Strongly Typed Memory Areas

Programming Systems-Level Data Structures in a Functional Language

Iavor S. Diatchki
OGI School of Science & Engineering
Oregon Health & Science University
Portland, Oregon, USA
diatchki@csee.ogi.edu

Mark P. Jones
Department of Computer Science
Portland State University
Portland, Oregon, USA
mpj@cs.pdx.edu

Abstract

Modern functional languages offer several attractive features to support development of reliable and secure software. However, in our efforts to use Haskell for systems programming tasks—including device driver and operating system construction—we have also encountered some significant gaps in functionality. As a result, we have been forced, either to code some non-trivial components in more traditional but unsafe languages like C or assembler, or else to adopt aspects of the foreign function interface that compromise on strong typing and type safety.

In this paper, we describe how we have filled one of these gaps by extending a Haskell-like language with facilities for working directly with low-level, memory-based data structures. Using this extension, we are able to program a wide range of examples, including hardware interfaces, kernel data structures, and operating system APIs. Our design allows us to address concerns about representation, alignment, and placement (in virtual or physical address spaces) that are critical in some systems applications, but clearly beyond the scope of most existing functional languages.

Our approach leverages type system features that are well-known and widely supported in existing Haskell implementations, including kinds, multiple parameter type classes, functional dependencies, and improvement. One interesting feature is the use of a syntactic abbreviation that makes it easy to define and work with functions at the type level.

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, memory areas, memory manipulation, systems programming, qualified types, improvement

1. Introduction

Many of the computers in use today are hidden in embedded systems where they provide functionality for a wide range of devices, from household appliances, to safety and security critical applications such as vehicle navigation and control, 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. As such, it is disappointing that industry still relies quite heavily on older, less robust languages, or even on lower-level assembly code programming.

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 of systems development. Other projects have already explored the potential for using higher-level languages for lower-level programming: a small sample includes the Fox Project [10], Ensemble [19], Cyclone [12], and Timber [15], for example. Based on these, and on our own experience using Haskell [16] to develop device drivers and an operating system kernel [9], 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.

In our previous work [4], we focused on *bitdata* (i.e., bit-level data structures that can fit in a single machine word or register). Bitdata values are widely used in systems software, for example, to describe the values stored in a device control register or the flags passed to an operating system call. Of course, it is possible to manipulate such values using standard bit-twiddling techniques, without special language support. The resulting code, however, can be hard to read and error prone: we know (first-hand!) that it is very easy to specify the wrong number of bits for a shift operation, or to give the wrong mask for a bitwise 'and', and that the resulting bugs can be subtle and hard to find. Moreover, when we reduce all bitdata to the lowest-common denominator of a single machine word, we also lose the ability to catch errors like these using compile-time type checking. To address these issues, we designed and implemented language features that allow programmers to define strongly typed, high-level views, comparable to programming with algebraic datatypes, on the underlying bitdata structures. A critical detail in making this work is the ability to specify bit-level layout and representation information precisely and explicitly; this is important because the encodings and representations that are used for bitdata are often determined by third-party specifications and standards that must be carefully followed by application programmers and language implementations.

Haskell'06 September 17, 2006, Portland, Oregon, USA.

In this paper, we describe new language extensions that provide direct support for manipulating memory-based data structures. Our approach does not require a new type system, but instead leverages the support for kinds, qualified types and improvement that is already provided in existing Haskell implementations. Although some syntactic extensions are used to allow programmers to describe and reserve storage for memory-based data structures, most of the features that we need are provided by a collection of new built-in functions, types, and predicate/type class symbols. In essence, our goal is to provide the same levels of flexibility and strong typing for byte-oriented data structures in memory as our bitdata work provides for bit-oriented data structures in registers. Given that analogy, one might expect us to refer to these memory structures as 'bytedata'. However, because we want to focus on higher-level views rather than the underlying byte values, we will instead call them 'memory areas', or simply 'areas'.

1.1 Characteristics of Memory Areas

We have found a wide range of uses of memory areas in our explorations of systems-level programming. It is instructive to describe some of these examples, and to emphasize their markedly different character to the list- and tree-like data structures that are common in traditional functional programming. Specific details of these applications, however, will not be assumed in the rest of the paper.

One notable feature is that many examples are specific to a particular processor architecture, hardware device, or OS kernel. Examples from the Intel IA32 processor family [11] include: page directories and page tables; interrupt and segment descriptor tables; the task state segment, which contains data used to support hardware-based multitasking; and the exception frame where user registers, like the stack and instruction pointer, are saved when an interrupt occurs. Examples for the L4 microkernel [25] include: the kernel information page (KIP) that is mapped into every address space, and the user-space thread control blocks (UTCBs) that are used to communicate values to and from the kernel. Similar examples can be found for other processor architectures and for other operating systems, both inside the kernel implementation, and outside in the interfaces that the kernel presents to user processes.

These memory area structures have fixed sizes, rigidly defined formats/representations, and are often subject to restrictions on the addresses at which they are stored. An IA32 page table, for example, 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 (i.e., by a 32 bit address in which the least significant 12 bits are zero). 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).

1.2 Example: Video RAM

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 in Haskell. This is a particularly easy device to work with because it can be programmed simply 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 Haskell because it does not include mechanisms for identifying or writing to physical addresses.

Instead, if we want to code or use operations like this from a functional program, then we will typically require the use of a foreign function interface [24, 2]. For example, 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. But now 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.

1.3 This Paper: Strongly Typed Memory Areas

In this paper, 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 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 easily detected and subjected to the appropriate level of scrutiny and review.

Using the language features described in the rest of this paper, 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 ScreenChar)
area videoRAM = 0xb8000 :: Ref Screen
```

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 `videoRAM` 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 `videoRAM` 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 (a) using the native representations that are required by the video hardware and (b) 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 the ideas introduced in this paper to implement code for clearing the video screen:

```
cls = forEachIx (λ i →
       forEachIx (λ j →
         writeRef (videoRAM @ 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.

1.4 Paper Outline

The following outline summarizes the topics that are covered in the rest of this paper, and also includes the relevant section numbers.

- The key features in the design of our language extension for memory areas is introduced in Section 2; this includes details of the types that are used to describe memory areas, and of the operations that are defined in each case.
- Many of the areas that we encounter in practice are actually structured as tables or arrays; Section 3 describes the mechanisms that we provide to allow efficient and safe access to the elements of these structures.
- As we have already seen, it is sometimes necessary to ensure that the starting address of a memory area is aligned on a particular byte boundary. We deal with this in Section 4 by including alignment information in pointer and reference types. The resulting system allows us to enforce alignment constraints using compile-time type checking.
- Storage for memory areas is introduced using the **area** declaration described in Section 5. Using type information, the compiler can determine the amount of space, the alignment constraints, and the start address of each memory area that is declared. It is also possible to specify an explicit *region* as part of an **area** declaration, which will be used at link-time to constrain or guide the start address of the area to a particular address or range of addresses. This is the only form of **area** declaration that can compromise safety. Fortunately, however, it is easy to identify and extract all of the declarations of this form from a given program so that they can be checked carefully for possible errors.
- Although we have a strong preference for strongly typed memory areas, there are situations when it is useful to be able to 'cast' between different region types. Section 6 describes the operations that we provide to support this, and the limitations that we impose to ensure that the conversions are safe.
- One interesting feature of our design is the use of an abbreviation mechanism for describing functions on types. Section 7 explains how this relates to the recent proposal for associated type synonyms [3].
- In the closing sections of the paper, we provide brief comments about the structure of our prototype implementation (Section 8); a survey of related work in areas of language design, foreign function interfaces, and interface definition languages (Section 9); a suggestion for some future work (Section 10); and a conclusion to summarize our contributions (Section 11).

2. Overview: Language Design

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 this section, we describe a collection of primitive types that can be used to describe memory areas, and a corresponding collection of primitive operations. We begin by describing some preliminary details of our type system (Section 2.1), and then we introduce types for describing references (Section 2.2), representations for stored values (Section 2.3), pointers (Section 2.4), arrays (Section 2.5), and structures (Section 2.6).

2.1 Preliminaries

Our design assumes a fairly standard, Haskell-like type system that uses *kinds* to classify different types, and *predicates*, much like Haskell's type classes, to capture relationships between types. Kind and type inference for such a system does not require any significant new work, and can be implemented using the same, well-developed machinery that is included in existing Haskell compilers.

Kinds. The syntax of kinds, represented here by κ, is as follows:

κ `::= * | Area | Nat |` $\kappa \to \kappa$ `| Pred`

The kind `*` classifies the standard types in a functional languages, whose values are first-class and can be passed freely as function arguments or results. A distinct kind, `Area` is needed to classify memory areas because 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. Function kinds of the form $\kappa \to \kappa$ are used to classify type constructors. For example, the pointer type, `Ptr`, introduced below, has kind `Area` $\to$ `*`, which tells us that, if `a` is a valid area type then, `Ptr a` is a well-formed type of kind `*`. We also include a kind `Nat`, whose elements are written as natural numbers: `0`, `1`, `2`, and so on. This is essentially the same framework that we used in our work on bitdata [4]. For example, we include a type constructor `Bit` of kind

`Nat → *`, and write `Bit n` for the type containing all bit vectors of length `n`.

Predicates. Our system also includes a kind `Pred` that is used to classify predicates on types. The standard Haskell type class `Eq`, for example, corresponds to a type constructor of kind `* → Pred`. In addition, we will also use several predicates to describe arithmetic relations between the different types of kind `Nat`:

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

For example, as the notation suggests, a predicate of the form `x+y=z` is an assertion that the sum of the numbers corresponding to the natural number types `x` and `y` is equal to the number corresponding to `z`. In practice, of course, any two of the three variables here will uniquely determine the third. We can represent this with a collection of functional dependencies, { (`x y ⇝ z`), (`y z ⇝ x`), (`z x ⇝ y`) } [14] and then use improvement [13] to instantiate and simplify many of the addition predicates that are generated during type inference. For example, if we infer a predicate of the form `2+3=z`, then we can use improvement to determine that `z` must be `5`. Conversely, given a predicate of the form `3+y=2`, we can use improvement to infer that there are no solutions for `y` and then abort the type inference process with an appropriate error. Clearly, we can also do similar things with multiplication (`x*y = z`), greatest common divisor (`GCD x y = z`) and power of two (`2^x = y`) predicates.

Syntactic Abbreviations for Functions on Types. Although we have formalized the operations on `Nat` as relations, it is often convenient to think of them as functions. For example, in our original work on bitdata, we defined the following operation for concatenating bit vectors:

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

While this gives a general polymorphic type for `(#)`, the notation is a little unwieldy and it might seem more natural to write the following type instead:

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

Attractive as this might appear, the introduction of an associative, commutative function into the type language makes type inference much more complicated. In addition, 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 type for `(#)` 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`) at the front of the type. Using this translation, for example, 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 predicate forms described previously. In fact, we can generalize this idea even further. 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 sections, and we expect that it will prove to be useful in other applications beyond the scope of this paper. We discuss this further, particularly with respect to the recent proposal for 'associated type synonyms', in Section 7. Our approach is also very similar to the 'functional notation for functional dependencies' suggested by Neubauer *et al.* [20], but it is slightly 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.

2.2 References

A *reference* is the address of a memory area, and reference types are introduced using the following primitive type constructor:

```
Ref :: Area → *
instance Eq (Ref a)
```

For example, if `T` is a description of a memory area (i.e., it is of kind `Area`), then `Ref T` (of kind `*`) is an address in memory where such an area is stored. 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&`) [22], and Cyclone's `notnull` pointers [12].

2.3 Stored Values

In our previous work on bitdata [4], we described a mechanism for specifying and working with types that have explicit bit-pattern representations. We would like to use such types as 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 layout for multi-byte values. To account for this we provide two type constructors that create basic `Area` types:

```
LE, BE :: * → Area
```

The constructor `LE` is used for little endian (least significant byte first) encoding, and `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.

To manipulate memory areas that contain stored values we use the class `ValIn`, which relates memory areas to the type of value that they contain:

```
class ValIn r t | r → t where
  readRef  :: Ref r → IO t
  writeRef :: Ref r → t → IO ()
```

The predicate `ValIn r t` asserts that the memory area `r` contains an abstract value of type `t`. The operations of the class are used to read and write the stored values. Depending on the target architecture and the particular encoding used for a memory area, an implementation should generate instructions to perform the necessary conversions.

Note that because the type `t` is uniquely determined by the representation type `r`, we can use the syntactic abbreviation notation

introduced previously and use `ValIn` in type expressions as a partial function of kind `Area → *`. For example, we could write the type of `readRef` like this:

```
readRef :: Ref a → IO (ValIn a)
```

Our system provides instances of the `ValIn` class for the little and big endian encodings of bit-vectors and **bitdata** types with sizes that are a multiple of 8.

The types of `readRef` and `writeRef` include a monad, `IO`, that encapsulates the underlying memory state. For the purposes of this paper, we need not worry about the specific monad that is used. For example, we might replace `IO` with the standard `ST` monad, while in House [9], we might use the `H` (hardware) monad.

2.4 Pointers

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 can think of pointers as an algebraic data type:

```
data Ptr a = Null | Ref (Ref a) deriving Eq
```

Note that in general implementations have to treat the `Ptr` type specially, because we allow pointers to be stored in memory areas, and so they need to have a known representation. In our implementation we follow the standard convention of representing `Null` with 0, and other pointers with a non-zero address.

2.5 Arrays

Memory-based array or table structures can be described using the following type constructor:

```
Array :: Nat → Area → Area
```

The type `Array n t` describes a memory area that has `n` adjacent `t` areas. We can obtain a reference to an element of an array using the function `(@)`.

```
(@) :: SizeOf a b ⇒ Ref (Array n a) → Ix n → Ref a
```

This operation performs pointer arithmetic to compute the offset of an element. The type `Ix n` describes the set of valid array indexes, and enables us to avoid run-time bounds checking. The details of how this works are described in Section 3. The predicate `SizeOf` indicates that the indexing function needs to know the size of the elements in the array; this is discussed in Section 5.

2.6 Structures

Another way to introduce types of kind `Area` is by writing structure declarations, each of which begins with the keyword **struct**, specifies a name and optional parameters for the new type, and lists a number of fields and constraints on the type. Figure 1 shows the syntax of the different kinds of fields that can be used. The list of all fields determines the layout of the corresponding memory area: The first field is placed at the lowest memory addresses, and subsequent fields follow at increasing addresses.

field	=	*label* :: *type*	Labelled field
	|	..	Computed padding
	|	*type*	Anonymous field (padding)

Figure 1. Syntax of fields

As an example, the following definition describes the static part of the header of an IP4 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
```

It is particularly important in this application to specify that multi-byte values are stored using big-endian representation to ensure `IP4StaticHeader` structures are interpreted correctly, even on platforms where the default encoding is little-endian; we accomplish this here by using `BE` instead of `Stored`.

Labelled fields are the most conventional form of field. They define an offset into the structure that contains a region of memory described by the type of the field. To access the fields of a structure, we reuse the family of selector operators (one for each label) from our work on bitdata [4]:

```
class Has_l a t where
  (.l) :: t → a
```

To make this possible, structure declarations introduce rules that are used to discharge the `Has` constraints. The rule for a field `l :: a` in a struct `T b` is of the form: `... ⇒ Has_l (Ref a) (Ref (T b))`. Thus, if we have a reference `r` to some structure `T b`, then `r.l` is a reference to a piece of memory of type `a` that corresponds to the field `l` of `r`. The `...` in the axiom contains assumptions about the type parameters of the structure that enable us to verify that we have a well-defined structure and to compute the offset of the field.

The other fields that may appear in a structure provide various ways to specify padding. By 'padding', we mean a part of the structure that takes up space, even though the programmer does not intend to use it. Most commonly, padding is used to conform to an external specification, or to satisfy alignment constraints for the fields in 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 have a fixed size but do not use all of that 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 determine the sizes of 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 of size 4K where
  nextFree :: Stored (Ptr FreePage)
  ..
```

This example illustrates a few new pieces of concrete syntax: (1) The optional **of** `size t` part of a **struct** declaration 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) The field declarations follow the keyword **where**, and follow the usual Haskell convention for declarations: they can either be explicitly separated with semicolons, or (like the example above) use layout to reduce clutter in the code. (4) The `..` in the above example is concrete syntax for a computed padding field and is not a meta-notation.

3. Accessing Array Elements

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 :: Nat → *
```

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.) There is an instance of `ValIn` for `Ix` types, so that we can store them in memory.

3.1 Basic Indexes

We can use the following operations to work with `Ix` values:

```
toIx    :: Index n ⇒ Int → Ix n
fromIx  :: Ix n → Int
minIx   :: Index n ⇒ Ix n
maxIx   :: Index n ⇒ Ix n
bitIx   :: (2^m = n, Index n) ⇒ Bit m → Ix n
```

The function `toIx` enables us to treat integers as indexes. As with other integral types, index literals have a modulo arithmetic semantics, which may be a bit confusing. For example, the literal 7 considered as an index of type `Ix 5`, is the same as the literal 2; both can be used to access the third element of an array. We use the type `Int` to be consistent with other library functions in Haskell, but a more appropriate type would be an unsigned integer type, similar to `Word32`. Note that this operation in general requires a division (or dynamic check), although, for statically known expressions such as literals, we can perform the division at compile time. For index sizes that are powers of 2, we can replace the division with a cheap mask operation; this special case is captured using the `bitIx` operator, which turns `n`-bit values into indexes of size 2^n.

The predicate `Index :: Nat → Pred` identifies natural numbers that are valid indexes. We require that 0 is not a valid index, because, as we already mentioned, `Ix 0` is an empty type, and so we should not be able to make values of that type (e.g., if it was not for the `Index` constraint, `toIx` would result in division by 0). In addition, we may want to put an upper limit to the size of arrays. For example, because indexes can be stored into memory areas, they have a fixed size (a machine word), and so we do not support arbitrarily large indexes.

The function `fromIx` 'forgets' that a value is an index and turns it into an ordinary number (again an unsigned type would be more appropriate). The values `minIx` and `maxIx` refer to the smallest and largest index in an `Ix` type. Indexes may also be compared for equality and are ordered.

3.2 Indexes as Iterators

The operations 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 is quite expensive. In this section we explain how such overheads can be avoided in common cases.

Programs often need to traverse an array (or a sub-range of an array). Usually this is done with a loop that checks for the end of the array at every iteration, and, if not, manipulates the array directly without any dynamic checks. We could get something similar by adding increment and decrement functions:

```
inc :: Index n ⇒ Ix n → Maybe (Ix n)
dec :: Index n ⇒ Ix n → Maybe (Ix n)
```

These functions increment (or decrement) an index, unless we are at the end (or beginning) of an array. Using these functions, 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 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.

A problem with this approach is that we perform *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 after in the **case** statement. Furthermore, this is perhaps the most common use of `inc` and `dec`, so it would be nice to avoid the extra check.

To solve the above problem we introduce an increment and decrement *pattern*. The pattern either fails if we cannot increment or decrement an index, or succeeds and binds a variable to the new value. We observe 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'
```

If for some reason we need the functions `inc` and `dec` we can implement them in the language like this:

```
inc (j - 1) = Just j
inc _       = Nothing

dec (j + 1) = Just j
dec _       = Nothing
```

Notice that to increment we use a minus in the pattern, and to decrement we use a plus. This is reasonable because we are pattern matching, but it can be confusing, and might lead us to prefer an alternative notation. In practice, we expect that programmers will not write too many explicit loops like the examples above. 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 above 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`:

```
sumArray a = accEachIx 0 (λ tot i →
               do x ← readRef (a @ i)
                  return (tot + x))
```

What we have described so far works well for traversing an entire array, but in some situations we may need to traverse only a sub-range of the array. This may happen in situations where we use an array to store data, but perhaps not all locations in the array contain meaningful data. We can deal with such situations by using a guard on a pattern that terminates a loop before we reach the end of the array. Unfortunately, then we end up having two checks per iteration again: one to check if incrementing would produce a valid index, and another to see if we have reached the last element that is of interest to us. It seems plausible that a compiler may be able to optimize away the first of the checks. However we can achieve the same by generalizing the increment and decrement patterns to contain a guard that determines when the pattern fails. For example, we can rewrite the function `accEachIx` to operate on a sub-range of an array like this:

```
accEachIxFromTo :: Index n ⇒
  Ix n → Ix n → a → (a → Ix n → IO a) → IO a
accEachIxFromTo start end a _ | end < start = return a
accEachIxFromTo start end a f = loop a start
  where
  loop a i = do b ← f a i
                case i of
                  (j - 1 | j ≤ end) → loop b j
                  _ → return b
```

The pattern `(j - 1 | j ≤ end)` succeeds if we can increment `i` by 1, and the result is less then or equal to `end`. In the example above `i` and `end` are both expressions of type `Ix n`, and if the pattern succeeds we introduce the variable `j` of type `Ix n`. Notice that the type of `end` ensures that we cannot get an index value that is too large. Omitting the guard (e.g., as in the previous examples) is a short-hand for `(j - 1 | j ≤ maxIx)`. The case for decrementing patterns is symmetric, where we allow the guard to specify a different lower bound than `minIx`. Figure 2 contains the formal semantics of the increment and decrement patterns. Note that we interpret pat-

$$
\begin{aligned}
(x - k \mid x \leq e) = \lambda i.\ \textbf{do let}\ & x' = i + k \\
& \mathit{guard}\ (x' \leq^u e) \\
& \mathit{return}\ \{x \mapsto x'\} \\
(x + k \mid e \leq x) = \lambda i.\ \textbf{do let}\ & x' = i - k \\
& \mathit{guard}\ (e \leq^s x') \\
& \mathit{return}\ \{x \mapsto x'\}
\end{aligned}
$$

Figure 2. Semantics of index patterns

terns here as functions from values to a `Maybe` type that returns the bindings produced by a successful match. In both cases we compute an intermediate value x', which is not necessarily a valid index, and then perform a check to determine if the pattern can succeed. Because we think of indexes as being integers on a real machine, it is important to not forget that we are using modulo arithmetic. This is why in incrementing patterns we use an unsigned comparison, and in decrementing patterns we use a signed comparison.

4. Alignment

In some situations (often when communicating with hardware), data stored in memory has to satisfy certain alignment constraints. This means that the address where the data is stored should be a multiple of some number k (i.e., if we consider the entire memory as an array of elements of size k bytes, then the data will be a valid element). Usually k is a power of 2, and so the lowest $\log_2 k$ bits in the address of the data are 0. Such assumptions are often used in hardware devices to pack more data in fewer bits.

To support aligned data, we use a more general version of `Ref` that we call `ARef` (for aligned reference):

```
ARef :: Nat → Area → *
type Ref = ARef 1
```

Compared to what we have seen so far, we have added a new parameter to track the alignment of the reference. For example, `ARef 4K FreePage` is a 4K aligned reference to a free page. Note that the reference type, `Ref`, used in previous sections is just a special case that allows alignment on any byte boundary.

With this change in mind we need to revisit the types of the operations that manipulate references. The building blocks of all memory areas are stored abstract values. Because they do not have any sub-components, the change to aligned references is trivial. For example, the type of `readRef` becomes:

```
readRef :: ARef a r → IO (ValIn r)
```

For arrays and structures, we need to do some work because we want to determine the alignment of their sub-components from the alignment of the entire structure. To see how we do this, consider a structure S, that has a sub-component of type T, at offset n (see Figure 3). Given an a aligned pointer to S, the sub-component will

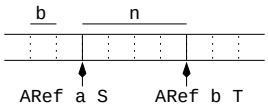

Figure 3. Alignment of a sub-component

be aligned on any boundary b that divides both a and n. The largest alignment we can deduce for the sub-component is therefore the greatest common divisor of a and n. 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 a 2 byte boundary, because $\gcd(4, 6) = 2$.

With this insight, we can give a new, more precise type for the array indexing operator:

```
(@) :: ARef a (Array n t) → Ix n
       → ARef (GCD a (SizeOf t)) t
```

Accessing the fields of a structure is similar: the maximum alignment of a field is the greatest common divisor of the alignment of the structure, and the sum of the sizes of the preceding fields. For example, consider a structure `Pair`, defined like this:

```
struct Pair s t where
  fst :: s
  snd :: t
```

Then the two selectors have the following types:

```
fst :: ARef a (Pair s t) → ARef a s
snd :: ARef a (Pair s t) → ARef (GCD a (SizeOf s)) t
```

Occasionally we may need to 'forget' that a reference has a given alignment. We can do this with the function `realign`:

```
realign :: GCD a b = b ⇒ ARef a t → ARef b t
```

Alternative Design Choice. In the design we outlined above the accessor operations computed the largest possible alignment for the sub-component. An alternative approach is to drop the requirement that we compute the *largest* alignment. This results in more polymorphic types for the accessors. For example, array indexing could be typed like this:

```
type a `Divs` b = (GCD a b = a)
(@) :: (b `Divs` a, b `Divs` SizeOf t) ⇒
       ARef a (Array n t) → Ix n → ARef b t
```

The predicate `a `Divs` b` states that `a` divides `b`. The benefit of this approach is that because the type is more polymorphic programmers do not need to use `realign` to forget alignments. The drawback is that it may lead to more ambiguous types, because of 'intermediate' alignments that cannot be determined uniquely, for example if we need to index in a two dimensional array.

The two designs are equivalent in expressive power: we can get the first design from the second by simply adding the more restrictive type signature; to get the second from the first we can compose accessors with `realign`.

Nested Structures. Programmers should be careful when nesting structures in code where alignment is important. To illustrate what may go wrong, consider the following two types:

```
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. The functions that access the fields, 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 (L 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 $\gcd(\gcd(4,3),1) = 1$. The reason this happens is that we lose some information when we access a field, namely the fact that the field was a part of the larger context in the structure.

5. Area Declarations

To declare memory references, programmers use an **area** declaration, which resembles a type signature:

area *name* [**in** *region*] :: *type*

Area declarations specify a name for the reference to the area, an optional region (to be discussed shortly), and a monomorphic type for the reference. We use the type to determine how much memory is required to accommodate the area, and also how that storage should be aligned. We can limit the alignments supported by an implementation using the `Alignment :: Nat → Pred` predicate; alignments are often restricted to powers of 2.

To compute the size of an area we use the predicate `SizeOf`:

```
class SizeOf (t :: Area) (n :: Nat) | t → n where
  sizeOf  :: Ref t → Int
  memCopy :: Ref t → Ref t → IO ()
  memZero :: Ref t → IO ()

instance Index n ⇒ SizeOf (Array n t) (n * SizeOf t)
instance SizeOf (Pair s t) (SizeOf s + SizeOf t)
```

To compute the size of an array we multiply the size of each element by the number of elements in the array. The size of a structure is the sum of the sizes of the fields in the structure. The sizes of stored pointers and indexes are as wide as the machine word on the target architecture. The sizes of stored bitdata types are determined by the number of bits in the representations.

For example, here is how we can define an area called `buffer` that contains 4096 bytes and is aligned on a 4 byte boundary:

```
type Byte    = Bit 8
area buffer :: ARef 4 (Array 4K (Stored Byte))
```

Note that even though programmers can write strange types like `LE (Int → Int)` they cannot use them to declare memory areas, because they lack `SizeOf` instances, and so the implementation cannot determine how much memory to allocate.

5.1 Area Representation

Areas do not (need to) reside in the heap like ordinary abstract values. In addition, we were careful to arrange things in such a way, that areas do not contain any abstract values, and, in particular, they cannot contain pointers to the heap. For these reasons there is no need to garbage collect areas.

Initialization. We were also careful to ensure that only types that contain 0 can be stored in memory. We can see this by considering what types can be stored in memory: `Ix`, `Ptr`, and bitdata types. Notice that references cannot be stored in memory. This choice makes it easy to initialize memory areas: an implementation just needs to place the memory areas in a part of memory that contains zero values (the `bss` segment, for example).

An alternative design choice would be to provide a means for programmers to specify the initial values for memory areas, either for each storable type, or for each declaration. With such a design, an implementation would have to initialize memory areas properly before executing the main program. There is still the question of how to initialize areas. For example, Cyclone provides special notation to make initializing arrays easier. Also, we need to make sure that the initializers themselves do not use uninitialized areas. This is one place where purity helps because operations that read and write to memory have monadic types, and we can restrict initializers to being pure. This also removes the problem of specifying in what order the initializers should be executed.

Area Attributes. In some special circumstances, there are more restrictions on where areas reside in memory and how they should be initialized. In such situations 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. All areas annotated with the same region are grouped together. Areas with explicit region annotations are not automatically initialized or allocated by an implementation, instead the implementation requires an external specification describing what to do with such memory regions. At present we do not track the regions in the types of the references as Cyclone [12], but this may be a useful future extension.

One example of when such advanced configuration may be needed is when working with memory mapped devices (e.g., video RAM). Then the memory area is required to be at a particular fixed location in memory.

```
type Row = Array 80 ScreenChar
area screen in videoRAM :: Ref (Array 25 Row)
```

As another example consider implementing a resource manager in an OS kernel. A common way to do that is to reserve a large area of virtual addresses, but only to back them up with concrete physical memory on demand. In such situations, we cannot initialize the (virtual) memory area ahead of time, because that would require us to back up the entire virtual area with physical memory.

```
area memPages in virtual
  :: ARef 4K (Array PageNum FreePage)
```

In general, programs that manipulate page tables can be type unsafe because updating page tables can have a profound effect on the entire program, even though it appears to be a simple write to memory. To ensure the correctness of such programs one needs something more advanced than the Hindley-Milner type system.

6. Conversion Primitives

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 only provide a limited set of casts, that does not compromise the invariants we enforce with types.

Arrays of Bytes. One set of casting operations deals with converting between structured descriptions of memory areas and arrays of bytes. These operations are safe for all memory areas, except ones that contain stored pointers or indexes, because these types have special invariants that we need to preserve. We use the class `Bytes` to classify types that can be converted to and from arrays of 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)
```

There are built-in instances for representation types of abstract values, except for the representation types of pointers and indexes. We also have instances for arrays and structs:

```
instance (Index n, Bytes t) ⇒ Bytes (Array n t)
instance (Bytes s, Bytes t) ⇒ Bytes (Pair s t)
```

The instance for arrays allows to turn 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 the fields of a structure are in the `Bytes` class.

As an example of how we might use these operations, we show an alternative implementation of a function that clears the screen:

```
cls' = forEachIx (λ i → writeRef (arr @ i) blank2)
  where
  blank2 = toBits blank # toBits blank

  arr :: (4 * n = SizeOf Screen)
      ⇒ Ref (Array n (Stored (Bit 32)))
  arr = fromBytes (toBytes videoRAM)
```

The function `cls'` first casts the video RAM to an array of double words, and then uses a single loop to write pairs of blank screen characters to the entire area.

Views on Arrays. Another set of useful casting operations involve arrays. The operations do not perform any computation, instead they change the way we perform indexing on arrays. The first operation allows us to split an array into two smaller arrays:

```
splitArr :: (Index x, Index y, GCD a (x * SizeOf t) = b) ⇒
            ARef a (Array (x + y) t) →
          ( ARef a (Array x t), ARef b (Array y t) )
```

When we change the view on an array, often we need to convert indexes that we had into the original array into indexes to the new array. We do this with the following operation:

```
splitIx :: (Index x, Index y) ⇒
           Ix (x + y) → Either (Ix x) (Ix y)
```

The type `Either` is a simple sum type, that tells us if the original index was in the left (first) array, or in the right array.

Another thing we might want to do with arrays is to change their dimension. We can use the following two functions to do that:

```
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)
```

The corresponding operations to convert indexes are:

```
divIx :: (Index x, Index y) ⇒
         Ix (x * y) → (Ix x, Ix y)

mulIx :: (Index x, Index y, x * y = z, Index z) ⇒
         (Ix x, Ix y) → Ix z
```

The operation `divIx` turns an index of a one dimensional array into an index into a two dimensional array by performing a division. The operation `mulIx` does the opposite, using multiplication.

7. Associated Type Synonyms

In this section we explore in more details what we can do using the abbreviation notation for functional predicates. This discussion is of a general nature, and readers that are mostly interested in the system programming aspects of our work can skip this section.

Associated Type Synonyms. Recently there was a proposal to allow type synonyms to be associated with Haskell's type classes [3]. For example, this is how we could define a class that captures some general operations on graphs:

```
type Edge 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 defined the type `Node g`, as well as the methods 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. It is interesting to note that we can achieve something very similar by just using ordinary functional dependencies and the abbreviations that we used throughout this paper. For example here is how we program the `Graph` example. First we define (the kind of) 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 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`. It is not clear how to do this with associated type synonyms, although their notation could probably be extended to accommodate such examples.

Arithmetic Predicates Another interesting set of examples are the arithmetic operators that we used extensively in this paper. Using associated type synonyms we could try to define addition like this:

```
class Add a b where
  type a + b
```

Then we can write `a + b` in types much in the same way as we would with the abbreviation we suggested. However, in addition we have to annotate the context with an extra `Add` constraint. This is not ideal, because now we have two names (`Add` and `(+)`) for a single concept. Also with more complex expressions, such as `x + (y + z)`, the extra constraints could get complex (e.g., `(Add y z, Add x (y + z))`).

8. Implementation

We have implemented the ideas described in this paper in hobbit, a prototype compiler that also includes support for bitdata [4]. The implementation is fairly standard for a pure strict language with a Haskell-like type system: the front-end checks that user specified types have valid kinds, and then it type checks the program. We eliminate the functional predicate notation during kind checking, although, in principle, we could do that earlier. During type checking, we compute and annotate programs with evidence for the predicates.

Our implementation follows the ideas of Tolmach [26] for compiling a functional language to C, although we generate assembly directly in the final pass (other approaches should also work). We perform a whole program analysis to make programs completely monomorphic. This enables us to resolve evidence completely at compile-time, and also means that we can avoid having to change representations when calling polymorphic functions. After that, we perform defunctionalization, turning unknown higher-order functions and monadic computations into explicit values.

9. Related Work

There has been a lot of research on how to make writing system software safer and simpler. In this section we point the reader to work that is closely related to our own, and which explores alternative ways of achieving goals that are similar to ours. The related work falls broadly into three categories: programming language design, foreign functions interfaces (FFIs), and interface description languages (IDLs). We consider our work to belong to the first category, but many of the topics that we discuss in this paper also show up in the design of FFIs, in particular when interfacing a high-level with a low-level language. IDLs are not as closely related, but are still of interest because they provide various ways of describing data layouts.

9.1 Programming Languages

In this section we compare our design to the choices made in some other systems programming languages (the list is by no means exhaustive).

C The de facto language for systems programming at present is C [17]. The literature on Cyclone [12] has a good description of the general benefits and drawbacks of using C. Usually C is considered to be suitable for systems programming because it grants programmers control over low-level data representation. However, the specification of the language leaves a number of representation details to particular implementations. For example, the fields in a C structure should be in consecutive (increasing) memory locations, but implementations are allowed to insert padding to properly align fields. This can lead to subtle bugs and inconsistencies when a program written for a particular implementation is compiled with a different one. The fundamental problem is that structures are used for two rather different purposes: to define abstract records and to describe memory regions. In our design there is a clear separation between these two cases, as records are of kind `*`, while memory structures are of kind `Area`.

Another interesting difference between our design and C is the treatment of alignment. In C alignment is a property of each type, while in our design it is a property of references (and by extension pointers). If alignment is associated with types, then care needs to be taken when constructing compound types such as structures and arrays, because an implementation needs to check that all subfields are properly aligned (there is still the design choice if implementations should insert implicit padding). We explored these ideas in an earlier version of the design presented in this paper. Separating the description of the layout of memory areas, and the restriction of where it can exist in memory is conceptually simpler, and more flexible, because we can impose different alignment constraints on areas with the same layout.

Cyclone Cyclone is a safe dialect of C [12, 23, 8]. At the surface Cyclone programs look quite similar to C programs, but in fact the Cyclone system supports many high-level language features, in particular an advanced type system. This makes it closely related to our work, and it is interesting to briefly compare some of the major design decisions. At a high-level there is a big difference in style: Cyclone is an imperative language, while we are interested in using a (pure) functional language. This is not a clear cut distinction because in either paradigm we can write programs that resemble the other, but the basic paradigm of a language influences what programs are easier to write. For example, notice that in our design there are very few monadic functions, reflecting the fact that a lot of the operations we need are pure functions.

The discussion about C structures also applies to Cyclone: the Cyclone compiler inserts implicit padding to align fields within structures.

There are many similarities in the handling of arrays and pointers in Cyclone and our design. Cyclone annotates arrays with their sizes like we do, they support not-nullable pointers (like our `Ref`), nullable pointers (like our `Ptr`), and pointers to sequences of elements, which are similar to a reference (or pointer) to an array in our design. The main difference between the two approaches is how to ensure that the invariants that these types state are preserved. Cyclone uses a flow analysis described in Chapters 6 and 7 of Daniel Grossman's PhD dissertation [8] to ensure that the invariants hold. We rely on a restricted set of operations (including pattern matching) to ensure that the invariants for the types hold.

9.2 Foreign Function Interfaces

Moby Moby is an experimental programming language. The reason we mention it here is that its FFI introduces the important notion of data level interoperability [7]. The idea is that in addition to an FFI, we should also have a foreign data interface (FDI) that enables us to manipulate foreign data without first having to marshal it. In the Moby system this is done by using tools that understand the foreign data and can generate intermediate code for the Moby compiler to manipulate it.

Our design is in much the same spirit, but instead of using tools to translate an external data specification, we specify the layout of the data directly in the high level language using the types of kind `Area`. We do this because our goal is to write as much as possible of our systems programs in the high level language. We can still write a bit of C or assembly code when we have to, because the memory types have concrete representations. If we want to use our approach to interface to large C programs than we would have to write tools that translate the C types to our `Area` types.

SML/NJ The C FFI of Standard ML of New Jersey (SML/NJ) [2], is also based on data level interoperability. To manipulate C data, programmers use a library that provides an ML encoding of the C type system. The ML types are implemented using a library of unsafe operations that should not be used by FFI programmers. There are many resemblances between this work and our own, for example in the FFI library arrays are also annotated with their size (but this is not used to provide safe indexing). There are also a number of differences, largely due to our differing goals, and the use of different technology.

The SML/NJ FFI tries to closely follow the design of C, which we don't because we are not designing an interface to C. This makes our job simpler, because we do not need to worry about many of the details of the C type system (e.g., `void*`). In addition we can provide things that are not in C, for example references with alignment constraints.

Another goal of the FFI is to not modify ML (except perhaps for the unsafe library). Our design was not restricted by such constraints. For example, we utilize a built-in natural numbers kind, instead of encoding natural numbers in the ML type system. We also provide special notation (e.g., pattern matching) to enable programmers to express exactly what they mean. This potentially leads to more readable programs, and simpler implementations.

Finally we make extensive use of qualified types, which are not available in ML. This enables us to avoid passing explicit type parameters or resorting to using 'fat' pointers, as is done in the FFI.

9.3 Interface Description Languages

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 their roots in remote procedure calls — 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. This process is called marshalling the data. The same idea has been used for communication between programs written in different languages, as they often use different representations for values. For example the HaskellDirect tool [5] uses an IDL to automatically generate marshalling code for Haskell values. A similar approach is used by CamlIDL [18] to marshal OCaml values. Other interesting IDLs include PADS [6], which is used to describe the format of data streams, DataScript [1] which can nicely describe the formats of various binary files, and SLED [21] which can describe the formats of machine instructions. HaskellDirect and CamlIDL bare resemblances to our work because they are fairly restricted — the IDLs resemble C header files with additional annotations. PADS, DataScript, and SLED are used to describe more complex data formats and are more similar to parsing tools like YACC.

9.4 Interfacing with C

Not only is C a very popular systems programming language, but it has also become the universal 'foreign' language — most FFIs interface to C. Clearly there are many situations where this is quite useful. However, for the kinds of applications we have in mind this approach introduces unnecessary overhead. Our concern is not so much for the possible performance overhead, but rather for the greater potential for errors: programmers need to be familiar not only with the high-level language, but also with the FFI, C, and subtle issues about their interaction. In addition, when manipulating the machine via C we have to rely on the correctness of the tools involved in the additional translations.

10. Future Work

In this paper we showed how we can annotate references with alignment constraints. It may be useful to annotate references with other properties: the SML/NJ FFI keeps track of which references are read-only [2], while the Cyclone system can differentiate between references that point to data in different regions of memory [12]. We could simply add more parameters to the reference type, but this quickly gets unwieldy. One way to solve this problem would be to add records at the type level. With such an extension, references could still have a single parameter describing their properties, and programmers can impose constraints only on the properties that are of interest. It seems likely that this feature could also be used in many other situations to reduce clutter in types or predicates with multiple arguments.

11. Conclusions

We showed how to extend a modern functional language with support for manipulating data with an explicit memory representation. Our design uses advanced but fairly well understood programming language tools, such as polymorphism, qualified types, and improvement. The motivation for working with such data stems from our desire to write system-level software in a high-level functional language.

We use the kind system to separate types with abstract and concrete representations. We use types to enforce a number of invariants about values: we distinguish between references and pointers, we use special types to index safely into arrays, and we can enforce and compute alignment constraints on data.

We also presented a simple, yet novel way of working with predicates with functional dependencies. 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.

Acknowledgments

This work was supported, in part, by the National Science Foundation award number 0205737, "ITR: Advanced Programming Lan-

guages for Embedded Systems." We would like to thank Thomas Hallgren, Andrew Tolmach, and Rebekah Leslie for many useful discussions and for their enthusiasm for writing operating systems in Haskell.

References

[1] 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.

[2] Matthias Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". In *BABEL 2001: 1st Workshop on Multi-Language Infrastructure and Interoperability*, September 2001.

[3] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP 2005: ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, 2005.

[4] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. High-level views on low-level representations. In *ICFP 2005: ACM SIGPLAN International Conference on Functional Programming*, pages 168–179, 2005.

[5] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *ICFP 1998: ACM SIGPLAN International Conference on Functional Programming*, 1998.

[6] 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.

[7] Kathleen Fisher, Ricardo Pucella, and John Reppy. A framework for interoperability. In *BABEL 2001: 1st Workshop on Multi-Language Infrastructure and Interoperability*, September 2001.

[8] Daniel Joeseph Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, 2003.

[9] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ICFP 2005: ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, 2005.

[10] 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.

[11] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volumes 1–3*. Available online from `http://www.intel.com/design/Pentium4/documentation.htm`.

[12] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.

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

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

[15] 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.

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

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[18] Xavier Leroy. *CamlIDL User's Manual (Version 1.05)*. INRIA Rocquencourt.

[19] 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 17th ACM Symposium on Operating System Principles (SOSP '99)*, Kiawah Island Resort, SC, December 1999.

[20] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A functional notation for functional dependencies. In *Proceedings of The 2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.

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

[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[23] Cyclone Team. *Cyclone User's Manual (0.8.2)*, August 2004.

[24] Haskell FFI Team. *Haskell 98 Foreign Function Interface (1.0)*, 2003.

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

[26] 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.

A. Quick Reference

The following code lists the main type constructors and predicates that are used by our language design. These were introduced incrementally in the body of the paper, but are repeated here together as a simple quick reference.

```
-- Type constructors
Bit, Ix    :: Nat → *
ARef, APtr :: Nat → Area → *
LE, BE     :: * → Area
Array      :: Nat → Area → Area

-- Manipulate stored values
class ValIn r t | r → t where
  readRef  :: ARef a r → IO t
  writeRef :: ARef a r → t → IO ()

-- Manipulate memory areas
class SizeOf t (n :: Nat) | t → n where
  sizeOf  :: ARef a t → Int
  memCopy :: ARef a t → ARef b t → IO ()
  memZero :: ARef a t → IO ()

-- Convert to/from byte arrays
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)

-- Subsets of the natural numbers
Index     :: Nat → Pred  -- Array index size
Alignment :: Nat → Pred  -- Alignment
```

Figure 4. A summary of types and predicates