \{\}}$$

$$\frac{\Gamma \vdash_{qual} q_1 : \Delta_1 \quad \Gamma, \Delta_1 \vdash_{qual} q_2 : \Delta_2}{\Gamma \vdash_{qual} q_1; q_2 : \Delta_1, \Delta_2}$$

Figure 6.2: Typing rules for qualifiers.

6.4 Patterns

Patterns provide a way to make decisions by examining a value. Different languages have different pattern forms, and there have even been proposals for user-defined patterns. In this section, we shall restrict ourselves to a tiny set of patterns that would be useful in many languages. The syntax of patterns is specified by the following grammar, and the typing rules are in Figure 6.3.

```
pat = x              -- variable
    | (pat | qual)   -- guarded pattern
```

Variable patterns are standard and, because they match any arguments without examining them, they never fail. Guarded patterns, on the other hand, are more unusual, but they provide a flexible way to examine values. A value matches a guarded pattern `(p | q)`, if it matches `p` and, in addition, it also satisfies the tests specified by the qualifier `q`. We may think of a guarded pattern as a qualifier with a parameter.

In a denotational framework, a pattern of type $\tau \to \Delta$ is a function of type $[\![\Gamma]\!] \times [\![\tau]\!] \to M[\![\Delta]\!]$. Again, the monad M is the same as the one in matches and qualifiers. There are two equations that relate guarded patterns

$$\frac{}{\Gamma \vdash_{pat} x : \tau \to \{x : \tau\}}$$

$$\frac{\Gamma \vdash_{pat} p : \tau \to \Delta_1 \quad \Gamma, \Delta_1 \vdash_{qual} q : \Delta_2}{\Gamma \vdash_{pat} (p|q) : \Delta_1, \Delta_2}$$

Figure 6.3: Typing rules for patterns.

to matches and qualifiers:

$$\begin{aligned}(p \mid q) \to m &= p \to (q; m)\\ (p \mid q) \leftarrow e &= (p \leftarrow e); q\end{aligned}$$

Nesting of guarded patterns corresponds to sequencing of qualifiers, as illustrated by the following equation:

$$((p \mid q_1) \mid q_2) = (p \mid q_1; q_2)$$

Using this basic framework, we can add other pattern forms that are common in modern functional languages. For example, a numeric literal pattern can be easily represented as a guarded pattern like this:

```
k  ≡  (x | if x == k)
```

In this definition, `k` is a numeric literal, and `x` is a new variable that we use to name the argument to a function. Notice that guarded patterns are more general than ordinary guards because they allow us to have guards nested within other patterns. Ordinary guards in Haskell may only appear at the end of each definitional equation.

A slightly more advanced form of numeric pattern is the so called `(n + k)` pattern:

```
n + k  ≡  (x | if x >= k; let n = x - k)
```

This definition shows us precisely how `(n+k)` patterns work: first we check that the argument is greater than the literal `k` and, if so, then we bind the variable `n` to the new value. In Haskell, `k` has to be a literal. But, as this equation shows, we could easily extend this to also allow an arbitrary expression.

Haskell's *as* patterns (written with `@`) also fit naturally in this framework:

```
x @ p  ≡  (x | p ← x)
```

We can even add algebraic patterns in a similar fashion, assuming the presence of suitable primitives that can: (i) distinguish between values created with different constructors; and (ii) project fields from constructors. If we have patterns for algebraic datatypes, then we can use our language to write down rules that essentially perform pattern matching compilation. For example, here is a rule that shows how to simplify algebraic patterns:

```
C p1 p2 ≡  (C x y | p1 ← x; p2 ← y)  -- if defs(p1) not in fvs(p2)
```

The side condition states that variables defined by `p1` should not be mentioned in the free variables of `p2`. In many functional languages, patterns do not contain free variables at all, so the side condition would be trivially satisfied.

Rules like the previous one are useful because they make many details of how patterns work explicit. For example, we can see that the fields in a constructor are examined from left to right. Such details are important in languages like Haskell because the order in which patterns are examined determines the order of evaluation of the program, and so it may affect the termination properties of the program. They are also important in impure languages because then the order of evaluation determines which side effects are executed.

6.5 Expressions and Declarations

In general, different languages have different expressions and declarations. To integrate matches and qualifiers with expressions and declarations, we add a new expression and a new declaration:

```
expr  = { mat } -- match
      | ...     -- other expressions

decl  = { qual } -- qualifier
      | ...      -- other declarations
```

The new expression enables us to embed matches within expressions, and the new declaration enables us to use qualifiers as declarations.

An expression of the form `{m}` evaluates the match `m` and, if the matching is successful, then the result of the expression is the result of the match. If

the match fails, then the value of the expression is undefined. In a strict language, this would result in the program terminating early, while, in a lazy language, the termination may be delayed until the value of the expression is used.

Declarations of the form `{q}` behave in a similar fashion. The values defined by the declaration are the same as the values defined by the qualifier. If the qualifier succeeds, then all values are defined as expected. If the qualifier fails, then the values introduced to the environment are undefined. In a strict language a failure of the qualifier would result in a run-time pattern match failure, similar to what happens if a function is not defined for all possible input arguments.

As we discussed previously, matches and qualifiers are given semantics with the aid of a monad that supports exceptions. We may think of the braces, `{_}`, as an operation that "eliminates" the failure component of the monad, by turning it into undefined values. In a denotational semantics, such values are represented with the bottom element of the relevant domain (i.e., the element that contains no information). Note that, in the case of qualifiers, environments are given semantics as an element of an unlifted product of domains, and so the bottom element is a tuple that contains the bottom elements for each of the values that are being defined.

The braces are not necessary when there is no possibility of failure. We can capture this observation with the following equations:

$$\{\textbf{return } e\} = e$$
$$\{\textbf{let } d\} = d$$

6.5.1 Simplifying Function Definitions

Recall the definition of `filter` from the beginning of this chapter:

```
filter = { p → [] → return []
         [] p → (x:xs) → let rest = filter p xs;
                         ( if p x; return (x:rest)
                         [] if otherwise; return rest
                         )
         }
```

We can now apply the rules of the calculus to simplify this definition. For example, we can factor the patterns `p` out of the branches of the `[]`. By using

a bit of inlining, we notice that `otherwise` is simply defined to be `True`, and so we do not need the second guard in the nested []. This would make it easy for a tool that analyzes definitions to notice that the definition of `filter` is exhaustive, and to infer that the program will not crash at run-time due to a pattern match failure. Another useful transformation that an implementation might do is to pick explicit names for the arguments of the function. This could be achieved with a rule like this (it is interesting to note that this rule resembles the η rule of the λ calculus):

```
p = (x | p ← x)
```

We should mention one small detail concerning rules like this one that introduce new variables: If we look carefully at the types of the patterns on the left and right hand side of the equation, we notice that they differ. This happens because the types of the patterns contain the variables that are defined by the pattern, and on the right-hand side we have an extra variable. It is a bit unusual to state equalities between values that are of different types, but what we mean is that the two patterns are the same, except for the newly introduced variable. One way to fix that would be to introduce an explicit operator that delimits the scope of the variables defined by patterns.

By using the naming rule together with the other rules in the calculus, an implementation could systematically eliminate matches of the form `p → m` by first naming all patterns:

```
p → m
= (naming)
(x | p ← x) → m
= (guarded pattern in match)
x → (p ← x) ; m
```

Variable patterns are a lot more "mobile" then other patterns because they don't make any decisions. In particular, we can move them across qualifiers (so long as no name capture occurs):

```
q ; x → m  =  x → q ; m     -- if x not in fvs(q)
```

This, together with the distributivity rule for patterns, enables us to move all arguments to the beginning of a match. Finally, we can use a rule that allows us to move variable patterns out of matches (assuming that we have function expressions):

```
{x → m} = λx.{m}
```

If we apply these transformations to the example with the definition of `filter`, then we get the following result, which is useful for generating code.

```
filter = λp. λys. { [] ← ys; return []
                  [] (x:xs) ← ys;
                     let rest = filter p xs;
                     ( if p x; return (x:rest)
                     [] return rest ) }
```

6.5.2 Pattern Bindings

We can also use our calculus to encode pattern bindings. The most common form of a pattern binding is:

```
pat = expr
```

We can express this directly in our calculus with a declaration like this:

```
{ pat ← expr }
```

In addition to ordinary pattern bindings, Haskell also supports more exotic pattern bindings as illustrated by the following example:

```
(version,name)
  | debug     = (1, name ++ " (DEBUG)")
  | otherwise = (2, name)
  where name = "Program"
```

This is a pattern binding that uses guards, and it has a local definition. Notice also that the local definition shadows one of the names that is being defined. The corresponding expression in our calculus is:

```
{ (version,name) ← { let name = "Program"
                   ; ( if debug; return (1, name ++ " (DEBUG)")
                     [] if otherwise; return (2, name) ) } }
```

In a similar fashion, we can also represent Haskell's lazy patterns in the calculus. These patterns are unusual because, unlike other Haskell patterns, they do not force evaluation. Instead, they behave like a pattern binding and succeed immediately. The check if an argument matched the pattern is only performed when the value of one of the bound variables is demanded. If the pattern match fails at this point, then the program crashes. We can make all of this explicit in the calculus like this:

```
~p ≡ (x | let { p ← x })
```

Here is an example that uses the calculus to show that we never need to use a lazy pattern in a pattern binding:

```
~p = e
= (defn of pattern binding)
{ ~p ← e}
= (defn of lazy pattern)
{ (x | let { p ← x }) ← e}
= (simplify guarded pattern)
{ x ← e; let { p ← x } }
= (pattern guard-let)
{ let x = e; let { p ← x } }
= (let sequence)
{ let (x = e; { p ← x }) }
= (let does not fail)
x = e; { p ← x }
= (inline x)
{ p ← e }
= (definition of pattern binding)
p = e
```

6.6 Summary

In this chapter, we have presented a calculus that is useful for manipulating and understanding the various definitional constructs in a modern functional language. The calculus is rich enough to support a direct encoding of advanced definitional constructs. We have identified a number of equational laws that enable us to simplify expressions in the calculus in a systematic fashion.

In this dissertation, we use the definitional calculus to give precise semantics for the new pattern-matching constructs that are described in later chapters. In addition, we used a variant of this calculus as an intermediate language for the prototype implementation that we developed to experiment with our design.

Part II

Language Design

Chapter 7

Working With Bitdata

In this chapter, we explain how a modern functional language like ML or Haskell can be extended with mechanisms for specifying and using bitdata. Our design provides fine-control over the choice of low-level representation while also supporting the higher-level programming notations and constructs that are associated with strongly typed, algebraic datatypes. The original motivation for this work grew out of two ongoing projects using Haskell for device driver and operating system implementation, respectively. We have strived, however, for a general approach that is broadly compatible with any modern functional language, and also for a flexible approach that is useful in many different application domains[1].

This chapter is structured as follows: In Section 7.1, we present an overview of our design, which has two components:

- Support for working with bit vectors is described in Section 7.2,
- Support for user-defined bitdata types is described in Section 7.3.

We discuss the interaction between these two components in Section 7.4.

[1]The material in this Chapter is based on: Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. High-level Views on Low-level Representations. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 168–179, Tallinn, Estonia, September, 2005.

7.1 Overview of the Approach

Algebraic datatypes promote a high-level view of data that hides many low-level implementation details. We can easily create new values using constructor functions, while pattern matching provides a way to inspect values without concern for their underlying, machine-level representation. We would like to provide programmers with the same flexibility for manipulating bitdata types. But how will a compiler determine the appropriate external representation for the type? Looking again at the examples from Chapter 1, it is easy to construct the following algebraic datatypes for PCI addresses, the L4 Time type, and for the Z80 encoding of bit twiddling instructions:

```
data PCIAddr = PCIAddr { bus::Int8, dev:: Int5, fun :: Int3 }

data Time  = Now
           | Period { e::Int5, m::Int10 }
           | Never

data BitOp = Shift {shift::S, reg::R}
           | BIT   {reg::R,   n::Int3}
           | RES   {reg::R,   n::Int3}
           | SET   {reg::R,   n::Int3}
data S     = RLC | RRC | RL | RR | SLA | SRA | SRL
data R     = A | B | C | D | E | H | L | MemHL
```

While it is possible that we might be able to infer the correct layout for types like `PCIAddr`, it is clear that the general case requires a more expressive notation for specifying bitdata representations. For example, there are no indications in these definitions alone that `Time` values should be represented using 16 bits; that `Now` and `Never` should be coded as special cases of `Period`; that the bit pattern `110` should not be used in the encoding of `S`; or that the bit pattern `111` should be used to represent `A` instead of the bit pattern `000`. To address these issues, we present a new approach to specifying and working with bitdata that is structured as two language extensions called *bit vectors* and *user defined bitdata*.

Bit Vectors. Bit vectors support basic, bit-level manipulation with a family of primitive `Bit n` types, a syntax for bit literals, and a `#` operator that can be used both for concatenating bit values and, in the context of pattern

matching, splitting bit values. The following examples give a brief flavor of the programs that we can write with this language extension:

```
mkPCIAddr             :: Bit 8 → Bit 5 → Bit 3 → Bit 16
mkPCIAddr bus dev fun  = bus # dev # fun

bitOpType             :: Bit 8 → Bit 2
bitOpType (tag # args) = tag
```

Programs like this can be parsed, type checked, and executed using `hobbit` (a **h**igher-**o**rder language with **bit**-level data), which is a prototype compiler that we have built to test and evaluate our bitdata extensions. The following extract shows how the `mkPCIAddr` and `bitOpType` functions might be used in an interactive session with `hobbit` (the `>` character is the `hobbit` prompt):

```
>show (mkPCIAddr 1 6 3)
"B0000000100110011"
>show (bitOpType 0x7f)
"B01"
```

The output from these examples also shows the syntax that is used for bit literals; an initial `B` followed by a sequence of binary digits.

User Defined Bitdata. User defined bitdata adds a mechanism for defining new bitdata types that are distinguished from their underlying representation. In special cases, the layout of these `bitdata` types can be inferred from the way that the type is written. For example, our system will infer the intended 16 bit representation of a `PCIAddr` from the following definition:

```
bitdata PCIAddr = PCIAddr { bus::Bit 8, dev::Bit 5, fun::Bit 3 }
```

In general, however, it is necessary to specify layout explicitly by annotating each constructor with an appropriate `as` clause. The following definition shows how the `Time` type can be described in this notation.

```
bitdata Time = Now                            as B0 # 1 # (0::Bit 10)
             | Period { e::Bit 5, m::Bit 10 } as B0 # e # m
             | Never                          as 0
```

Note that the representation for `Never` is written simply as `0`; the fact that a sixteen-bit zero is required here is inferred automatically from the other two **as** clauses.

The representation for Z80 bit twiddling instructions can be described in a similar way. In this case, we must specify the appropriate bit patterns for each of the constructors in the enumeration types `S` and `R`.

```
bitdata BitOp = Shift { shift::S, reg::R   } as B00 # shift # reg
              | BIT   { reg::R,   n::Bit 3 } as B01 # reg   # n
              | RES   { reg::R,   n::Bit 3 } as B10 # reg   # n
              | SET   { reg::R,   n::Bit 3 } as B11 # reg   # n

bitdata S     = RLC as B000 | RRC as B001 | RL  as B010 | RR  as B011
              | SLA as B100 | SRA as B101 |               SRL as B111

bitdata R     = A   as B111 | B   as B000 | C as B001 | D   as B010
              | E   as B011 | H   as B100 | L as B101 | MemHL as B110
```

With these definitions, we can construct byte values for different Z80 instructions using expressions like `Shift{shift=RRC, reg=D}` and `SET{n=6, reg=A}`, but attempts to construct encodings using arguments of the wrong type—as in `SET{n=6, reg=B010}`—will be treated as type errors, even in cases where values might otherwise be confused because they have the same number of bits in their representation.

Our system also includes generic `toBits` and `fromBits` operators that can be used to convert arbitrary bitdata to and from its underlying bit-level representations, and to provide a connection between the two language extensions. These are generalizations of the `toPCIAddr` and `fromPCIAddr` operations that were described in Section 1.3.2. The following example shows how the first of these function can be used to inspect the bit pattern for one particular Z80 instruction:

```
>show (toBits (SET{n=6, reg=A}))
"B11111110"
```

7.2 Bit Vectors

We introduce a new type constant called `Bit` (of kind `Nat` $\rightarrow$ `*`) that we use to type bit sequences. In other words, `Bit` is a type constructor that, given a natural number, produces the type of bit sequences of the corresponding length. For example, bytes are 8-bit sequences and have type `Bit 8`.

We focus on manipulating bit sequences that will fit in the registers of a CPU or a hardware device. It is therefore desirable to restrict the lengths of bit sequences that can be used in a program. Furthermore, operations on bit vectors of different sizes behave differently: for example, addition is performed modulo the size of the bit-vector, so `(bitAdd 3 3 = 6)` in 8 bits, but `(bitAdd 3 3 = 2)` in 4 bits. For these reasons, instead of providing operations that are completely polymorphic in the sizes of the bit-vectors, we use qualified types and overload the operations:

```
class Width n where
  bitEq       :: Bit n → Bit n → Bool
  bitCompareU :: Bit n → Bit n → Ordering
  bitCompareS :: Bit n → Bit n → Ordering
  bitAdd      :: Bit n → Bit n → Bit n
  bitAnd      :: Bit n → Bit n → Bit n
  bitFromInt  :: Integer → Bit n
  ...
```

Bit vectors are equipped with all the usual operations that one might expect. In practice, an implementation would provide instances for the bit vector widths that it supports (typically up to the size of a machine register `MaxWdith`, but in principle an implementation could provide support for larger bit vectors as well):

```
instance Width 0 where ...
instance Width 1 where ...
instance Width 2 where ...
...
instance Width MaxWidth where ...
```

Programmers do not need to work with the built-in bit operations directly. Instead, we could use the standard Haskell classes (for example) to provide a more conventional interface for working with bit vectors. Here is how we could define the instance for Haskell's class `Eq`:

```
instance Width n ⇒ Eq (Bit n) where
  x == y = bitEq x y
```

We may provide similar instances for a number of Haskell classes (e.g. Read, Show, Ord, Bounded, etc.), so that programmers may use bit vectors just like any other type that belongs to the relevant class.

Some of the operations on bit vectors come in different flavors. For example, if we think of bit vectors as binary representations for numbers, then it makes sense to order and compare them. However, the same bit pattern might represent different numbers depending on the encoding. For example, it is common to distinguish between signed and unsigned numbers. Thus, if we think of 3 bit vectors as unsigned numbers, then `010` (i.e., 2) is smaller then `100` (i.e., 4). However, if we think of them as signed numbers, then we are comparing 2 with -4 and so we get a different result.

For this reason, in the built-in operations, we provide two different comparison operators (`bitCompareU` and `bitCompareS`). However, if we want to provide a bit vector instance for Haskell's `Ord` class, then we have to pick one of them (e.g., bit vectors represent unsigned numbers). To provide instances for the other encoding we may use a new type that is isomorphic to the bit-vector types. For example:

```
newtype SignedBit n = Signed (Bit n)

instance Width n ⇒ Ord (Bit n) where
  compare x y = bitCompareU x y

instance Width n ⇒ Ord (SignedBit n) where
  compare x y = bitCompareS x y
```

Having a separate type for signed numbers is handy independent of Haskell's class system because the type explicitly indicates if we intend to work with signed or unsigned numbers.

7.2.1 Literals

One way to introduce a bit sequence in a program is to use a *binary literal*. This notation is useful when a bit vector is used as a name, for example, to identify a device, a vendor, or perhaps a particular command that needs to be sent to a device. A binary literal is written as a `B` followed by a number in base two. An `n` digit binary literal belongs to the type `Bit n`, as long as `n` is a valid width (i.e., it belongs to the `Width` class). Leading zeros are important because they affect the type of the literal. Here are some examples of binary literals, and the corresponding types:

```
> :t B11
Bit 2
```

```
> :t B011
Bit 3
> :t B00000000000000000000000000000000
FAIL
33 is not a valid width
```

This example uses the `:t <expr>` command in `hobbit` to show the type of an expression. In our implementation, the largest allowed width is 32, so the last example is not type correct because there are 33 zeros in the literal.

Binary literals may be used in both expressions and patterns. Indeed, we may think of `Bit n` as a kind of algebraic datatype that has the `n`-digit binary literals as constructors. For example:

```
data Bit 3 = B000 | B001 | B010 | B011 | B100 | B101 | B110 | B111
```

The only exception is the case when `n = 0`, where the name of the constructor is `NoBits`. To be consistent, we could have used the name `B` for the inhabitant of this type but we found that this can be confusing in practice.

It is often convenient to think of bit sequences as numbers and we introduce *numeric literals* to accommodate this. An interesting challenge is to allow numeric literals for all types of the form `Bit n`, without introducing a baroque notation. We do this by overloading the notation for octal, hexadecimal, and decimal literals, as in Haskell [76]. The trick is to define an instance of Haskell's `Num` class using the primitive function:

```
bitFromInt :: Width a ⇒ Integer → Bit a
```

A numeric literal n in the text of a program, can then be treated as syntactic sugar for the constant `fromInteger` applied to the value `n` of type `Integer`. Usually the type of an overloaded literal can be inferred from the context where it is used. If this is not the case, programmers can use a type signature to indicate the number of bits they need. Numeric literals may also be used in patterns and will match only if the argument is a value that is the same as the literal. Here are some examples that illustrate how literals work:

```
> :t 1
(Num a) => a
> :t bitAnd 1 B00110000
Bit 8
```

Notice that, when used on its own, the literal 1 has a polymorphic type—the system is telling us that 1 belongs to any numeric type. However, if used in a particular context, as in the second example where an 8 bit literal is required, then 1 will be converted to the appropriate type using the function `fromInteger`.

7.2.2 Joining and Splitting Bit Vectors

Another common programming task is joining and splitting bit sequences. The usual way of doing this is to use shift and mask operations to get bits into the correct positions. This is a complicated way to achieve a conceptually simple task, and it is all too easy to shift a bit too much, or to use the wrong bit mask. To make this task simpler, we introduce the operator `(#)` to join sequences. One way to type this operator is like this:

```
(#) :: Bit a → Bit b → Bit (a+b)
```

Notice that we use the notation for functional predicates here, as described in Chapter 5. In its desugared form, the type of `(#)` has a constraint indicating that we are essentially working with an overloaded operator:

```
(#) :: (a + b = c) ⇒ Bit a → Bit b → Bit c
```

We could use an operator of this type to join bit sequences of any type. While in principle this is possible, we introduced the predicate `Width`, so that implementations can control the sizes of the bit-vectors that they support. To be consistent with this design, we give `(#)` a slightly different type:

```
class (Width a, Width b, Width c, a + b = c)
   ⇒ (a # b = c) | a b ⇝ c, b c ⇝ a, c a ⇝ b where
  (#) :: Bit a → Bit b → Bit c
```

Here is the full type of the method `(#)`:

```
(#) :: Bit a → Bit b → Bit (a # b)
```

We can discharge the predicate `(#)`, as long as the bit vectors involved have acceptable widths, and the length of the result is equal to the sums of the lengths of the arguments. The super classes on the declaration can be used to simplify redundant contexts such as `(Width a, a # b = c)` to the equivalent `(a # b = c)`.

Here is an example of concatenating two bit vectors:

```
> show (B100 # B111)
"B100111"
```

As a another example, consider the following function:

```
mask x y = bitAnd (x # y) B100
```

The function `mask` is interesting because it is polymorphic in its arguments, which is accurately captured by its type:

```
mask :: (a # b = 3) ⇒ Bit a → Bit b → Bit 3
```

The concatenation operator is similar to a constructor because we can also use it in patterns to split bit sequences. A split pattern has the form `p # q` and matches bit vectors whose most significant part matches `p` and least significant part matches `q`. For example, a function to get the upper 16 bits of a 32 bit quantity could be written like this:

```
upper16        :: Bit 32 → Bit 16
upper16 (x # _) = x
```

Note that `#` patterns do not specify *how* to split a value into two parts, but simply what the two parts should match. How the sequence will be split depends on the types of the sub-patterns `p` and `q`. These types may be determined using type inference or from explicit signatures in the patterns. For example, if we define another function called `upper` that is the same as `upper16`, but we omit the type signature, then we get the following type:

```
> :t upper
(a # b = c) ⇒ Bit c → Bit a
```

As an example of a situation where we need to use a signature in a pattern, consider the function that extracts the bus component of a PCI address:

```
pciBus                              :: Bit 16 → Bit 5
pciBus ((dev :: Bit 8) # bus # fun) = bus
```

If we were to omit the annotation on the *dev* pattern, then type inference would fail. For example, our prototype reports the following error:

```
FAIL
Cannot solve goals: ?a + ?b = 16, ?c + 5 = ?a
```

The system needs to split a 16 bit quantity into two parts: one of width a (`dev # bus`), and one of width b (`fun`). It also has to split the a component into two parts: one that is c bits wide (`dev`), and one that is 5 bits wide (`bus`). There is not enough information in the program to determine how this splitting should be done, which is why we get the type error.

Signature patterns resemble the explicit types on functions in the presentation of the lambda calculus à la Church. At present, our design does not allow type variables in signature patterns. We could use lift this restriction by using scoped type variables [77].

7.2.3 Semantics of the (#) Pattern

In the previous example, we defined `upper` in terms of the pattern `(#)`. Alternatively, we can take `upper`, together with the symmetric function `lower` as the built-in primitives:

```
class (Width a, Width b, Width c, a + b = c)
   ⇒ (a # b = c) | a b ⇝ c, b c ⇝ a, c a ⇝ b where
  (#)   :: Bit a → Bit b → Bit c
  upper :: Bit c → Bit a
  lower :: Bit c → Bit b
```

This makes it explicit that we are essentially dealing with a product operation. Furthermore, using the guarded patterns from Chapter 6, we can also try to define the pattern `(#)` in terms of `upper` and `lower`:

```
p # q ≡ (x | p ← upper x; q ← lower x)
```

Unfortunately this is not quite correct because nothing in the definition states that we intend to split the value into two adjacent but non-overlapping parts. Consider, for example, the pattern `(x # y)`, where `x` and `y` are pattern variables. Our intention is that this pattern has the type (we use the keyword **pattern** to emphasize that we are specifying the type of a pattern, and not an expression):

```
pattern (x # y) :: (a # b = c) ⇒ Bit c → { x :: Bit a, y :: Bit b }
```

However, the type of the proposed 'definition' differs in an important way:

```
pattern (z | x ← upper z, y ← lower z) ::
  (a # b' = c, a' # b = c) ⇒ Bit c → { x :: Bit a, y :: Bit b }
```

The difference lies in the contexts. In the first case, we have only a single piece of evidence because we perform a single split. In the second case, we have two pieces of evidence, one arising from the use of `upper`, and one from the use of `lower`. The problem is that the two splits are completely independent of each other: what we want is `a = a'` and `b = b'`, but nothing in the definition forces these equalities.

We can remedy this problem in several different ways. One option is to replace the two operations `upper` and `lower` by a single operation `split`, which would result in a `(#)` class like this:

```
class (Width a, Width b, Width c, a + b = c)
   ⇒ (a # b = c) | a b ⇝ c, b c ⇝ a, c a ⇝ b where
  (#)   :: (Bit a, Bit b) → Bit c
  split :: Bit c → (Bit a, Bit b)
```

The intention here is that `(#)` and `split` form an isomorphism pair (we have uncurried `(#)` to emphasize the symmetry). It is then easy to define the pattern `(#)` in terms of split:

```
p # q ≡ (x | (p,q) ← split x)
```

Another option is to stick with `upper` and `lower` as the basic built-in primitives, but to modify the definition of the pattern `(#)` to make it explicit that `upper` and `lower` should use *the same* evidence. We can do this in the explicit calculus that we used in Chapter 3, resulting in the following definition:

$$(p\#q) \equiv \Lambda abc.\ \bar{\lambda} e : (a\#b = c).\ (\ x \mid p \leftarrow upper_{abc} \cdot e\ x;\ q \leftarrow lower_{abc} \cdot e\ x\ )$$

In this definition, Λ represents type abstraction, while type application is written with a subscript. We write evidence abstraction with $\bar{\lambda}$, while evidence application is written with a centered dot. The important difference from the previous definition is that `upper` and `lower` use the same evidence, e, which ensures that the value examined by the pattern is split correctly.

7.3 User-Defined Bitdata

In the context of systems programming, bit vectors are often used as representations for values that have more structure. To enable programmers to

capture this extra structure we introduce **bitdata** declarations to the language (Fig. 7.1). The grammar is specified using extended BNF notation: non-terminals are in italics and terminals are in a bold font; constructs in brackets are optional, while constructs in braces may be repeated zero or more times.

The syntax of **bitdata** declarations resembles **data** declarations in Haskell, because this is a common way to specify structured data. However, while there are many similarities between **data** and **bitdata** declarations, there are also important differences. For example, the type defined by a **bitdata** declaration is not the free algebra of its constructors (see Section 8.1). Instead, the type provides a kind of view [93] on the underlying bit sequences. Each constructor provides a convenient way to construct and recognize particular forms of bit sequences, while fields provide a means to access or update data components.

bdecl	=	**bitdata** *con* `=` *cdecl* {`|` *cdecl*}	type decl.
cdecl	=	*con* `{` [*fdecls*] `}` [**as** *layout*] [**if** *expr*]	constr. decl.
fdecls	=	*fdecl* {, *fdecl*}	field decl.
fdecl	=	*label* [`=` *expr*] `::` τ	
layout	=	*layout* `#` *lfield*	*layout* `::` τ	field layout
lfield	=	*lit*	`_`	`(` *layout* `)`	

Figure 7.1: The syntax of user-defined bitdata declarations.

7.3.1 Constructors

To illustrate how **bitdata** declarations work, we present some definitions for a device driver for a NE2000 compatible network card [70]. The exact details of how the hardware works are not important here; our goal is simply to illustrate the features of **bitdata** declarations. One of the commands to the NE2000 card contains a 2-bit field that specifies what the card should do. We can represent the format of this field with the following **bitdata** declaration:

```
bitdata RemoteOp
  = Read as B01 | Write as B10 | SendPacket as B11
```

This is essentially an enumeration type. The definition introduces a new type constant `RemoteOp` and three constructors `Read`, `Write`, and `SendPacket`.

Note the **as** clauses specify a bit pattern for each constructor; these patterns will be used to construct values with the constructor, or to recognize them in patterns. All constructors should have representations that are of the same width. The following examples use these constructors:

```
> :t Read
RemoteOp
> show Read
"B01"
> Read & B00
FAIL Type mismatch: Bit ?a vs. RemoteOp
```

The last example emphasizes the point that, even though `Read` is represented with the bit sequence `01`, it is not of type `Bit 2`.

The type `RemoteOp` captures only a fragment of the DMA commands available on NE2000 cards. The full set of DMA commands is described in the following definition:

```
bitdata DMACmd = Remote { op :: RemoteOp } as B0 # op
               | AbortDMA                  as B1 # _
```

This definition uses some more features of the **bitdata** declarations. In general, constructors may have a number of fields that describe sub-components of the value. For example, the constructor `Remote` has one field called `op` of type `RemoteOp`. The types of the fields should all have concrete bit vector representations, which enables us to compute a representation for the value. We shall make this property more formal in the next section.

To construct values with fields, we use the notation `C {`$\bar{l}$ `=` $\bar{e}$`}`, where `C` is the name of a constructor, $\bar{l}$ are its field name labels, and $\bar{e}$ are the values for the fields. The order of the fields is not significant. The fields of a constructor in a **bitdata** declaration may contain *default values*. The default value for a field is written after the field name and should be of the same type as the field. If a programmer does not initialize a field while creating a value with a particular constructor, then the field will be initialized with the default value for the field. If the field does not have a default value, then the program is invalid and the system will report a compile-time error.

There is also a corresponding pattern `C x`, which can be used to check if a value was constructed with the constructor `C`. The test is computed based on the **as** clause for the corresponding constructor: if all tag bits (i.e., non-field bits) match, then the pattern succeeds, otherwise it fails. It may be useful

to think of these patterns as a more structured version of the `(#)` operator that we used for bit vectors.

If the pattern `C x` succeeds, then the variable `x` is bound to a value that contains the fields of the constructor. If we think of a bitdata type as describing a sum-of-products, then pattern matching with a constructor eliminates the sum part and binds the variable `x` to the product part. We introduce a new type for the product component of each constructor. The name of the product type for a constructor is obtained by adding a prime to the name of the constructor. For example, the product type for `Remote` is called `Remote'`. We can use constructor functions to convert a product type back to the bitdata type. Here are the types of the constructor functions for `DMACmd`:

```
Remote    :: Remote' → DMACmd
AbortDMA  :: DMACmd
```

Note that, for constructors that have no fields (e.g., `AbortDMA`), we do not generate a product type, and the constructor function simply creates the appropriate value.

7.3.2 Product Types

The product types associated with a constructor are useful because they capture statically the extra information that the tag bits of the value match the pattern for the given constructor. Indeed, a similar approach could be useful for ordinary algebraic datatypes as well. To manipulate product types, we provide operations that can access the value for each of the fields in the product, as well as operations for updating field values. These operations are generated by the compiler and can be implemented internally using bit-twiddling. The names of the operations are obtained by combining the operation name, the name of the constructor, and the field name. For example, the product type `Remote'` has one field called `op`, and so we get the following two operations:

```
(get'Remote'op) :: Remote' → RemoteOp
(set'Remote'op) :: RemoteOp → Remote' → Remote'
```

To see how we might use these functions, here is an example of a function that will change remote read commands into remote write commands and leave all other DMA commands unchanged:

```
readToWrite :: DMACmd → DMACmd
readToWrite (Remote x) = Remote (upd (get'Remote'op x))
  where
  upd Read  = set'Remote'op Write x
  upd _     = x
readToWrite x = x
```

Clearly, the notation for manipulating product types is a little verbose. Our intention is that we should not work with these functions directly but instead, if possible, we should reuse the record system of the language that is being extended with bitdata. For example, Haskell 98 has a simple record system that allows field labels to be used only in a single type. If we were to take this approach, then we would not need to annotate the operations with the constructor names because the field names would be unique.

Records. Alternatively, we can use qualified types to overload the operations that manipulate records [41, 32]. In this way, we can use the same label for fields in different types and then rely on type inference to resolve the actual type that we are manipulating. The basic idea is to introduce a family of classes, one for each label, `l`:

```
class Field'l r t | l r ⇝ t where
  get'l :: r → t

class Field'l r t ⇒ UpdField'l r t | l r ⇝ t where
  set'l :: t → r → r
```

We split the operations in two (families of) classes so that we can support both read-only and updateable fields (we shall make use of this in Chapter 10). The predicate `Field'l r t` asserts that the record type `r` has a readable field `l` of type `t`. Similarly, `UpdField` asserts that we can both get the value of a field, and also create a new record with an updated value for the field. When we declare a new bitdata type, the compiler generates the appropriate instances for the two classes based on the fields for each constructor. For example, for the `DMACmd` type, the compiler will generate the following instances:

```
instance Field'op Remote' RemoteOp where
  get'op = get'op'Remote
```

```
instance UpdField'op Remote' RemoteOp where
  set'op = set'op'Remote
```

In addition to overloading the operations for working with product types, we also use some syntactic sugar when we work with records:

```
r.l           = get'l r
{ r | l = v } = set'l v r
```

The update notation also supports updating multiple fields, which can be desugared into nested uses of `set`. For example, writing `{ r | l1 = v1, l2 = v2 }` is the same as writing `set'l2 v2 (set'l1 v1 r)`.

Record Patterns. When we pattern match with a constructor, we often need to access the fields and perhaps make decisions based on their values. To make this easier, we introduce a special pattern that examines records: `{` $\overline{l}$ `=` $\bar{p}$ `}`. Such a pattern succeeds if the value of the field `l` matches the pattern `p`. More formally, we can translate a record pattern to the notation from Chapter 6 like this:

```
{ l1 = p1, l2 = p2 } = (x | p1 ← x.l1, p2 ← x.l2)
```

As an example, here is how we might rewrite the function `readToWrite` using the record notation:

```
readToWrite :: DMACmd → DMACmd
readToWrite (Remote { op = Read }) = Remote { op = Write }
readToWrite x = x
```

Note that, because we overloaded the operations for manipulating records, the record patterns are not specific to bitdata:

```
{ l = y } :: Field'l a b ⇒ a → { y :: b }
```

In Chapter 10, we will make use of this generality by reusing the same notation to access the fields of structures that are stored in memory.

7.3.3 The 'as' Clause

The **as** clause of a constructor may contain literals, field names, and wildcards (`_`), separated by `#`. Field names must appear exactly once, but can

be in any order. Type signatures are also permitted in the **as** clause. The representation for a constructor is obtained by placing the elements in the layout specification sequentially with the left-most component in the most significant bits of the representation. For example, the layout specification for the constructor `Remote` says that we should place `0` in the most significant bit and that we should place the representation for the field `op` next to it:

```
> show (Remote { op = Read })
"B001"
```

The **as** clause is also used to derive tests that will recognize values corresponding to the constructor. The matching of a pattern `C p` proceeds in two phases: first, we see if the value is a valid `C`-value, and then we check if the nested pattern `p` matches. The tests to recognize `C`-values check if the bits of a value corresponding to literals in the **as** clause match. For example, to check if a value is a `Remote`-value we need to check that the most significant bit is 0.

It is possible to use an alternative choice for the semantics of bitdata constructor patterns `C x`. In particular, when we match to see if we have a `C` value, we could consider not only the literals in the **as** clause, but also the possible values for the fields of the constructor. To see the difference between the two choices, consider the following example:

```
bitdata Tag1 = A as B00 | B as B11
bitdata Tag2 = C as B01 | D as B10

bitdata T    = Tag1 { tag :: Tag1, val :: Bit 30 } as tag # val
             | Tag2 { tag :: Tag2, val :: Bit 30 } as tag # val

f (Tag1 x)   = 1
f (Tag2 x)   = 2
```

If we consider the definition of `T`, there are no literals ('tag' bits) in the **as** clauses of either of the constructors. Therefore, using the first semantics of pattern matching on bitdata constructors we cannot distinguish between `Tag1` and `Tag2` values, and so the function `f` will produce a result of 1 for any input. If we use the second semantics, however, and consider the possible values for the fields, then the `Tag1` pattern will only succeed if the two most significant bits are `B00` or `B11`, as these are the only possible values for the type `Tag1`.

The first design choice has the benefit that it is simpler and results in more efficient code than the second choice, but it has the drawback that, in some cases, like the previous example, may result in surprising behavior. The first design is simpler because, to decide if a pattern matches we only need to look at the **as** clause for the relevant constructor, while with the second choice we would need to examine the definitions of the types for all the fields (and in turn, the definitions of their fields). Of course, the implementation could compute the set of matching bit patterns and simply show it to the programmer upon request. Still, because the predicate associated with matching a constructor is more complex with the second design choice, the code that we have to generate is less efficient. Furthermore, with some more unusual bitdata, such as the pointers that we will introduce in later chapters, it is difficult to compute a set of valid bit patterns. For these reasons, in our implementation we used the first design choice. Programmers can recover the behavior of the second design choice by using explicit **if** clauses together with the automatically generated `isJunk` predicate, both of which are described in the following sections. Here is how we could do this for previous example:

```
bitdata Tag1 = A as B00 | B as B11
bitdata Tag2 = C as B01 | D as B10

bitdata T    = Tag1 { tag :: Tag1, val :: Bit 30 } as tag # val
                                                    if not (isJunk tag)
             | Tag2 { tag :: Tag2, val :: Bit 30 } as tag # val
                                                    if not (isJunk tag)
```

Wild cards in the layout specifications represent "don't care" bits and do not play a role in pattern matching. For value construction, they have an unspecified value. The only constraint on a concrete implementation is that the "don't care" bits for a particular constructor are always the same. This is necessary if we want to convert bitdata values to bit vectors. In Section 7.4, we shall discuss such a function, called `toBits`. Using this function, programmers may observe the value of the "don't care" bits. Therefore, it is important that `toBits` returns the same bit pattern when applied to arguments created with the same constructor (and same fields). For example, the `AbortDMA` constructor only specifies that the most significant bit of the command should be 1 and the rest of the bits are not important, but in a particular implementation the compiler could choose a representation in which all the remaining bits are also zeros.

Constructors that have no **as** clause are laid-out by placing their fields sequentially, as listed in the declaration. This is quite convenient for types that do not contain any fancy layout (e.g., the type `PCIAddr`). Following this rule, the representation of constructors with no fields and no **as** clause, is simply `NoBits`, the value of type `Bit 0`. Such examples are not common, but this behavior has some surprising consequences. By analogy with the syntax of algebraic datatypes in Haskell, a new user may try to define a type for Boolean values like this:

```
bitdata MyBool = MyFalse | MyTrue
```

This is a legal definition, but it is probably not what the user intended: both constructors end up being represented with `NoBits` and are thus the same. Our implementation examines **bitdata** declarations for constructors with overlapping representations and warns the programmer to alert them of potential bugs. In hobbit, this example will trigger a warning about overlapping representations that will alert the programmer to what is, in this case, a likely bug.

7.3.4 The 'if' Clause

In some complex situations, the pattern derived from the layout of a value is not sufficient to recognize that the value was created with a particular constructor. Occasionally it may be necessary to examine the values in the fields as well. For example, the `LD` instruction of the Z80 processor should never contain the register `MemHL` as both its source and its destination. In fact, the bit pattern corresponding to such a value is instead used for the `HALT` instruction:

```
> show (LD { src = MemHL, dst = MemHL })
"B01110110"
> show HALT
"B01110110"
```

One way to deal with complex definitions is to include an explicit guard [76] in any definition that pattern matches on `LD`. For example:

```
instrName (LD { src = x, dst = y })
  | not (x == MemHL && y == MemHL) = "Load"
instrName Halt = "Halt"
```

This approach works but it is error prone because it is easy to forget the guard. To avoid such errors, a bitdata definition allows programmers to associate a guard with each constructor by using an **if** clause with a Boolean expression over the names of that constructor's fields. The expression is evaluated after the tests derived from the **as** clause have succeeded and before any field patterns are checked. If the expression evaluates to `True`, then the value is recognized as matching the constructor, otherwise the pattern fails. For example, this is how we could modify the definition of `Instr` to document the overlap between `LD` and `HALT`

```
bitdata Instr
 = LD { dst::Reg, src::Reg } as B01 # dst # src
   if not (src == MemHL && dst == MemHL)
 | HALT as 0x76
 ...
instrName (LD _)  = "Load"
instrName HALT    = "Halt"
```

We may use the function `instrName` to experiment with this feature:

```
> instrName (LD { src = A, dst = MemHL })
"Load"
> instrName (LD { src = MemHL, dst = MemHL })
"Halt"
> instrName HALT
"Halt"
```

As the second example illustrates, the **if** clause is used only in pattern matching and not when values are constructed. We made this design choice because it is simple and avoids the need for partiality or exceptions, which could otherwise arise when the **if** clause is used during value construction. The cost of this choice is minimal because, if necessary, programmers always have the option to define 'smart constructors' that validate the fields before constructing a bitdata record. For example, here is how we could define a smart constructor for the `LD` instruction:

```
ld :: Reg → Reg → Maybe Instr
ld MemHL MemHL = Nothing
ld x y         = Just (LD { src = x, dst = y })
```

7.4 Bitdata and Bit Vectors

7.4.1 Conversion Functions

There is a close relation between the bit sequence types `Bit n` and bitdata types like `RemoteOp`, because they are both represented with fixed bit-patterns. However, not all types in the language may be converted to and from bit vectors. Some types are completely abstract and cannot easily be converted to bit sequences (e.g., function values). Other types have concrete bit representations, but may also have extra constraints (an example are the array indexes to be discussed in Chapter 10). Finally, we have types that are basically *views* on bit vectors, and so we can convert them both to and from bit-vectors. To accommodate these different levels of abstraction we use two classes: `BitRep` for types that can be converted to bits, and `BitData` for types that can be converted both to and from bits. Both of these predicates track the number of bits that are required to represent a value. We use a functional dependency to specify that the number of bits is uniquely determined by the type of the value.

```
class BitRep t n | t ⇝ n where
  toBits :: t → Bit n

class BitRep t n ⇒ BitData t n | t ⇝ n where
  fromBits :: Bit n → t
  isJunk   :: t → Bool
```

The function `toBits` converts values into their bit-vector representations, while the function `fromBits` does the opposite, turning bit sequences into values of a given type. The function `isJunk` identifies 'malformed' values that do not match any of the constructor patterns of the type. We may think of `toBits` as a pretty-printer, and of `fromBits` as a parser that use bits instead of characters. These functions can be very useful when a programmer needs to interact with 'the outside world'. The function `toBits` will be used when data is about to leave the system, and the function `fromBits` is used when data enters the system. Our intention is that bitdata values are represented internally with the bit patterns specified in their declaration. In this case, these two functions can be implemented as simple identity functions that do not inspect or modify their arguments, and serve only as a restricted form of unchecked type-cast.

Properties. We expect that `fromBits` and `toBits` are inverses of each other in the sense that the equation `toBits (fromBits n) = n` holds for all bit-vectors `n`, independent of the type of the intermediate value that we create. This is useful because it enables programmers to propagate 'junk' values to other parts of the system without changing them. Saying that `toBits` and `fromBits` are inverses suggests that the equation `fromBits (toBits n) = n` should also hold. But what do we mean by equality in this case? We use operational equivalence and we consider two expressions to be the same if we can replace the one with the other in any piece of program. Because expressions of bitdata types are represented with bit patterns, then two expressions are the same if they are represented with the same bit pattern: $x == y \iff \mathit{toBits}\ x == \mathit{toBits}\ y$. Using this definition for equality we can see that the second equation follows from the first.

Relation to Views Views [93] provide the ability to pattern match on an abstract type as if it were an algebraic datatype. This is accomplished by defining a 'view' type, which specifies a set of functions for converting values of the abstract type into the view type. Values of the view type are used in pattern matching, but the programmer cannot construct them outside of a view type declaration.

One can regard bitdata declarations as defining views on a specific class of types, namely `Bit n` types. Limiting the scope in this way allows us to provide considerably more powerful functionality. In Wadler's work, view types are phantom types that can only be used in pattern matching. In contrast, a bitdata declaration creates a new type that may appear in type signatures and whose values may appear on both the left and right hand sides of an equation. In addition, our compiler automatically generates the marshalling functions. In most cases, the application of these functions imposes no performance penalty because they are identity functions on the underlying representations, whereas view transformation functions may perform arbitrary computation.

7.4.2 Instances for 'BitRep' and 'BitData'

In this section we discuss which types have instances for the classes `BitRep` and `BitData`. This is important because it determines what we may assume about the representations of different types. In particular, we use the class `BitRep` to formalize what we mean by bitdata: all types that have concrete bit

representations belong to `BitRep`. The class `BitData` identifies those bitdata types that are essentially 'views' on bit vectors.

Built-in types, such as the `Bit n` types, have predefined instances:

```
instance Width n ⇒ BitRep (Bit n) n where ...
instance Width n ⇒ BitData (Bit n) n where ...
```

In later chapters we shall introduce other built-in types that have bit representations (and so they belong to `BitRep`), but they also satisfy special constraints on their representation and thus they lack instances for `BitData`.

Types that are defined with **bitdata** declarations automatically get instances of `BitRep`, which are derived from their **as** clauses. However, to get an instance for `BitData` programmers have to provide an explicit `deriving` clause to the declaration. Doing so provides a `fromBits` function, but may add 'junk' values to the type (i.e., values that cannot be defined using the constructors of the type). For example, consider the following declaration:

```
bitdata T = A as B01 | B as B10
```

If we were to derive `fromBits` for `T`, then we could write `fromBits B00` to obtain a value of type `T`, which cannot be defined either with `A` or with `B`. We can use the function `isJunk` to test for such values.

We mentioned before that the fields of bitdata declarations may only contain types that have concrete bit representations. We are now in position to make this more formal by using the `BitRep` and `BitData` classes. We require that all fields of all constructors in a bitdata declaration should be members of, at least, the `BitRep` class. This ensures that we can compute a bit-pattern for each field. In addition, if a programmer wishes to derive an instance of the `BitData` class for a given type, then all the fields in the type should be in the `BitData` class.

As we have already stated, we would like to ensure that types that do not belong to the `BitData` class contain only values that can be defined using their constructors. To ensure that this invariant holds, in some situations we require that overlapping fields in a bitdata declaration are members of the `BitData` class, even if the type itself is not. If we omit this requirement, then it is possible to convert arbitrary bit patterns into values of types that do not belong to the `BitData` class, thus violating our invariant. To see how this could happen, consider some type `T` (of width `n`), that is not a member of the `BitData` class. Then we can write a function to convert arbitrary bit patterns to `T` values by using an additional **bitdata** declaration like the following:

```
bitdata Views = Abs { abs :: T } | Raw { raw :: Bit n }

fromBitsT :: Bit n → T
fromBitsT bits = case Raw { raw = bits } of
                   Abs { abs = t } → t
```

This function works because the constructors of the type `Views` provide different views on `T` values: one abstract and one as a bit vector. Notice that the representations for the two constructors 'overlap' because there are no 'tag' bits to distinguish the two. Because we can 'confuse' the values of one type with values of another, we have lost the abstraction that may have been encapsulated by the types of the fields. This is why we require that the fields of 'overlapping' constructors, should be not just members of the `BitRep` class, but also they should be in the `BitData` class. With this extra requirement, the definition of the type `Views` from the previous example would be rejected on the grounds that the field `abs` of constructor `Abs` is of type `T`, which is not in the `BitData` class, but the constructors `Abs` and `Raw` overlap. In Chapter 8, we describe an algorithm that analyzes **bitdata** declarations to detect when different constructors overlap.

7.4.3 The Type of 'fromBits'

In a robust system, programmers must allow for the possibility that data values that were obtained from the 'real-world' may not be valid. An important design decision related to this issue shows up in the type of `fromBits`. It does not include the possibility of failure because `fromBits` will always produce a value of the target type, even if the input sequence does not correspond to anything that may be created using the constructors of that type. We call such values 'junk', and they will not match any constructor pattern in a function definition. Programmers may, however, use variable or wild-card patterns to match these values. Consider, for example, defining a function that will present human readable versions of the values in the `RemoteOp` type:

```
showOp Read       = "Read"
showOp Write      = "Write"
showOp SendPacket = "SendPacket"
showOp _          = "Unknown"
```

Now we can experiment with different expressions:

```
> showOp (fromBits B01)
"Read"
> showOp (fromBits B00)
"Unknown"
> show (toBits (fromBits B00 :: RemoteOp))
"B00"
```

The first example recognizes the bit-pattern for `Read`. The second example does not match any of the constructors as none of them are represented with `B00`. The last example illustrates that we can convert a bit sequence into a value of type `RemoteOp` and then back into the original bit sequence without loss of information.

An alternative design decision would be to adopt a checked semantics for `fromBits`, where the function result belongs to the `Maybe` type: the result `Nothing` could signal that the bit-pattern does not belong to the type, while we could indicate success by using `Just`. We chose to use the unchecked semantics because it is simple and has practically no overhead (in hobbit, it is implemented as an identity function). Furthermore, using the checked semantics may be misleading because —as we discussed earlier—junk values could arise from pattern matching in the presence of confusion and *not* because we have used the `fromBits` function directly. Programmers can identify 'junk' values using the `isJunk` function. This makes it easy to define a version of `fromBits` that supports the checked semantics:

```
checkedFromBits :: (BitData t n) ⇒ Bit n → Maybe t
checkedFromBits x
  | isJunk v  = Nothing
  | otherwise = Just v
  where
  v = fromBits x
```

If the language that is being extended with bitdata supports exceptions, we could just as easily implement a function that raises an exception if the bit-pattern does not correspond to a value that can be defined using the constructors. This diversity of possible implementations provides further motivation for taking the simplest alternative as primitive, and allowing other operations to be built on top of it.

7.5 Summary

In this chapter we presented two language extensions that make it easy to manipulate data with programmer-defined bit representations.

The first extension adds support for working with bit vectors, which is well integrated with a Haskell-like language. We use overloading to provide a uniform interface to working with vectors of different sizes, and we also support pattern matching on bit vectors for readable definitions.

The second extension supports working with structured bitdata. With this extension, programmers can manipulate bitdata like they manipulate algebraic datatypes. We use pattern matching to examine the values of predefined 'tag' bits, and we automatically generate functions to access and update bit fields.

The following is a summary of the (overloaded) constants from our design:

```
-- Bit vectors
Bit :: Nat → *

-- Bit literals
-- data Bit 2 = B00 | B01 | B10 | B11   (etc.)

-- Operations on bit vectors
-- Instances for valid bit-vector sizes
class Width a where

  -- comparisons
  bitEq       :: Bit a → Bit a → Bool
  bitCmpU     :: Bit a → Bit a → Ordering
  bitCmpS     :: Bit a → Bit a → Ordering

  -- logical operations
  bitAnd      :: Bit a → Bit a → Bit a
  bitOr       :: Bit a → Bit a → Bit a
  bitXor      :: Bit a → Bit a → Bit a
  bitNot      :: Bit a → Bit a

  -- shifts
  bitShiftL   :: (Width b) ⇒ Bit a → Bit b → Bit a
  bitShiftR   :: (Width b) ⇒ Bit a → Bit b → Bit a
```

```
  -- arithmetic
  bitAdd      :: Bit a → Bit a → Bit a
  bitSub      :: Bit a → Bit a → Bit a
  bitNeg      :: Bit a → Bit a
  bitMul      :: Bit a → Bit a → Bit a
  bitDivU     :: Bit a → Bit a → (Bit a, Bit a)
  bitDivS     :: Bit a → Bit a → (Bit a, Bit a)

  -- conversion
  bitFromInt  :: Integer → Bit a


-- Join and split bit vectors
class (Width a, Width b, Width c, a + b = c)
  ⇒ a # b = c |  a b ⇝ c, b c ⇝ a, c a ⇝ b where
  (#)         :: Bit a → Bit b → Bit c
  bitSplit    :: Bit c → (Bit a, Bit b)
  bitSignExt  :: Bit b → Bit c
-- Instances for a, b, and c that satisfy super-class constraints


-- Data with bit representation
class BitRep t n | t ⇝ n where
  toBits      :: t → Bit n
-- Instances for bit vectors
-- Instances for bitdata declarations
-- Required for bitdata fields


-- A 'view' on a bit vector
class BitRep t n ⇒ BitData t n | t ⇝ n where
  fromBits    :: Bit n → t
  isJunk      :: t → Bool
-- Instances for bit vectors
-- Instances for bitdata with explicit deriving annotation
-- Required for overlapping bitdata fields
```

Chapter 8

Static Analysis of Bitdata

In this chapter, we show that, when we work with certain kinds of bitdata, it is useful to perform static analysis that goes beyond type checking. In Section 8.1, we describe how bitdata types may fail to satisfy two important properties of algebraic datatypes. In the rest of the chapter, we present two algorithms that help us detect and work with such bitdata. In Section 8.2, we describe an algorithm that analyzes **bitdata** declarations to detect if the defined type satisfies the properties of ordinary algebraic datatypes. We do not reject declarations that fail to satisfy the algebraic laws, but we provide diagnostics to help programmers work with such bitdata. In Section 8.3, we describe an algorithm that examines function definitions for potential errors, such as missing or unreachable equations.

8.1 Junk and Confusion!

Standard algebraic datatypes enjoy two important properties that are sometimes referred to as 'no junk' and 'no confusion' [33], both of which are useful when reasoning about the behavior of functional programs. The former asserts that every value in the datatype can be written using only the constructor functions of the type, while the latter asserts that distinct constructors construct distinct values. In the language of algebraic semantics, which is where these terms originated, the combination of 'no junk' and 'no confusion' implies that the semantics of a datatype is isomorphic to the initial algebra generated by its constructor functions.

Because programmers specify the representations for the values of bitdata

types, it is possible that such types may contain 'junk' or 'confusion' (or both). For example, 'confusion' arises when the bit patterns for different constructors in a **bitdata** declaration overlap. We can introduce 'junk' by deriving an instance for the `BitData` class (see Chapter 7). Then applying `fromBits` to a bit pattern that does not correspond to a constructor results in a 'junk' value.

Usually, we try to avoid having junk and confusion in bitdata types. However, in some situations, a designer might still opt for a representation that sacrifices one or both of these properties to conform to an external specification, or because it simplifies the tasks of encoding and decoding. Recall, for example, the type `Time` from Chapter 1:

```
bitdata Time = Now                              as B0 # 1 # (0::Bit 10)
             | Period { e::Bit 5, m::Bit 10 } as B0 # e # m
             | Never                            as 0
```

This type contains confusion (because the `Now` and `Never` cases overlap with the `Period` case), and would also contain junk if we were to derive an instance of `BitData` (because the most significant bit can never be set).

It is also worth commenting that the potential for 'confusion' in a bitdata type can sometimes be used to our advantage. The following example shows a `DWord` type that reflects different interpretations of a 32 bit word on an IA32 platform as either a full word, a virtual address with page directory and page table indexes, or a collection of four bytes:

```
bitdata DWord
  = Int32    { val :: Bit 32 }
  | VirtAddr { dir :: Bit 10, tab :: Bit 10, offset :: Bit 12 }
  | Bytes    { b3 :: Byte, b2 :: Byte, b1 :: Byte, b0 :: Byte }
```

Each of these constructors can encode an arbitrary 32 bit value, but we can use pattern matching to select the appropriate view for a given setting.

Bitdata types that contain junk or confusion (or both!) do not satisfy the usual properties that one might expect of algebraic datatypes, and so programmers need to exercise caution when working with such types. For example, pattern matching on bitdata that has confusion is similar to working with overlapping patterns: when programmers define functions, they need to check for more specific cases first. When working with bitdata that contains junk, programmers may wish to provide 'default' cases to function definitions, even if a function provides definitions for all constructors in the datatype.

Such cases are used to handle junk values, and to increase the robustness of the program—the exact action taken by the programmer is, in general, application specific but, for example, a programmer may raise an exception, or, alternatively, they may choose to propagate junk values unchanged in the result of the function.

To help programmers work with bitdata that contains junk and confusion, we should first warn them when a bitdata type does not satisfy the usual algebraic properties. In Section 8.2, we develop an algorithm that can examine **bitdata** declarations and report accurately when values created with different constructors overlap (i.e., when the type contains confusion), or when there are bit-patterns that cannot be constructed with any constructor (i.e., when the type contains junk). Such an analysis is useful for two reasons: (i) it shows programmers exactly how the usual algebraic laws are violated; and (ii) it helps programmers to detect erroneous bitdata declarations because unexpected junk or confusion may reveal mistakes in the declarations.

In addition to analyzing bitdata declarations, it is also useful to analyze the definitions of functions in the program in an attempt to detect mistakes that result from junk and confusion. The symptoms that we look for are unreachable or incomplete functions definitions. Unreachable definitions often result from the presence of confusion, while incomplete definitions may be caused by junk (or simply by forgetting a relevant case!). Incomplete definitions are dangerous because they may cause the program to crash at run-time. Unreachable equations cannot crash the program, but do not serve a useful purpose. Because of that, their presence in the program often reveals mistakes. For example, we might (incorrectly) define a function that converts `Time` values to microseconds like this:

```
toMicro :: Time → Maybe Integer
toMicro (Period { m = m, e = e }) = Just (2^e * m)
toMicro Now                       = Just 0
toMicro Never                     = Nothing
```

There would be nothing wrong with this definition if `Time` was an ordinary algebraic datatype. However, from the definition of `Time`, we can see that the representations of `Now` and `Never` values overlap with the representations of `Period` values, and so the second and third equations in the definition are unreachable. This is the case because, as in Haskell, evaluation prefers earlier equations in a function definition. Our algorithm would alert the

programmer of this anomaly, and they can correct the definition by placing the most general case, `Period`, last in the list of equations defining `toMicro`:

```
toMicro :: Time → Maybe Integer
toMicro Now                        = Just 0
toMicro Never                      = Nothing
toMicro (Period { m = m, e = e }) = Just (2^e * m)
```

8.2 Checking Bitdata Declarations

In this section, we describe an algorithm that detects junk and confusion in types defined with **bitdata** declarations. The overall idea is simple:

1. Compute the set of bit vectors that can be constructed with each constructor of the declaration (we shall write $[\![C]\!]$ for the set of vectors that can be constructed with the constructor C).

2. A declaration, d, contains confusion if the sets of bit vectors for any two constructors overlap.

   $$\begin{aligned} \mathit{overlap}\ C\ D &= [\![C]\!] \cap [\![D]\!] \\ \mathit{confusion}\ d &= \bigcup\{\mathit{overlap}\ C\ D \mid C, D \in \mathit{ctrs}\ d \wedge C \neq D\} \neq \emptyset \end{aligned}$$

   The set $\mathit{overlap}\ C\ D$ contains those bit vectors that can be constructed with either constructor C or D. Our algorithm is constructive—when we detect that a **bitdata** declaration contains confusion, we can also display which bit vectors can be constructed with the different constructors.

3. A declaration, d, contains junk if there are bit vectors that cannot be constructed with any constructor.

   $$\begin{aligned} \mathit{cover}\ d &= \bigcup\{[\![C]\!] \mid C \in \mathit{ctrs}\ d\} \\ \mathit{junk}\ d &= \mathit{cover}\ d \neq \mathit{Univ} \end{aligned}$$

The set $\mathit{cover}\ d$ contains the bit vectors that can be constructed with the constructors of d. Any remaining bit vectors correspond to junk values. Again, our algorithm is constructive and can display junk values.

As an example of the output of the algorithm, consider what happens when we analyze the L4 type `Time` that we have already seen a few times:

```
bitdata Time = Now                              as B0 # 1 # (0::Bit 10)
             | Period { e::Bit 5, m::Bit 10 } as B0 # e # m
             | Never                            as 0
               deriving BitData
```

The diagnostics produced by our prototype are as follows:

```
Warning: The type Time contains junk:
        1---------------

Warning: Constructors Period and Never of type Time overlap
        0000000000000000

Warning: Constructors Now and Period of type Time overlap
        0000010000000000
```

The first warning states that any value that has 1 as its most significant bit is a junk value (notice that all **as** clauses start with a 0). The other two warnings indicate the presence of confusion: the given bit patterns can be constructed with either of the listed constructors (e.g., `0000000000000000` can be interpreted either as `Period` or as `Never`).

8.2.1 'as' clauses

To compute the set $[\![C]\!]$ for a constructor C we use its layout specification (i.e., the **as** and **if** clauses). In this section, we consider **as** clauses and in the following section, we shall turn our attention to **if** clauses.

Recall that the **as** clause of a constructor contains a list of fields separated by the symbol `#`. A field can be either a literal, or a named field, or a wildcard pattern. To compute the set of bit vectors for a constructor we compute the set of bit vectors for each field and then we 'concatenate' them. To 'concatenate' two sets of bit vectors we concatenate all their elements:

$$A\#B \equiv \{xs\#ys \mid xs \in A \wedge ys \in B\}$$

The sets corresponding to literal and wildcard fields are straight-forward: for literal fields we use a singleton set containing the literal, and for wildcard fields we use the set of all bit vectors that are of the same length as the field.

Choosing the set of possible bit vectors for a named field presents us with some choices. Consider, for example, the following declarations:

```
bitdata A = A as B1
bitdata B = B { x :: A } as x
```

In this form, neither `A` nor `B` contain junk values because they do not belong to the `BitData` class. Now suppose that we derived a `BitData` instance for `A`. In that case, `A` contains the junk value `fromBits B0` but does `B` contain any junk? The answer to this question boils down to deciding if the value `B { x = fromBits B0 }` should be considered to be a junk value of `B`. We are interested in junk values because they do not match any of the constructors for the type, and so to avoid partial definitions we need to provide extra 'catch all' cases. Therefore, the decision if `B { x = fromBits B0 }` is a junk value should be related to the semantics of pattern matching with a bitdata constructor. Recall from Chapter 7 that, when we pattern match with a constructor we consider only the 'tag' bits from the **as** clause, and *not* the possible values for the fields. In this example, the constructor `B` has no tag bits, and so matching against it will never fail. Therefore, we do not need an extra 'catch all' equation in definitions involving the type `B` and thus, it does not contain junk values. If, however, we had adopted the alternative design described in Chapter 7, then the type `B` would inherit the junk from `A`.

The situation is similar when we analyze for confusion. If a type contains confusion, then different constructors may match the same value. Therefore, detecting confusion depends on the semantics of pattern matching. Because when we pattern match we do not consider the possible values for fields, but rather we just examine the 'tag' bits for constructors, it follows that a type contains confusion if any two constructors have the same 'tag' bits. For an example, consider the following declarations:

```
bitdata A = A as B0 deriving BitData
bitdata B = B as B1 deriving BitData
bitdata C = C1 { x :: A } | C2 { y :: B }
```

Here, the type `C` contains confusion because there are no tag bits to distinguish the constructor `C1` from the constructor `C2`. Note also that, as we have discussed in Chapter 7, because of this overlap the types `A` and `B` have to be in the `BitData` class.

The conclusion from this discussion is that because when we pattern-match we do not consider the possible values for the fields of a bitdata constructor, it follows that in our analysis here we should treat named fields in **as** clauses in the same way as we treat wildcards. In particular, when

we analyze the declarations from `A`, `B`, and `C`, we should get the following warnings:

```
Warning: The type A contains junk:
        1
Warning: The type B contains junk:
        0
Warning: Constructors C1 and C2 of type C overlap:
        -
```

8.2.2 'if' clauses

In its most general form, the layout specification for a constructor also supports an **if** clause that may make decisions based on the values in the fields of the constructor. When we pattern match with a constructor whose specification has an **if** clause, the pattern will only succeed if the Boolean expression of the **if** clause evaluates to `True`. Therefore, when we compute the set of bit vectors that correspond to a constructor, we need to constrain it with the Boolean expression from the **if** clause.

This leads to several challenges. First, because **if** clauses may contain arbitrary Boolean expressions, computing the set of bit vectors for a constructor becomes undecidable in general. The second problem is that we only use **if** clauses in pattern matching but not when we construct values (we made this decision to avoid partial constructor functions). As a result, constructors may be used to introduce junk values and so we may not completely ignore the **if** clauses during analysis. Consider, for example, a definition like the following:

```
bitdata T = T { x :: Bit 4 } if x > 10
```

If we omit the **if** clause, then the system will not report junk in this type, even though patterns like `T { x = x }` will fail if $x \leq 10$.

Our solution is to restrict the set of possible bit vectors with the **if** clause when the Boolean decision is fairly simple and to revert to a conservative strategy for complex **if** clauses. In practice this works pretty well because it is common to have fairly simple decisions in **if** clauses (e.g., comparisons with constants) and so undecidability is not a problem.

We say that a strategy is *conservative* if it guarantees that the algorithm will not miss any problems with the declarations. However, because the

algorithm is approximating the actual behavior of the program, it may report some false-positives (i.e., reporting junk or confusion when there is none).

The conservative strategy that we use depends on how we plan to use $[\![C]\!]$. When we are computing $[\![C]\!]$ to detect confusion between constructors, the safe strategy is to compute a superset of the real value of $[\![C]\!]$ and so we approximate complex Boolean expressions by `True`. When we are computing $[\![C]\!]$ to detect junk in **bitdata** declaration, the safe strategy is to compute a subset of the actual value of $[\![C]\!]$, and so we approximate complex Boolean expressions by `False`.

8.2.3 Working with Sets of Bit Vectors

In the previous sections we described how to analyze **bitdata** declarations for junk and confusion in terms of sets of bit vectors. In this section we describe an efficient implementation for such sets and their corresponding operations. More concretely the set operations that we need are: union, intersection, complement, and concatenation. It is also convenient to have a method for displaying the elements in a set in a concise fashion.

The idea is to represent a set of bit vectors by its characteristic function (i.e., a function that given a bit vector will compute if the vector belongs to the set or not). Any such function is going to be a function of the individual bits in the bit vector. A well-understood and widely used method for representing such Boolean functions is to use ordered binary decision diagrams (OBDDs) [16], and so in the reminder of this section we provide a brief introduction to these ideas.

A BDD is a binary decision tree with a variable at each node, and two branches specifying the value of the expression for when the variable is true or false, respectively. In fact, it is common to use an acyclic graph instead of a tree to represent BDDs. This is useful because it can make the representation more compact, and thus enhance the efficiency of the operations. For the purposes of this section, however, we shall ignore this detail and use a tree. The operations that we define behave identically on trees and graphs, but if we use the graph representation, then we have to take care to preserve the common structure. Furthermore, it is not clear that the sharing is really necessary for our purposes: we used the tree representation in the implementation of our static analysis algorithm, and it seems to work quite well. This can be explained by the relatively low number of variables (e.g. at most 32) in our Boolean expressions. This is in contrast to other

applications of BDDs, such as hardware circuit verification, where Boolean expressions may contain hundreds of variables. We use the following data structure to represent BDDs:

```
type Var = Int
data BDD = F | T | ITE Var BDD BDD
```

The nodes of the tree are like a special form of an if-then-else expression (hence the name `ITE` for the constructor) where the condition is always a variable. For example, the tree `ITE x T F` encodes the Boolean expression x, while the tree `ITE x F T` encodes the Boolean expression *not* x because the expression evaluates to `F` when the variable `x` is true, and to `T` when `x` is false.

It is convenient to define the normal if-then-else operator, *ite*, that allows expressions in the decision. We can then use this operator to define a number of other operations on BDDs:

```
bddAnd    :: BDD -> BDD -> BDD
bddAnd p q = ite p q F

bddOr     :: BDD -> BDD -> BDD
bddOr p q  = ite p T q

bddNot    :: BDD -> BDD
bddNot p   = ite p F T
```

Note that these correspond to set intersection, union, and complement respectively.

Our first attempt to define the `ite` operator might look something like this:

```
ite1 T t e           = t
ite1 F t e           = e
ite1 (ITE x p q) t e = ITE x (ite1 p t e) (ite1 q t e)
```

While this definition is not wrong, it may produce decision trees that are not very good. Consider, for example, the expression `ite1 x x x`. Clearly this expression behaves in exactly the same way as `x`. However, here is what happens if we use `ite1` (remember that the expression `x` is represented as `ITE x T F`):

```
ite1 (ITE x T F) (ITE x T F) (ITE x T F)
= (case 3 of ite1)
ITE x (ite1 T (ITE x T F) (ITE x T F)) (ite1 F (ITE x T F) (ITE x T F))
= (case 1 and 2 of ite1)
ITE x (ITE x T F) (ITE x T F)
```

The problem is that we are not propagating information that we learn at ITE nodes to the branches. The basic observation is that, while making a decision, we never need to examine the value of a variable more than once: once we test a variable `x`, we can assume that `x` is true in the left branch, and that it is false in the right branch. In terms of our representation, this translates into the property that variables should appear at most once on any path in a BDD. One way to solve this problem is to write a function that eliminates redundant tests:

```
simp :: [(Var,Bool)] → BDD → BDD
simp xs F = F
simp xs T = T
simp xs (ITE x t e)
  = case lookup x xs of
      Just True  → simp xs t
      Just False → simp xs e
      Nothing    → ITE x (simp ((x,True) :xs) t)
                         (simp ((x,False):xs) e)

ite2 p t e = simp [] (ite1 p t e)
```

The function `simp` has an extra argument that stores information about variables: after we test a variable, we record the value in the respective branches. If we encounter another test on the same variable, then we eliminate it by using the known value.

This solution is correct, but we can do even better if we assume that all decision trees have the property that they do not contain redundant tests. The idea is to pick some fixed order in which we examine variables and then use this order in all decision trees. Such BDDs are called *ordered* BDDs because of the ordering on the variables. More concretely, an OBDD is a BDD in which the variables on any path in the tree are strictly decreasing from the root to the leaves. The fact that variables are *strictly* decreasing ensures that they are not repeated. The fact that they always decrease gives us an efficient way to know what variables may appear in the sub-trees. For

example, if we know that $x \geq y$, and that the value of x is b, then we can use the following function to optimize a decision tree:

```
with :: Var → Bool → BDD → BDD
with x b (ITE y p q)
  | x == y = if b then p else q
with _ _ t = t
```

Notice that we do not need to examine the sub-branches of the tree because the ordering property of OBDDs and the assumption that $x \geq y$ combine to guarantee that if x appears in the tree, then it will appear exactly once and only at the root.

We are now ready to present the final version of the function `ite`:

```
ite         :: BDD → BDD → BDD → BDD
ite T t _    = t
ite F _ e    = e
ite p t e
  = let x    = maximum [ x | ITE x _ _ ← [p,t,e] ]
        t'   = ite (with x True  p) (with x True  t) (with x True  e)
        e'   = ite (with x False p) (with x False t) (with x False e)
    in if t' == e' then t' else ITE x t' e'
```

The base cases are as before: if the condition is a constant, then we just pick the appropriate alternative. Otherwise, we need to make a decision, but there is an interesting twist here: OBDDs require us to check variables in a fixed (decreasing) order. This means that the variable that we should examine next is not necessarily the one that is in `p`, but rather the largest variable that occurs in all three of `p,t,e`. This variable, `x`, is quite simple to compute because the largest variable (if any) of an expression is always at the root. Having picked the variable to examine, we can compute a new 'then' and 'else' branch by modifying the current problem to take into account the value of `x`. One final detail in the definition of `ite` is that, if both the 'then' and the 'else' branch turn out to be the same, then there is no need to examine the variable after all.

To show how this works in practice, here is a worked out example of the function `ite`. Notice how the requirement that we test variables in decreasing order changes the original expression into an equivalent, but different expression.

```
-- if 1 then 2 else 3

  ite (ITE 1 T F) (ITE 2 T F) (ITE 3 T F)
    x  = 3
    t' = ite (ITE 1 T F) (ITE 2 T F) T
          x  = 2
          t' = ite (ITE 1 T F) T T
             = T
          e' = ite (ITE 1 T F) F T
                x  = 1
                t' = ite T F T
                   = F
                e' = ite F F T
                   = T
             = ITE 1 F T
       = ITE 2 T (ITE 1 F T)

    e' = ite (ITE 1 T F) (ITE 2 T F) F
          x  = 2
          t' = ite (ITE 1 T F) T F
                x = 1
                t' = ite T T F
                   = T
                e' = ite F T F
                   = F
             = ITE 1 T F

          e' = ite (ITE 1 T F) F F
             = F
       = ITE 2 (ITE 1 T F) F

= ITE 3 (ITE 2 T (ITE 1 F T)) (ITE 2 (ITE 1 T F) F)

-- if 3 then (2 or not 1) else (2 and 1)
```

The following diagram shows the initial BDD and the resulting OBDD in graphical format:

One detail that we have not addressed yet is: what constitutes the whole set? We have not imposed any restrictions on the variables that may appear in a BDD, and so the whole set is bit vectors of any length. For our purposes,

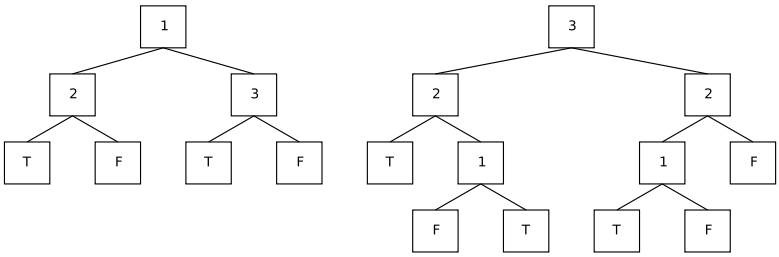


Figure 8.1: Transforming a BDD to satisfy the OBDD ordering.

this is not very convenient because we work with bit vectors of a fixed size. For example, it is nice to display a 16 bit value with 16 bits, but if we work with just plain BDDs, then nothing in the representation tells us the width of the value that the BDD represents. For this reason, in our implementation we used a pair containing the BDD for the pattern and the width of the encoded expression. An additional property that we expect from the BDD is that it should only mention variables that are smaller than the width: thus the BDD for bit vectors of width 8 may mention variables up to 7, while bit patterns of width 0 should not contain any variables at all.

```
type Width = Int
data Pat   = Pat Width BDD
```

Knowing the widths of the bit vectors in a set is also useful when we want to compute the 'concatenation' of two sets. Given two OBDDs `p` and `q` we can compute their concatenation by shifting `p` to the left by incrementing all of its variables by the width of `q` and then computing the intersection of the resulting BDDs. In this way, the BDD `p` asserts what we know about the most significant bits of the vectors in the resulting set, and the BDD `q` contains the information about the least significant bits.

Finally, we need a good way to display the elements of a set of bit vectors. In many cases enumerating all the elements is not a good option because there are simply too many of them. Fortunately we can easily define a function that represents a decision tree as a set of mutually exclusive patterns:

```
showPat             :: Pat → [String]
showPat (Pat w T)    = [replicate w '_']
showPat (Pat _ F)    = []
showPat (Pat w f@(ITE v p q))
```

```
  | w' > v          = [ '_' : p | p ← showPat (Pat w' f) ]
  | otherwise       = [ '0' : p | p ← showPat (Pat w' q) ]
                   ++ [ '1' : p | p ← showPat (Pat w' p) ]
    where w'         = w - 1
```

The first equation displays the entire set: we can do this with a single wildcard pattern of the appropriate length. The second equation is for the empty set, which is represented with no patterns. The interesting case is the third equation, which deals with decisions. If the decision is based on a variable that is not the most significant bit in the pattern, then, by the property of OBDDs, we know that the most significant bit will not affect the value of the expression, and so we display a wildcard (this is done in the first guard). Otherwise the most significant bit matters, and so we produce two sets of patterns, one starting with a 0 for the false branch, and one starting with a 1 for the true branch. For example, here is the output of the function when applied to the decision tree for the complement of the pattern `B0001` (i.e., all 4 bit integers, except for 1):

```
0000
001_
01__
1___
```

8.3 Checking Function Declarations

In this section, we describe how to analyze the function declarations in the language, so that we can detect partial or redundant definitions. Such an analysis is useful in ordinary functional languages, but it is even more important here because of the presence of 'junk' and 'confusion' in some bitdata types.

8.3.1 The Language

To illustrate the algorithm in a more concrete setting, we use a simple but expressive language, based on the calculus of definitions from Chapter 6. This language is obtained by using the rules of the calculus to eliminate some of the constructs systematically. In particular, we expect that the arguments to functions were named and factored out of matches (using a procedure like

the one described in Chapter 6). This eliminates matches of the form `p → m`. Then we may also eliminate patterns entirely by using the definitions of the various patterns, and the following two rules:

```
      x ← e  =  let x = e
(p | q) ← e  =  p ← e ; q
```

After these simplifications, we end up with the following language:

```
mat  = mat [] mat    -- alternatives
     | qual ; mat    -- check
     | return expr   -- success

qual = if expr       -- guard
     | let d         -- local declaration
     | qual ; qual   -- sequencing
```

The last technical detail about the language is the assumption that declarations do not shadow existing values, which is easily ensured by renaming local variables that shadow existing names. To see why we need this, consider the following expression:

```
λx → { if p x; let x = e1; if q x; return e2 }
```

The definition of this function may fail in two different ways: either the argument does not satisfy the predicate `p`, or the local variable does not satisfy the predicate `q`. Unfortunately, it is difficult to formalize this statement without renaming the variables because both the function argument and the local variable have the same name.

Renaming the variables solves the problem but, in practical implementations, we have to do something to indicate to the programmers what the new names refers to. For example, in a graphical development environment we could highlight the variables directly on the screen. In a text based implementation, we should make sure that the new name is related to the original name. One way to achieve this is to annotate names with the location in the program where they were defined.

8.3.2 The Logic

The conditions under which a match or qualifier may fail are expressed in a simple propositional language:

```
Prop = F | T | Prop ∧ Prop | Prop ∨ Prop | Not prop | Atom expr
```

Besides the basic propositions `T` and `F`, we also allow arbitrary Boolean expressions in the formulas. This is necessary to express the constraints arising from guards in the language. Consider, for example, a definition like the following:

```
f x y
  | x < y   = e1
  | x == y  = e2
  | x > y   = e3
```

The analysis of `f` will result in a warning stating that `f` may fail if:

```
not (x < y) ∧ not (x == y) ∧ not (x > y)
```

In this expression, `(x < y)`, `(x == y)`, and `(x > y)` are atoms. Of course, we probably expect that one of the three conditions should hold, but the system cannot be sure of that without examining the definitions of the functions `<`, `==`, and `>`. In general, our algorithm does not examine definitions because: (i) the system becomes too complex; (ii) the definitions may be part of a library whose code is not available; and (iii) with overloaded functions, like `f`, we may not even know the definitions of some of the functions until `f` is completely instantiated. In many situations, it is possible to rewrite the guards in the system in such a way that it becomes obvious that the guards are exhaustive. For example, we could replace the last branch in the definition of `f` with a trivially true guard, such as `otherwise`.

8.3.3 The Algorithm

The algorithm performs abstract interpretation on matches and qualifiers to determine under what circumstances they might fail. As an input, we get a predicate that asserts a number of facts that hold so far, and the match (or qualifier) to examine:

```
fails_m :: Prop → Match → Prop
fails_q :: Prop → Qual → Prop
```

For example, `fails_m p m` is a predicate that describes the conditions that will lead to `m` failing, provided that `p` holds before we evaluate `m`. When we start analyzing a new match we do not have any special pre-conditions, so we simply use `T`. Here is the definition of `fails_m`:

```
fails_m p (m1 [] m2)   = fails_m (fails_m p m1) m2
fails_m p (q ; m)      = fails_q p q ∨ fails_m (succeeds q ∧ p) m
fails_m p (return e)   = F

succeeds :: Qual → Prop
succeeds (if e)      = Atom e
succeeds (let d)     = T
succeeds (q1 ; q2)   = succeeds q1 ∧ succeeds q2
```

The first equation deals with alternatives: the match fails if the second alternative fails, and before examining the second alternative we may assume that the first one must have failed. The second equation is where decisions are made in matches. Such qualified matches can fail in two different ways: either the qualifier fails, or the qualifier succeeds, but the remaining part of the match fails. The final equation deals with matches that choose a particular alternative. Such matches cannot fail.

The function `succeeds` computes new facts that we may assume if we know that the qualifier succeeded. Note that, for the case of **let** qualifiers, we do not add any assumptions. We could get a more accurate analysis if we examined the declarations: for example, if a variable is defined with a constructor (e.g., `x = Nil`), then we could also use this fact to simplify the inferred conditions—but we have found that our simpler version works well in practice.

The code that analyzes qualifiers for failure is similar to the code for matches:

```
fails_q p (if e)      = Not (Atom e) ∧ p
fails_q p (let d)     = F
fails_q p (q1 ; q2)   = fails_q p q1 ∨ fails_q (succeeds q1 ∧ p) q2
```

8.3.4 Unreachable Definitions

As we discussed previously, we would like to know not only when a match fails, but also if a match contains alternatives that are unreachable. The analysis is similar to analyzing matches for failure: a match contains an unreachable alternative, if the pre-condition in the third equation (**return** `e`) is unconditionally false. It is fairly easy to modify the above equations to emit and propagate these extra conditions, for example using an output monad to automatically collect the warnings:

```
fails_m p (m1 [] m2)   = do q ← fails_m p m1
                            fails_m q m2
fails_m p (q ; m)      = do r ← fails_m (succeeds q ∧ p) m
                            return (fails_q p q ∨ r)
fails_m p (return e)   = do when (p == F) (warn "Unreachable")
                            return F
```

Here is an example of how this works on a simple definition:

```
f x   = e1
f Nil = e2

-- translated version
f = λx → { return e1 [] if isNil x; return e2 }

-- analyzing the match:
fails_m T (return e1 [] if isNil x; return e2)
= using equation (1)
  do q ← fails_m T (return e1)
     fails_m q (if isNil x; return e2)
= using equation (3), then simplifying
  fails_m F (if isNil x; return e2)
= using equation (2)
  do r ← fails_m F (return e2)
     return (fails_q F (if isNil x) ∨ r)
= using equation (3), then simplifying
  do warn "Unreachable"
     return (fails_q F (if isNil x))
= using definition of  fails_q
  do warn "Unreachable"
     return F
```

Because the first alternative cannot fail, the precondition when we reach the second alternative is `F`, and hence this code is unreachable.

8.3.5 Simplifying Conditions

So far, we have completely ignored patterns by hiding all decisions inside qualifiers. To make the algorithm useful, we need to teach it about the properties of various data declarations. Omitting this step can lead to many warnings. For example, consider the following definition:

```
null (_ : _) = False
null []      = True

-- translated version
null = λx → { if isCons x; return False
            | if isNil x; return True
            }
```

When we perform our analysis we would get a warning that `null` may fail if the following condition holds:

```
not (isCons x) ∧ not (isNil x)
```

Clearly this cannot happen because `Cons` and `Nil` are the only possible constructors for the type; but the system does not know that.

In our language, we can pattern match on two forms of types: algebraic data types, and bitdata. For each of these, we introduce a new formula to the logic language:

```
Prop = ... | ADT var {ctrs} | BDT var Pat
```

The formula `ADT x { C1, .. Cn }` asserts that the variable `x` is an algebraic value that was constructed with one of the constructors `C1` ... `Cn`. For example, instead of writing `isCons x`, now we can write `ADT x { Cons }`. We may use the following rules to simplify formulas:

```
ADT x { } = F
ADT x { C1 .. Cn } = T  -- if C1 .. Cn are all constructors
ADT x cs ∨ ADT x ds = ADT x (cs ∪ ds)
ADT x cs ∧ ADT x ds = ADT x (cs ∩ ds)
Not (ADT x cs) = ADT x (compl cs)
```

The formula `BDT x p` asserts that the variable `x` is one of the bit-patterns described by the binary pattern `p`, as discussed in Section 8.2. These formulas satisfy a similar set of rules, except that the set operations are implemented using the corresponding BDD operations.

In principle, we can introduce `ADT` and `BDT` formulas by examining `Atom` predicates and eliminating atoms of the form `isC`. In our implementation, however, we did not completely eliminate patterns: we just simplified them until they were simple algebraic patterns, or binary patterns. We then modified the above algorithms to deal with patterns directly (which is straightforward), generating either `ADT` or `BDT` formulas as needed.

8.4 Summary

We have shown that, because programmers specify the representations for bitdata types, two important algebraic laws, known as 'no junk' and 'no confusion', may be violated. A type contains 'junk' if there are values that do not match any of the constructor-patterns for the type. Functions that are defined by pattern matching on such types need to provide 'catch-all' equations to avoid partial definitions. A type contains 'confusion', if there are values that match multiple constructor-patterns. Pattern matching on such types is similar to working with overlapping patterns, and the order of the equations in a function definition is important.

To help programmers, we present two algorithms that can be used in the static analysis phase of an implementation. The first algorithm analyzes **bitdata** declarations and reports the presence of junk or confusion. The algorithm is constructive and computes the representations for the values that violate the algebraic laws. The second algorithm analyzes function definitions to detect missing or redundant definitions. Such an analysis is useful for ordinary algebraic datatypes (and indeed many implementations perform a similar analysis), but it is particularly useful in the presence of types that contain junk and confusion. In this area, our contribution is to show that the algorithm can be extended to handle bitdata types that may contain junk and confusion.

Chapter 9

Memory Areas

In this chapter, we describe a new language extension that provides direct support for manipulating memory-based data structures. We would like to work in a high-level functional language, reaping the benefits of strong static typing, polymorphism, and higher-order functions, and, at the same time, be able to manipulate values that are stored in the machine's memory, when we have to. In essence, our goal here has been to provide the same levels of flexibility and strong typing for byte-oriented data structures in memory as our bitdata work provided for bit-oriented data structures in registers[1].

We start with an overview of our design in Section 9.1. Then, in Section 9.2, we introduce our method for describing memory. To manipulate memory we use references, which are discussed in Section 9.3. Then, in Section 9.4, we discuss the memory representations for stored values. In Section 9.5 we show how to introduce memory areas to a program. We conclude the chapter with a discussion of alternative design-choices in Section 9.6.

9.1 Overview

To illustrate these ideas in a practical setting, we will consider the task of writing a driver for text mode video display on a generic PC. This is a particularly easy device to work with because it can be programmed simply

[1]The material in this Chapter is based on the following paper:
Iavor S. Diatchki and Mark P. Jones. Strongly Typed Memory Areas. In *Proceedings of ACM SIGPLAN 2006 Haskell Workshop.*, pages 72–83, Portland, Oregon, September, 2006.

by writing appropriate character data into the video RAM, which is a memory area whose starting physical address, as determined by the PC architecture, is 0xB8000. The video RAM is structured as a 25×80 array (25 rows of 80 column text) in which each element contains an 8 bit character code and an 8 bit attribute setting that specifies the foreground and background colors. We can emulate a simple terminal device with a driver that provides the following two methods:

- An operation, `cls`, to clear the display. This can be implemented by writing a space character, together with some suitable default attribute, into each position in video RAM.

- An operation, `putc`, that writes a single character on the display and advances the cursor to the next position. This method requires some (ideally, encapsulated) local state to hold the current cursor position. It will also require code to scroll the screen by a single line, either when a newline character is output, or when the cursor passes the last position on screen; this can be implemented by copying the data in video RAM for the last 24 lines to overwrite the data for the first 24 lines and then clearing the 25th line.

There is nothing particularly special about these functions, but neither can be coded directly in ML or Haskell because these languages do not include mechanisms for reading or writing to memory addresses.

Instead, if we want to code or use operations like these from a functional program, then we will typically require the use of a foreign function interface [21, 15]. For example, in Haskell, we might choose to implement both methods in C and then import them into Haskell using something like the following declarations:

```
foreign import ccall "vid.h cls"  cls  :: IO ()
foreign import ccall "vid.h putc" putc :: Char → IO ()
```

Although this will provide the Haskell programmer with the desired functionality, it hardly counts as writing the driver in Haskell!

Alternatively, we can use the `Ptr` library, also part of the Haskell foreign function interface, to create a pointer to video RAM:

```
videoRAM :: Ptr Word8
videoRAM  = nullPtr `plusPtr` 0xB8000
```

This will allow us to code the implementations of `cls` and `putc` directly in Haskell, using `peek` and `poke` operations to read and write bytes at addresses relative to the `videoRAM` pointer. These two functions are a part of Haskell's foreign function interface, which allow programmers to read and write values to memory:

```
peek :: Storable a ⇒ Ptr a → IO a
poke :: Storable a ⇒ Ptr a → a → IO ()
```

The class `Storable` identifies which types may be stored in memory.

Unfortunately, because we have to use pointer arithmetic, we have lost many of the benefits that we might have hoped to gain by programming our driver in Haskell! For example, we can no longer be sure of memory safety because, just as in C, an error in our use of the `videoRAM` pointer at any point in the program could result in an unintentional, invalid, or illegal memory access that could crash our program or corrupt system data structures, including the Haskell heap. We have also had to compromise on strong typing; the structured view of video RAM as an array of arrays of character elements is lost when we introduce the `Ptr Word8` type. Of course, we can introduce convenience functions, like the following definition of `charAt` in an attempt to recreate the lost structure and simplify programming tasks:

```
charAt    :: Int → Int → Ptr Word8
charAt x y = videoRAM `plusPtr` (2 * (x + y*80))
```

This will allow us to output the character `c` on row `y`, column `x` using a command `poke (charAt x y) c`. However, the type system will not flag an error here if we accidentally switch `x` and `y` coordinates; if we use values that are out of the intended ranges; or if we use an attribute byte where a character was expected.

9.1.1 Our Approach: Strongly Typed Memory Areas

In this part of the dissertation, we describe how a functional language like Haskell or ML can be extended to provide more direct, strongly typed support for memory-based data structures. In terms of the preceding example, we can think of this as exploring the design of a more tightly integrated foreign function interface that aims to increase the scope of what can be accomplished in the functional language. We know that we cannot hope to retain complete type or memory safety when we deal with interfaces between hardware and

software. In the case of our video driver, for example, we must trust at least that the video RAM is located at the specified address and that it is laid out as described previously. Even the most careful language design cannot protect us from building a flawed system on the basis of false information. Nevertheless, we can still strive for a design that tries to minimize the number of places in our code where such assumptions are made, and flags each of them so that they can be detected easily and subjected to the appropriate level of scrutiny and review.

Using the language features described in the rest of this chapter, we can limit our description of the interface to video RAM to a single (memory) area declaration like the following:

```
type Screen   = Array 25 (Array 80 (Stored SChar))
area screen in VideoRAM :: Ref Screen
```

(The bitdata type `SChar` describes the content of a single location on the screen—for details see Chapter 11, Section 11.1). It is easy to search a given program's source code for potentially troublesome declarations like this. However, if this declaration is valid, then any use of the `screen` data structure, in any other part of the program, will be safe. This is guaranteed by the type system that we use to control, among other things, the treatment of references and arrays, represented, respectively, by the `Ref` and `Array` type constructors in this example. For example, our approach will prevent a programmer from misusing the `screen` reference to access data outside the allowed range, or from writing a character in the (non-existent) 96th column of a row, or from writing an attribute byte where a character value was expected. Moreover, this is accomplished using the native representations that are required by the video hardware, and without incurring the overhead of additional run-time checks. The first of these, using native representations, is necessary because, for example, the format of the video RAM is already fixed and leaves no room to store additional information such as type tags or array bounds. The second, avoiding run-time checks, is not strictly necessary, but it is certainly very desirable, especially in the context of many systems applications where performance is an important concern.

As a simple example, the following code shows how we can use our ideas to implement code for clearing the video screen:

```
cls = forEachIx (λ i →
       forEachIx (λ j →
         writeRef (screen @ i @ j) blank))
```

In this definition, `forEachIx` is a higher-order function that we use to describe a nested loop over all rows and columns, writing a `blank` character at each position.

9.2 Describing Memory Areas

In this section, we describe a collection of primitive types that can be used to describe the layout of memory areas, and a corresponding collection of primitive operations that can be used to inspect and modify them.

New Kind. We start by introducing a new kind called `Area` that classifies the types used to describe memory-area layouts. We use a distinct kind to emphasize that these are not first-class entities. For example, a memory area cannot be used directly as a function argument or result and must instead be identified by a reference or a pointer type:

```
cls :: Screen → IO ()     -- incorrect
cls :: Ref Screen → IO () -- correct
```

The first signature for `cls` results in a kind error because the function-space constructor `(→)` expects an argument of kind `*` but `Screen` is of kind `Area`.

Unlike types of kind `*`, implementations do not have the freedom to pick the representation for `Area` types. Instead, each `Area` type has a fixed representation that corresponds to the shape of the particular memory area that it describes. In practice, this is useful when we have to interact with external processes (e.g., hardware devices, or an OS kernel) and so we need to use a specific data representation.

Stored Values. In Chapters 7 and 8, we described a mechanism for specifying and working with types that have explicit bit-pattern representations. We now use such types as the basic building blocks for describing memory areas. To do this, we need to relate abstract types to their concrete representations in memory Unfortunately, knowing the bit pattern for a value is not sufficient to determine its representation in memory because different machines use different layouts for multi-byte values. To account for this, we provide two type constructors that create basic `Area` types:[2]

[2]In Section 9.4 we shall discuss exactly which types of kind `*` may be used as arguments to `BE` and `LE`.

```
LE, BE :: * → Area
```

The constructor `LE` is used for little endian (least significant byte first) encoding, while `BE` is used for big endian encoding. For example, the type `BE (Bit 32)` describes a memory area that contains a 32 bit vector in big endian encoding. In addition to these two constructors, the standard library for a particular machine provides a type synonym `Stored` which is either `LE` or `BE` depending on the native encoding of the machine. Thus writing `Stored (Bit 32)` describes a memory area containing a 32 bit-vector in the native encoding of the machine.

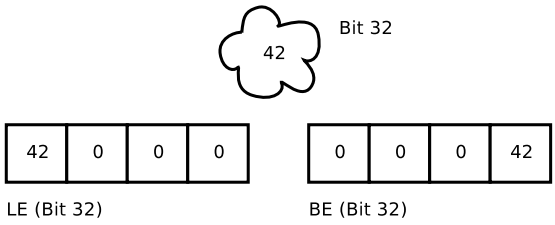


Figure 9.1: Different memory representations of multi-byte values.

Arrays. We may also describe memory-based array or table structures with the `Array` constructor:

```
Array :: Nat → Area → Area
```

The first argument indicates how many areas belong to the array, while the second argument describes the format of each sub-area. Thus the type `Array n t` describes a memory area that has `n` adjacent `t` areas. As the kind of the constructor suggests, we may use any area type to describe the sub-areas of an array. Here are some examples of different arrays:

```
Array 1024 (Stored (Bit 8))          --1024 bytes
Array 64 (LE (Bit 32))               --64 little endian dwords
Array 80 (Array 25 (Stored SChar))   --An array of arrays
```

Note that the array of arrays in the last example is a completely flat structure because arrays simply specify adjacent structures. Another way to describe the same physical area would be to use:

```
Array (80 * 25) (Stored SChar)
```

These two types are not exactly the same, however, because they differ in how programmers access the sub-areas of the array.

Structures. Arrays are useful to describe a sequence of contiguous memory areas that all have the same representation. Programmers may also define their own combinations of labeled adjacent areas with potentially different types for each component by using **struct** declarations. We describe these in detail in Section 10.1, but a simple example of a struct is a small area that contains two adjacent stored words, called `x` and `y`:

```
struct Point where
  x :: Stored (Bit 32)  --lower address
  y :: Stored (Bit 32)  --higher address
```

As with arrays, the fields of a structure may contain arbitrary areas, including arrays or other structures.

9.3 References and Pointers

Because area types belong to a kind different from `*`, we cannot use ordinary functions to manipulate areas directly. This is reasonable because `Area` types do not correspond to values but rather they describe regions of memory. The values that we work with are the *addresses* of various memory areas. We call such values *references* and we introduce a new type constructor for their types:

```
ARef :: Nat → Area → *
type Ref = ARef 1
```

The first argument to the constructor is an alignment constraint (in bytes), while the second argument is the type of the area that lies at the particular address. Thus, `ARef N R` is an `N`-byte-aligned address of a memory region, described by the `Area` type `R`.

Saying that an address is *aligned* on an `N`-byte boundary (or simply `N`-byte aligned) means that the address is a multiple of `N`. There are two common reasons to align data in memory: (i) to ensure efficient access to the data, and (ii) to represent memory addresses with fewer bits. For example,

on the IA32 it is more efficient to access 4-byte-aligned data then it is to access unaligned (1-byte aligned) data. Using fewer bits to represent aligned addresses is a technique commonly used in hardware design, which is why it also shows up in systems programming. It is based on the observation that, if an address is aligned on a 2^n-byte boundary, then the last n bits of its binary representation are 0, and so they do not need to be stored. Therefore, when we work with such devices we need to ensure that data is properly aligned, or else the device will not be able to access it. In situations where we do not have any particular alignment constraints we use the type synonym `Ref`. Here are some concrete examples of reference types:

```
Ref (Stored (Bit 32))
ARef 2 (Array 4 (BE (Bit 16)))
```

The first type is the address of a 32 bit value (in the native representation of the machine), stored at an arbitrary location in memory. The second type is for the address of a memory area that contains 4 big endian 16 bit values, and we know that this area is aligned on a 2 byte boundary.

Clearly, our references serve a purpose similar to pointers in C but they support a much smaller set of operations. For example, we cannot perform reference arithmetic directly, or turn arbitrary integers into references. Such restrictions enable us to make more assumptions about values of type `Ref`. In particular, references cannot be 'invalid' (for example `null`). This is in the spirit of C++'s references (e.g., `int&`) [85], and Cyclone's `notnull` pointers [46].

In some situations, it is convenient to work with *pointers*, which are either a valid reference, or else a null value. From this perspective, we may think of pointers as values of an algebraic datatype:

```
data APtr a r = Null | NotNull (ARef a r)
```

This simple abstraction provides an elegant way to avoid dereferencing `null` pointers because we provide only operations to manipulate memory via references. For example, if a function has a pointer argument, then before using it to manipulate memory, a programmer would have to pattern match to ensure that the value contains a valid reference. Attempting to use the pointer directly would result in a type error.

9.3.1 Relations to Bitdata

As the kind `*` suggests, references are ordinary abstract values, and so, in principle, an implementation is free to choose its own concrete representation. On the other hand, references also resemble a specialized form of bitdata because memory addresses are simply bit-vectors. It is therefore useful to provide an instance of the `BitRep` class for reference types. Of course, we do not provide an instance of `BitData` for references, as this would compromise the property that references always refer to valid memory areas.

We can provide the `BitRep` instance in two different ways. The first option is to use an instance like the following:

```
instance BitRep (ARef a r) AddrSize
```

The type synonym `AddrSize` is the number of bits used by the hardware to represent addresses. This solution is fairly easy to implement and is sufficient if all we need is to convert memory references to bit-vectors.

The other option is to take the alignment of the references into consideration and ignore bits that are guaranteed to be 0 by the alignment. We can do this with an instance like the following:

```
instance BitRep (ARef (2 ^ n) r) (AddrSize - n)
```

If we were to choose this instance, then we could define types like the following:

```
bitdata Perms = Perms { read :: Bit 1, write :: Bit 1 }
bitdata T = T { mem :: ARef 4 Data, perms :: Perms }
```

This type describes a bitdata type `T`, which contains a reference to some data (of type `Data`), and two permission bits. Because we know that the reference will be aligned on a `4`-byte boundary, we only need `(AddrSize - 2)` bits to represent the address, and therefore we can represent `T` values in exactly `AddrSize` bits. Such encodings seem to be quite common in systems programming.

Similar arguments apply to pointer types, except that they contain the extra `Null` value. It is quite common to encode `Null` with the 0 bit-vector, thus assuming that no memory area will start at address 0. We also adopt this convention in the `BitRep` instance for pointers. It is interesting to note that we could (nearly) define the type of pointers using a bitdata declaration like this:

```
bitdata APtr a r = NotNull { ref :: ARef a r }
                 | Null as 0
```

For this declaration to work, an implementation would have to know that 0 is not a valid value for references, otherwise the constructors would overlap, and the system will require a `BitData` instance for references, and so fail. Fortunately, this is easy to arrange. The problem with this definition is that it relies on *parameterized* bitdata, which is more difficult to analyze. For this reason, we do not support parameterized bitdata in our current implementation.

9.3.2 Manipulating Memory

In this section, we describe the operations that we use to store and retrieve values in memory areas. These operations are overloaded because, in general, values of different types have different representations, and it would be far too inconvenient to have a different name for each different type that may be stored in memory:

```
class Storable t where
  readRef  :: Align a ⇒ ARef a (Stored t) → IO t
  writeRef :: Align a ⇒ ARef a (Stored t) → t → IO ()
```

This is a simplified version of our final design, but it is sufficient to explain a number of common features. We used the name `Storable`, because there is a similar class in the Haskell FFI [21].

First, notice that the read and write operations only work on memory areas that contain stored abstract values. For example, we cannot read directly through a reference that points to an array, because there is no corresponding value type that we would get from such a reference.

The overloading of the operations allows us to generate different code for different values. Depending on the (implementation specific) representation of a particular value type, we would, in general, access memory in different ways. For example, if we work with boxed values, then we would have to box values after we read them from memory, and unbox them before writing them back.

Another interesting property of `Storable` is that the operations are guarded by an alignment constraint. The predicate `Align` identifies the subset of the natural numbers that can be used as alignment. This is useful because, for

example, an implementation that generates code for hardware that only supports 4 byte aligned memory access would not solve predicates like `Align 1`. In this way, attempting to access memory that is not properly aligned results in static type errors, rather than run-time crashes.

The types of `readRef` and `writeRef` include a monad [66], `IO`, that encapsulates the underlying memory state. For the purposes of our work, we need not worry about the specific monad that is used. For example, another alternative in Haskell would be to replace the `IO` monad with the `ST` monad [57], while in House [39], we might use the `H` (hardware) monad.

Recall that the type synonym `Stored` is for the native encoding of multi-byte values on a particular machine. However, we would like to manipulate values that are stored in other encodings too (e.g., big-endian values in an IA32 machine). To do this, we can abstract the type `Stored` from the types, and add it as an extra parameter to the class:

```
class Storable enc t where
  readRef  :: Align a ⇒ ARef a (enc t) → IO t
  writeRef :: Align a ⇒ ARef a (enc t) → t → IO ()
```

This is an example of a *constructor* class [50], because `enc` is a type constructor. For every type that we can store in memory, we would then have one instance for every encoding that we support. For example, for 32-bit vectors we have the instances:

```
instance Storable LE (Bit 32)
instance Storable BE (Bit 32)
```

There is an alternative, and slightly more general way to achieve the same result. The idea is to think of a class that encodes the relation between memory areas and the abstract values that are stored in them:

```
class ValIn r t | r ⇝ t where
  readRef  :: Align a ⇒ ARef a r → IO t
  writeRef :: Align a ⇒ ARef a r → t → IO ()
```

The predicate `ValIn r t` asserts that the memory area `r` contains an abstract value of type `t`. The functional dependency asserts that a memory area may contain at most one type of abstract value. This is not strictly necessary, but is quite useful in practice (e.g., if we read twice through a reference, then we know that we would get the same type of a value). In addition, the

functional dependency enables us to use the notation for functional predicates from Chapter 5. Thus we may think of `ValIn` as a partial function of kind `Area → *`. For example, we could write the type of `readRef` like this:

```
readRef :: Align a ⇒ ARef a r → IO (ValIn r)
```

The class `ValIn` is more general than `Storable` because we can encode all `Storable` instances like this:

```
instance Storable F T
-- corresponds to
instance ValIn (F T) T
```

For example, the instances for 32-bit vectors are like this:

```
instance ValIn (LE (Bit 32)) (Bit 32)
instance ValIn (BE (Bit 32)) (Bit 32)
```

As we can see, the instances are a little more verbose (this is quite common when we use a functional dependency). There are also instances of `ValIn` that we cannot represent with `Storable`. For example:

```
instance ValIn (Array 2 (LE (Bit 32))) (Bit 32, Bit 32)
```

Our implementation currently implements the class `ValIn`, although we have not yet made any essential use of its extra generality.

9.4 Representations of Stored Values

So far, we have seen how to describe memory areas and how to store and retrieve values from memory. In this section, we discuss exactly which types of kind `*` may be stored in memory, and what representations they should use. In our formalism, this amounts to specifying the instances for the class `ValIn`, and describing what these instances do to the memory. For example, we shall not provide an instance for memory areas of type `LE (Int → Int)` (because there is no standard way to represent such values) and so programmers have no operations to read or write `Int → Int` values into memory areas. For other types, such as `Bit 32` for example, we have one instance for each of the little- and big-endian representations. The essential difference between these two types is that `Bit 32` has a concrete bit-vector representation,

while the type `Int → Int` is an abstract type. Formally, this is captured by the fact that `Bit 32` belongs to the `BitRep` class while `Int → Int` does not. Membership in the `BitRep` class is a good starting point to determine which types may be stored in memory areas, but we still need to make some more decisions.

The first has to do with the size of the bitdata types involved. Typically, machines manipulate memory at a granularity of at least one byte (and some machines have more restrictions). We have to choose therefore, either to allow only bitdata whose width (in bits) is a multiple of 8, or else to pad other bitdata implicitly so that it occupies a whole number of bytes. Currently, in our implementation, we do not add any implicit padding, and so we do not allow storing bitdata whose size is not a multiple of 8. Instead, programmers have to specify how to pad such values explicitly.

The second decision that we have to make is if requiring a `BitRep` instance is sufficient, or if we should impose additional constraints on the type. At present, we have an extra requirement, namely that 0 is a valid value of all the types that are stored in memory. Clearly, this requirement is somewhat ad-hoc, but it makes it easy to initialize areas, simply by placing 0 at all positions. It also makes it safe to provide a `memZero` operation that clears all the data in a memory area. This function is quite useful in security sensitive applications when we reuse the same memory area for different purposes and hence must clear its content before each use to avoid potential information leaks. In Section 9.5, we shall discuss alternative approaches to initializing areas. The main effect of requiring memory-storable types to contain 0 is that we do not allow references to be stored in memory, but we do allow pointers.

In general, the representation of a stored value is determined by its `BitRep` instances. If the value belongs to a type that is `8*n` bits wide, then its area will occupy `n` consecutive bytes. The type-constructor that we use to define the area determines if the bit-vector is stored using a little- or a big-endian encoding. For single byte values, the representation is the same for `BE` and `LE` areas. There is one exception to these rules, and it has to do with how we store pointers. Recall from Section 9.3.1 that we represent a 2^n byte aligned pointer with `AddrSize - n` bits. For example, on a 32 bit machine, we represent a 4 byte aligned pointer with `30` bits. If we choose to follow the rules that we just stated, then we would have to disallow storing such pointers directly in memory. Instead, programmers would either have to use a wrapper bitdata type to specify the extra 2 bits in the representation,

or else 'forget' the alignment constraint before storing pointers in memory. Neither of these approaches is very satisfactory. For this reason, we allow for arbitrarily aligned pointers to be stored in memory, and the bit pattern that we use for a pointer `p` is `toBits p # 0`. Memory areas always take the number of bytes that are required to store an entire machine address. Simply put, the representation of a pointer is just the address to which it refers, including the 0s due to alignment. We can achieve this with the following instances:

```
instance ValIn (LE (APtr a r)) (APtr a r)
instance ValIn (BE (APtr a r)) (APtr a r)
```

If a programmer needs to store a pointer using its 'small' representation for some reason (i.e., to omit the bits that are guaranteed to be 0 by alignment), then they will have to use a wrapper bitdata type, for example like this:

```
bitdata Pack = Pack { ptr :: APtr 256 Data }
```

Because bitdata types consistently use the `BitRep` representation for their values, the `Pack` type would be represented with 3 bytes instead of 4. Of course, this again is a design choice that we made so that we have this functionality if we need it. The alternative would have been to generate the instances for types like `Pack` that contain only a single pointer in the same way as we did for pointers. We did not make this choice because: (i) it removes some functionality, namely the ability to store a pointer with its small representation; and (ii) it introduces more special cases to the system, which makes it more difficult to understand and implement.

9.5 Area Declarations

In this section, we describe a conservative, yet practical method for introducing reference values to a program. To declare memory references, programmers use an **area** declaration, which resembles a type signature:

```
area name [in region] :: type
```

Every such declaration introduces a distinct, non-overlapping memory area that may be accessed with the name specified in the declaration. The alignment constraints and the size of the area are computed from the type in the declaration. For example, we can declare a 4-byte aligned area to an array of 256 double words like this:

```
area arr :: ARef 4 (Array 256 (Stored (Bit 32)))
```

In the previous section, we discussed which abstract types may appear in memory areas. By limiting the instances to the `ValIn` class, we ensured that we have no operations to read and write from 'malformed' memory areas. While, in principle, this is sufficient to ensure the safety of the system, it would be nice if we could detect and reject declarations that contain such bad areas. To identify well-formed memory areas, we use the class `SizeOf`, which, as the name suggests, also serves to compute the size of the area:

```
class SizeOf r (n :: ℕ) | r ⇝ n where
  sizeOf  :: ARef a r → Int
  memCopy :: (Align a,Align b) ⇒ ARef a r → ARef b r → IO ()
  memZero :: (Align a) ⇒ ARef a r → IO ()
```

The methods of the class are generally useful functions that may be applied to all well-formed areas. The function `sizeOf` returns the size, in bytes, of the area referenced by the argument. The function `memZero` fills the memory area with 0. The function `memCopy` transfers the data stored in one memory area to another memory area of the same type. Implementations are free to choose the concrete algorithm used to implement these functions based on the features of the target architecture. Furthermore, note that implementations may use statically known information about the sizes and alignment of areas to produce more efficient code: for example, for small areas it may be beneficial to unroll the loops that copy (or zero) memory areas, thus avoiding any software bounds checking.

The predicate `SizeOf` can be discharged for atomic areas (as already discussed in the previous sections), and also for arrays and user defined structures. These instances for the latter are defined inductively in the obvious way: an array is valid if its element type is valid, while a structure is valid if all of its fields are valid. The size of an array is computed by multiplying the number of elements by the size of the elements, while the size of a structure is the sum of the sizes of its fields, plus space allocated for padding (see Chapter 10, Section 10.1 for a discussion of **struct** declarations). For example, this is the instance for arrays (the instances for structures are derived automatically from their declarations):

```
instance SizeOf (Array n r) (n * SizeOf r)
```

Notice that the syntax of **area** declarations does not place any restrictions on the type in the signature. Of course, not all types may be placed in such

declarations: we allow only types that are references to valid areas. We can formalize this with the `AreaDecl` class:

```
class AreaDecl t where
  initialize :: t → IO()

instance (Align a, SizeOf r n) ⇒ AreaDecl (ARef a r) where
  initialize x = memZero x
```

The class `AreaDecl` is used to validate **area** declarations. When an implementation encounters an **area** declaration, it has to prove that its type belongs to the `AreaDecl` class. This process rejects **area** declarations that do not describe valid memory areas but it also supplies useful information for the implementation: the size of the memory area, its alignment constraints, and how to initialize the area.

The memory areas introduced by **area** declarations are static, in the sense that they reside at a fixed location in memory, and the have a life-time that is as long as the life-time of the entire program. These properties make our areas very similar to a more structured version of `.space` directives found in some assemblers, and also to `static` declarations in C. Because the life-time of areas is the entire program, we never need to deallocate the memory occupied by such static areas and so memory areas do not (need to) reside in the heap like ordinary abstract values. In addition, the restrictions we place on what values can be stored guarantee that memory areas do not contain references to the heap containing the abstract values of the language. Because of this, the garbage collector does not need to examine memory areas.

Local Areas Currently we allow **area** declarations to appear both at the top level of a program and as local declarations. The semantics we use for local declarations is similar to the semantics of local static declarations in C: a local area declaration always refers to the same area, no matter how many times we call a function. This choice ensures that memory areas take a statically known amount of space, which can be very useful in systems applications. In Section 9.6.2, we briefly discuss some of the issues that arise when we work with dynamic areas. The choice that locally defined areas always refer to the same memory region has an effect on how we reason about programs (which itself impacts the way that we optimize programs). In pure languages, we usually expect that we can replace functions with their definitions (i.e., perform inlining). For example:

```
let f x = x in (f 1, f 2)
=
(1,2)
```

Unfortunately, this rule does not work (in general) if we allow local areas with the static semantics:

```
let f x = let area r :: T in r
in (f 1, f 2)
≠
( let area r :: T in r
, let area r :: T in r)
```

In the first case, both components of the pair contain a reference to the same area while, in the second, we declare two different areas. This is not too bad in implementations because they could first lift **area** declarations to the top level, before performing inlining optimizations. Still, it complicates the semantics of the language. In our experience with the system, we have not made essential use of local **area** declarations, although it appears that they may be useful to enforce abstraction properties such as hiding the reference to a local counter within a function definition.

9.5.1 External Areas

In some special circumstances, we may need to work with memory areas that were created by an external source. Examples include memory mapped devices (e.g., video RAM), or memory areas created by code written in a different language. To declare such external areas, programmers may annotate **area** declarations with an optional region using the keyword **in**. Such region annotations are similar to the different segments provided by assemblers. In general, implementations will not reserve memory for such areas. Instead, they should use the name of the region to determine how to initialize the reference for the appropriate area. For example, the video RAM on PCs resides at physical address `0xB8000`. We could declare a memory area to manipulate the screen like this:

```
type Row = Array 80 SChar
area screen in VideoRAM :: Ref (Array 25 Row)
```

When we compile the program, we would have to instruct the implementation what to do with `VideoRAM`, for example, by passing a flag like:

```
--region VideoRAM=0xB8000
```

An implementation may also accept regions with sizes and then check that the areas in a particular region really fit in the region. If more than one area is declared to be in the same region, an implementation would require some instruction on how to organize the areas in the region. This approach might seem a little vague, but we leave it vague on purpose so that we can accommodate various different ways to associate memory areas with external regions.

Clearly, working with such external areas is not entirely safe because nothing in the program guarantees that the external area corresponds to the declared type in the program. Notice, however, that the area declaration makes explicit the assumptions that we have about the external area. Furthermore, if the external area really corresponds to the declared type, then we know that our program will preserve these constraints (e.g., it will not be able to write outside of the declared area).

9.6 Alternative Design Choices

In this section, we discuss different designs that we have considered. We list them here because they all represent valid points in the design space. We have experimented with some (e.g., associating alignment with areas, rather than references), and rejected them in favor of the design we presented earlier in this chapter. Others, such as the generalized method of initializing areas seem promising, but we leave experimenting with them as future work.

9.6.1 Initialization

Our design adopts a simple method for initializing memory areas: we simply fill areas with 0. This has the benefit of being simple, and it does not require any space in a binary file to store the areas. The main draw-back of this approach is that memory areas may not contain types for which 0 is not a valid value. The main example for such types were references but, by the same principle, we would also have to reject bitdata types that cannot contain 0.

Default Values. We could relax this restriction in several different ways. (If we were to do so, then we would have to remove the method `memZero` from

the `SizeOf` class). As we saw in Section 9.5, there is a class `AreaDecl` that identifies the types that may be introduced with **area** declarations. This class contains a method to initialize the areas when they are created. The instance that we provided uses the function `memZero` to fill the area with 0s: Of course, an implementation may arrange for areas to be filled with 0 in different ways (e.g., by placing them in the `bss` segment). A different option would be to provide different instances for different types. This approach is more flexible because it allows us to initialize different types of areas in different ways. Still, this is not as useful as it may appear at first because, for some types, there is no good default value. For example, there is no good way to pick a default value for reference types unless we assume that there is some default area that all references of a particular type should point to. This is not likely to be useful in practice because the implementation would spend time initializing all references to point to the default area and then the program would have to spend some more time to initialize references to their actual values.

Initializers. A more useful feature would be to have a general mechanism for initializing areas on a per-declaration basis. We could do this by associating an *initializer* type with every area type. The initializers for a particular area are abstract values (i.e., of kind `*`) that can be used to initialize the area. We could encode this relation with another class:

```
class Initializer (r :: Area) (t :: *) | r ⇝ t
instance Initializer (LE (Bit 32)) (Bit 32)
instance Initializer (BE (Bit 32)) (Bit 32)
instance Initializer (Array n a) (Ix n → Initializer a)
```

These declarations specify that, to initialize a memory area containing a stored `Bit 32` value (in either encoding), we must use a `Bit 32` value. To initialize an array, however, we provide a function that maps array indexes to initializers for the appropriate element (the type `Ix n` is discussed in Chapter 10, but it is essentially a subset of the natural numbers that are valid array indexes). For example, the initializer for an area of type `Array 8 (LE (Bit 32))` would be a function of type `Ix 8 → Bit 32`. Note that this a pure function, which means that we cannot use the values stored in areas to initialize the array. Clearly, this restricts us in how we can initialize areas, but it also has the benefit that we do not need to worry that an initializer is using a value stored in an area that has not itself been initialized yet. Here is an example

of how we may use initializers to define an area that contains a reference to itself.

```
struct List a where
  hd :: a
  tl :: Ptr (List a)

area cycle = (hd = 0, tl = NotNull cycle) :: List (Stored (Bit 32))
```

This example declares a structure type with two fields. We then declare an area that uses the same structure and we provide an initializer for it, which creates a circular list. The initializer for a structure type is a record, whose fields contain initializers for the fields of the structure.

Marshalling It is interesting to note that there is a similarity between the class `Initializer` and `ValIn` because they both specify the kind of values stored in an area. Another point in the design space would therefore be to unify these two classes, replacing `Initializer` with `ValIn`, and providing extra instances for arrays and structures as we did for initializers. The operations to read and write from such areas would essentially marshal data between memory areas and the abstract heap: arrays would be turned into abstract arrays (i.e., functions), while structures would be converted to records.

Uninitialized References. Yet another way to deal with initialization of areas would be to introduce a new type for uninitialized references, say:

```
URef :: Nat → Area → *
```

We would then change the instance for the `AreaDecl` class to use `URef`, rather than `ARef`, and we could eliminate the `initialize` methods. Because `URef`s do not belong to the `ValIn` class, we would not be able to manipulate uninitialized areas. To initialize areas, we would use a function that would marshal-out a value into an area, after which it would return an `ARef` to the area:

```
init :: Align a ⇒ URef a r → Initializer r → IO (ARef a r)
```

This method is a little more general than the previous one in that it allows us to use the values from memory areas that have already been initialized. A draw-back of this approach is that we cannot use the global reference values

directly because they have the uninitialized reference type. Instead, after initializing an area, we have to pass the 'new' reference to all functions that need to use it.

9.6.2 Dynamic Areas

In some applications, programmers may prefer to work with a more dynamic forms of memory area, for example, in which it is possible to allocate and deallocate memory areas on demand. As is, our design does not support such operations. There are a few different ways in which we could generalize the design. For example, we could add an operation to allocate a new area dynamically, perhaps like this:

```
newArea :: AreaDecl t ⇒ IO t
```

The `IO` effect is there because this operation has a side effect (e.g., it allocates a new area, and the operation may fail). To deallocate areas we have a number of different choices: (i) do not deallocate areas; (ii) garbage collect memory areas; (iii) use an explicit deallocation schema, perhaps using the techniques developed for working with memory regions [35]. The first option is useful if the areas are used for communication with external processes, but we do not statically know how many areas we will need. Such options are also useful for systems that perform some dynamic configuration at startup. Adding a garbage collector could enable us to reuse some memory used by areas. Notice that this collector does not need to use the same algorithm as the collector for the abstract heap (e.g., we probably want a collector that does not move areas). Writing a collector for memory areas is not entirely trivial because we would have to deal with references that point in the middle of areas (objects) (e.g., a pointer to the 5^{th} element of an array). The third option, providing programmers with the ability to deallocate areas, is more low-level than having a garbage collector, but gives programmers more control over the life-time of areas. Here, the main issue is how to make the system safe (i.e., how to avoid using areas that have been deallocated while there are still pointers to those structures in other parts of the program). A promising line of research in this direction is to use regions analysis and linear/unique types [94], and indeed these ideas have already been used in the design of some languages [46, 79].

9.7 Summary

In this chapter, we described the basics of our design for working with data that has explicit memory representation. We introduced a new kind, `Area`, that classifies descriptions of contiguous memory regions. Programmers identify and manipulate such memory regions via references. The type of a reference contains a description of the memory region at the given address, as well as an alignment constraint. References support only a limited set of operations, which preserve the invariants specified by their types.

There are two ways to introduce references to a program: (i) by using **area** declarations; (ii) by using a trusted external source (i.e., an external function that produces a reference value). The areas defined with **area** declarations occupy disjoint regions of memory and have a life-time that is as long as the execution of the program.

The following declarations provide a summary of the types and (overloaded) functions that we used in this chapter:

```
-- Types of kind Area
BE    :: * → Area           -- big-endian
LE    :: * → Area           -- little-endian
Array :: Nat → Area → Area -- array of areas

-- References
ARef  :: Nat → Area → *     -- alignment, area type


-- Valid alignments
Align :: Nat → Pred

-- Well-formed area types
class SizeOf r (n :: Nat) | r ⇝ n where
  sizeOf  :: ARef a r → Int
  memCopy :: (Align a,Align b) ⇒ ARef a r → ARef b r → IO ()
  memZero :: (Align a) ⇒ ARef a r → IO ()


-- Areas containing stored values
class ValIn r t | r ⇝ t where
  readRef  :: (Align a) ⇒ ARef a r → IO t
  writeRef :: (Align a) ⇒ ARef a r → t → IO ()
```

Chapter 10

Structures and Arrays

In this chapter, we focus on memory areas that contain sub-areas, such as user-defined structures and arrays. Section 10.1 describes **struct** declarations, which are used to declare user-defined memory areas. Section 10.2 describes how to access array elements. Section 10.3 discusses functions that allow us to change the 'view' on a particular region of memory. Finally, in Section 10.4 we provide a summary of our design for working with memory areas[1].

10.1 User Defined Structures

As we have already discussed in Chapter 9, programmers may define new types of kind `Area` using **struct** declarations. In this section, we describe the details of this new form of declaration. The syntax of user defined `Area` types resembles the syntax of Haskell's **class** declarations:

```
struct struct-head where
  fields

field = label :: type  -- Labeled field
      | type           -- Anonymous field (padding)
      | ..             -- Computed padding
```

[1]The material in this Chapter is based on the following paper:
Iavor S. Diatchki and Mark P. Jones. Strongly Typed Memory Areas. In *Proceedings of ACM SIGPLAN 2006 Haskell Workshop.*, pages 72–83, Portland, Oregon, September, 2006.

The *head* of a **struct** declaration is similar to a class declaration, in that it specifies the name for the structure, together with optional type parameters, context, and attributes. We shall discuss attributes shortly, but syntactically, they are written in the same place where we specify the functional dependencies for a Haskell class. The fields of the structure determine the layout of the corresponding memory area: the first field is placed at the lowest memory addresses, and subsequent fields follow at increasing addresses. As with Haskell's class declarations, syntactically we may use either implicit or explicit layout to specify the list of fields.

As an example of a user-defined structure, consider the following declaration which describes the static part of the header of an IPv4 packet that might be used in a network protocol stack:

```
struct IP4StaticHeader where
  ipTag            :: Stored IPTag
  serviceType      :: Stored ServiceType
  total_length     :: BE (Bit 16)
  identification   :: BE (Bit 16)
  fragment         :: BE Fragment
  time_to_live     :: Stored (Bit 8)
  protocol         :: Stored Protocol
  checksum         :: BE (Bit 16)
  source_addr      :: BE Addr
  destination_addr :: BE Addr
```

This declaration introduces a new type, called `IP4StaticHeader`, which is of kind `Area`. The area that it describes is made up of a number of adjacent stored values, the first (at lowest address) being `ipTag`. Note that, in this application the network standard specifies that multi-byte values are transmitted using big-endian encoding. For this reason, it is particularly important to use the `BE` constructor, so that the multi-byte fields of `IP4StaticHeader` are interpreted correctly, even on platforms where the default encoding is little-endian.

10.1.1 Accessing Fields

Structures have operations that allow us to access their sub-components using the labels on their fields. We can reuse the record system from Chapter 7 to avoid introducing a new notation for accessing the fields of a structure.

In this case, the record values are references to user defined memory areas. The type of the fields of such records are references to the sub-components of the structure. For example, the declaration of `IP4StaticHeader` introduces instances similar to the following:

```
instance Field'total_length (Ref IP4StaticHeader) (Ref (BE (Bit 16)))
```

These instances enable us to obtain references to the sub-components of a field using the standard record operations. For example, here are two different ways to read the value stored in the `total_length` part of an `IP4StaticHeader`:

```
getLen1 r                      = readRef r.total_length
getLen2 { total_length = l } = readRef l
```

The first function uses a projection operation to obtain a reference to the field, while the second uses a record pattern. Notice that the record operations are pure because they just compute a reference to the field. To access the value that is stored in the corresponding area, we still need to use the `readRef` function. The references produced by the projection operations are based on the structure declaration, and so they are guaranteed to be valid references.

Recall that in Chapter 7, we also used the class `UpdField`, which had a method for updating the value of a field. We do not provide instances of `UpdField` for user-defined **struct**s because this operation does not make sense: the values of the fields in such 'records' are at fixed offsets and we cannot change them to arbitrary references.

10.1.2 Alignment

The example instance for the field `total_length` is a little simplified because it omits the alignments on the references. The real instance that we generate from the structure declarations computes an alignment for the sub-field that we access. This alignment depends on two parameters: the alignment of the entire structure, and the offset of the field within that structure. The alignment of the reference to the resulting field is computed by taking the greatest common divisor of these two parameters. To see why this is the case, recall that, if a structure is aligned on an a-byte boundary, then its address must be $k * a$ for some natural number k. The address of the field that is n bytes from the beginning of the structure would therefore be $k * a + n$. This address is aligned on any byte boundary, b, that divides $k * a + n$. Therefore,

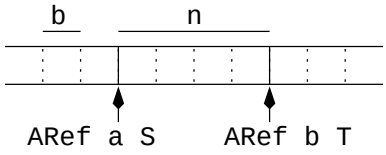


Figure 10.1: Alignment of a Field

to ensure that b is a valid alignment for the field, it has to divide both a and n, because, in general, we do not know the value of k. For example, if a structure is aligned on a 4 byte boundary (i.e., $a = 4$), a field that is at offset 6 bytes (i.e., $n = 6$) would be aligned on any boundary that divides both 4 and 6 (i.e., 1 or 2).

At this point we face a design choice: (i) we may fix the alignment of the field to the largest possible alignment, given by the greatest common divisor of a and n; or (ii) we may leave the alignment polymorphic, subject to the constraints that it divides both a and n. This choice does not affect the expressiveness of programs, but it has an impact on how programmers work with the fields. Choice (i) leads to more concrete types in the program because, if we know the alignment of a structure reference and the offset of the field, then we know the exact alignment of the field. The upside of this is that we tend to infer simpler types, and also we pose simpler problems to our constraint solver. The down-side is that, in some situations, programmers may have to 'forget' the alignment constraints on a reference explicitly before they can use it, with a function like this one:

```
realign :: (GCD a b = b) ⇒ ARef a t → ARef b t
```

This would happen if a function expects a 1-byte aligned reference as an argument, but instead we have a 4-byte aligned reference. Such casts would not be necessary if we left the alignment of sub-fields polymorphic (although the same situation may arise for other reasons). Besides inferring slightly more complicated types, option (ii) has the draw-back that it makes it easier to write programs that are ambiguous. This happens when there is not enough information in the program to determine what alignment to use when manipulating a reference. In contrast, this is less frequent if we use option (i) because then we would simply use the largest possible alignment. We could, of course, avoid such ambiguities by using type annotations to specify the alignment explicitly, but this would clutter the program, because option (ii)

leads to quite a few ambiguities. Consider, for example, a function like the following:

```
getLen :: ARef 4 IP4StaticHeader → IO (Bit 16)
getLen r = readRef r.total_length
```

From the definition of `IP4StaticHeader`, we know that `r.total_length` is: (i) a reference to an area containing a big-endian 16 bit value; and (ii) it resides 8 bytes from the beginning of the structure. Using choice (i) we would infer the type `ARef 4 (BE (Bit 16))`, and so this definition would be accepted. If we use option (ii) however, the system would report an ambiguity, because nothing in the program indicates if we should use 1,2, or 4 byte alignment, all of which are valid.

Based on the previous discussion, we choose option (i), and so the instances that we generate for the fields in a structure use the greatest common divisor to compute the alignments of the sub-fields. For example, the full instance that would be generated for the field `total_length` looks like this:

```
instance Field'total_length (ARef a IP4StaticHeader)
                            (ARef (GCD a 8) (BE (Bit 16)))
```

Nested Structures. When we compute the alignment of a sub-field, we loose some information, namely the fact that field belongs to a larger structure, which itself was accessed through an aligned reference. This is why programmers should be careful when nesting structures in code where alignment is important. To illustrate what how alignment information may be lost, consider the following example:

```
struct Pair a b where
  fst :: a
  snd :: b

type L s t u = Pair (Pair s t) u
type R s t u = Pair s (Pair t u)
```

These types describe essentially the same memory areas: both have three fields that are of types `s`, `t`, and `u` respectively. Their accessor functions, however, have subtly different types. Consider, for example, the type of the function the accesses the third field:

```
thirdL  :: GCD a (SizeOf s + SizeOf t) = b ⇒
           ARef a (L s t u) → ARef b u
thirdL x = x.snd

thirdR  :: GCD (GCD a (SizeOf s)) (SizeOf t) = b
           ARef a (R s t u) → ARef b u
thirdR x = x.snd.snd
```

While both functions return references to the same type of area, the references have different alignment constraints. To see this, consider the case where `s` is of size 3 bytes, `t` is of size 1 byte, and the alignment `a` is 4. Then `thirdL` will produce a 4 byte aligned reference, because $\gcd(3 + 1, 4) = 4$, while `thirdR` will produce a 1 byte aligned reference, because $\gcd(\gcd(4, 3), 1) = 1$.

10.1.3 Padding

In some situations, it is useful to specify that a part of a memory area is not used. We can do this by using special 'padding' fields in **struct** declarations. Such fields occupy space, but we do not provide accessor functions to access the memory. Most commonly, padding is used to conform to an external specification, or to satisfy alignment constraints for the fields of a structure. The simplest way to provide padding in a structure is to use an *anonymous field*: the programmer writes a type, but does not provide a label for it. We can even introduce a type synonym:

```
type PadBytes n = Array n (Stored (Bit 8))
```

and then write `PadBytes n` in a **struct** to add `n` bytes of padding.

Some memory areas occupy a fixed amount of space, but do not utilize all of their space. Such structures typically have a number of fields, and then there is 'the rest', the unused space in the structure. To define such structures using only anonymous fields, programmers would have to compute the amount of padding in the structure by hand. To simplify their job, we provide the *computed padding field*, which is an anonymous field whose size is automatically computed from the context, and from size constraints specified by the programmer. There can be at most one such field per structure because there is no way to specify how the available space should be distributed between multiple padding fields. As a concrete example, consider implementing a resource manager that allocates memory pages, each of which

is 4096 bytes long. A common way to implement such a structure is to use a 'free list': each page contains a pointer to the next free page, and nothing else. Using some more special syntax, we may describe the memory region occupied by a page like this:

```
struct FreePage | size 4K where
  nextFree :: Stored (Ptr FreePage)
  ..
```

This example illustrates a few new pieces of concrete syntax: (1) The optional attribute `size t` is used to specify the size of the structure; an error will be reported if the declared size is either too small, or else if there is no computed padding field for any surplus bytes. (2) The literal `4K` is just a different way to write 4096. We also support `M` (for 'mega', times 2^{20}) and `G` (for 'giga', times 2^{30}). Like other literals, these can be used in both types and values. (3) Note that the `..` in the above example is concrete syntax for a computed padding field and is not a meta-notation.

10.2 Working with Arrays

In this section, we describe a simple mechanism that allows us to access the elements of an array both efficiently and without compromising safety. Instead of using arbitrary integers as array indexes, we use a specialized type that guarantees that we have a valid array index:

```
Ix :: ℕ → *
```

Values of type `Ix n` correspond to integers in the range `0` to `n-1`, so they can be used to index safely into any array of type `Array n a`. (In particular, `Ix 0` is an empty type.). In this respect `Ix` types are a special case of the ranged types available in the Pascal family of languages [18]. Notice that `Ix` types satisfy the requirements for values that may be stored in memory (discussed in Chapter 7), because they have a well-defined bit representation, and because 0 is always a valid index. The `BitRep` instance for `Ix` types is simply a bit-vector that corresponds to the index. We represent all `Ix` types with the same number of bits, which, like the representations for pointers and references, is machine dependent. There are no instances of `BitData` for `Ix` types, because the guarantee that `Ix` types correspond to valid indexes would be broken if we were to allow arbitrary bit vectors to be cast into indexes.

10.2.1 Index Operations

We group the operations on indexes into a class called `Index`. The parameter to the class is a natural number, and the corresponding predicate, `Index n`, asserts that `n` may be used to form index types. In this respect, the predicate `Index` resembles the predicate `Align`, which identifies valid alignments. It is useful to restrict the indexes to ensure that the operations on `Ix` types have well-defined semantics, and that they can be implemented efficiently. For example, `0` is not a valid index because `Ix 0` is an empty type, and therefore we cannot get the smallest index of the type. We also mentioned that, in memory areas, indexes occupy the space of a machine word. This places an upper bound on the `Ix` types that can be used in the system, which we can also formalize with the class `Index`.

```
class Index n where
  (@)     :: ARef a (Array n t) → Ix n → ARef (GCD a (SizeOf t)) t

  toIx    :: Int → Maybe (Ix n)
  fromIx  :: Ix n → Nat
  bitIx   :: (Width m, 2^m = n) ⇒ Bit m → Ix n

  minIx   :: Ix n
  maxIx   :: Ix n

  inc     :: Int → Ix n → Maybe (Ix n)
  dec     :: Int → Ix n → Maybe (Ix n)
```

We use the operation `(@)` to access elements of arrays. This operation works in much the same way as the accessor functions in structures: in particular, it is pure because it simply uses pointer arithmetic to compute a reference to a memory area. If the area contains a stored value, then we can use the operations from the `ValIn` class to manipulate the memory. The alignment constraints are computed in the same way as for structures except that, because we do not know the value of the index, we do not know the exact offset of the area. This is why we use only the size of the array elements to compute the alignment.

The operations `toIx`,`fromIx`, and `bitIx` are used to convert index values to/from other types. The function `fromIx` is essentially the same as `toBits` in that it 'forgets' that a value has the restrictions imposed by the `Ix` type. The function `bitIx` converts a $\log n$-bit vector to an `Ix n` value. Here, the

types ensure that we always get a valid index because $\log n$-bit vectors have exactly the same values as `Ix n` indexes. The most interesting operation is `toIx`, which performs a dynamic check to turn an integer into an index.

The values `minIx` and `maxIx` are the lower and upper bounds of the `Ix n` type, while `inc` and `dec` are partial functions that if possible, increment or decrement an index by the given amount. Values of type `Ix` also support operations to compare indexes for equality and ordering.

10.2.2 Iterating over Arrays

The operations that we have described so far are safe and expressive enough to support any style of indexing because `toIx` can turn arbitrary numbers into indexes. This, however, comes at the cost of a division (dynamic check), which can be quite expensive. In this section, we explain how such overheads can be avoided in many cases.

Programs often need to traverse an array (or a sub-range of an array). Usually this is done with a loop that checks if we have reached the end of the array at every iteration, and if not, manipulates the array directly without any dynamic checks. We could program examples like this by using the `inc` and `dec` operations of the `Index` class. For example, we could write a function to sum up all the elements in an array like this:

```
sumArray a  = loop 0 minIx
  where
  loop tot i  = do x ← readRef (a @ i)
                   let tot' = tot + x
                   case inc 1 i of
                     Just j → loop tot' j
                     Nothing → return tot'
```

This function can be compiled without too much difficulty into code that is quite similar to the corresponding C version, with two exceptions: (i) it probably performs *two* checks at every loop iteration to see if we have reached the end of the array, one in the function `inc`, and another one immediately afterwards in the **case** statement; (ii) it may create junk `Maybe` values in the heap. This is not very nice, because this increment-and-check pattern is perhaps the most common way in which we use indexes. It is, of course, possible that an optimizing compiler could detect and eliminate these redundant checks by inlining the definition of `inc`, and to eliminate the junk `Maybe` values by using

vectored returns. The cost of this would be a more complicated compiler, and perhaps slower compilation times, but on the positive side, the same optimizations could be useful to improve other parts of the program.

An alternative to relying an a compiler optimization would be to introduce increment and decrement *patterns*. The patterns either fail if we cannot increment or decrement an index, or succeed and bind a variable to the new value. It is interesting to note that Haskell's $n+k$ patterns do exactly this for decrementing. For example, we may write the factorial function in Haskell like this:

```
fact x = case x of
           n + 1 → x * fact n
           _ → 1
```

If `x` is a (positive) non-zero number, then the first branch of the **case** succeeds and `n` is bound to a value that is 1 *smaller* then the value of `x`.

To support *incrementing*, we can use a symmetric `n-k` pattern (not present in Haskell), which succeeds if we can increment an index by `k` and still get a valid index. Using these ideas we can write the above loop more directly:

```
sumArray a  = loop tot minIx
  where
  loop tot i  = do x ← readRef (a @ i)
                   let tot' = tot + x
                   case i of
                     j - 1 → loop tot' j
                     _ → return tot'
```

We can give semantics for these two patterns by using the calculus from Chapter 6:

```
(n + k) ≡ (x | Just n ← dec k x)
(n - k) ≡ (x | Just n ← inc k x)
```

Notice that we use a minus in the pattern to increment, and we use a plus to decrement, which may be a little confusing at first. In practice, we expect that programmers will not write too many explicit loops like the previous examples. Instead, they will use higher-level combinators, analogous to `map` and `fold` for lists, that are implemented with the incrementing and decrementing patterns. For example, we can abstract the looping part of the `sumArray` example like this:

```
accEachIx :: Index n ⇒ a → (a → Ix n → IO a) → IO a
accEachIx a f = loop a minIx
  where
  loop a i = do b ← f a i
                case i of
                  j - 1 → loop b j
                  _ → return b
```

This function, `accEachIx` is quite general, and can be used to give a much more compact definition of `sumArray` (and, of course, also to define other similar functions):

```
sumArray a = accEachIx 0 (λ tot i →
               do x ← readRef (a @ i)
                  return (tot + x))
```

10.2.3 Related Work

The approach that we have described here uses range types to index safely into arrays. Another possibility is to use *singleton* types [101, 36]. Singleton types can be used to get some of the benefits of dependent types without using a system that fully supports dependent types. The idea is that each value of a given type is introduced as a new type that contains only the corresponding value. For example, we can 'lift' the natural numbers to the type level by using a type constructor like this:

```
SNat :: Nat → *
```

Then the type `SNat 42` is a type that contains only a single value, namely the number 42. Then we can reformulate the operations on array types using singleton types, instead of range types, an idea described by Xi and Pfenning [102]. For example, the function to access an element of an array could be given a type like this:

```
(@) :: (m < n) ⇒ ARef a (Array n t) → SNat m
               → ARef (GCD a (m * SizeOf t)) t
```

This type differs from the previous type that we used, because now we know the value of the index statically (it is recorded in the type of the second argument). The constraint `m < n` ensures that the index is within the range of the array. Because we know the value of the index we can also compute

a more accurate alignment: when using a range type we had to approximate the alignment of an element because we only knew the range of the index, but not its actual value. Of course, in most cases when we work with arrays we do not know the position that we need to access statically. To work around this problem, systems that use singleton types often rely on existential quantification to represent information that is not known statically. For example, this is how we can define ranged types by using singleton and existential types:

```
data Ix n = ∀ m. (m < n) ⇒ Ix (SNat m)
```

The quantifier in this data declaration introduces a type variable that is local to the constructor. Because such type variables do not appear in the result type of the constructor (in this case `m` does not appear in `Ix n`), they correspond to existentially quantified types. Here is the full type of the constructor `Ix`:

```
Ix :: ∀ m n. (m < n) ⇒ SNat m → Ix n
```

This type also illustrates how existential types hide information: note that the type `m` appears in the argument of the constructor, but not in the result, thus "hiding" the value of the singleton type. Now we could try to define our old ranged array indexing operator like this:

```
rangedIx a (Ix n) = a @ n
```

Unfortunately, this does not work because the result type of the function mentions the type (`m`), which is 'hidden' in the existential:

```
rangedIx :: ARef a (Array n t) → Ix n → ARef (GCD a (m * SizeOf t)) t
```

This is not allowed, because `m` is not known statically, and so it cannot openly appear in the types of values. To work around this, we would have to declare an explicit type for the operation, and then (paradoxically!) use the function `realign` to 'forget' the information that we did not know in the first place:

```
realign :: (GCD a b = b) ⇒ ARef a t → ARef b t

rangedIx :: ARef a (Array n t) → Ix n → ARef (GCD a (SizeOf t)) t
rangedIx a (Ix n) = realign (a @ n)
```

To check this definition an implementation would have to prove the following:

```
(SizeOf t x, GCD a x = b) ⇒ GCD (GCD a (m * x)) b = b
```

This proof requires knowledge of the interaction between `GCD` and `*`. The rules that we presented in Chapter 4 are quite weak and cannot prove this equation in its polymorphic form. Of course, we could provide additional rules to specify more fully the operations on natural numbers.

The main conclusion we can draw from this example is that a system with singleton types and support for existential quantification is more expressive than the system of ranged index types that we have used. This is because singleton types enable us to specify and exploit statically known information, but still use existential types in situations where information is not statically known. The cost of this extra expressiveness is that it results in more complex constraints that have to be discharged by the system. The constraints are more complex, because existentials introduce goals with unknown variables, which cannot be solved by simple evaluation, but instead have to be solved using general rules.

The type system used in the designs that we have discussed so far is based on the Hindley-Milner type system extended with qualified types. It is also possible to use a more general type system, for example, a system based on the calculus of inductive constructions [7]. Such type systems are very expressive and have been used as the basis for the design of automated proof assistants such as Coq [12]. If we were to base our design on such a type system, then we could give the indexing operator the following type:

```
(@) :: (a :: Nat) → (t :: Area)
    → (ix :: Nat) → (size :: Nat) → (p :: ix < size)
    → ARef a (Array size t) → ARef (GCD a (ix * SizeOf t)) t
```

This type requires some explanation. The arguments `a` and `t` specify the alignment and the type of elements for the array that we are indexing. Note that in this system we do not need to introduce a new *kind* `Nat` but, instead, we may reuse the *type* `Nat`. The arguments `ix`, `size` and `p` specify the index of the element that we need to access, the size of the array, and a proof that the index is not outside of the array bounds. The final argument is a reference to the array that is being indexed. The functions `GCD` and `(*)` are ordinary functions that manipulate natural numbers. The function `SizeOf` computes the sizes of `Area` types. Finally, the relation `(<)` is encoded as a type by using the ideas of the Curry-Howard isomorphism [90].[2]

[2]We could reduce the number of arguments in the type by packaging some of the arguments together: for example, we could store references to arrays together with their

Besides the fact that such type systems are very expressive, they are also quite elegant because the language treats uniformly values, types, proofs, and properties. Unfortunately, the generality of the system makes the static analysis of programs much more complex. For example, full type inference for such systems is not possible, however various systems may still support limited forms of type inference (e.g., Cayenne's hidden arguments [4]). Furthermore, blurring the distinction between types and values makes the generation of good code more complex because the compiler has to determine which parts of the program are needed at run-time, and which parts represent static information that is not required while executing the program. Consider, for example, the arguments `a` (alignment) and `ix` (index) from the previous type signature. In the signature they appear identical (i.e., they are both values of type `Nat`). However, we might expect that the alignment of the reference is a static property, while the index of the element that we need to access is a dynamic value. There are various ways to work around this problem, for example, Coq introduces different *sorts* for terms that correspond to statically known information and dynamic values, while Cayenne [4] extends the type-system to track terms that need to be represented at run-time.

In summary, our design is fully compatible with a type system that is based on the calculus of inductive constructions. However, in this dissertation our goal is to explore how far we can get by using the more restricted system of Hindley-Milner with qualified types. Our reason for this choice is that this type system has a number of nice properties, and has proved to work remarkably well in the design of Haskell.

10.3 Casting Arrays

In this section, we discuss a number of operations that are used to change the description of a memory area. Most of the operations are like type casts in C/C++, in that they do not need to perform any work at run-time. Unlike C/C++ however, we provide only a limited set of casts that does not compromise the invariants we enforce with types.

---

sizes, and we could also group array indexes together with the proof that they are in bounds, thus recovering a type similar to `Ix`.

10.3.1 Arrays of Bytes

One set of casting operations deals with converting between structured descriptions of memory areas and arrays of bytes. These operations are restricted to memory areas that contain only values in the `BitData` class, a property formalized with the class `Bytes`:

```
type BytesFor t = Array (SizeOf t) (Stored Byte)

class Bytes t where
  fromBytes :: ARef a (BytesFor t) → ARef a t
  toBytes   :: ARef a t → ARef a (BytesFor t)

instance (BitData t (8 * n)) ⇒ Bytes (BE t)
instance (BitData t (8 * n)) ⇒ Bytes (LE t)
instance (Bytes t) ⇒ Bytes (Array n t)
```

The instance for arrays allows turning an array of structured data into an array of bytes. Programmers may also use a `deriving` mechanism to derive instances for user defined structures. These only work if the system can ensure that all of the fields of a structure are in the `Bytes` class.

The functions `toBytes` and `fromBytes` resemble the bit operations `toBits` and `fromBits` from Chapter 7 but they work on memory areas, instead of bit vectors. There is an important difference, however, because memory areas are mutable. It was beneficial to place `fromBits` and `toBits` into separate classes, because it enabled us to distinguish two forms of bitdata: one that is more abstract, because it does not contain arbitrary bit patterns, and another that is more like a 'view' [93] on a bit vector. Placing `fromBytes` and `toBytes` into separate classes does not lead to the same distinction for memory areas. Each of these operations provides us with two different views on the same memory region. Because memory is mutable, whenever we have two different views on the same region, we can perform 'casts' by writing to memory through the one view, and reading back from it through the other. This is why we allow only types that belong to the `BitData` class to be stored in memory areas that support casting to byte arrays.

To see why this restriction is important consider the following example, which should be rejected:

```
badIx :: Ref (LE (Ix 8)) → IO (Ix 8)
badIx r = do let bs = toBytes r :: Ref (Array 4 (Stored (Bit 8)))
```

```
writeRef (bs @ 0) 10
readRef r
```

This function attempts to invalidate the property that the type `Ix 8` contains only the values `{0 .. 7}` by converting the reference `r`—which points to an area containing an index—into a reference `bs` that points to an array of bytes. The problem is that both of these references point to the same region of memory but they have incompatible types. This allows us to use `bs` to place 10 (an invalid value for `Ix 8`) in the memory region and then use `r` to read a malformed `Ix 8` value. In our design, the function `badIx` is ill-typed because the use of `toBytes` requires that we discharge the predicate `Bytes (LE (Ix 8))`. This, in turn, amounts to discharging the predicate `BitData (Ix 8) (8 * n)` but there are no instances of `BitData` for `Ix` types.

10.3.2 Reindexing Operations

Another useful set of casting operations involve array indexes. These operations do not perform any computation but instead, they change the way that we perform indexing on arrays. For each casting operation we also provide a corresponding reindexing function that allows us to convert indexes of the original array to indexes in the new array.

The first casting operation allows us to split an array into two smaller arrays, at a statically known position:

```
splitArray :: (Index x) ⇒ Ref (Array (x + y) t) →
                          ( Ref (Array x t),  Ref (Array y t) )

subIx      :: (Index x) ⇒ Ix (x + y) → Either (Ix x) (Ix y)
```

The corresponding operation on indexes is called `subIx`, and it subtracts `x` from its argument. If the subtraction is successful (i.e., the result is not negative) we return a `Right` value, otherwise we return the index unmodified as a `Left` value with a more precise type. This is exactly what we need to convert indexes of the original array into indexes of either the left or the right sub-array. For example, if we split an array with 10 elements into two parts: one with 3 and the other with 7 elements, then index `5` of the original array would correspond to index 2 of the second array (i.e., `Right 2`).

We can formalize the relation between `splitArray` and `subIx` with the following equation:

```
a @ i = let (a1,a2) = splitArray a
        in case subIx i of
             Left j  → a1 @ j
             Right j → a2 @ j
```

Here, `a` is a reference to an array and `i` is an index of `a`. The equation asserts that indexing with `i` into `a` yields the same reference, as splitting `a` into two parts, converting the index `i`, and then using the result to index into the corresponding sub-component of `a`.

In the type of `splitArray` above, we did not show the alignments on the references for simplicity. If we take the alignments into consideration, then we get the following type:

```
splitArray :: (Index x) ⇒ ARef a (Array (x + y) t) →
                          ( ARef a                       (Array x t),
                            ARef (GCD a (x * SizeOf t)) (Array y t) )
```

The address of the left sub-array is the same as the address of the original array so it inherits the same alignment. We compute the alignment of the right sub-array using the method that we described in Section 10.1 (take the `GCD` of the alignment of the whole structure and the offset within the structure where the sub-component starts).

Another useful operation is to convert a flat array into an array of arrays. We can do this with the function `toMatrix`:

```
toMatrix :: ARef a (Array (x * y) t) → ARef a (Array x (Array y t))
divIx    :: (Index y) ⇒ Ix (x * y) → (Ix x, Ix y)
```

This function splits an array with `(x * y)` elements into an array with `x` elements, each of which is an array with `y` elements (this operation does not change the address of the array, so the alignments of the references are the same). The corresponding operation on indexes is `divIx` and it divides its argument by `y`. The resulting pair contains the quotient and the reminder of the division. This is what we need to convert an index of the original array into a pair of indexes for the new view. For example, if we turn an array with 256 elements into 64 arrays of 4 elements each, then index `11` of the original array would correspond to the pair `(2,3)`. We can formalize the relation between `toMatrix` and `divIx` with the following equation:

```
a @ x = let (i,j) = divIx x
        in toMatrix a @ i @ j
```

In this equation, `a` is a reference to an array and `x` is an index of `a`. The equation asserts that indexing with the converted index into the converted array, is the same as indexing with original index in the original array.

We can also perform a casting operation that is the opposite of `toMatrix`. It turns an array of arrays into a single flat array, and its name is `fromMatrix`:

```
fromMatrix :: ARef a (Array x (Array y t)) → ARef a (Array (x * y) t)
mulIx      :: Index (x * y) ⇒ Ix x → Ix y → Ix (x * y)
```

The corresponding operation on indexes converts a pair of indexes identifying an element of the original array, into a single index of the flat array (notice that we have curried the type of `mulIx`). For example, if we convert an array that has 4 elements, each of which is an array with 16 elements into a single array of 64 elements, then the pair of indexes `(2,1)` in the original array would correspond to the index `17` of the flattened array. We can formalize the relation between `fromMatrix` and `mulIx` with the equation:

```
a @ i @ j = fromMatrix a (mulIx i j)
```

In this equation, `a` is a 2-dimensional array, `i` is an index in the first dimension, and `j` is an index in the second dimension. The equation asserts that indexing with `i` and `j` in the original array, is the same as indexing with `mulIx i j` in the flattened array.

Based on the duality between `fromMatrix` and `toMatrix` we might expect an operation `joinArray` that performs the opposite transformation to `splitArray`. Such an operation would have the type (in its curried form):

```
joinArray :: Ref (Array x t) → Ref (Array y t) → Ref (Array (x+y) t)
```

To define this operation as a casting function we would need the arguments to be references to two *adjacent* arrays. However, nothing in this type enforces this requirement, so we cannot define a casting function of this type. This observation emphasizes the point that the function `splitArray` looses some information, namely that the resulting references point to adjacent pieces of memory. For this reason, programmers should be careful when using `splitArray`.

The function `splitArray` looses information because its result uses ordinary pairs, which are not aware of explicit memory issues. We can define an alternative version of `splitArray` that does not loose information (and hence, it has an inverse) by using a pairing *structure*:

```
struct Pair a b where
  fst :: a
  snd :: b
```

This structure describes a memory area that has two adjacent components, one of type `a` and the other of type `b`. Using `Pair`, we can define a new version of `splitArray` (we shall refer to the other version as `oldSplitArray`) like this:

```
splitArray :: ARef a (Array (x+y) t)
           → ARef a (Pair (Array x t) (Array y t))

subIx :: (Index x) ⇒ Ix (x + y) → Either (Ix x) (Ix y)
```

This new version of `splitArray` works in a similar fashion to the old one, except that the result type captures the fact that when we split an array we get two adjacent arrays. The function `subIx` remains unchanged but we need to modify the equation that relates `splitArray` to `subIx`:

```
a @ i = let b = splitArray a
        in case subIx i of
                  Left j  → b.fst @ j
                  Right j → b.snd @ j
```

The equation asserts the same fact as before but, now we have to use the projection functions `fst` and `snd` to obtain references to the two parts of an array. Notice that this version of `splitArray` is more general than the old one because we can use it to recover the old behavior (but not vice versa):

```
oldSplitArray a = let b = splitArray a
                  in (b.fst, b.snd)
```

We can also define the dual operation of `splitArray`, which we shall call `joinArray`:

```
-- casting operator
joinArray :: ARef a (Pair (Array x t) (Array y t))
          → ARef a (Array (x+y) t)

-- reindexing functions
leftIx  :: (Index (x+y), Index y) ⇒ Ix x → Ix (x + y)
rightIx :: (Index (x+y), Index x) ⇒ Ix y → Ix (x + y)
```

```
-- equations
a.fst @ i = joinArray a @ leftIx i
a.snd @ j = joinArray a @ rightIx j
```

In the equations, `a` is a reference to a `Pair` of arrays, while `i` and `j` are indexes into the first and second array respectively. The two equations jointly state that we get the same results when we index into the two parts of the array separately, or when we join the two arrays into a single array and then convert the indexes with either `leftIx` or `rightIx`.

Note that we can define `leftIx` and `rightIx` in terms of a more general operation for adding indexes that has the following type:

```
addIx    :: (Index (a + b)) ⇒ Ix a → Ix b → Ix (a + b)

leftIx   :: (Index (a+b), Index b)  ⇒ Ix a → Ix (a + b)
leftIx a  = addIx a minIx

leftIx   :: (Index (a+b), Index a)  ⇒ Ix b → Ix (a + b)
rightIx b = addIx maxIx b
```

10.4 Summary

In this chapter, we described our design for working with structured memory areas, such as arrays and structures. We presented two groups of operations to manipulate such areas.

The first group of operations computes references to sub-areas from references to a structured area. These operations are automatically generated from the types describing the areas, which ensures that we never get invalid references. When we compute references to sub-areas we also compute alignment constraints for the sub-areas. To ensure that array indexes are within bounds we use a family of numeric range types, called `Ix`, which are equipped with operations that preserve their invariants.

The second group of operations allows us to change the 'view' on memory areas, for example to place a structured representation on an unstructured array of bytes. To ensure that we do not compromise the safety of the design, we allow such conversions only for types that can be represented with arbitrary bit-vectors, which is formalized with the class `Bytes`.

The following is a summary of the operations that we discussed in this chapter:

```
-- User-defined areas
struct Pair r s where
  fst :: r
  snd :: s

-- (generated)
(.fst) :: ARef a (Pair r s) → ARef a r
(.snd) :: ARef a (Pair r s) → ARef (GCD a (SizeOf r)) s

class Index n where
  (@)     :: ARef a (Array n t) → Ix n
          → ARef (GCD a (SizeOf t)) t

  -- Conversions
  toIx    :: Int → Maybe (Ix n)
  fromIx  :: Ix n → Nat
  bitIx   :: (Width m, 2^m = n) ⇒ Bit m → Ix n

  minIx   :: Ix n
  maxIx   :: Ix n

  -- Dynamically checked index arithmetic
  inc     :: Int → Ix n → Maybe (Ix n)
  dec     :: Int → Ix n → Maybe (Ix n)

  -- Statically checked index arithmetic
  addIx     :: (a + b = n) ⇒ Ix a → Ix b → Ix n
  subIx     :: (a + b = n, Index a) -- subtract a
            ⇒ Ix n → Either (Ix a) (Ix b)

  mulIx     :: (a * b = n) ⇒ Ix a → Ix b → Ix n
  divIx     :: (a * b = n, Index b) -- divide by b
            ⇒ Ix n → (Ix a, Ix b)

-- Views on byte arrays
type BytesFor t = Array (SizeOf t) (Stored (Bit 8))

class Bytes t where
  fromBytes :: ARef a (BytesFor t) → ARef a t
  toBytes   :: ARef a t → ARef a (BytesFor t)
```

```
-- Views on arrays
splitArray  :: ARef a (Array (x+y) t)
            → ARef a (Pair (Array x t) (Array y t))

joinArray   :: ARef a (Pair (Array x t) (Array y t))
            → ARef a (Array (x+y) t)

toMatrix    :: ARef a (Array (x * y) t)
            → ARef a (Array x (Array y t))

fromMatrix  :: ARef a (Array x (Array y t))
            → ARef a (Array (x * y) t)
```

Part III

Examples and Implementation

Chapter 11

Example: Fragments of a Kernel

In this chapter, we show how our design might be used in practice. This chapter does not introduce any new concepts; instead, it contains a lot of examples. The examples have a common theme: they are different components that may be useful in the implementation of an OS kernel, which explains the name of the chapter. In Section 11.1 we show how to implement a simple driver for the text console of the PC architecture. In Section 11.2 we present a brief overview of the IA-32 architecture. In Sections 11.3 and 11.4 we show how to initialize the segmentation and interrupt hardware when the kernel starts up. In Section 11.5 we show how we might implement the code the allows the kernel to switch between user and kernel mode execution. In Section 11.6 we show some data structures that would be useful in the implementation of the memory manager of a kernel.

11.1 A Text Console Driver

One of the examples that we discussed in Chapter 9 was a simple text console driver. In this section, we use the tools that we described to implement such a driver. On the PC, the text console is a memory-mapped device, which means that we manipulate it by writing data, in a particular format, to a designated part of memory. For the text console, each location in its memory area corresponds to a location on the screen. Depending on where exactly we write, we either affect what is displayed on the screen, or how it is displayed.

Screen Characters A screen character is a 16 bit quantity that specifies (i) which character to display, and (ii) how to display the character on the screen. We can use **bitdata** declarations to describe the exact format of screen characters:

```
bitdata SChar = SChar { attr = default_attr :: Attr, char :: Bit 8 }

bitdata Attr
  = Attr { flash  :: Bool,   -- flashing?
           bg     :: Color, -- background color
           bright :: Bool,  -- foreground bright?
           fg     :: Color  -- foreground color
         }

default_attr :: Attr
default_attr
  = Attr { fg = White, bg = Black, flash = False, bright = False }

bitdata Color
  = Black as 0 | Blue as 1 | Green as 2  | Cyan as 3
  | Red as 4 | Magenta as 5 | Yellow as 6 | White as 7 :: Bit 3
```

The type `Attr` describes the appearance of the character, while the actual character is represented as an 8-bit value. The definition of `SChar` specifies that if we do not provide a value for the field `attr`, then the compiler should use the value `default_attr`.

The Memory Region The text console has 25 rows of 80 characters each. Each row is an array of screen characters, the first element corresponds to the left-most character on the screen. The screen is made up of 25 consecutive rows, the first one being the row that is displayed at the top of the screen:

```
type Rows     = 25
type Cols     = 80

type Row      = Array Cols (LE SChar)
type Screen   = Array Rows Row

area screen in VideoRam :: Ref Screen
```

State of the Driver Our driver uses a region of memory to store the corrdinates of the current position on the screen, as well as the attributes that we should use when we display characters:

```
area cur_col  :: Ref (Stored (Ix Cols))
area cur_row  :: Ref (Stored (Ix Rows))
area cur_attr :: Ref (Stored Attr)
```

For the purposes of this example, we have chosen to store the driver state using memory areas. While the format of the screen is defined by the PC architecture, the driver state is not, so we could represent it in other ways as well. For example, it would also have been possible to store the state of the driver in the heap, instead of in memory areas. There are also other ways in which we could have used memory areas. For example, we could have grouped the state into a structure, which would guarantee that the state for the console resides in adjacent locations. This would be useful if we were to implement support for virtual consoles, because then we could use an array of driver states, one for each console.

Cursor movement. We may now introduce some simple operations for moving the cursor. These operations simply manipulate the driver state and have no effect on the hardware. A more sophisticated driver could, of course, choose to indicate the current position of the cursor by, for example, inverting the attributes for the character where the cursor is located, or use the hardware cursor.

```
-- Move cursor to a given location
goto :: Ix Cols → Ix Rows → IO ()
goto x y = do writeRef cur_col x
              writeRef cur_row y

-- Move cursor to the 'beginning' of the screen
goHome :: IO ()
goHome  = goto 0 0
```

Notice that the types of the arguments to `goto` guarantee that we have a valid position, so we do not need to perform any checks. Of course, we cannot entirely eliminate checks: when we display text, if we reach the end of a line, then we would like to continue displaying the text on the next line.

```
-- Move cursor to the position of the next character
goNext :: IO ()
goNext = do col ← readRef cur_col
            case col of
               next - 1 → writeRef cur_col next
               last     → goNextLine

-- Go to the beginning of the next line
goNextLine :: IO ()
goNextLine = do goDown
                writeRef cur_col minVal
```

The monadic action `goNextLine` moves the cursor to the beginning of the following line. This, of course, may also fail if we happened to be already at the last line of the screen. In this case, we have to scroll the screen, and clear the last line:

```
-- Go to the next row
goDown :: IO ()
goDown = do row ← readRef cur_row
            case row of
              next - 1 → writeRef cur_row next
              last     → scroll

-- Scroll the screen
scroll :: IO ()
scroll = loop 0 1
  where
  loop prev cur = do memCopy (screen @ cur) (screen @ prev)
                     case cur of
                       next - 1  → loop cur next
                       last      → clearRow last

-- Clear a row
clearRow :: Ix Rows → IO ()
clearRow row = do a ← readRef cur_attr
                  let blank = SChar { char = 32, attr = a }
                  forEachIx (λ col →
                    writeRef (screen @ row @ col) blank)
```

Driver Operations Finally, we can define the main functionality of the driver. We provide functions to display characters, manipulate the display attributes, and clear the screen:

```
-- Clear the screen
cls :: IO ()
cls = do forEachIx clearRow
         goHome

-- Manipulate attributes
setArrr :: Attr → IO ()
setAttr a = writeRef cur_attr a

getAttr :: IO Attr
getAttr = readRef cur_attr

-- Display a character, move to the next position
putChar :: Bit 8 → IO ()
putChar c = do a ← readRef cur_attr
               col ← readRef cur_col
               row ← readRef cur_row
               writeRef (screen @ row @ col)
                        (SChar { char = c, attr = a })
               goNext

-- Display a list of characters
putStr :: [Bit 8] → IO ()
putStr xs = mapM_ putChar xs

-- Display a list of characters, go to the next line
putStrLn :: [Bit 8] → IO ()
putStrLn xs = do putStr xs
                 goNextLine
```

Our definition of `cls` clears the screen one row at a time, and every row is cleared one screen character at a time. We could get a more efficient implementation if we were to clear two characters at the same time. We could do this, by viewing the `screen` memory region as an array of `Stored (Bit 32)`, rather then the more structured view that we use in the rest of the driver. Here is an example of how we could do this:

```
cls' :: IO ()
cls'  = do a ← readRef cur_attr
           let blank = toBits (SChar { char = 32, attr = a })
               blank2 = blank # blank
               arr :: Ref (Array (25 * 40) (LE (Bit 32)))
               arr = fromBytes (toBytes videoRAM)
           forEachIx (λ i → writeRef (arr @ i) blank2)
```

It is interesting to note that the type checker does quite a lot in this declaration. For example, it checks that the size of the memory for the screen is a multiple of 4, as otherwise, we would not be able to clear the entire screen using 32-bit words. Still, we need the type annotation on the array `arr` (or alternatively an annotation on the reference in the `writeRef` operation) because it specifies the encoding that we want to use when we write to memory.

11.2 Overview of IA-32

In this section, we present some of the details of the IA-32 architecture, when executing in protected mode. Our information is based on the Software Developer's Manual [45] provided by the Intel corporation.

The hardware provides a number of fairly advanced features, including paging, segmentation, multitasking, and four different protection levels at which code may execute. Depending on its particular design, a kernel may utilize these features to a greater or lesser extent but, at the very least, a kernel should be able to initialize the hardware correctly.

In protected mode, the processor works with *virtual* (sometimes called *logical*) addresses that are used to address a 4GB *linear* space. The paging hardware translates virtual addresses to concrete *physical* addresses that are then used to retrieve or store information. The abstraction provided by the paging hardware is often used to support multiple processes in an OS: by manipulating the translation between virtual and physical address, kernels can control how different processes interact through memory; similar mechanisms are also used to support a collection of processes whose total demand for memory exceeds the available physical memory—the memory for inactive processes can be backed up onto secondary storage to provide more working space for active processes.

The segmentation hardware of the IA-32 is used to split the linear space into a number of *segments*. A segment is a contiguous region in linear space that has some properties associated with it. Many instructions are executed with respect to a particular segment, so memory references are interpreted as offsets in the relevant segment. Thus, to obtain a linear address, the hardware adds the base of the relevant segment and then checks that the resulting linear address is within the segment boundaries (the hardware also performs other checks). Segmentation is not heavily used by modern kernels, but there is no way to turn it off on the IA32. To work around this, kernels often setup the machine once and forall at boot time with segments that span the entire linear space.

Multitasking is another feature of the IA-32 that does not appear to be heavily used by current kernels, perhaps because it tends to be fairly heavyweight. Still, the IA32 architecture requires at least one task-state segment, which is used while switching between the execution of kernel and user code. Task-state segments are used to save the state of a task, and are a part of the hardware implementation of multitasking.

11.3 Segments

The IA-32 has 6 segment registers that specify the code segment (CS), stack segment (SS), and four data segments (DS, ES, FS, GS). Instructions that affect the instruction pointer, such as jumps, are executed with respect to the code segment. As the name suggests, the stack segment is used for instructions that manipulate the stack, and the data segments are for different instructions that manipulate memory. In this section, we describe a simple way to initialize the segmentation hardware. As before, the details follow the specifications in the Intel documentation.

11.3.1 Segment Selectors

The segment registers are 16 bit registers, that contain *segment selectors*, which are used to identify the different segments. We use a bitdata declaration to define the exact format of segment selectors:

```
bitdata SegSel    = SegSel { segIx :: Bit 13
                           , table :: SegTable
                           , rpl   :: Bit 2 }
```

```
bitdata SegTable  = GDT as B0 | LDT as B1
```

The field `segIx` is an index into a table of *segment descriptors*, each of which describes a particular segment. The hardware supports indexing into two different tables containing segment descriptors: the local descriptor table (LDT), and the global descriptor table (GDT). The field `table` indicates which table should be consulted to find the appropriate segment descriptor. The last field, `rpl`, is the *requestor's privilege level*, which is used to restrict access to segments to particular privilege levels.

Because the index is a 13 bit quantity, the table can contain up to 8192 entries. A typical OS kernel that does not make use of segmentation is likely to use only five: a code and data segments for kernel and user mode respectively, and a special segment that contains the state of the current (and perhaps only) task:

```
seg_kernel_code = SegSel { segIx = 1, table = GDT, rpl = 0 }
seg_kernel_data = SegSel { segIx = 2, table = GDT, rpl = 0 }
seg_user_code   = SegSel { segIx = 3, table = GDT, rpl = 3 }
seg_user_data   = SegSel { segIx = 4, table = GDT, rpl = 3 }
seg_tss         = SegSel { segIx = 5, table = GDT, rpl = 3 }
```

11.3.2 Segment Descriptors

For the proper operation of the kernel, we need to provide at least a global descriptor table, which contains the descriptions of the various segments. A descriptor table is simply an array of segment descriptors:

```
type DescTab n = Array n SegDesc
area gdt      :: ARef 8 (DescTab 6)
```

The table contains 6 entries rather then the 5 we need, because the Intel specification states that the first entry in the table should contain a null segment descriptor.

Segment descriptors are two words each, and have a rather peculiar format, as illustrated by the following bitdata declarations. (Not all of the type annotations in the declarations are necessary, but the redundancy makes the code more readable and helps to prevent layout mistakes)

```
struct SegDesc | size 8 where
  seg0 :: Stored Seg0
```

```
  seg1 :: Stored Seg1

bitdata Seg0  = Seg0  { baseL :: Bit 16, limitL  :: Bit 16 }
bitdata Seg1  = Seg1
   { baseH        :: Bit 8
   , gran         :: Gran
   , userBit = 0  :: Bit 1
   , limitH       :: Bit 4
   , present      :: Bool
   , dpl          :: Bit 2
   , segType      :: SegType
   , baseM        :: Bit 8
   } as (baseH                          :: Bit 8)
      # (gran # B10 # userBit # limitH  :: Bit 8)
      # (present # dpl # segType        :: Bit 8)
      # (baseM                          :: Bit 8)

bitdata Gran  = InBytes as B0 | InPages as B1
```

The segment selector contains the starting address of the segment (scattered among the fields `baseL`, `baseM`, `baseH`), and the size of the segment, which is a 20 bit field (contained in `limL` and `limH`). The segment size can be interpreted in two different ways, depending on the field `gran`. If `gran` is set to `InBytes`, then the limit is in bytes. Otherwise, if `gran` is `InPages` then the segment limit is scaled by 4K (the size of a page). The field `userBit` is available for use by systems programmers, and the other fields specify other properties of the segment: `dpl` is the privilege level, and `segType` is the type of segment that we are describing.

There are three types of segment that are of interest, as illustrated by the `SegType` bitdata type:

```
bitdata SegType   = DataSeg { expDown  = False :: Bool
                             , writable = True  :: Bool
                             , accessed = True  :: Bool }
                        as B10 # expDown # writable # accessed

                  | CodeSeg { conform  = False :: Bool
                             , readable = True  :: Bool
                             , accessed = True  :: Bool }
                        as B11 # conform # readable # accessed
```

```
                | TaskSeg { busy = False :: Bool }
                    as B010 # busy # B1
```

Note that this declaration uses defaults to specify common settings for the different segment types.

Having defined the formats of all the data involved, we can write a function that initializes an entry in the `gdt` table:

```
gdtEntry :: SegSel → SegType → Bit 32 → Bit 20 → Gran → IO ()
gdtEntry (SegSel sel) ty (baseH # baseM # baseL) (limH # limL) gran
  = do let slot = gdt @ toIx (0 # sel.segIx)
       writeRef slot.seg0 (Seg0 { baseL = baseL, limitL = limL })
       writeRef slot.seg1 (Seg1 { baseH   = baseH
                                , gran    = gran
                                , limitH  = limH
                                , present = True
                                , dpl     = sel.rpl
                                , segType = ty
                                , baseM   = baseM })
```

Perhaps the most interesting aspect of the function is the use of type information to split the base and limit arguments automatically into the correct subfields.

All that remains to be done is to use `gdtEntry` to initialize the global descriptor table:

```
initGDT :: IO ()
initGDT = do gdtEntry seg_kernel_code (CodeSeg {}) minIx maxIx InPages
             gdtEntry seg_kernel_data (DataSeg {}) minIx maxIx InPages
             gdtEntry seg_user_code   (CodeSeg {}) minIx maxIx InPages
             gdtEntry seg_user_data   (DataSeg {}) minIx maxIx InPages
             gdtEntry seg_tss         (TaskSeg {})
                      (toBits tss) (lower (sizeOf tss - 1)) InBytes
```

We have initialized the segments to be trivial: they all span the entire linear space (although other options are certainly possible). The only exception is `seg_tss`, which contains the state of a task. Next we describe the structure of the task state segment.

11.3.3 Task-State Segment

The task-state segment,TSS, contains the state of a hardware task. We need to have at least one of these, and it is mostly used while switching between user and kernel mode execution. We can describe the format of the task segment, and declare a memory area for it like this:

```
struct TaskSegment | size 104 where
  Pad 4
  esp0    :: Stored (Bit 32)
  ss0     :: Stored SegSel; Pad 2
  ..
  ioMap   :: Stored (Bit 16)

area tss    :: ARef 128 TaskSegment
```

The task-state segment is at least 104 bytes long, and in general contains a number of fields. However, we are only planning to use a few of those, and so we only include the fields that we need (the `..` is formal notation). The fields that we need to specify are the stack segment and stack pointer, which change when we switch between user- and kernel-mode execution. The other field, `ioMap` can be used to configure access for IO ports in user mode.

The **area** declaration reserves space for the TSS. We request that the area is aligned on a 128 byte boundary: this is necessary to ensure that the TSS resides in a single page, which is a hardware requirement.

To initialize the TSS, we just specify the stack segment and an offset for the `ioMap`. We set the kernel stack just before we switch to user mode execution (to be discussed later):

```
initTSS :: IO ()
initTSS = do writeRef tss.ss0 seg_kernel_data
             writeRef tss.ioMap (lower (sizeOf tss))
```

Normally, the field `ioMap` should contain an offset in the segment containing configuration for the IO ports. In the initialization code, we provide an offset that is outside the segment, which means that access to the IO ports is disabled. The function `lower` converts the 32 bit size of TSS to a 16 bit quantity by extracting the lower 16 bits.

Having initialized all data structures appropriately, we need to tell the machine to use our data structures. This is done with a little bit of code

that resembles assembly. It loads the appropriate machine registers with the values we described.

```
do setGDT gdt
   setCS seg_kernel_code
   setDS seg_kernel_data
   setSS seg_kernel_data
   setES seg_kernel_data
   setFS seg_kernel_data
   setGS seg_kernel_data

   ltr seg_tss
```

Each of these statements is an IA-32 specific primitive that is expanded to the corresponding IA-32 instructions. For example, `setCS` has the following type:

```
setCS :: SegSel → IO ()
```

11.4 Interrupts and Exceptions

In addition to initializing the segmentation hardware, we also have to initialize the machine so that it can respond to interrupts and exceptions. The mechanism to do this is similar to what we used to initialize the segmentation hardware. Gate descriptors (similar to segment descriptors) specify what is to be done when a particular interrupt or exception occurs, and are stored in an interrupt descriptor table (IDT) that is similar to the global descriptor table.

```
type IDT n = Array n GateDesc

struct GateDesc where
  gate0 :: Stored Gate0
  gate1 :: Stored Gate1

bitdata Gate0 = Gate0 { segment :: SegSel, offsetL :: Bit 16 }

bitdata Gate1
  = Gate1 { offsetH        :: Bit 16,    -- ignored for task gates
            present        :: Bool,
```

```
            dpl            :: Bit 2,
            gateType       :: GateType,
          } as (offsetH # present # dpl # gateType # _) :: Bit 32

bitdata GateType
  = Interrupt32 as B01110000
  | Trap32      as B01111000
  | Task        as B00101 # _

area idt :: ARef 8 (IDT 256)
```

Despite the fact that the IA32 supports many different gates, in our small kernel we shall use only interrupt gates.

We can initialize a single entry in the `idt` with the following code:

```
idtEntry :: Bit 8 → Bit 32 → IO ()
idtEntry n (offH # offL)
  = do let slot = idt @ bitIx n
       writeRef slot.gate0 (Gate0 {
                              segment = seg_kernel_code,
                              offsetL = offL
                            })
       writeRef slot.gate1 (Gate1 {
                              offsetH = offH,
                              present = True,
                              dpl     = 0
                            })
```

The 32 bit integer is the address of an external procedure that is to be called when the particular exception number occurs.

11.5 User Mode Execution

The previous sections described fairly generic code to initialize the machine on start up. In this section, we describe how a kernel might implement support for executing multiple user mode processes. In general, this differs from one kernel to the next; what we present is a fairly simple design that might be suitable for a micro-kernel.

Once execution of user code starts, it continues until the occurrence of either a hardware interrupt or a software exception. At this point, control is

transferred to the kernel, which responds appropriately and then resumes a (possibly different) user process.

We use only two of the four privilege levels supported by the IA-32: the kernel executes at level 0 (the most privileged level), and user code executes at level 3 (the least privileged level). These were set up when we initialized the various segments of the machine.

When an interrupt or an exception occurs, the machine consults the IDT and starts execution of the appropriate handler (in the specified code segment: in our case the kernel's code segment). The details of exactly what happens are different depending on the privilege levels of the code that was executing when the interrupt occurred, and the privilege level of the handler. Presently, we are interested in the case when we were executing in user mode (i.e., at privilege level 3), and the interrupt handler operates in kernel mode (i.e., at level 0). In such situations, before executing the handler, the hardware uses the TSS to access a kernel 'stack' (stack segment and stack pointer); then it switches to this stack (by loading the appropriate registers); finally, it pushes a the user state on the new stack. The format of the state on the stack can be described with the following structure declaration:

```
struct IRet where
  eip    :: Stored (Bit 32)    -- user's code
  cs     :: Stored SegSel      -- user's code segment
          ; Pad 2
  eflags :: Stored (Bit 32)    -- state of the flags
  esp    :: Stored (Bit 32)    -- user's stack
  ss     :: Stored SegSel      -- user's stack segment
          ; Pad 2
```

In addition, for some interrupts/exceptions the hardware also pushes an error code. After this, execution proceeds at the location specified by the IDT.

The hardware only saves the code and stack of the user process. Typically, we also need to save the other registers of the machine so that we can restore them later before resuming the process. We can do this with the instruction `pusha`, which pushes the machine registers on the stack in the following format:

```
struct PushA where
  edi :: Stored (Bit 32)
  esi :: Stored (Bit 32)
  ebp :: Stored (Bit 32)
```

```
      ; Stored (Bit 32)  -- esp, unused
  ebx :: Stored (Bit 32)
  edx :: Stored (Bit 32)
  ecx :: Stored (Bit 32)
  eax :: Stored (Bit 32)
```

The instruction `pusha` saves all general purpose registers, including `esp`, but we do not provide a name for the corresponding field of `PushA`. The reason for this is that the value in the `esp` register does not contain the user's stack pointer. Recall that, when the hardware switches from user to kernel execution, it updates the stack pointer using the data stored in the task-state segment. For this reason, the user's stack pointer is part of the `IRet` structure, and not the `PushA` structure.

Besides saving the general purpose registers, we also save the user's data segment registers. The user's stack and code segments are automatically saved by the hardware in the `IRet` structure so they do not need to included again here:

```
struct SegRegs where
  gs :: Stored SegSel; Pad 2
  fs :: Stored SegSel; Pad 2
  es :: Stored SegSel; Pad 2
  ds :: Stored SegSel; Pad 2
```

Finally, we also save the pointer to the user's page directory, which completes the definition of the user's context—the information that we need to resume a suspended process:

```
struct Context where
  page_dir   :: Stored (Bit 32)
  seg_regs   :: SegRegs
  regs       :: PushA
  error_code :: Stored (Bit 32)
  resume     :: IRet
```

Having decided on the format of a user's context we can write a little bit of assembly code that starts a user process, given a pointer to a `Context`:

```
# Expects a pointer to a 'Context' in %eax
asmCallUser:
        movl    %esp, stack_top   # save RTS state
```

```
        movl    %ebp, heap_top    # ...
        movl    %eax, %esp        # context becomes the stack

        popl    %eax              # set page directory
        movl    %cr3, %ebx        #
        cmpl    %eax, %ebx        # update cr3 only if
        je      1f                # the pdir is different
        movl    %eax, %cr3        #

1:      popl    %gs               # restore segment registers
        popl    %fs
        popl    %es
        popl    %ds

        popa                      # restore other registers
        addl    $4, %esp          # remove error code

        iret                      # we are off to the user
```

First we save the state of the run-time system in some memory that belongs to the kernel. The run-time system of our compiler is very simple so all we need to do is save the heap and stack pointers. In general, we might also save some of the general purpose registers. Having done that, we switch the 'stack' to the user's context and pop-off the new machine state (general purpose and segment registers). Finally, we use the instruction `iret` (mnemonic: return from interrupt) which pops off the remaining state off the stack, switches to the user's stack, and resumes execution of the user code.

When the execution of user code is interrupted, the hardware switches to kernel mode and start executing the relevant interrupt handler. The interrupt handlers in our example kernel all perform essentially the same task: they save the user state on the stack, then restore the state of the RTS, and finally they return to the kernel indicating what happened. The assembly code that performs these tasks is almost a mirror image of the previous piece of code, so we'll omit it here.

11.6 Paging

We mentioned briefly that the IA-32 translates between virtual and physical addresses with the aid of some paging hardware. Now we shall examine this

hardware in more detail. While looking at the code in this section it may be useful to consult Figure 11.1, which illustrates the datatypes used by the machine. The main point of control for the paging hardware is the *page*

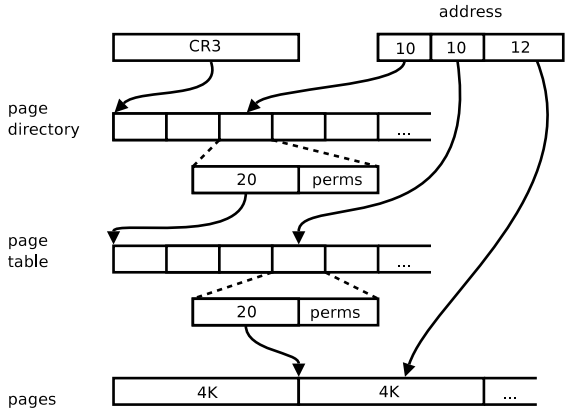


Figure 11.1: The Paging Hardware of IA-32

directory, which is a table that contains 1024 page directory entries:

```
type PDir = Array 1024 (Stored PageDirectoryEntry)
```

When paging is enabled, there is always exactly one active page directory, and its location in physical memory is stored in an internal hardware register (called CR3). To translate a virtual address, the hardware uses the 10 most significant bits of the address as an index in the page directory to access a page directory entry, which specifies how to translate the virtual address. The format of page directory entries can be defined with the following bitdata declaration:

```
bitdata PageDirectoryEntry

  = NoEntry
  { userBits      :: Bit 31
  } as userBits # B0

  | PageTable
  { pageTable                    :: Bit 20
  , userBits      = 0            :: Bit 3
```

```
  , global        = False         :: Bool
  , status        = PStatus {}    :: PStatus
  } as pageTable # userBits # global # B00 # status # B1

  | SuperPage
  { physAddr                      :: Bit 10
  , pat           = False         :: Bool
  , userBits      = 0             :: Bit 3
  , global        = False         :: Bool
  , dirty         = False         :: Bool
  , status        = PStatus {}    :: PStatus
  } as physAddr # _ # pat # userBits # global # B1 # dirty # status # B1

bitdata PStatus
  = PStatus
  { accessed      = False :: Bool
  , noCache       = False :: Bool
  , writeThrough  = False :: Bool
  , user          = False :: Bool
  , writeable     = True  :: Bool
  }
```

There are three kinds of data that may appear as an entry in a page directory. If the entry corresponding to a virtual address is of the form `NoPage`, this means that the virtual addresses in that range are not mapped anywhere in physical memory. An attempt to use such an address results in a special exception called a page fault. Notice that such 'blank' entries are recognized by a 0 in the least significant bit of the entry. The remaining bits are available for use by the systems programmer. For example, a particular memory manager may choose to temporarily backup the data in a part of physical memory onto some secondary storage (e.g., a hard-disk). In such a situation, the 'userBits' could be used to identify the new location of the data, so that it can be retrieved later.

Page directory entries of the form `SuperPage` refer to 4MB chunks of physical memory (often called *super pages*). The field `physAddr` contains the 10 most significant bits of the physical address of the super page that contains the data for the virtual address. The field is only 10 bits wide, because super pages are always aligned on 4MB boundaries, and so their 22 lowest bits are all zero. The 22 least significant bits of the virtual address are then used

as an offset within the super page. To make things more concrete, consider a virtual address `(10 :: Bit 10) # (17 :: Bit 22)`. Now, if the 11^{th} entry (index 10) of the page directory is a `SuperPage`, then the hardware will use it to get the address of the relevant physical super page and then add 17 to it to get the location in memory that contains the data for our virtual address.

Super pages provide a fairly coarse granularity for manipulating memory. The hardware also supports paging using ordinary 4KB pages. This happens when the entry in a page directory is of the form `PageTable`. Then the field `pageTable` contains the physical location of a *page table*. Page tables are similar to page directories, and have the following format:

```
type PTab         = Array 1024 (Stored PageTableEntry)

bitdata PageTableEntry
  = NoPage
  { userBits                      :: Bit 31
  } as userBits # B0

  | Page
  { physAddr                      :: Bit 20
  , userBits      = 0             :: Bit 3
  , global        = False         :: Bool
  , pat           = False         :: Bool
  , dirty         = False         :: Bool
  , status        = PStatus     :: PStatus
  } as physAddr # userBits # global # pat # dirty # status # B1
```

If the entry in the page directory is a `PageTable`, then the next 10 bits of the virtual address are used as an index in the appropriate page table. The page table entries are similar to page directory entries: `NoPage` indicates that the corresponding virtual address is not mapped anywhere; if the entry is of the form `Page`, then the field `physAddr` is the physical address of 4KB of physical memory that contain the data for the virtual address.

To summarize, depending on the entry in the page directory, the hardware views a virtual address in one of two ways:

```
bitdata VirtAddr = SuperP { pdirIx :: Bit 10
                          , offset :: Bit 22 }
                 | PTab   { pdirIx :: Bit 10
                          , ptabIx :: Bit 10
                          , offset :: Bit 12 }
```

11.7 Summary

In this chapter, we presented some examples that illustrate how our language design might be used in practice. We presented three examples, each of which might be useful in the implementation of an OS kernel. The first example showed a simple driver for a memory mapped device, the text console of the PC architecture. The second example showed how to write some fairly low-level code that is useful when we initialize the machine at boot time. The third example showed how to program some data structures that would be useful in the implementation of a memory manager for an OS kernel. We managed to program the examples in a direct style, without many artificial encodings. This gives us some confidence that our design may scale to the needs of other systems software, as well.

Chapter 12

Implementation

12.1 Introduction

We implemented the ideas described in this dissertation in a prototype compiler. Developing the implementation was an important part of the design process because it allowed us to try out different ideas and see what worked and what did not. Furthermore, having an implementation is a first step towards showing the feasibility of the language design. In this chapter, we briefly describe the more interesting aspects of the back-end of our compiler. While we have not conducted any formal measurements, the method that we used suggests that our language design can be implemented efficiently on modern machines.

The source language that we implemented is a strict version of Haskell, extended with support for bitdata and memory areas. More precisely, the front-end (syntax, type system) is similar to that of Haskell, while the back-end is similar to that of ML. We compile this language into IA32 assembly code, and we have a small run-time system (described in Section 12.4) that enables us to run our programs under the Linux operating system. We also have a modification or the run-time system that enables us to run our program on a 'bare' machine (i.e., a machine without a host operating system).

The front-end of our implementations is similar to the front-ends of Haskell implementations, extended with support for the new declarations **bitdata**, **struct**, and **area**. It checks that user-specified types have valid kinds, and then it type checks the program. We eliminate the functional predicate notation (Chapter 5) during kind checking, although, in principle, we could

do that earlier. During type checking, we make type abstractions and applications explicit, as well as evidence applications and abstractions (recall the explicit calculus from Chapter 3). We use different types of evidence for the different predicates, for example, the evidence for the `BitData` class is an integer that indicates the number of bits that are needed to represent a value. In this respect, our implementation differs from most other Haskell implementations, which use 'dictionaries' as evidence for all predicates.

The back-end of our compiler follows the ideas of Tolmach and Oliva [91] for compiling a functional language to C, although, at the very end, we generate assembly instead of C. We start by performing a whole program analysis to make programs completely monomorphic by generating specialized instances of polymorphic functions (a technique also used by the MLton [100] optimizing compiler for Standard ML). This approach has the benefit that we can choose more efficient representations for types (e.g., we do not need to change the representations of values when we call polymorphic functions) and also, we can compute the evidence for overloaded functions at compile-time. The drawback of this technique is that it does not work for programs that use polymorphic recursion because such programs may need an unbounded number of instantiations of a polymorphic function. Our implementation rejects such programs, but it is also possible to use a hybrid approach that specializes most functions, but still uses a uniform representation for functions that use polymorphic recursion.

After monomorphization we perform defunctionalization, which is an algorithm that makes the manipulation of computation values explicit, thus turning higher-order into first-order programs. Finally, we generate assembly code.

This chapter is structured as follows: in Section 12.2 we discuss how to implement computation values; in Section 12.3 we discuss various alternative representations for bitdata types; and in Section 12.4 we describe some aspects of code generation and the run-time system for our compiler.

12.2 Computation Values

Throughout this work, we use a higher-order functional language with monadic computations. This means that both functions and monadic computations are first-class values, and so they can be passed as arguments to functions, returned as results, or stored in data structures. To implement such values,

we use *defunctionalization* [82, 9, 91], which is a method for transforming higher-order into first-order code. In the resulting programs, we explicitly distinguish between data (i.e., values) and code (i.e., computations). In addition, our implementation also preserves the distinction between pure and effectful code that is already present in the source program because this information is useful for analyzing and optimizing programs.

Values and Computations. The main operation on a function of type `A → B` is to apply it to a value of type `A` to compute a result of type `B`. The main operation for a monadic value of type `IO A` is to execute it, performing the necessary effects, to get a result of type `A`. We can describe these operations as first-order functions:

```
app :: (A → B, A) ⟶ B
run :: IO A ⟶I A
```

Notice that in the types we have two different arrows: the short arrow is for the type of *values* that represent functions, while the long arrow is for *computations*. We also annotate the types of computations that perform effects with a label describing the effects [87]. It is interesting to note the difference between the types `A` $\longrightarrow_I$ `B`, and `A` $\longrightarrow$ `IO B`. The former describes a computation that, given an `A` value, will produce a `B` value and, in the process may perform the effect `I`. The latter type describes computations that, given a value of type `A`, produce a value of type `IO B`, and do not perform any effects in the process (later, we may use `run` on the resulting value to perform the effects that it encapsulates).

Implementation of Computation Values. The implementation of a computation value consists of three parts: (i) a concrete representation for the value that is used, for example, when we store the value in a data structure; (ii) a first-order function that describes the behavior of the value when we execute it; (iii) a definition for `app` or `run` that relates the representation of a value to its behavior. For example, consider the monadic value `getChar :: IO Char`. When executed, this computations reads a character from the standard input and returns it as its result. To represent values of type `IO Char`, we use an algebraic datatype which has one constructor for each distinct value of that type:

```
data IO Char = GetChar
             | ... constructors for other values ...
```

Note that in this declaration, `IO Char` is the name of an atomic type which happens to contain a space, and not a type constructor applied to an argument. Later in this section we shall come back to this point in the context of generalized algebraic datatypes.

The second component of the implementation of `getChar` is an effectful piece of code, `primGetChar`, that reads a character from standard input. Finally, we need an equation in the definition of `run` that relates the constructor `GetChar` to the computation `primGetChar`:

```
-- (i) value
getChar          :: IO Char
getChar           = GetChar

-- (ii) behavior
primGetChar      :: () ⟶I Char
primGetChar       = ... read from stdin ...

-- (iii) Value related to behavior
run              :: IO Char ⟶I Char
run GetChar       = primGetChar ()
run ...           = ...
```

Representing computation values as the constructors of a datatype is quite convenient because it allows us to reuse the tools that we have for working with datatypes in this new setting. Of course, other representations are possible [78], but we aways have the same three basic components.

Free Variables. Because we work in a language with static scoping, computation values may have to store parts of the context in which they were created so that these values can be used when the computation is executed. Representing computation values with algebraic datatypes supports this quite naturally because we may store the context in the fields of the constructor representing the value. Consider, for example, the function `putChar :: Char → IO ()` which is used to write a character to standard output. This function is more complex than `getChar` because it involves two computation values. The first is the function `putChar`, and the the second is the monadic computation that is returned when we apply `putChar` to an argument. Because these values have different types, they belong to different representation types:

```
data Char → IO ()  = PutChar
                   | ...

data IO ()         = PutChar_1 Char
                   | ...
```

The constructor `PutChar_1` represents the value that we get when we apply `putChar` to one argument. Then `putChar` has the following implementation:

```
-- (i) value
putChar          :: Char → IO ()
putChar           = PutChar

-- (ii) behavior of putChar
funPutChar       :: Char ⟶ IO ()
funPutChar x      = PutChar_1 x     -- stores free variable

-- (ii) behavior of the result of putChar
primPutChar      :: Char ⟶I ()
primPutChar x     = ... write character to stdout ...

-- (iii) Relate values to behaviors
app              :: (Chat → IO (), Char) ⟶ IO ()
app (PutChar,x)   = funPutChar x
app ...           = ...

run              :: IO () ⟶I ()
run (PutChar_1 x) = primPutChar x
run ...           = ...
```

Other IO Primitives. We have already seen the definitions for two `IO` primitives, `getChar` and `putChar`. Other basic operations of the monad are added in a similar fashion. The result is that the different `IO` values correspond to the abstract syntax of an effectful language, and the `run` function defines an interpreter for this language. Sequencing of operations is done with the monadic operation `bindIO`, which is defined like this:

```
bindIO             :: IO a → (a → IO b) → IO b
bindIO              = BindIO
```

```
primBindIO          :: (IO a, a → IO b) ⟶_I b
primBindIO (m,f)     = do x ← run m
                          run (app(f,x))

app (BindIO, m)      = BindIO_1 m
app (BindIO_1 m,f)   = BindIO_2 m f
run (BindIO_2 m f)   = primBindIO (m,f)
```

In the definition of `primBindIO` we use a `do` construct which sequences computations. It is quite similar to sequencing statements in C. Notice that this construct is different from the `do` construct of the source language, which is simply syntactic sugar for `bindIO`.

Defunctionalization Algorithm. The algorithm has two parts. First we perform lambda lifting [47] to move locally defined computation values to the top level of a program. The lambda-lifted computations may acquire extra arguments, corresponding to their free variables, and the algorithm adjusts uses of these local variables to pass the free variables explicitly.

The second part of the algorithm makes the manipulation of computation values explicit. As we saw in the previous examples, for every function definition we need to generate three components:

- First, we add a new constructor to the appropriate representation datatype. Then, we replace the function definition with a new definition that uses the constructor to create a representation for the function.
- Second, we define a behavior function. Its definition is obtained by inserting explicit calls to `app` in the original function definition. For example, if the function definition contains an expression of the form `f x`, then we replace this expression with `app(f,x)`. This is necessary because, after we transform the program, `f` will be the representation of a function value and so we need to use `app` when we want to execute the computation capturing its behavior.
- Third, we generate a new equation for `app` that relates the new constructor and the new behavior.

As in Haskell, the entry point to the program is an `IO ()` action called `main`. Of course, `main` is simply a value; to execute the program, we need to `run` main, which we do by adding a new entry point:

```
entry :: () ⟶_I ()
entry = run main
```

Inlining. After defunctionalization, we have a first-order program that can be compiled using fairly standard technology. Unfortunately, the programs that we obtain from the algorithm we showed are rather inefficient because they contain large amounts of interpretive overhead. For example, we call all functions using the application operator, which examines the function value and takes an appropriate action. The same happens with monadic values and the `run` interpreter. Fortunately, we can obtain efficient programs by performing aggressive inlining: our goal is to eliminate calls to `app` and `run` when the function or monadic value is known statically. This eliminates unnecessary interpretive overhead by evaluating the program at compile-time, and only switching to the interpreter for unknown values. We note that the degree to which the static evaluation succeeds is affected by the quality of the lambda-lifter: noticing that something is a global value and hence that it does not need to be passed as a free variable, generally results in better programs [47]. Here is an example that illustrates how inlining eliminates interpretive overhead:

```
main = bindIO getChar putChar

run (app (app(bindIO,getChar),putChar))
=
run (app (BindIO_1 getChar,putChar))
=
run (BindIO_2 getChar putChar)
=
do x ← run getChar
   run (app(putChar,x))
=
do x ← primGetChar
   run (app(putChar,x))
=
do x ← primGetChar
   run (PutChar_1 x)
=
do x ← primGetChar
   primPutChar x
```

Optimizing Monadic Values. As the previous example illustrates, inlining works well in a number of cases. We can eliminate even more interpretive overhead, however, if we distribute `run` operations across the different constructs of the language. For example, inlining does not help for expressions like the following:

```
run (case x of
       'a' → e1
       _ → e2)
```

This computation will evaluate the argument to `run`, to get a computation value, and then use the interpreter to execute the value. We get better code if we 'lift' the **case** on expressions to a **case** in the statement language like this:

```
run (case x of
       'a' → e1
       _ → e2)
=
case x of
  'a' → run e1
  _ → run e2
```

A similar situation occurs when we try to `run` the result of a pure function (e.g, `run (f x)`). We can avoid the call to `run` by generating an effectful version of `f`, which is obtained by wrapping `run` around the definition of `f`. Thus, functions that return `IO` computation have two behaviors, one pure and one effectful:

```
f     :: A → IO B   -- a value
funF  :: A ⟶ IO B   -- pure behavior
procF :: A ⟶I B     -- effectful behavior

run (funF x) = procF x  -- property of procF
```

Now we can replace `run (funF x)` with `procF x` to eliminate interpreting the results of functions. Note that we do not need to do this for computation values because inlining automatically generates the effectful code. However, if a value results in a large computation, and if it is used many times, then this could make the code that we generate large. We could avoid this by generating an effectful behavior (with no arguments) for such computations, and then calling it where the value is used.

Optimizing Function Values. Note that, while we mainly discussed the liftings for the `run` operator, similar optimizations can be used for functions that return functions as their results. Consider, for example, a function like this:

```
f x = case x of
        'a' → e1
        _   → e2
```

Here `e1` and `e2` are function values. Now, if we encounter an expression `app(f x,y)`, then we would have to interpret the result of the application. We would get better code if we generated a specialized version of `f` by η-expanding:

```
f2 (x,y) = case x of
             'a' → app (e1,y)
             _   → app (e2,y)
```

Now we can replace `app(f x, y)` with `f2 (x,y)`. In fact, it may be tempting to rewrite the definition of `f` just to create a closure, and then add a new equation to `app` that uses `f2` when this closure is applied to an argument. This would eliminate the code duplication that arises when we have two functions. Unfortunately, this is only safe, if we know that `e1` and `e2` are guaranteed to terminate, otherwise this transformation could turn diverging programs into terminating ones. For example, if `e1` diverges, then `f 'a'` diverges. However, if we changed the definition of `f`, then `f 'a'` would simply create a closure, and so terminate immediately. This is an unfortunate consequence of the presence of non-termination in the language. In fact, for the same reason, we can only remove dead code that we know will terminate, as this could also make programs more terminating. While such transformations change the meanings of some programs, it is not clear if this change is important in practice: the visible effect would be that, in rare cases, a program that does not work when not-optimized, will start working when optimized. Still, an implementation could enable or disable such 'unsafe' optimizations with a switch.

Types. In our implementation, we have a completely monomorphic program by the time we perform defunctionalization. This enables us to have different representations for different values, and to avoid having to box and

unbox values when working with polymorphic functions. For this reason, we have a number of different `run` and `app` operators, one for each different monadic or function type (in the previous examples we used the same name to avoid cluttering the definitions with complicated names). Alternatively, we could use a single type to represent all computation values, and then we would only need one `app` and `run` function. This is easy to do in an untyped intermediate language, however, if we want a typed intermediate language, then we need a slightly more advanced type system. For example, we could use generalized abstract datatypes (GADTs) [92, 80] to represent function types. Here is an example using the notation used by the GHC compiler (as of 2006):

```
data FunVal a b where
  Id              :: FunVal a a
  PutChar         :: FunVal Char (IOVal ())

data IOVal a where
  GetChar         :: IOVal Char
  PutChar_1       :: Char →  IOVal ()

app               :: (FunVal a b, a) → b
app (Id, x)        = x
app (PutChar, x)   = PutChar_1 x

run               :: IOVal a → IO a
run GetChar        = getChar
run (PutChar_1 x) = putChar x
```

GADTs differ from ordinary abstract datatypes in that they allow constructors to produce values, whose types are *instances* of the general type for the datatype. This is useful because when we pattern match on such values, we can assume the more concrete types in the relevant alternative. For example, the type of `Id` states that the argument and result of the function have the same type, which is why, in the definition of `app`, we can view `x` as having both type `a` and `b`. Once we know about GADTs, implementing polymorphic defunctionalization becomes fairly easy because we are essentially generating a well-typed interpreter for the language, which is one of the classic applications for using GADTs.

12.3 Representations for Bitdata

In this section we discuss various unboxed representations for bitdata and how they affect the operations on bitdata. The representation that we are discussing here is the internal representation used by an implementation, which may differ from the fixed representation which is used when bitdata is stored in memory areas. In particular, the internal representation has to support bitdata values that have sizes that are not multiples of 8. The internal representation is used when bitdata is used as an ordinary value (e.g., function arguments and results, or values in closures or datatype values).

The main issues arise with bitdata whose size does not match the size of the registers of the machine. Our discussion will be focused on the representation of bitdata that is smaller than the registers of the machine (e.g., how to represent a 20-bit quantity in a 32-bit register). For the purposes of this discussion we shall assume that all bitdata values are internally represented using the same number of bits, W (e.g., 32 on a 32-bit machine). This is not necessary in general: for example, we could choose to represent an n bit value with the smallest number of bytes that can contain the n bits. This extra generality does not affect the issues that we discuss, which is why we stick to a single size.

When we choose how to represent an n-bit value in W bits we face at least two choices: (i) we have to decide where in the W bits to place the n meaningful bits; (ii) we have to decide what to do with the remaining bits of W. In addition, we have to decide if all bitdata values use the same representation, or if it is useful to choose different representations depending on the type. Two reasonable places for the n meaningful bits are either the least or the most significant bits of W. For the remaining bits, we could choose a *normalized* representation, where the unused bits all have a fixed value (e.g., 0), or a representation where the unused bits may contain arbitrary values. The benefit of using a normalized representation is that it provides a unique representation for an n bit quantity, and the drawback is that, in some operations, we may have to perform extra work to maintain the invariant.

We may convert between values that use the different encodings. To change from normalized to unnormalized representation we do not need to do anything, because the normalized representation is a special case of the unnormalized. For the opposite conversion, we need to mask-out any bits that are not meaningful. To change from most- to least-significant representation

we need to shift right by $W - n$, while the opposite transformation is done with a shift to the left. Notice that, because shifting fills the new positions with 0, the result is always normalized.

The operations that we perform on bitdata include: arithmetic operations, logic operations, comparisons, accessing and updating bit-fields, conversions to and from bit-vectors, and storing and retrieving values in memory areas. Notice that, because pointers and references are also a form of bitdata, we have to consider the operations for dereferencing as well. For some operations (e.g., and, or, xor), we can reuse the operations on values of width W directly, independent of the actual representation we choose. For other operations we may have to normalize some of the arguments or the result. For example, the binary complement of a normalized value is not normalized, because the meaningless bits become 1, instead of 0. If we choose to use a normalized representation, then we would have to apply a conversion (i.e., mask) to get a valid result. Other situations where we would have to normalize the result include operations that may overflow (if we use a least-significant representation) or underflow (if we use a most-significant representation). If, alternatively, we chose to work with an unnormalized representation, then for some operations (e.g., comparisons) we would have to convert the arguments to a normalized form. There are cases where we may have to change from most- to least-significant representation. This happens, for example, in the definition of multiplication when using the most-significant encoding. In this encoding, the value x is represented as $x * 2^{W-n}$. The following expression gives us the formula for multiplying two values in this encoding:

$$(x * y) * 2^{W-n} = (x * 2^{W-n}) * y$$

Notice that one of the values has to be changed into the least-significant encoding before we perform the multiplication. To access a bit-field we shift either left (if we use the most-significant encoding) or right (for least-significant encoding). If, in addition, we are using a normalized representation, then we would have to also mask to clear the bits that are not part of the field. If the bitdata and bit-vectors use the same representation, then the conversion functions do not need to do anything. Otherwise, they would have to use the function that changes the representation appropriately.

As far as bit-twiddling is concerned, there isn't a big difference between using the most- or least-significant encoding for ordinary bitdata. However, references and pointers are naturally represented using the most-significant

normalized encoding because this exactly coincides with the address that they represent. It may therefore be tempting to represent all bitdata using this encoding. Unfortunately, this has a negative effect on the operations that manipulate stored bitdata because, when we read data from an address, the result is in least-significant encoding. Consider, for example, dereferencing a pointer to a byte: after we read the byte from memory into a register, we would have to shift the value to the left, so that we turn it into the most-significant normalized representation. If, on the other hand, we decided to represent all bitdata using a least-significant encoding, then we would have to shift aligned references to the left before dereferencing them.

The problem is that references and pointers are naturally represented with the most-significant (normalized) encoding while, for other bitdata, it is more suitable to use the least-significant encoding. This mixed representation is attractive because it eliminates the spurious shifts that occur when use the same encoding for all values: the encoding of references is their address, so we don't need adjustments before dereferencing them; when we read data from memory areas we do not need to adjust the value because bitdata uses the least-significant encoding. The only exception would be pointers that are stored in memory. However, recall (Chapter 9, Section 9.4) that the representation of a pointer in memory is simply its address (or 0 for null pointers) so again we do not need any adjustments.

The mixed representation has no effect on the arithmetic and logical operations because they manipulate values of the same type. However, we need to adjust the operations that manipulate bit-fields, and also the operations that compute the bit-vector representations for pointers and references. Because all operations are overloaded, it is easy to perform such type-dependent manipulations.

When using the least-significant (unnormalized) encoding to access a bit-field we simply shift to the right. Here is an example to make this concrete:

```
bidata PCI = PCI { bus :: Bit 8, dev :: Bit 5, fun :: Bit 3 }

getDev :: PCI → Bit 5
getDev (PCI { dev = x }) = x

-- implementation
getDev pci = pci >> 3
```

However, when we generate the accessor (and update) functions for bit-fields

that contain references, we also have to shift to the left, so that the resulting value uses the most-significant normalized representation:

```
bitdata MyRef = MyRef { ref :: ARef 4 T, bits :: Bit 2 }

getRef :: MyRef → ARef 4 T
getRef (MyRef { ref = x }) = x

-- implementation
getRef ref = (ref >> 2) << 2
```

Notice that shifting right and then left exactly ensures that we clear the bits in the reference that have to be 0 (due to the alignment constraint). A peephole optimizer could replace such pairs of shifts with a mask, if that is more efficient on a particular architecture.

The other change that we need to make is in the implementation of to `toBits` for references and pointers, which becomes a shift to the right, changing back to the least-significant representation.

12.4 Run Time System

Our implementation uses a fairly standard run-time system that supports garbage collection and efficient tail calls. There are two memory regions used by the run-time system: the stack and the heap. The stack is a mixture of a data and control stacks: it is used to store function arguments, local variables, and return addresses for function calls. The heap stores dynamically allocated objects, such as the representations for algebraic datatypes and closures.

12.4.1 Calling Convention

To implement function calls we use the convention that the caller allocates the arguments of a function and the callee deallocates them. Our calling convention is similar to the 'standard' calling convention (e.g., as used in Pascal) with the difference that the return address of a function is pushed on the run-time stack *before* the arguments to the function. The main benefit of this calling convention is that it naturally supports arbitrary tail-calls, which are very common in programs written in functional languages.

Another common calling convention is for the caller to both allocate and deallocate the arguments of a function (e.g., as done in many C compilers). This calling convention does not support tail-calls of functions whose arguments require more space than is occupied by the arguments of the calling function because, upon return, not all arguments will be deallocated. Consider, for example, the following C functions:

```
void f(int x,int y) { ... }
void g(int x) { f(x,x); }
void h(int x) { g(x); }
```

The definition of `h` contains a call to `g` that is in a tail position. In this case, we can perform a tail-call simply by jumping to the code for `g`. When the function `g` finishes its work, it will return directly to the caller of `h` which will deallocate the argument `x`. Now consider the definition of `g`, which also contains a function call (to `f`) in a tail-position. In this case, we cannot perform a tail-call because after `f` is finished it needs to return to `g` so that the second argument can be deallocated.

12.4.2 Stack Frames

The run-time stack of our programs is split into a number of *stack frames*. Each stack frame contains the state of a function that is currently active (i.e., is in the process of computing a result). The order of the stack frames on the stack depends on the order in which functions call each other at run-time. Our stack grows 'down' (towards lower addresses), so, if a function `f` calls a function `g`, then the stack frame for `g` would be at lower addresses then the stack frame for `f`. This, together with the layout of stack frames is illustrated in Figure 12.1. The layout of stack frames is fairly standard, except for the location of the return address for the function: in our stack frames it comes *before* (at a higher address) the other arguments. It is more common to push the return address after the arguments, just before jumping to the code for the called function (so much so that there is special instruction for this on the IA32) . One of the reasons we deviate from the status quo, is that our frame layout is more natural for tail calls: because the return address is essentially the first argument to all functions, when we perform a tail call we do not need to adjust the location of the return address. If we placed the return address after the arguments, then we would have to move the return address to its new location when we tail-call a function with a different number of

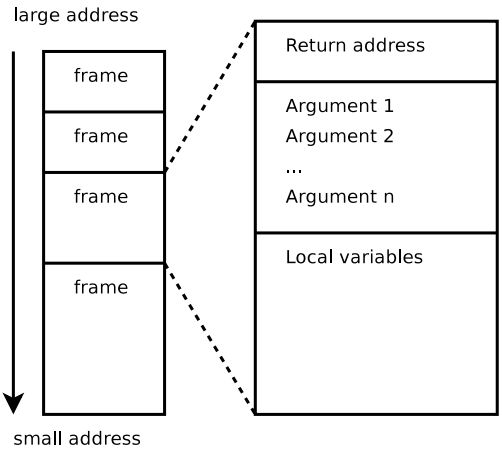


Figure 12.1: The stack, and the layout of a stack frame.

arguments. The code that we generate for a typical call to a function `F` is as follows:[1]

```
    push $1f
    push arg_1
    ...
    push arg_n
    jmp F
1:
```

The code for tail calls differs in that we overwrite the caller's stack frame, rather than pushing a new frame on the stack. While over-writing the current stack frame, we have to be careful to avoid prematurely overwriting data that we need. We implemented this with a dependency analysis between stack locations, and using an auxiliary register to break dependency cycles. When we return from a function, we deallocate its stack frame by adding the size of the frame to the stack pointer and then using the `ret` instruction to pop the return address from the stack, and jump to it:

```
    add $frame_size, %esp
    ret
```

[1]In the next section, we shall discuss a small change to this code

12.4.3 Traversing the Stack

To perform garbage collection we need to identify the 'root set' of live pointers into the heap. Pointers to the heap that reside on the stack belong to functions that are still active, and so they refer to potentially live data. Unfortunately, not all values on the stack are pointers, so we need to do some work to distinguish pointers to the heap from other values.

At compiler time, we know the layouts of all possible frames that may reside on the stack because each function can keep track of the types of its variables. However, we do not know until run-time in what order these frames will appear on the stack when a garbage collection occurs because this depends on the execution path of the program. Thus, if the garbage collector is to find the pointers on the stack, we need to arrange for some way to recognize the frames on the run-time stack.

A simple way to recognize frames at run-time would be to add a special 'tag' field to each frame. This essentially amounts to passing an extra argument to all functions, which results in a fairly expensive solution. We can get a better solution by noticing that we can use the return address, to identify the *previous* frame on the stack. This works, because the return address uniquely identifies the place in the program from which the current function was called.

All we are missing now is a means to associate return addresses with the information about their corresponding frames. A simple solution might be to use a hash-table or another similar data structure, but this gives us a fairly expensive solution. To get a more lightweight solution, we use a trick that exploits the slightly non-standard way that we use to generate code for function calls. More specifically, we modify the code that is generated for a function call to embed the information about the frame directly in the code, after the jump to the callee, and before the address where we continue to execute after the callee returns.

```
    push $1f
    push arg_1
    ...
    push arg_n
    jmp F
    .long frame-info
1:
```

With this convention, we can easily access the information about the frame

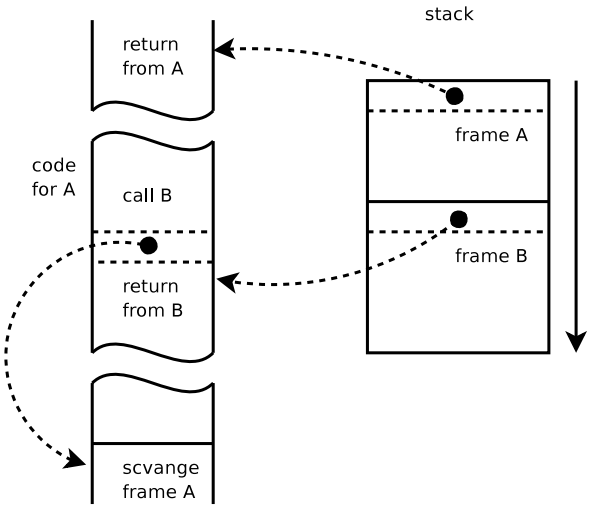


Figure 12.2: Walking the stack. Function A called function B. The figure illustrates how B can find the code to process the frame for A.

of our caller, simply by looking at the address in memory before the return address of the function.

In our implementation, we generate specialized pieces of code to garbage-collect (scavenge) each shape of frame or heap object. So the *frame-info* in the previous example is a pointer to the code that scavenges the appropriate frame (see Figure 12.2). Having set up things appropriately, the code that scavenges the stack during garbage collection becomes fairly simple, and it may be written like this in C:

```
void walk_stack(DWord* frame, ScavFun scav) {
  do {
    DWord* ret;
    frame = scav(frame);
    ret   = (DWord*)(*frame);
    scav  = (ScavFun)ret[-1];
    --frame;
  } while (scav != 0);
  scav_globals();
}
```

The function `walk_stack` uses two arguments: `frame`, which is a pointer to

the end (smallest address) of the current frame, and `scav` which is a function to scavenge this frame. The scavenging functions expect a pointer to the end of a frame and return a pointer to the beginning of the frame as a result. To terminate the process, we arrange that the location before the return address of the main function contains 0.

Given these details, the rest of the garbage collector follows the standard structure of a copying collector [2].

12.5 Summary

In this section we presented some of the details of the implementation of the back-end for our prototype compiler. Our implementation preserves types throughout the compilation process and uses this information to pick the most suitable representation for each value.

We generate specialized versions for polymorphic functions and in this way we convert polymorphic to monomorphic programs. We use defunctionalization to provide explicit representations for computation values. It is important to perform aggressive inlining to avoid excessive interpretive overhead due to computation values.

To implement function calls we used a calling convention that supports efficient tail-calls. In addition, our calling convention makes it easy to add extra information about stack frames, which is useful for the implementation of the garbage collection algorithm.

Chapter 13

Conclusions and Future Work

In this final chapter, we summarize the key contributions of our dissertation (Section 13.1), and present some ideas for future work (Section 13.2).

13.1 Summary of Contributions

The main contribution of this dissertation is that it shows how to extend a modern functional language with support for manipulating data with rigid representation constraints. We identified two kinds of such data: *bitdata*, which is data that is represented using fixed-width bit-patterns, and *memory areas*, which is data that has a fixed layout in memory. The motivation for working with such data stems from our desire to write system-level software in a high-level functional language. The following list describes our key contributions in more detail:

- We showed how to formalize a system for natural numbers at the type level by using qualified types and improvement (Chapter 4). Our design uses this feature to keep track of various invariants about data, such as the size of bit-vectors, the alignment of references, and the sizes of arrays.

- We presented a simple, yet novel way of working with predicates with functional dependencies (Chapter 5). It enables us to treat such predicates as type functions, which improves the readability of the types in our programs. This idea is general and can be used in any program that uses functional predicates.

- We described a calculus that can be used to understand and analyze the various definitional constructs of modern functional languages (Chapter 6). We used this notation to define and discuss the various pattern matching constructs introduced in our design.

- We described a new declaration that allows programmers to define types whose values have programmer-defined bit-pattern representations (Chapter 7). Our design provides a high-level, type-safe approach for manipulating low-level data, that enables programmers to work with bitdata as if it was ordinary algebraic data.

- We developed algorithms that can analyze programs to detect bitdata that fails to satisfy important algebraic properties—the lack of junk and confusion in the representation (Chapter 8). Working with such bitdata requires caution but we showed that we can extend the exhaustiveness/reachability analysis for function definitions to also support such unusual bitdata.

- We designed a system of types that can be used to give a precise description of the layout of memory areas (Chapters 9 and 10). Our design provides a safe interface for manipulating memory areas by utilizing the type system to enforce a number of invariants at compile time. In addition, our design supports the manipulation of aligned data by computing and propagating alignment information for references.

- We implemented a compiler for a strict, pure, functional language extended with support for bitdata and memory areas (some details are presented in Chapter 12). This implementation demonstrates the feasibility of our design, and also, it enabled us to experiment with the design in a working system, which gives us confidence that our design can be used for developing practical systems software. We tried our design on a number of examples, including the implementation of the boot code for an operating-system kernel.

13.2 Future Work

In this section, we outline a number of ideas for future work that we have considered but not yet implemented or fully explored.

13.2.1 Parameterized Bitdata

In Chapter 7, we presented **bitdata** declarations that can be used to define new types with explicit bit-vector representations. One limitation of these declarations is that, unlike ordinary **data** declarations, they do not support type parameters. While the monomorphic form is sufficient in many situations, occasionally it is useful to parameterize a bitdata type by another type. To see how parameterized bitdata might be used, consider the following example:

```
bitdata Pair a b = Two { x :: a, y :: b }
```

This type describes bitdata types that are represented by concatenating any two other types (i.e., it is similar to the operator `(#)` for bit-vectors). For example, the type `Pair (Bit 16) (Bit 16)` is represented with 32 bits, while the type `Pair (Bit 5) (Bit 3)` is represented with 8 bits. In the following paragraphs, we illustrate the different types and operations that we have to introduce because of this declaration.

Recall that, for each constructor of a **bitdata** declaration, we introduce a new 'product' type describing the values created with that constructor. When we have a parameterized bitdata type, we also have to parametrize the product types for any constructors that mention one or more of the parameters. For example, because the constructor `Two` uses both type parameters, its corresponding product type would have the following kind:

```
Two' :: * → * → *
```

Types that are introduced by a **bitdata** declaration should belong to the `BitRep` class to indicate that they have well-formed bit-vector representations. When we work with parameterized bitdata these instances are polymorphic:

```
instance (BitRep a + BitRep b = n, Width n) ⇒ BitRep (Pair a b) n
instance (BitRep a + BitRep b = n, Width n) ⇒ BitRep (Two' a b) n
```

The assumptions on the instances assert two things: (i) each of the fields in a constructor have valid bit-vector representations, and (ii) the representation of the values in the type are not too wide. In general, the constraints in the instance for a bitdata type are computed by combining the constraints from the instances of the product types of its constructors.

For each **bitdata** declaration, we introduce a number of functions that manipulate the values. For parameterized bitdata, these functions are polymorphic. For example, the constructor and the accessor and update functions for `Pair` have the following types:

```
Two       :: Two' a b → Pair a b

get'Two'x :: (BitRep a m, BitRep b n) ⇒ Two' a b → a
get'Two'y :: (BitRep a m, BitRep b n) ⇒ Two' a b → b

set'Two'x :: (BitRep a m, BitRep b n) ⇒ a → Two' a b → Two' a b
set'Two'y :: (BitRep a m, BitRep b n) ⇒ a → Two' a b → Two' a b
```

The contexts on the types of the `get` and `set` functions restrict them to types that have bit representations. From an operational point of view, the context specifies how to access (or update) the relevant field. The type of the constructor does not need a context because this function is simply an embedding and so we do not need any extra information. When we use the constructor to create `Pair` values directly, we still need to check that the types of the values belong to the `BitRep` class:

```
mkTwo :: (BitRep a m, BitRep b n) ⇒ a → b → Pair a b
mkTwo e1 e2 = Two { x = e1, y = e2 }
```

It is also possible to give the `set` functions more general types that allow them to change the type of a field. If we do this, then we should add extra constraints that ensure that the resulting value is not too wide (i.e., we can still discharge the `Width` predicate for its width).

The key technical difficulty in adding type parameters to **bitdata** declarations is adjusting the junk and confusion-analysis to work in the presence of type parameters. This is important because we use the confusion analysis to determine if a field needs to be a member of the `BitData` class, or if membership in `BitRep` is sufficient. To see the difficulty, consider the following example:

```
bitdata T a b = C1 { x :: a, y :: b } as x # B0 # y
              | C2 { x :: a, y :: b } as y # B1 # x
```

The type `T` has two constructors `C1` and `C2`. The two constructors differ in that `C1` places the field `x` in the most significant part of the representation, while `C2` places `x` in the least significant part. We cannot accurately determine

if the values constructed with the two types overlap because we do not know the widths of the fields `x` and `y`. More precisely, if they are of the same width, then the two constructors do not overlap because the tag bit will distinguish them, but otherwise they will overlap.

We can avoid this problem by being conservative in our analysis: if an implementation cannot determine for certain that two constructors do not overlap, then it may assume that they do, thus requiring `BitData` instances for all of their fields. This is likely to work well in practice because examples like `T` are contrived, and it is much more likely that the tag bits in the bitdata type are in a fixed location so that the analysis can tell if the constructors overlap or not.

13.2.2 Computed Bitfields

In our current design, the fields of a **bitdata** declaration are always represented by a contiguous sequence of bits. Some more exotic encodings, however, may require that a single conceptual value is constructed out of several non-adjacent components (we saw an example of this in the description of segment descriptors in Chapter 11). Another instance of the same problem is when the conceptual value for a field differs from its concrete representation. For example, if we want to work with quantities measured in bytes but, in the actual representation they are stored in units of words.

Using our current design, programmers would have to write functions that assemble the non-adjacent components into a single value or perform any necessary conversions on the fields. This may not be a big problem in practice because, in our experience, such bitdata types are not very common. Another possibility, however, would be to extend the **bitdata** declarations to support such bitdata directly.

The basic idea is to allow the values for fields to be *computed* from the **as** clause, so long as this computation is reversible to an extent. This requirement is necessary because we want to use the constructors both to construct and pattern match on values. To illustrate how this might work, consider the following example (the notation that we use conflicts with the notation for default values on the fields but that is not important here).

```
bitdata T = C { size = s # B00 :: Bit 8,
                opts           :: Bit 2
              } as s # opts
```

Values of type `T` have two fields: an 8-bit field called `size`, and a 2-bit field called `opts`. Ordinarily, such values would require 10 bits to represent but, in this case, we use only 8 bits because the field `size` has the additional constraint that its last two bits are always `B00` and so we do not need to store them. This constraint may arise because we would like to work with sizes measured in bytes but, for this type, the size should be a whole number of words.

The main difference from our original design is that each conceptual field has an annotation that specifies how to compute its value from the representation specified in the **as** clause. If this specification is omitted, then it is assumed to be just the name of the field. For example, the following definition of `T` is equivalent to the previous one:

```
bitdata T = C { size = s # B00 :: Bit 8,
                opts = opts    :: Bit 2
              } as s # opts
```

The annotations on fields are used when we pattern match on a value, or when we use accessors for fields to extract their values. For example, at the bit level, the `get` operations for the type `C'` behave like this:

```
get'C'size :: Bit 8 → Bit 8
get'C'size (s # opts) = s # B00

get'C'opts :: Bit 8 → Bit 2
get'C'opts (s # opts) = opts
```

Note that the patterns correspond to the **as** clause and the results correspond to the annotations on the fields.

When programmers specify the value for a field (e.g., when they use a constructor or an update function), they use the computed value. Therefore, we need to invert the computation specified for the field to determine the concrete representation that we should use. The main design work that is required to complete this extension is to determine what expressions should be allowed in the computation for a field so that we can determine a sensible inversion function. One option would be to use patterns as specifications of how to compute fields. Then we could define the update and the constructor functions like this:

```
set'C'size :: Bit 8 → Bit 8 → Bit 8
set'C'size (s # B00) (_ # opts) = s # opts
```

```
set'C'opts :: Bit 2 → Bit 8 → Bit 8
set'C'opts opts (s # _) = s # opts

mkC :: Bit 8 → Bit 2 → Bit 8
mkC (s # B00) opts = s # opts
```

Note that the results of the functions correspond to the **as** clause, and that we use the field annotations to examine fields. If we choose this approach, then the constructor and update functions may be partial, as is the case for `set'C'size` and `mkC`. These functions will fail if we try to construct a value by using invalid values for the fields (e.g., in this case, a value for the `size` field that does not end in `B00`).

To avoid partiality in constructors, we consider two alternative designs. One option is to restrict the patterns that can be used as field computations to allow only irrefutable patterns. This limits the computations that we can do when accessing fields, but it ensures that the constructors and update functions will not fail due to invalid values. Irrefutable patterns are sufficient for working with non-contiguous fields but we cannot use them to define examples like the bitdata type `T`.

Another option would be to allow arbitrary patterns in the fields definitions but to use an irrefutable version of the pattern when we construct values. We can obtain an irrefutable version of a pattern by replacing subpatterns that may fail with wildcard patterns (for bitdata constructors this amounts to ignoring the tag bits). Using this approach, the update and constructor functions for `T` would be as follows:

```
set'C'size :: Bit 8 → Bit 8 → Bit 8
set'C'size (s # (_ :: Bit 2)) (_ # opts) = s # opts

set'C'opts :: Bit 2 → Bit 8 → Bit 8
set'C'opts opts (s # _) = s # opts

mkC :: Bit 8 → Bit 2 → Bit 8
mkC (s # (_ :: Bit 2)) opts = s # opts
```

The difference from the previous example is that we have replaced the pattern `B00` with its irrefutable version `(_ :: Bit 2)`. The net effect of this design choice is that constructors do not validate the values for their fields, which is similar to what we did with **if** clauses (they are not evaluated when we

construct values, see Section 7.3.4 of Chapter 7). In this case, however, we need to be more careful to avoid violating the properties enforced by the types. Consider, for example, the following declaration:

```
bitdata Zero = Z as B0
```

This defines a bitdata type `Zero` that has only a single value represented as `B0`. We can violate this property of `Zero` by using a strategy similar to the one that we followed in Chapter 7 (Section 7.4) where we exploited the confusion between constructors to convert values to incompatible types. In this case, instead of using confusion between constructors, we can exploit the fact that the constructors for computed fields do not validate their values.

```
bitdata Sum  = A { x :: Zero } as B0 # x
             | B { x :: Bit 1 } as B1 # x

bitdata Bad = Bad { f = A { x = bits } :: Sum } as bits

cast :: Bit 1 → Zero
cast b = case Bad { f = B { x = b } } of
           Bad { f = A { x = z } } → z
```

First, we define the type `Sum`, which has two constructors, `A` and `B`, that have fields of incompatible types. This is not a problem, however, because `Sum` values are represented with 2 bits: one tag bit, and one bit for the value of the field. Next, we define the type `Bad`, which has one computed field, `f`, of type `Sum`. `Bad` values are represented with one bit, which should be the representation of a `Zero` value. When we access the field `f`, we use the constructor `A` to construct a value of type `Sum`. When we use the constructor `Bad`, however, we are performing the operation in the opposite direction, essentially turning `Sum` values into `Zero` values. This is where the problem occurs: because the constructor does not check the tag on the `Sum` value, we can violate the properties of the types, which is what we do in the function `cast`. Had we used the checked semantics for constructors, then the function `cast` would terminate with a run-time error when we try to construct the `Bad` value that is scrutinized by the `case` expression.

In conclusion, if we choose the unchecked semantics for constructors, then there is no guarantee that the parts of the representation defined by a possibly failing pattern will satisfy the property suggested by their type (e.g., we have

no guarantee that `bits` from the previous example is a valid value of type `Zero`). Therefore, we should allow such representations only if their types belong to the `BitData` class (i.e., if they have no special invariants). With this requirement, the declaration `Bad` would be rejected because the constructor `A` may fail, but the value `bits` is of type `Zero`, which does not belong to `BitData`.

13.2.3 Views on Memory Areas

In Chapter 10, we described a number of functions that allow us to 'cast' between different views on arrays. For example, one of these functions is `toMatrix`, which enables us to turn a one-dimensional array into a two-dimensional one:

```
toMatrix :: Ref (Array (x*y) r) → Ref (Array x (Array y r))
```

A common property of all casting functions is that they do not perform any computation but, instead, they change the 'view' on a memory area. In other words, given a reference to an area, a casting function returns the exact same reference but at a different type. Therefore, we may think of casting functions as being derived from 'functions' on areas that simply 'modify' the types of references. It is useful to capture this property of casting functions in their types because then we could define casting functions in a modular fashion (e.g., we can have casting functions that are parameterized by other casting functions). To achieve this, we introduce a type constructor called `(⇝)` which classifies 'functions' between compatible memory areas, and a new kind called `View`:

```
(⇝) :: Area → Area → View
```

An expression of type `A ⇝ B` is an assertion that it is safe to convert a reference to an `A` area into a reference to a `B` area. We shall call such expressions *views* on memory areas, and we will use them in the casting operator:

$$\frac{\Gamma \vdash v : r \rightsquigarrow s}{\Gamma \vdash \mathit{cast}\ v : \mathit{ARef}\ a\ r \to \mathit{ARef}\ a\ s}$$

The casting operator is an identity function in disguise because the address `cast v r` is the same as the address `r`. The purpose of the argument `v` is to ensure that the cast is safe. Therefore, we have to be careful to provide only operations that construct valid views.

Basic Views. We can start with some simple examples:

$$\frac{}{idV : r \rightsquigarrow r}$$

$$\frac{v_1 : r \rightsquigarrow s \quad v_2 : s \rightsquigarrow t}{(v_2 \circ v_1) : r \rightsquigarrow t}$$

The first rule introduces the identity view, idV, which does not change the view on an area. The second rule introduces the operator $(\circ)$, which allows us to compose views. These operations satisfy the usual identity and associativity laws for composition:

$$\begin{aligned} idV \circ f &= f \\ g \circ idV &= f \\ f \circ (g \circ h) &= (f \circ g) \circ h \end{aligned}$$

Furthermore, the operation *cast* satisfies the functorial laws:

$$\begin{aligned} cast\ idV &= id \\ cast\ (f \circ g) &= cast\ f\ .\ cast\ g \end{aligned}$$

Of course, we would like to have some more interesting casts between memory areas. For example, the casting operations from Chapter 10 fit naturally in this framework:

$$\frac{}{splitArr : Array\ (x + y)\ r \rightsquigarrow Pair\ (Array\ x\ r)\ (Array\ y\ r)}$$

$$\frac{}{toMatrix : Array\ (x * y)\ r \rightsquigarrow Array\ x\ (Array\ y\ r)}$$

In addition, we could provide the inverse views *joinArr* and *fromMatrix* and also the byte-array functions *fromBytes* and *toBytes*.

Inverse Views. All of the views so far have an inverse, so instead of defining the views in isomorphism pairs, we might want to add a general inversion operation:

$$\frac{v : s \rightsquigarrow r}{inv\ v : r \rightsquigarrow s}$$

This allows us to define *fromMatrix* as *inv toMatrix*, for example. The inversion operator satisfies the usual inversion laws:

$$\begin{aligned} inv\ idV &= idV \\ inv\ (f \circ g) &= inv\ g \circ inv\ f \end{aligned}$$

The tradeoff of having a general inversion operator is that we restrict ourselves to working only with views that do not lose information. To see what we lose, suppose that we add a type constant `Blank :: Area`, which occupies 1 byte, but we do not provide any operations on references to such areas (e.g., there are no instances for the `ValIn` and `Bytes` classes). Then, it would make sense to have a view that 'forgets' the format of an area:

$$\frac{}{forget : t \rightsquigarrow Array\ (SizeOf\ t)\ Blank}$$

Clearly, it is not safe to invert the view *forget*, because then we would be able to cast between arbitrary memory areas of the same size. For example, we could define a view, `bad`, that allows us to turn arbitrary integers into pointers:

```
bad :: Stored (Bit 32) ⇝ Stored (Ptr a)
bad = inv forget ∘ forget
```

Area Constructors as Functors. So far, we have shown that the casting operations from Chapter 10 fit into this framework. However, we can also define new operations that we could not define before. For example, we can define an operation called *mapArr* that exploits the functorial nature of the type constructor *Array*:

$$\frac{v : r \rightsquigarrow s}{mapArr\ v : Array\ n\ r \rightsquigarrow Array\ n\ s}$$

This operation allows us to apply a cast to all the elements in an array. It satisfies the functor laws:

$$\begin{aligned} mapArr\ idV &= idV \\ mapArr\ (f \circ g) &= mapArr\ f \circ mapArr\ g \end{aligned}$$

As an example of how we might use *mapArr*, consider the problem of converting a 2-dimensional array into a 3-dimensional one. There are two ways

to achieve this because we can either split along the first or the second dimension. Here is how we can define the function that splits along the first dimension:

```
split1 :: Ref (Array (x + y) (Array z r)) →
          Ref (Array x (Array y (Array z r)))
split1 r = cast toMatrix r
```

We can define this function with the operations from Chapter 10 because `(cast toMatrix)` is the same as the function that we previously called `toMatrix`. We can define the function that splits along the second dimension like this:

```
split2 :: Ref (Array x (Array (y+z) r)) →
          Ref (Array x (Array y (Array z r)))
split2 r = cast (mapArr toMatrix) r
```

The function `split2` provides an example of something that we can do with area views that we cannot do with the operations defined in Chapter 10.

13.2.4 Reference Fields

Our design uses types of kind `Area` to describe the layout of memory areas. To identify memory areas we use references. For example, consider the following declaration:

```
struct T where
  fst :: Stored (Bit 16)
  snd :: Stored (Bit 32)
```

Then a value of type `Ref T` is the start address of an area that has two adjacent sub-areas described by the types of the fields.

The Idea. In our design, we chose to identify areas by their initial address. It is possible to avoid this (somewhat arbitrary) choice by allowing programmers to specify the address that identifies an area. One possibility is to extend the syntax of **struct** declarations to allow programmers to specify the *reference* field of a structure. Then, when we define a reference to the entire structure, we would use the reference field to identify the structure, instead of using the first field as in our original design. Here is an example:

```
struct T1 where
    fst :: Stored (Bit 16)
    ^
    snd :: Stored (Bit 32)
```

The special field `^` is the reference field. In this case, it specifies that `T1` areas should be identified by the address *between* the two sub-areas. Note that the field `fst` resides at a negative offset relative to the `T1` reference, something that cannot not happen in our original design. Figure 13.1 illustrates this graphically. A structure may have at most one reference field. If there is no specified reference field, then it is implicitly added before the first field, which is compatible with our current design.

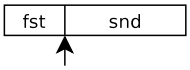


Figure 13.1: A reference to `T1`

New Functionality. Using this extension, we can specify some interesting areas that we cannot express directly in our design. For example, consider the following declaration:

```
area r :: ARef 4 T1
```

As before, the alignment constraint restricts the possible values for the reference `r`. However, now `r` refers to the the middle of the structure instead of the beginning and so we know that the field `snd` will be aligned on a 4 byte boundary instead of `fst`. To achieve this in our original design we have to add some padding to the beginning of the structure to ensure that the second field is properly aligned:

```
struct T1' where
    pad :: Stored (Bit 16)
    fst :: Stored (Bit 16)
    snd :: Stored (Bit 32)

area r' :: ARef 4 T1'
```

Offsets We have to account for reference fields when we compute references to sub-areas. In our original design, computing a reference to a sub-field is fairly easy: we simply calculate the offset of the field in the structure. In the extended design, we have to adjust the computation to account for the reference fields of both the entire structure and the sub-field. Consider the following example:

```
struct Wrap r where
  wrapped :: r
```

References to `Wrap r` areas point to the beginning of the area because we have no explicit reference field. Note, however, that the selector function `wrapped` will behave differently depending on how we instantiate `r`. For example, to access the field of `Wrap T1` we need to add 2 bytes (because of the reference field), but to access the field of `Wrap (LE (Bit 32))` we do not need to add anything. Clearly, these different computations also affect the alignments of the resulting references, so we need to adjust the type of the selectors to account for this.

We can formalize all these computations by using the following type function:

```
class Offset (r :: Area) (n :: Nat) | r ⇝ n

instance Offset (LE r) 0
instance Offset (BE r) 0
instance Offset (Array n r) 0
instance Offset T1 2
```

The predicate `Offset r n` asserts that the reference point for area `r` is `n` bytes from the beginning of the structure. The fact that we use the beginning is not important, but we do need a common reference point to perform the computations. The instances specify that we reference stored values and arrays by using the first address of the area (as in our original design). Non-zero offsets arise from the use of reference fields in **struct** declarations. For example, the offset for `T1` is two because the reference field for `T1` is after the field `fst` which occupies two bytes:

```
instance Offset T1 2
```

Using the class `Offset`, we can define the type for the selector function `wrapped` like this:

```
(.wrapped) :: ARef a (Wrap r) → ARef (GCD a (Offset r)) r
```

Operationally, the evidence for `Offset r n` could be the integer `n` so `wrapped` simply needs to add the evidence to its argument. In general, the alignment of the sub-field can be computed by the greatest common divisor of the alignment of the reference and the distance between the reference to the structure and the reference to the field. Here is another example, which illustrates what happens when we access fields from structures that have a reference field:

```
struct T3 r s t where
  f1 :: r
  f2 :: s
  ^
  f3 :: t

(.f3) :: (Offset t o)
      ⇒ ARef a (T3 r s t) → ARef (GCD a o) t
(.f2) :: (SizeOf s - Offset s = o)
      ⇒ ARef a (T3 r s t) → ARef (GCD a o) s
(.f1) :: (SizeOf s + SizeOf r - Offset r)
      ⇒ ARef a (T3 r s t) → ARef (GCD a o) r
```

13.2.5 Implementation and Additional Evaluation

In the previous sections we presented some ideas for possible extensions and generalizations of our design. However, before we proceed with such extensions it would be useful to spend some more time experimenting and evaluating the design in its present form. In particular, it would be useful to develop more systems software and compare our implementations with existing code for safety, expressiveness, and performance. Before we can proceed with this, we would need a fairly robust and efficient implementation of the language. The prototype compiler that we developed for experimentation during the course of this work is lacking a number of optimizations (e.g., no register allocation and no inlining) which makes it difficult to use it in comparisons of performance. An interesting area for future exploration would be to consider optimizations of the bit-twiddling code resulting from **bitdata** declarations.

Bibliography

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, Jr. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised Report on the Algorithmic Language Scheme. *Journal of Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[3] Joe Armstrong. Erlang—A Survey of the Language and Its Industrial Applications. In *Proceedings of the Ninth Symposium and Exhibition on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, October 1996.

[4] Lennart Augustsson. Cayenne–A Language With Dependent Types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, Baltimore, MD, USA, September 1998.

[5] Lennart Augustsson, Jacob Schwartz, and Rishiyur S. Nikhil. *Bluespec Language Definition*. Sandburst Corporation, December 2002.

[6] Godmar Back. DataScript—A Specification and Scripting Language for Binary Data. In *Proceedings of the ACM Conference on Generative Programming and Component Engineering*, pages 66–77, Pittsburgh, PA, USA, October 2002.

[7] Henk Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[8] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1997.

[9] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven Defunctionalization. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, Netherlands, June 1997.

[10] Bell Labs. The Creation of the UNIX* Operating System. `http://www.bell-labs.com/history/unix`, date viewed: 25 April 2007.

[11] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[13] Edoardo S. Biagioni. Sequence Types for Functional Languages. Technical Report CMU-CS-95-180, School of Computer Science, Carnegie Mellon University, August 1995.

[14] Edoardo S. Biagioni, Robert Harper, and Peter Lee. A Network Protocol Stack in Standard ML. *Journal of Higher-Order and Symbolic Computation*, 14(4):309–356, 2001.

[15] Matthias Blume. No-Longer-Foreign: Teaching an ML Compiler to Speak C "Natively". *Electronic Notes in Theoretical Computer Science*, 59(1):36–52, November 2001.

[16] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[17] C99 Committee. *C99 Specification with TC1 and TC2*, May 2005. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`, date viewed: 25 April 2007.

[18] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Report (revised). Technical Report SRC-RR-52, HP Labs, November 1989.

[19] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[20] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated Type Synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, Tallinn, Estonia, September 2005.

[21] Manuel M. T. Chakravarty and the Haskell FFI Team. *Haskell 98 Foreign Function Interface (1.0)*, 2003. `http://www.cse.unsw.edu.au/~chak/haskell/ffi`, date viewed: 25 April 2007.

[22] Christopher L. Conway and Stephen A. Edwards. NDL: A Domain-Specific Language for Device Drivers. In *Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 30–36, Washington, DC, USA, June 2004.

[23] Olivier Dubuisson. *ASN.1—Communication Between Heterogeneous Systems*. Morgan Kaufmann Publishers, September 2000.

[24] Martin Erwig and Simon Petyon Jones. Pattern Guards and Transformational Patterns. *Electronic Notes in Theoretical Computer Science*, 41(1):12.1–12.27, August 2001.

[25] Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, May 1996.

[26] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: A Binary Foreign Language Interface for Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 153–162, Baltimore, MD, USA, September 1998.

[27] Kathleen Fisher and Robert Gruber. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, pages 295–304, Chicago, IL, USA, June 2005.

[28] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The Next 700 Data Description Languages. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, USA, 2006.

[29] Kathleen Fisher, Ricardo Pucella, and John Reppy. A Framework for Interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1):3–19, November 2001.

[30] Guangrui Fu. Design and Implementation of an Operating System in Standard ML. Master's thesis, University of Hawaii at Manoa, August 1999.

[31] Galois Inc. *Cryptol Reference Manual*, January 2004. `http://www.cryptol.net/doc.htm`, date viewed: 25 April 2007.

[32] Benedict R. Gaster. *Records, Variants, and Qualified Types*. PhD thesis, University of Nottingham, July 1996.

[33] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1):68–95, January 1977.

[34] Andrew D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, 1992.

[35] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[36] Daniel Joeseph Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, August 2003.

[37] Per Gustafsson and Konstantinos Sagonas. Native Code Compilation of Erlang's Bit Syntax. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Erlang*, pages 6–15, Pittsburgh, Pennsylvania, October 2002.

[38] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In *Lecture Notes from the First International Spring School on Advanced Functional Programming Techniques*, pages 137–182, Bastad, Sweden, May 1995.

[39] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A Principled Approach to Operating System Construction in Haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, Tallinn, Estonia, September 2005.

[40] Robert Harper, Peter Lee, and Frank Pfenning. The Fox project: Advanced Language Technology for Extensible Systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, January 1998.

[41] Robert W. Harper and Benjamin C. Pierce. Extensible Records Without Subsumption. Technical Report CMU-CS-90-102, School of Computer Science, Carnegie Mellon University, Feburary 1990.

[42] P. Henderson. Purely Functional Operating Systems. In *Proceedings of the Conference on Functional Programming and Its Applications*, pages 177–192, 1982.

[43] Wilson C. Hsieh, Marc E. Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian N. Bershad. Language Support for Extensible Operating Systems. Technical Report TR-95-11-02, University of Washington, 1995.

[44] Glen C. Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[45] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual (Volume 3a)*, January 2006. `http://www.intel.com/products/processor/manuals/index.htm`, date viewed: 25 April 2007.

[46] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, USA, June 2002.

[47] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, Nancy, France, September 1985.

[48] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, November 1994.

[49] Mark P. Jones. Simplifying and Improving Qualified Types. Technical Report YALEU/DCS/RR-1040, Yale University, New Haven, CT, USA, June 1994.

[50] Mark P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[51] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the Ninth European Symposium on Programming*, pages 230–244, Berlin, Germany, March 2000.

[52] Mark P. Jones, Magnus Carlsson, and Johan Nordlander. Composed, and in Control: Programming the Timber Robot. Technical report, OGI School of Science & Engineering at OHSU, August 2002.

[53] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[54] L4ka Team. *L4 eXperimental Kernel Reference Manual*, January 2005. `http://l4ka.org/`, date viewed: 25 April 2007.

[55] Peter J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Parts I/II. *Communications of the ACM*, 8(2/3):89–101:158–165, February/March 1965.

[56] Konstantin Läufer and Martin Odersky. Polymorphic Type Inference and Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[57] John Launchbury and Simon Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, FL, USA, June 1994.

[58] Xavier Leroy. *CamlIDL User's Manual*. INRIA Rocquencourt. `http://caml.inria.fr/pub/old_caml_site/camlidl`, date viewed: 25 April 2005.

[59] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the Twenty Second ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, CA, USA, January 1995.

[60] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 237–250, Copper Mountain Resort, CO, USA, December 1995.

[61] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert Constable. Building Reliable, High-Performance Communication Systems from Components. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, pages 80–92, Kiawah Island Resort, SC, USA, December 1999.

[62] Fabrice Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for Hardware Programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, CA, USA, October 2000.

[63] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[64] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML—Revised*. MIT Press, May 1997.

[65] John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[66] Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, USA, June 1989.

[67] Eugenio Moggi and Amr Sabry. Monadic Encapsulation of Effects: A Revised Approach (extended version). *Journal of Functional Programming*, 11(6):591–627, 2001.

[68] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995.

[69] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. In *Proceedings of the ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.

[70] National Semiconductor. *DP8390D/NS32490 NIC Network Interface Controller*, July 1995. `http://www.national.com/opf/DP/DP8390D.html`, date viewed: 25 April 2007.

[71] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A Functional Notation for Functional Dependencies. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 101–120, Firenze, Italy, September 2001.

[72] Marius Nita, Dan Grossman, and Craig Chambers. A Theory of Implementation-Dependent Low-Level Software. Technical Report UW-CSE Technical Report 2006-10-01, University of Washington, October 2006.

[73] Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology and Göteborg University, 2000.

[74] PCI Special Interest Group. *PCI Local Bus Specification*, June 1995. `http://www.pcisig.com/specifications/conventional`, date viewed: 25 April 2007.

[75] Simon Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in

Haskell. In *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.

[76] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[77] Simon Peyton Jones and Mark Shields. Lexically Scoped Type Variables. This paper is available from: `http://research.microsoft.com/~simonpj/papers/scoped-tyvars`, date viewed: 25 April 2008, March 2004.

[78] Simon Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Proceedings of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, Charleston, SC, USA, 1993.

[79] Rinus Plasmeijer and Marko van Eekelen. *The Concurrent Clean Language Report*, December 2001. `http://clean.cs.ru.nl/`, date viewed: 25 April 2007.

[80] François Pottier and Nadji Gauthier. Polymorphic Typed Defunctionalization and Concretization. *Higher-Order and Symbolic Computation*, 19(1):125–162, 2006.

[81] Norman Ramsey and Mary F. Fernandez. Specifying Representations of Machine Instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, 1997.

[82] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, MA, USA, 1972.

[83] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a Verified, General-Purpose Operating System Kernel. In *NICTA FM Workshop on OS Verification*, pages 1–19, 2004.

[84] Emin Gün Sirer, Stefan Savage, Przemyslaw Pardyak, Greg P. DeFouw, Mary Ann Alapat, , and Brian Bershad. Writing an Operating System

with Modula-3. In *Proceedings of the First Workshop on Compiler Support for System Software*, pages 134–140, Tucson, AZ, USA, February 1996.

[85] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[86] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with Type Equality Coercions. In *The ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66, Nice, France, January 2007.

[87] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173, Santa Cruz, CA, USA, 1992.

[88] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compilation for Standard ML*. PhD thesis, Carnegie Mellon University, 1996.

[89] The GCC Team. GCC, the GNU Compiler Collection. `http://gcc.gnu.org`, date viewed: 25 April 2007, 2007.

[90] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[91] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed Language Interoperability via Source Translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[92] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Simple Unification-Based Type Inference for GADTs. Technical Report MS-CIS-05-2, University of Pennsylvania, April 2006.

[93] Philip Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the Fourteenth Symposium on Principles of Programming Languages*, pages 307–312, 1987.

[94] Philip Wadler. Linear Types Can Change the World! In *Working Conference on Programming Concepts and Methods*, pages 347–359, Sea of Galilee, Israel, 1990.

[95] Philip Wadler and Stephen Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the Sixteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76, Austin, TX, USA, January 1989.

[96] Philip Wadler and Peter Thiemann. The Marriage of Effects and Monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003.

[97] Malcolm Wallace. *Functional Programming and Embedded Systems*. PhD thesis, University of York, 1995.

[98] Malcolm Wallace and Colin Runciman. Lambdas in the Liftshaft: Functional Programming and an Embedded Architecture. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 249–258, La Jolla, CA, USA, 1995.

[99] Malcolm Wallace and Colin Runciman. The Bits Between the Lambdas: Binary Data in a Lazy Functional Language. In *Proceedings of the First International Symposium on Memory Management*, pages 107–117, Vancouver, British Columbia, Canada, 1998.

[100] Stephen Weeks. Whole-Program Compilation in MLton. Invited talk for the 2006 wokshop on ML. `http://mlton.org/`, date viewed: 25 April 2007.

[101] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, September 1998.

[102] Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.

[103] Zilog, Inc. *Z80-CPU, Z80A-CPU Technical Manual*, 1977. Information about the Z80 is also available from `http://www.z80.info/`, date viewed: 25 April 2007.

Biographical Note

Iavor Sotirov Diatchki was born on May 1, 1977, in Sofia, Bulgaria. In 1993 he moved with his family to Zimbabwe and later he attended the University of Cape Town in South Africa. His interests in mathematics and computer science lead him to pursue a double major in these subjects, and in 1999 he was awarded the degree of Bachelor of Science with Honours.

In 2000, Iavor moved to the United States to continue his education. He started his Doctoral studies at the (then) Oregon Graduate Institute, which later became the OGI School of Science & Engineering at OHSU. He has worked with Prof. Mark P. Jones on a number of topics related to programming language design and implementation.

In 2006, Iavor started a practical training program at Galois, Inc, a software development company based in Beaverton, Oregon. There he uses his experience with functional programming to develop high-assurance software.