High-level Views on Low-level Representations

Iavor S. Diatchki
OGI School of Sci. & Eng. at OHSU
diatchki@cse.ogi.edu

Mark P. Jones
OGI School of Sci. & Eng. at OHSU
mpj@cse.ogi.edu

Rebekah Leslie
Portland State University
rebekah@cs.pdx.edu

Abstract

This paper explains how the high-level treatment of datatypes in functional languages—using features like constructor functions and pattern matching—can be made to coexist with *bitdata*. We use this term to describe the bit-level representations of data that are required in the construction of many different applications, including operating systems, device drivers, and assemblers. We explain our approach as a combination of two language extensions, each of which could potentially be adapted to any modern functional language. The first adds simple and elegant constructs for *manipulating* raw bitfield values, while the second provides a view-like mechanism for defining distinct new bitdata types with fine-control over the underlying *representation*. Our design leverages polymorphic type inference, as well as techniques for improvement of qualified types, to track both the type and the width of bitdata structures. We have implemented our extensions in a small functional language interpreter, and used it to show that our approach can handle a wide range of practical bitdata types.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]: Data types and structures

General Terms Design, Languages

Keywords Data representation, bit manipulation, bitdata, bitfields, pattern matching, views, polymorphism, qualified types

1. Introduction

Algebraic datatypes promote a high-level view of data that hides low-level implementation details. Storage for new data structures is allocated automatically by applying constructor functions, while pattern matching provides a way to inspect values without concern for their machine-level representation. By abstracting from such details, we can obtain code that is more succinct and easier to reuse.

Many applications, however, require the use of 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 used by device drivers; and in programs like assemblers that work with machine code instruction encodings. We will refer to examples like this collectively as *bitdata*.

In this paper, 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 representations 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. Although there are occasional differences in syntax and semantics, the examples in this paper still follow the basic conventions of Haskell. 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. Our goal in this paper is to treat fixed-width bitdata types of the kind that can potentially be stored in a single machine register. This emphasis distinguishes our work from approaches, some of which will be described in Section 5, that focus instead on handling variable length streams of bitdata, as used in applications like multimedia and compression codecs, and machine language instruction stream encoders.

ICFP'05 September 26–28, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-064-7/05/0009…$5.00.

1.1 Examples of 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 application developer must conform*. For example, an operating system standard might fix the encoding for the set of flags that are passed to a system call, while the datasheet for a hardware device specifies the layout of the fields in each 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 might identify a particular opcode, while others specify the operands.

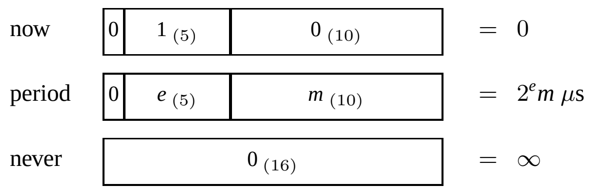
We will now describe a small collection of examples to illustrate some of the challenges of dealing with bitdata.

PCI Device Addresses PCI 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 are identified by a 16 bit address that consists of three 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 block diagram that specifies the name and width (as a subscript) of each field, from which we can infer the corresponding positions of each field.

bus $_{(8)}$	dev $_{(5)}$	fun $_{(3)}$

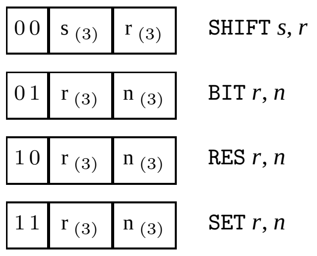
We draw diagrams like this with the most significant bit on the left and the least significant bit on the right. With this encoding, function 3 of device 6 on bus 1 is represented by the 16 bit value that is written `0x0133` in hexadecimal or as `00000001 00110 011` in binary, using spaces to show field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

Timeouts in the L4 Microkernel L4 was developed as a minimal, flexible, and high performance operating system kernel [13]. The most recent version of the L4 specification includes a detailed ABI (application binary interface) that describes the format for system call arguments and results [17]. 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 wait for a corresponding receive request in another process. Simplifying the details a little, there are three forms of timeout value, as shown in the following diagrams:



There are two special values here: A 'now' timeout 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 (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 (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 timeouts with the most significant bit set, which it uses to 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 as a separate type.

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 uses today as a low-cost microcontroller [19]. The Z80 has 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, the `0xCB` prefix byte signals that the next byte in the instruction stream should be interpreted as a bit manipulation instruction using one of four different formats:



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. Note 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.

1.2 The Perils of Bit Twiddling

From a high-level, it is clear that bitdata structures can have quite a lot in common with the 'sum-of-products' 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 use other bits for 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 overspecify and obfuscate the semantics of the operation that it implements. Our two examples show how a 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 too must rely on sophisticated optimization and instruction selection schemes to recover the intended semantics and, where possible, substitute more direct implementations.

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, place conceptually different types of value in different fields. Bit twiddling, however, usually bypasses this structure, treating all data homogeneously as some kind of machine word, with few safeguards to ensure that fields are accessed at the correct offset, with the correct mask, and with appropriately typed contents.

Some of the most widely used systems programming languages, notably C/C++ and Ada, do 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 make significant use of bitdata often define 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 specific bit twiddling operations. This approach also isolates portability concerns in a software layer that can be rewritten to target different compilers or platforms. In effect, this amounts to defining a simple, domain-specific language for each application—which can work quite well in a single program, but involves duplication of effort in identifying, implementing, and learning to use the set of abstractions that are provided. These problems can be addressed by designing a single domain-specific language that can be used across multiple applications. The PADS and SLED systems (described in Section 5) follow this approach using separate programming and data description languages. One of the goals of the current paper is to investigate a different point in the design space that treats data description as an integral part of the programming language by providing general and flexible programming constructs for working with bitdata.

1.3 Low-level Representations for Bitdata

In this section, we consider how issues of data representation should influence our design of bitdata language extensions. Suppose, for example, that we want to write programs that manipulate PCI addresses, as described in Section 1.1. If the language had already been extended with an appropriate collection of Int*n* types representing integers of different bit widths, then it would be possible to represent PCI addresses with a standard Haskell data type:

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

Ideally, we might hope that a ‘smart enough’ compiler would 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 every type—including PCIAddr as well as each of its component types Int8, Int5, and Int3—has an additional bottom element, $\perp$. 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 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, and, to the best of our knowledge, no existing Haskell (or ML) compiler even attempts it.

The representation that a compiler chooses for a given data type is usually only important when values of that type must be communicated with the outside world. Using either of the previous representations for PCIAddr, we could define functions like the following to marshal back and forth between external and internal representations of PCI addresses (we take some liberties with 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 a higher-level PCIAddr representation can be used exclusively in the remaining portions of the Haskell code. In theory, at least, if we follow this approach, then the details of the representation that the Haskell 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. 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 difficult to write functions like toPCIAddr and fromPCIAddr, it is tedious work, especially when dealing with more complex examples. It is also 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. The deriving mechanism of Haskell cannot be used for this purpose because standard datatype definitions do not provide information about layout. It is possible to generate the necessary code from specifications in a separate ‘interface definition language’ (IDL) that provides these details. Instead of using an external IDL, we extend the programming language to enable programmers to describe the layout of data directly. In this way the compiler can use the same underlying representation for both the internal and external forms of PCIAddr. The functions toPCIAddr and fromPCIAddr could be implemented as identity functions and their uses optimized away in the back end of the compiler.

Our conclusion from this discussion is that we are likely to incur significant overheads if the representation that is used inside the functional language differs from the representation that is used externally. Given that many bitdata formats are often determined by third parties, we cannot expect to change them to match the representations used by our compilers; instead, the only option is to change our compilers so that they will use the external representation directly. In this respect, we share the goal of *data-level interoperability* that guided the design of PADS [4] and of Blume’s foreign function interface for ML [3]. However, neither of these systems deals with the construction and matching of bit-level structures that motivates our work. We also consider that it is beneficial to maintain strong typing distinctions between different kinds of bitdata values. In this way, we are able to catch more type errors at compile-time while allowing compiler-generated analogues of the from and to functions to be used where necessary as explicit type conversions, without incurring runtime overhead.

1.4 Describing Bitdata Layout

We have concluded that our compilers will need to use the same external representation for bitdata as everybody else. But how will such a compiler determine what that external representation might be? Looking again at the examples in Section 1.1, it is easy to construct the following algebraic datatypes for the L4 time type and for the Z80 encoding of bit twiddling instructions:

```
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
```

However, it is also easy to see that there is not enough information in these definitions for a compiler, no matter how sophisticated it might be, to infer the corresponding external representations. For example, there are no indications in these definitions 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`, rather than `000`, should be used to represent `A`.

While a compiler might be able to infer the appropriate layout for `PCIAddr`, the general case requires a more expressive notation for specifying bitdata representations. One possibility would be to adopt a separate (and perhaps language-neutral) interface definition language (IDL) for describing low-level encodings of data. In this scenario, we would also need a compiler for compiling IDL descriptions into stub code that could be used to import the bitdata types and operations into the functional language as foreign types.

1.5 Our Approach

In this paper, we present a new approach to specifying and working with bitdata that is structured as two language extensions. These extensions allow programmers to capture essential details of low-level data formats directly within the functional language. This approach is attractive because it enables tighter integration of bitdata with the rest of the language than would be possible using an external IDL alone, including stronger type checking, the possibility of checking for non-exhaustive and/or overlapping pattern matches, and the potential for more aggressive and effective optimization.

Our first extension supports 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 interpreter 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.

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

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

In general, it is necessary to specify layout explicitly by annotating each constructor with an appropriate `as` clause. The following definition shows how `Time` 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 encoding of Z80 bit twiddling instructions can be described in a similar way. In this case, we 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 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}`—are treated as type errors, even where values might otherwise be confused because their representations have the same number of bits.

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. These are generalizations of the `toPCIAddr` and `fromPCIAddr` operations described in Section 1.3. 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"
```

The remaining sections of this paper present our approach in more detail, beginning with an overview of the language design, including its type system, in Section 2. An extended example, demonstrating some of the more advanced features of our design, is presented in Section 3. Details of our current implementation are described in Section 4, while Section 5 documents related work. We conclude with ideas for future work in Section 6.

2. Language design

This section provides a more thorough description of our design, covering features for bit manipulation in Section 2.1, and mechanisms for defining new bitdata types in Section 2.2. New syntax and typing rules are presented as we go along.

The core of our design is the small functional language described in Figure 1. However, our extensions are largely independent of the details of this language, and it should be quite easy to integrate them with other functional languages. As usual, a program consists of a number of (possibly recursive) declarations.

e	$=$	x	variable	κ	$=$	$* \mid \kappa \to \kappa$	kind
	$\mid$	$e\,e$	application	σ	$=$	$\forall\bar{\alpha}.\,\bar{\pi} \Rightarrow \tau$	scheme
	$\mid$	$e :: \tau$	type sig.	τ	$=$	$\tau\,\tau \mid \alpha \mid c$	type
	$\mid$	$\ldots$		π	$=$	$\ldots$	predicate

Figure 1. The core language

The type system is based on the Hindley-Milner system [14], extended with qualified types [8]. We use kinds (κ) to classify different types. Types that contain values are classified by $*$, and function kinds classify type constructors. The core language should have at least the following type constants:

Bool	:	$*$	Boolean value
$(\to)$	:	$* \to * \to *$	function space

We use schemes (σ) to type expressions that have multiple simple types (τ). Such expressions are said to be *polymorphic*. When a polymorphic expression appears in a particular context, the type variables (α) in the schema are instantiated to concrete types. In certain situations, it is convenient to impose restrictions on how type schemes may be instantiated. We do this by qualifying type schemes with *predicates* (π). An expression of a qualified type may only appear in contexts where the type variables are replaced by concrete types that satisfy the predicates. In the following sections, we present a set of rules for solving predicates. The rules are of the form $\Pi \vdash \pi$, which states that, from the set of assumptions Π, we can conclude that the predicate π holds.

2.1 Bit Manipulation

We introduce a new type constant called `Bit` that we use to type bit sequences. `Bit` is a type constructor that, given a natural number, produces the type of bit sequences of the corresponding length. To complete the definition of `Bit` we introduce a new kind $\mathbb{N}$, that is inhabited by the natural numbers.

Bit	:	$\mathbb{N} \to *$	bit sequence
$0, 1, 2, \ldots$	:	$\mathbb{N}$	natural number

For example, bytes are 8-bit sequences and have type `Bit 8`.

In this paper 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. An elegant way to achieve this, without compromising the generality of the system, is to use qualified types. We introduce a predicate, called *Width*, that can only be solved for natural numbers that are valid bit sequence widths.

$$\mathit{Width} : \mathbb{N} \to \mathit{Prop} \qquad\qquad \frac{n \leq \mathit{max}}{\Pi \vdash \mathit{Width}\ n}\ \text{[Width]}$$

The kind *Prop* classifies predicates. In the rule *Width*, Π is a set of assumptions and *n* is a natural number. The rule states that, to solve the predicate *Width n*, the number *n* should be smaller than *max*, which might be chosen as the size of the largest hardware register. In this way, implementations that are optimized to work on a particular machine will reject programs that attempt to create bit sequences that are too large. An implementation may instead allow the use of arbitrary (fixed-size) bit sequences by choosing to solve arbitrary *Width* predicates.

Now we can introduce standard operators on `Bit` values:

$$\begin{array}{lllll} (\&) & : & \forall \alpha. & \mathit{Width}\ \alpha & \Rightarrow & \mathit{Bit}\ \alpha \to \mathit{Bit}\ \alpha \to \mathit{Bit}\ \alpha \\ (==) & : & \forall \alpha. & \mathit{Width}\ \alpha & \Rightarrow & \mathit{Bit}\ \alpha \to \mathit{Bit}\ \alpha \to \mathit{Bool} \end{array}$$

Other common operations on bit sequences include logical and arithmetic operations (e.g., disjunction and addition), relational operations (e.g., equality tests and comparisons), and sequence manipulation (e.g., concatenation and splitting). Readers familiar with the Haskell class system may think of these constants as the methods of a class called *Width*. Furthermore, both signed and unsigned versions of `Int`, and the basic operations on these types, are easy to define using `Bit` types.

2.1.1 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. Leading zeros are important because they affect the type of the literal. Here are some examples of binary literals:

```
> :t B11
Bit 2
> :t B011
Bit 3
> :t B000000000000000000000000000000000
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.

Binary literals may be used in both expressions and patterns. Indeed, we can think of `Bit` *n* as an algebraic data type that has the *n*-digit binary literals as constructors. 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.

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 the design of Haskell [11]. The trick is to introduce a new primitive function *fromLit* : $\forall \alpha.\ \mathit{Width}\ \alpha \Rightarrow \mathit{Bit}\ \mathit{max} \to \mathit{Bit}\ \alpha$ (corresponding to `fromInteger` in Haskell). A numeric literal *n* in the text of a program, can then be treated as syntactic sugar for the constant `fromLit` applied to the value *n* of type `Bit` *max*. Bit sequences represent numbers using the standard two's complement encoding. 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 2
(Width a) => Bit a
> :t 2 & B11
Bit 2
```

Notice that, when used on its own, the number 2 has a polymorphic type—the system is telling us that 2 has type `Bit` *a* for any *a* that is a valid width. However, if used in a particular context, 2 will be converted to the correct length using the function `fromLit`.

2.1.2 Joining and Splitting Bit Sequences

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 to use an addition operator at the type level, and to use the type $(\#) : \forall \alpha\beta.\ \mathit{Bit}\ \alpha \to \mathit{Bit}\ \beta \to \mathit{Bit}\ (\alpha + \beta)$. At first, this seems quite attractive because it captures our intuition of what $(\#)$ does. However the situation is more complex than it appears, especially in a system that supports type inference. Having the $(+)$ operator at the type level makes it more difficult to decide if two types are the same: checking for structural equivalence is not sufficient. There are also situations in which it is not clear what type should be inferred. Consider, for example, the following definition:

```
mask x y = (x # y) & B101
```

This `mask` function may be applied to any two bit sequences whose lengths add up to 3—but we cannot express this using an addition operator at the type level. For this reason, we choose instead to introduce an addition predicate:

$$(\_ + \_ = \_) : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathit{Prop}$$

$$\frac{n_1 + n_2 = n_3 \quad \Pi \vdash \mathit{Width}\ n_i \quad i \in \{1,2,3\}}{\Pi \vdash n_1 + n_2 = n_3}\,[\mathit{Add}]$$

The predicate $n_1 + n_2 = n_3$ is a relation between natural numbers that holds when the sum of the first two numbers is the same as the third number. We also require that all the numbers are valid bit sequence widths. This trick of representing functions with relations should be very familiar to Prolog programmers and to users of Haskell's class system. Now we can give the join operator the following type:

$$(\#) : \forall\alpha\beta\gamma.\ (\alpha + \beta = \gamma) \Rightarrow \mathit{Bit}\ \alpha \to \mathit{Bit}\ \beta \to \mathit{Bit}\ \gamma$$

Here are some examples that use this operator:

```
> :t mask
(a + b = 3) => Bit a -> Bit b -> Bit 3
> show (B100 # B111)
"B100111"
```

The first line shows the type that the system inferred for `mask`. The second line shows the result of joining together two sequences.

We use pattern matching to split bit sequences. The split pattern has the form $p_1 \# p_2$. This pattern matches bit vectors whose most significant part matches p_1, and least significant part matches p_2. 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_1 and p_2. These types may be determined using type inference, or 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 we get the following type:

```
> :t upper
(a + b = c) => Bit c -> Bit a
```

An interesting point about this type is that the order of the variables in the predicate $a + b = c$ is important. If we were to switch a and b around we would get the type of the function that accesses the lower bits of a bit sequence.

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 omit the annotation on the `dev` pattern, the system fails with 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. We do not currently allow type variables in signature patterns but this restriction could be lifted using scoped type variables [12].

2.2 User-Defined Bitdata Types

In the context of systems programming, bit sequences 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. 2). 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 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 2.2.4). Instead, the type provides a kind of view [18] on the underlying bit sequences. We use constructors to construct and recognize bit sequences, while fields provide a means to access or update contiguous sub-sequences.

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 2. The syntax of user defined data types

To illustrate how `bitdata` declarations work, we present some definitions for a device driver for a NE2000 compatible network card [15]. The details of how the hardware works are not important; our goal is to illustrate the features of `bitdata` declarations. As a first example, consider a type that defines a number of commands:

```
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`. The `as` clauses specify a bit pattern for each constructor, which will be used to construct values and to recognize them in patterns. All of the constructors in a given `bitdata` declaration must have the same number of bits in their representation. The following examples use these constructors:

```
> :t Read
RemoteOp
> show Read
"B01"
> Read & B00
FAIL Type mismatch: RemoteOp vs. Bit 2
```

The last example emphasizes the point that, even though `Read` is represented with the bit sequence `01`, it is not of type `Bit 2`. There is a close relation between the bit sequence types `Bit` n and `bitdata` types like `RemoteOp`, captured by the following constants:

$$\begin{array}{lcllcl} \mathit{toBits} & : & \forall\alpha\beta. & \mathit{BitRep}\ \alpha\ \beta & \Rightarrow & \alpha \to \mathit{Bit}\ \beta \\ \mathit{fromBits} & : & \forall\alpha\beta. & \mathit{BitRep}\ \alpha\ \beta & \Rightarrow & \mathit{Bit}\ \beta \to \alpha \end{array}$$

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. We may think of `toBits` as a pretty-printer, and `fromBits` as a parser, that use bits instead of characters. These functions are very useful when a programmer needs to interact with the outside world. The function `toBits` is used when data is about to leave the system, and the function `fromBits` is used when data enters the system.

Not all types in the language may be converted to bit sequences. We use the predicate *BitRep* to restrict the contexts where the functions `toBits` and `fromBits` may appear:

$$\mathit{BitRep} : * \to \mathbb{N} \to \mathit{Prop} \qquad \frac{\Pi \vdash \mathit{Width}\ n}{\Pi \vdash \mathit{BitRep}\ (\mathit{Bit}\ n)\ n}\ [\text{Bit}]$$

The predicate *BitRep* τ n states that the type τ is represented with n bits. We can represent `Bit` n types in n bits, as long as n is a valid width, as described in the rule *Bit*. We may also solve *BitRep* predicates for types defined with `bitdata` declarations. These declarations introduce new assumptions to the system that enable us to solve the predicate directly by assumption. We can use the `:a` command to list the assumptions that are in scope:

```
> :a
BitRep RemoteOp 2
```

So far, we have defined only one type, so there is only one assumption. For readers familiar with Haskell we note that the automatic introduction of assumptions is like deriving an instance of the *BitRep* class for each bitdata declaration.

A critical design decision that shows up in the type of `fromBits` is that we do not include the possibility of failure: `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 wildcard 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` in which an exception is signaled when an unmatched bit pattern is passed in as an argument. We chose to use the unchecked semantics because it is simple, has practically no overhead (in `hobbit`, it is implemented as an identity function), and does not rely on support for exceptions in the core language. However, our design has all the machinery that would be needed to generate checked versions of `fromBits` or even to derive `isJunk` predicates that could be used to test for junk at runtime. Clearly, programmers must allow for the possibility that data values obtained from the "real-world" using `fromBits` may not be valid. The choice between the different designs that we have sketched here is about finding the most appropriate and/or convenient way to handle this. For `hobbit`, we have chosen an approach that requires programmers to deal with invalid data, where appropriate, by including equations with wildcards in function definitions, as in the code for `showOp`.

We expect that `fromBits` and `toBits` are inverses of each other, in the sense that the equation $\mathit{toBits}\ (\mathit{fromBits}\ n) = n$ holds. 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 $\mathit{fromBits}\ (\mathit{toBits}\ n) = n$ also holds. But what do we mean by equality in this case? We use operational equivalence—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: $x = y \equiv \mathit{toBits}\ x = \mathit{toBits}\ y$. Using this definition for equality, we can see that the second equation follows from the first.

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 `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`. Only types for which the *BitRep* predicate can be solved may be used in field declarations. The reason for this is that the fields become a part of the representation of the value, and so we need to be able to come up with a bit pattern for them. As we already saw, `RemoteOp` is a type defined with a `bitdata` declaration, and so the above declaration is valid.

To construct values with fields, programmers may use the notation $C\{l_i = e_i\}$, where C is the name of a constructor, l_i are its fields, and e_i are the values for the fields. The order of the fields is not significant. There is also a corresponding pattern $C\{l_i = p_i\}$ that may be used to check if a value matches the C constructor and the fields match the patterns p_i. The following definition is a function that will change remote read commands into remote write commands and leave all other DMA commands unchanged:

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

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.

2.2.1 The `as` Clause

The syntax of `as` clauses is more general than what we have seen so far. A layout specification 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 $C\{l_i = p_i\}$ proceeds in two phases: first, we see if the value is a valid C-value, and then we check that the listed fields match their corresponding patterns. 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`.

Wild-cards in the layout specifications represent 'don't care' bits. They 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 to make `toBits` a proper function. 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.

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. 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. Consider, for example, the definition:

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

This is legal, 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, like `MyBool` above.

2.2.2 Records

So far, we have seen how to access the fields of constructors using pattern matching. This approach is convenient in many situations but also has a drawback: if many functions need to access the fields of a value, each of them will have to pattern match to first ensure that the value has the expected format. Instead, we would like to have a mechanism that enables us to check that a value is of a particular form once, and then we should be able to simply access the fields without any additional overhead. To achieve this, `bitdata` declarations introduce a type constant for constructors that have fields. Consider, for example, a fragment of an encoding for the Z80 bit twiddling instructions:

```
bitdata Instr
 = LD { dst::Reg, src::Reg } as B01 # dst # src
 | HALT as 0x76
 ...
```

This definition introduces not just the type `Instr`, but also a type called `Instr.LD`. In general, the type of a constructor with fields C, in a `bitdata` declaration T, is $T.C \to T$. For example, `LD` is of type `Instr.LD` $\to$ `Instr`. Like other constructors, constructors with fields may be used in both patterns and expressions. A pattern of the form $C\ p$, will examine a value by using the tests derived from the `as` clause of C. If the tests succeed, the value will be promoted to type $T.C$, and then matched against the pattern p. Notice that the values of type T and $T.C$ have the exact same representation, but values of type $T.C$ are guaranteed to conform to the format specified by the constructor C. This is guaranteed because pattern matching is the only way to obtain a value of type $T.C$. In particular, there is no way to solve the *BitRep* predicate for $T.C$ types, and so we cannot use the function `fromBits` to create a $T.C$ value from raw bits.

The types $T.C$ are record types that contain the fields of the constructor. To capture this idea, we introduce another predicate:

$$(l :: \_) \in \_ : * \to * \to Prop$$

The predicate $(l :: \tau_1) \in \tau_2$ states that τ_2 is a record type that has a field of type τ_1. This idea is not new and has been used in the design of various record systems [7, 5]. The only way to solve such predicates is by assumption. These assumptions are introduced by `bitdata` declarations: one for each field of each constructor. For example, if we use the `:a` command to list all assumptions that are in scope for the types we defined so far, we get the following list: (our implementation prints $(l :: \tau_1) \in \tau_2$ as τ_2 *has* $l :: \tau_1$)

```
> :a
BitRep DMACmd 3
DMACmd.Remote has op :: RemoteOp
BitRep RemoteOp 2
BitRep MyBool 0
BitRep Instr 8
Instr.LD has dst :: Reg
Instr.LD has src :: Reg
BitRep Reg 3
```

This allows for the same field names to appear in different constructors, even if they belong to different `bitdata` declarations.

There are two families of constants, indexed by label names, that we use to manipulate records:

$$\begin{array}{llllll} access\ l & : & \forall\alpha\beta. & (l :: \alpha) \in \beta & \Rightarrow & \beta \to \alpha \\ update\ l & : & \forall\alpha\beta. & (l :: \alpha) \in \beta & \Rightarrow & \alpha \to \beta \to \beta \end{array}$$

The constant *access* l is used to get the field l of a record. A more concise notation for this operation is $e.l$, which is merely a short-hand for *access* $l\ e$. The constant *update* l is used to replace the value of the field l in a record. We have syntactic sugar for updates as well: $\{r \mid l_1 = e_1, l_2 = e_2\}$ is an abbreviation for *update* $l_2\ e_2$ (*update* $l_1\ e_1\ r$). Notice that this differs from Haskell's notation for manipulating records. As an example of how to use the record operations, consider defining a function that will set the source of the `LD` instruction to a particular register, and will leave other instructions unchanged:

```
setSrc (LD r) x = LD { r | src = x }
setSrc i _      = i

> :t setSrc
Instr -> Reg -> Instr
> show (LD { src = A, dst = B })
"B01000111"
> show (setSrc (LD { src = A, dst = B }) C)
"B01000001"
```

Another way to examine record values is to use *record patterns*. These patterns are of the form $\{p \mid l_i = p_i\}$. A value matches a record pattern if it matches the pattern p, and its fields l_i match the patterns p_i. For example, we may use a record pattern in combination with a constructor pattern to match on a constructor and a field, and at the same time to name the record of the constructor:

```
fromHL (LD { r | src = MemHL })   = r.dst
fromHL _                          = MemHL
```

2.2.3 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 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 [11] in any definition that pattern matches on `LD`. 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 (isMemHL src && isMemHL dst)
 | HALT as 0x76
 ...
instrName (LD _)  = "Load"
instrName HALT    = "Halt"
```

We 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 arise when the `if` clause is used during value construction. The cost of this choice is minimal because programmers may define 'smart constructors' to validate the fields before constructing a bitdata record.

2.2.4 Junk and Confusion!

Standard algebraic datatypes enjoy two important properties that are sometimes referred to as 'no junk' and 'no confusion' [6], 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. In more concrete terms, 'no junk' states that every bit pattern corresponds to some conceptual value, while 'no confusion' tells us that there is no overlap between the bit patterns for different constructors.

For bitdata types, we can only hope to avoid junk and confusion if the total number of representable values, N, is a power of two. In any other case, if $2^{n-1} < N < 2^n$, we need to use an encoding with at least n bits, and either accept some junk (i.e., some bit patterns that do not correspond to any of the N possible values), or else some the N values will be represented by more than one bit pattern.

In other cases, even when it is technically possible to avoid both junk and confusion, a designer might still opt for a representation that sacrifices one or both of these properties because it simplifies the tasks of encoding and decoding. The `Time` type in Section 1.5, for example, includes both junk (because the most significant bit can never be set) and confusion (because the `Now` and `Never` cases overlap with the `Period` case).

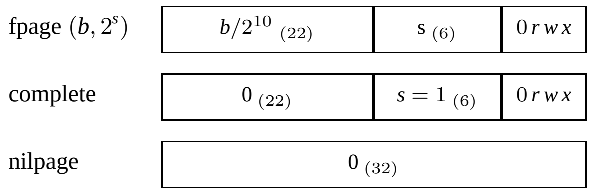
We cannot avoid the potential for junk and confusion in bitdata, but we can at least take steps to warn programmers about potential errors and pitfalls that they can cause. We have implemented a prototype static analysis for `hobbit` that captures the set of all possible bit patterns in each `bitdata` type using an ordered binary decision diagram (OBDD). Testing for junk in this setting amounts to testing the OBDD for the propositional constant *true*. Testing for confusion is accomplished by comparing pairs of OBDDs for individual constructors within a `bitdata` definition. The results of these tests are used to trigger appropriate warning diagnostics, and, from our experience to date, this seems to work well in practice.

It may also be possible to infer orderings between the representations of different constructors, and to use these results to check for non-exhaustive or overlapping pattern matches in arbitrary user-defined functions. This remains as a topic for future work.

3. Extended Example: Flexpages in L4

In this section, we present an extended example from L4 [17] to illustrate the use of our language on a real world problem.

Several operations in L4 manipulate regions of a process' virtual address space. The L4 specification introduces a type of *flexpages* to describe these regions in an architecture-independent manner.



In general, a flexpage contains a `base` address for the region of memory, a `size` equal to log_2(number of bytes in region), and a set of permissions that specify what operations can be performed on the region. The set of permissions contains three bits: read (`r`), write (`w`), and execute (`x`). Complete and nilpage values are special cases: 'complete' represents the entire virtual address space, and 'nilpage' represents an empty region of memory.

The size of the region described by a flexpage is restricted to whole numbers of physical pages on the target machine, and the starting address must be 2^s-aligned. It is up to a particular implementation to enforce this condition. For the IA32, the size of the flexpage must be greater than or equal to 12. We characterize flexpages using a type called `Fpage` and with permissions represented by the `Perms` type.

```
bitdata Fpage
  = Fpage { base::Bit 22, size::Bit 6, perms::Perms }
      as base # size # B0 # perms
      if (base `mod` (2 ^ size) == 0) && (size >= 12)
  | Complete { perms = nullPerms :: Perms }
      as 1 # B0 # perms
  | Nilpage as 0
bitdata Perms = Perms { r::Bit 1, w::Bit 1, x::Bit 1 }
nullPerms     = Perms { r = 0, w = 0, x = 0 }
```

The definitions of `Complete` and `Nilpage` follow directly from the layout in the specification. The representation of regular flexpages is more involved, because we capture the validity restrictions in the type. We use an `if` clause to guarantee that a flexpage will only match the `Fpage` constructor if its base field is aligned and its size is greater than or equal to 12.

Note there is junk in this type: an `Fpage` with an invalid size or incorrect alignment will not match any constructor. The L4 specification states that such a flexpage should be treated as a `Nilpage`. We capture this, and eliminate junk from the type, by reformulating the definition of `Nilpage` to catch invalid flexpages.

```
bitdata Fpage  = ... | Nilpage { bits = 0 :: Bit 32 }
```

While there is no longer junk in the type, we have introduced confusion: now `Nilpage` will match any flexpage, including a valid or complete one. To obtain the desired behavior for flexpages, we must take care when pattern matching on `Fpage` values: the `Nilpage` constructor should always come last. In addition, the representation of a `Nilpage` is no longer guaranteed to be zero. This is acceptable in the context of L4, but it is a choice that might not be right in all situations.

The discussion of the `Fpage` datatype exemplifies the reasoning behind some of our language design decisions, such as allowing junk and confusion. Of course it is undesirable to use a specification that includes invalid values and even worse to use an implementation that constructs such values. Unfortunately, these situations cannot always be avoided. An implementation of L4 must adhere to the specification, which contains datatypes, like flexpage, that can describe meaningless values. Often flexpages come from untrusted user-level programs, and there is no way to guarantee that these applications only construct valid values.

A frequent operation on a flexpage is the computation of the region's ending address, the `base` address plus the `size`. The types of `base` and `size` do not match, so the (+) operator, which has type *Bit n* $\rightarrow$ *Bit n* $\rightarrow$ *Bit n*, is insufficient. Zero-extending `size` to `Bit 22` before applying (+) rectifies this problem, but it is not an ideal solution. Adding quantities of different widths is a common occurrence, and manually zero-extending the smaller quantity is tedious and clutters the code. An alternative approach is to create a more polymorphic addition operation that automatically zero extends both arguments to the expected result type.

```
add    :: (b+a = d, e+c = d) => Bit a -> Bit c -> Bit d
add x y = (0 # x) + (0 # y)
```

We can use `add` without a type annotation whenever the width of the result type is evident from the context in which `add` appears. If the result type cannot be determined from the context, we must attach a type annotation to the expression containing `add`.

We use `add` to define a subset operation on flexpages. A flexpage `fp1` is a subset of another flexpage `fp2` if the region specified by `fp2` completely contains the region specified by `fp1`.

```
subset (Fpage fp1) (Fpage fp2)
  = (fp1.base >= fp2.base) && (end fp1 <= end fp2)
  where end fp = (add fp.base fp.size)::Bit 22
```

Only the case for normal, valid flexpages is shown. If the starting and ending addresses of the first argument, `fp1`, are both within the region of memory described by the second argument, `fp2`, then `fp1` is a subset of `fp2`. We check if `fp2` contains the starting address of `fp1` using a simple comparison of the `base` fields. Next, we check if the ending address for `fp1` is less than or equal to the ending address for `fp2`. We define a local function `end`, which calculates the ending address of a flexpage using `add`. In this case, the result type of `add` cannot be determined from the context alone, so we annotate the call to `add` with the desired result type.

4. Implementation

The ideas described in this paper are implemented in a prototype system written in Haskell. The system is about 3500 lines of code, approximately 1000 of which are in the parser. The implementation played an important part in the development of our ideas, as the process of building the prototype revealed a number of details that we had not noticed before. Having the prototype also enabled us to experiment with concrete examples, which revealed both problems with the design (that we then had to fix), and unexpected features, that we had not realized would be possible at first.

The implementation consists of four phases: parsing and static analysis in the front end, and a simplifier and interpreter in the back end. The simplifier eliminates patterns (by turning them into sequences of case statements and tests) and produces translated programs that can be executed using a standard interpreter for lambda-calculus with constants. We will therefore focus our attention in the rest of this section on the treatment of the type system, which is the most complex part of our implementation.

4.1 Type Inference

Type inference in our system is based on a modified version of Milner's algorithm *W* [14]. The modifications are to account for qualified types, and are described elsewhere [8]. In the absence of qualified types the two algorithms work in the same way. To implement qualified types we need a way to solve predicates. To do that we use the rules described in Section 2. We also add some *simplification rules* [9], described in Figure 3. They do not make the system any more expressive, in the sense that the same programs may be typed with or without these rules. The rules are useful, because they provide alternative ways to solve *Width* predicates: we can discharge a *Width* predicate even if the argument type is not a natural number. This makes it possible to infer simpler types. For example, using the rule *WRep* we can simplify the type $(\forall \alpha\beta.\ (\mathit{BitRep}\ \alpha\ \beta, \mathit{Width}\ \beta) \Rightarrow \tau)$ to $(\forall \alpha\beta.\ \mathit{BitRep}\ \alpha\ \beta \Rightarrow \tau)$

$$\frac{\Pi \vdash \tau_1 + \tau_2 = \tau_3 \quad i \in \{1,2,3\}}{\Pi \vdash \mathit{Width}\ \tau_i} \qquad \frac{\Pi \vdash \mathit{BitRep}\ \tau_1\ \tau_2}{\Pi \vdash \mathit{Width}\ \tau_2}\,[\text{WRep}]$$

Figure 3. Simplification rules

Readers familiar with the Haskell class system may think of these simplification rules as specifying that *Width* is a super-class of the *BitRep* and $(\_ + \_ = \_)$ classes.

4.2 Evidence for Predicates

As we have already seen, in the presence of qualified types, the system needs to solve predicates to make sure that type schemes may be instantiated at the given concrete types. A common way to implement such systems is to have a predicate solver that constructs *proof objects* as evidence that a predicate is true. We can think of a value that has a qualified type $\pi \Rightarrow \tau$ as a function that takes the evidence that the predicate π holds as an argument, and then produces a result of type τ. While the system performs type inference, it automatically constructs evidence for predicates, and applies values of qualified types to the evidence they need.

In our system, we have four different predicate forms. The evidence for each of these depends on the concrete data representation chosen in a particular implementation. The evidence we chose for our system is described in Table 1. It is quite general and can accommodate different data representations. We chose to implement all bit values with 32 bit words. However there are other options: for example, we could have implemented all values with up to 8 bits as bytes, values with between 8 and 16 bits as words, and values with between 16 and 32 bits as long words. There are also implementation choices that can use simpler evidence, based on particular assumptions about how data is represented.

Predicate form	Evidence
$(x :: \tau_1) \in \tau_2$	{offset :: Int, width :: Int}
Width τ	{width :: Int }
BitRep τ_1 τ_2	{width :: Int }
$\tau_1 + \tau_2 = \tau_3$	{upper :: Int, lower :: Int}

Table 1. Evidence for predicates

The evidence for record manipulation tells us the offset and number of bits that we need to access. The evidence for *Width* and *BitRep* tells us how many bits are necessary to represent some value. Addition predicates are used when we split or join bit vectors together. The evidence tells us the widths of τ_1 (the upper or most significant bits) and τ_2 (the lower or least significant bits).

4.3 Improvement

Improvement [9] is a technique for inferring types from predicates. Given a set of predicates, we examine them and compute an *improving substitution*. The substitution does not change the set of predicates, but can replace some of the type variables with concrete types. This in turn may enable the system to solve some of the predicates, which could not be solved before because the types were unknown. A popular language feature based on improvement is the addition of *functional dependencies*[10] to the Haskell class system. Figure 4 shows the functional dependencies that we used in our type checker as well as some additional rules for improvements that cannot be captured using functional dependencies.

$$
\begin{array}{lcl}
\alpha + \beta = \gamma & | & (\alpha, \beta) \rightsquigarrow \gamma,\ (\alpha, \gamma) \rightsquigarrow \beta, \\
 & & (\beta, \gamma) \rightsquigarrow \alpha \\
(l :: \alpha) \in \beta & | & \beta \rightsquigarrow \alpha \\
\mathit{BitRep}\ \alpha\ \beta & | & \alpha \rightsquigarrow \beta \\
 & & \\
\mathit{impr}\{0 + \tau_1 = \tau_2\} & = & \mathit{mgu}\ \tau_1\ \tau_2 \\
\mathit{impr}\{\tau_1 + 0 = \tau_2\} & = & \mathit{mgu}\ \tau_1\ \tau_2 \\
\mathit{impr}\{\tau_1 + \tau_2 = 0\} & = & \mathit{mgu}\ \tau_1\ 0 \circ \mathit{mgu}\ \tau_2\ 0 \\
\mathit{impr}\{\tau_1 + \tau_2 = \tau_1\} & = & \mathit{mgu}\ \tau_2\ 0 \\
\mathit{impr}\{\tau_1 + \tau_2 = \tau_2\} & = & \mathit{mgu}\ \tau_1\ 0 \\
\mathit{impr}\{\mathit{BitRep}\ (\mathit{Bit}\ \tau_1)\ \tau_2\} & = & \mathit{mgu}\ \tau_1\ \tau_2
\end{array}
$$

Figure 4. Functional Dependencies and Improvement Rules

To see how improvement works in practice, consider the expression `toBits B00`. The constant `B00` is of type *Bit* 2, and the constant `toBits` is of type $\forall\alpha.\ \mathit{BitRep}\ \alpha\ \beta \Rightarrow \alpha \rightarrow \mathit{Bit}\ \beta$. The type inference algorithm will infer the type $\mathit{Bit}\ \beta$, provided that the system can solve the predicate $\mathit{BitRep}\ (\mathit{Bit}\ 2)\ \beta$. Notice that we cannot use any of the rules to solve this predicate because we do not know what type β stands for. One possibility is to infer the type $\forall\beta.\ \mathit{BitRep}\ (\mathit{Bit}\ 2)\ \beta \Rightarrow \mathit{Bit}\ \beta$. This type is not wrong, but is unnecessarily complicated—we know that the type *Bit* 2 is represented with 2 bits. To get a more intuitive type, we use the improvement rules and determine that β is not a free type variable, but should be replaced by the type 2. Then we can use the rule *Bit* to discharge the predicate, and infer the expected type *Bit* 2.

5. Related Work

Cryptol Our use of the `(#)` notation comes from Cryptol, and our `Bit` types are a special case of Cryptol's more general sequences: for example, `Bit 32` in our system corresponds to the `[32] Bit` type in Cryptol. However, our approach differs from Cryptol because we have a different application domain in mind: we are interested in helping programmers develop systems software. The operations that are common in that setting are not based on sequence manipulation (which is one of Cryptol's main strengths) but are much more like operations on ordinary data types, which is what our design introduces. The work on Cryptol is orthogonal to our own—in principle, we could add more operators for sequence manipulation.

Bluespec Bluespec is a language for programming FPGAs [1]. There does not seem to be a freely available implementation that we could experiment with, but the language specification describes a number of ideas similar to ours. Bluespec supports bit vectors, much like the ones we described, but there are no operations to join or split bit vectors. Bluespec supports user defined data types and records, which provides convenient ways to pattern match on data, and to access fields in a record. However, because Bluespec's main goal is to program FPGAs, there is no need for the data types to conform to a particular predefined ABI. Bluespec does not allow programmers to specify explicit data layouts, and instead leaves the compiler to derive bit representations for data automatically in a predefined manner.

SLED The Specification Language for Encoding and Decoding (SLED) is a domain specific language for working with streams of machine language instructions [16]. Encoding and decoding machine instructions requires bit manipulation, so it is interesting to compare SLED's design to our own. An important difference is that SLED processes streams of instructions, while our approach deals only with fixed-width values. In our design, obtaining the data from a buffer or a device is separate from decoding and processing the data, while in SLED these two concepts are unified in a single `match` construct. Another difference is that SLED is a specification language that (at least in principle) is independent of the language in which the application is written, while with bitdata the specification and the application are written in the same language. The benefit of the SLED approach is that it is not necessary to extend the host language in which the application is written. The benefit of our approach is that we can obtain stronger static checking, and potentially gain some opportunities for optimization. The reason for this is that a language independent preprocessor can only take into consideration the SLED parts of a program, and the language specific tools only get to process a translated specification. Using a single language does not have these drawbacks and, in addition, enables us to have more accurate types.

From a language perspective, there are interesting differences between the mechanisms that SLED provides for specifying bit-level representations, and those that are provided by our bitdata definitions. In SLED, the position of a field within a bitdata structure is specified by giving a range of bit positions, and a language of patterns—including simple range/equality constraints on individual fields as well as constructs for conjunction, disjunction, and sequencing of patterns—is used to describe binary representations. This is strictly more expressive than our approach, although we have found that the layout specifications ("as" clauses) in our bitdata definitions provide a convenient and flexible way to accomplish the same thing in many practical examples. SLED also allows the use of equations that relate constructor values with bit fields using a small language that includes linear arithmetic as well as a few specialized operators such as sign extension. There is no corresponding feature in our system, but we have certainly encountered situations where this would be useful as a way to allow the data that is seen through bitdata views to be *computed* rather than simply *extracted* directly from the underlying bitstream.

PADS PADS is a declarative data description language [4] for processing large streams of data from ad hoc sources, such as web server logs. This is a separate problem from our goal of providing tools that enable easy manipulation of fixed size machine registers, but there are still some similarities with our work. In both cases, for example, the programmer describes data in a high level language, rather than directly writing parsers and pretty printers by hand. A PADS specification describes the physical layout and semantic properties of a data source, so it is analogous to a bitdata declaration, but on a different scale. Like ours and many other designs, a PADS specification can contain unions (sums), structs (products), types, and constraints.

DataScript DataScript is a language that uses types to describe the physical layout of binary file formats [2]. DataScript provides primitive types, set types, variants (sums), records (products), and arrays. Set types include bitmask types, which enable programmers to describe enumerations with specific bit representations in a fashion similar to bitdata. An alternative in a sum type is chosen based on constraints written in a specification. DataScript specifications are language independent, so processing a specification with different language bindings can produce code for different languages. In this respect it resembles the input to parser generator tools like Yacc. The exact representation of the parsed files depends on the particular language binding. For example, in one of the Java bindings [2] arrays are represented as lists. Unlike bitdata, DataScript does not provide mechanisms for defining functions by pattern matching; DataScript's design is based on the Java language, which does not support this feature.

Views Views [18] 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 (in fact, the proposed compilation scheme advocates that these values are never constructed at all).

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 which 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 marshaling functions. The application of these functions imposes no performance penalty because they are the identity, where as view transformation functions may perform arbitrary computation.

6. Future Work

Beyond registers In this paper we focused on data whose representation fits in the registers of a machine. This is sufficient for many systems level programming tasks, but not all. For example, when implementing a network protocol stack, programmers often think of a sequence of bytes as having some particular structure. It seems plausible that our ideas may be extended to handle such larger data items. However there are important details that need to be worked out, mostly to do with representing data in memory. For example, we need to be careful about the order in which bytes are stored in memory. Other important details include working with references, and interactions with automatic garbage collection.

Optimizations As we already discussed, we have a working implementation of all the ideas in the paper. A component that is missing from the implementation is a good optimizer to get efficient bit-twiddling code. Using the higher-level constructs we described should make it easier for a compiler to spot opportunities for optimization. It would also be useful to consider optimizations to our implementation of pattern matching: we would like to examine the patterns occurring in the equations of a function, and combine them into an efficient test on values. One promising approach is to use binary decision diagrams (BDDs) to calculate minimal representations for groups of patterns.

More static checking As we discussed in Section 2.2.4, programmers may define types with both junk and confusion. In some circumstances this is inevitable because a program has to conform to a predefined ABI, but an implementation may still warn programmers about such anomalies. To achieve this, it would be useful to develop algorithms that can detect overlapping or potentially redundant pattern matches in function definitions. It seems that this problem is related to pattern matching compilation, and we expect that the two problems may have similar solutions.

Generalizing data types Several features of `bitdata` declarations are orthogonal to bit manipulation, and hence may have more general interest. Allowing record declarations to contain default values makes sense in any language that supports records. Similarly, `if` clauses are not specific to `bitdata` and might also be useful in `data` declarations. A final observation idea is that `as` clauses are also not specific to bitdata. Instead of specifying that a value should be represented with a particular bit pattern, `as` clauses could specify a value of a different type to represent the constructor.

Acknowledgments

This work was supported, in part, by the National Science Foundation award number 0205737, "ITR: Advanced Programming Languages for Embedded Systems." We would especially like to thank Norman Ramsey, Andrew Tolmach, Thomas Hallgren, and the anonymous reviewers for their comments and suggestions.

References

[1] Lennart Augustsson, Jacob Schwartz, and Rishiyur S. Nikhil. *Bluespec Language definition*. Sandburst Corporation, 2002.

[2] Godmar Back. Datascript - a specification and scripting language for binary data. In *Proceedings of the ACM Conference on Generative Programming and Component Engineering Proceedings (GPCE 2002)*, pages 66–77, October 2002.

[3] Matthias Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01: First workshop on multi-language infrastructure and interoperability*, September 2001.

[4] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 295–304, 2005.

[5] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical report, University of Nottingham, 1996.

[6] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24(1):68–95, 1997.

[7] 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.

[8] Mark P. Jones. *Qualified Types Theory and Practice*. Cambridge University Press, 1994.

[9] Mark P. Jones. Simplifying and improving qualified types. Technical Report YALEU/DCS/RR-1040, Yale University, New Haven, Connecticut, USA, June 1994.

[10] Mark P. Jones. Type classes with functional dependencies. In *ESOP 2000: European Symposium on Programming*, March 2000.

[11] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[12] Simon Peyton Jones and Mark Shields. Lexically scoped type variables. March 2004.

[13] Jochen Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles*, December 1995.

[14] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[15] National Semiconductor. *DP8390D/NS32490 NIC Network Interface Controller*, July 1995.

[16] Norman Ramsey and Mary F. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, 1997.

[17] L4ka Team. *L4 eXperimental Kernel Reference Manual*, January 2005. Available online from `http://l4ka.org/`.

[18] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages*.

[19] Zilog, Inc. *Z80-CPU, Z80A-CPU Technical Manual*, 1977. Information about the Z80 is also available from `www.z80.info`.