The Habit Programming Language Preliminary Report

The High Assurance Systems Programming Project (Hasp)
Department of Computer Science, Portland State University
Portland, Oregon 97207, USA

August 2009

1 Introduction

This report presents a preliminary design for the programming language Habit, a dialect of Haskell [14] to support development of high quality systems software. The primary commitments of the design are as follows:

- *Systems programming*: Unlike Haskell, which was intended to serve as a general purpose functional programming language, the design of Habit focusses on features that are needed in systems software development. These priorities are reflected fairly directly in the new features that Habit provides for describing bit-level and memory-based data representations, the introduction of new syntactic extensions to facilitate monadic programming, and, most significantly, the adoption of a call-by-value semantics to improve predictability of execution. The emphasis on systems programming also impacts the design in less direct ways, including assumptions about the expected use of whole program compilation and optimization strategies in a practical Habit implementation.
- *High assurance*: Although most details of Haskell's semantics have been formalized at some point in the research literature, there is no consolidated formal description of the whole language. There are also known differences in semantics, particularly with respect to operational behavior, between different Haskell implementations in areas where the Haskell report provides no guidance. Although it is not addressed in the current report, a high-priority for Habit is to provide a full, formal semantics for the complete language that can be used as a foundation for reasoning and formal verification, a mechanism for ensuring consistency between

implementations, and a basis for reliably predicting details about memory allocation, asymptotic behavior, and resource utilization.

- *Simplicity*: We strive for a language design that is as simple as possible. The goals of providing a full and tractable semantics and of producing a practical working implementation are key drivers for a simple design; it would be very difficult to realize these goals for a complicated programming language with many constructs, intricate semantics, and awkward special cases. The emphasis on systems programming provides a lower bound in terms of functionality that must be included, but also provides opportunities for simplification because it allows us to focus on features that are needed in this particular domain, and to omit those that might only be useful in a more generally scoped language. For example, the design of Habit omits aspects of Haskell such as list comprehensions, lazy patterns, and defaults because these features are not typically required in systems programming. Perhaps more significantly, Habit avoids the complexities of features like the *monomorphism restriction*, a frequent source of confusion for new and seasoned Haskell programmers alike, because it is not relevant to the kinds of compilation techniques that we expect to be used in Habit implementations.

Another fundamental theme that underlies the design is the need to balance *abstraction* and *control*. Clearly, it is beneficial to use higher-level abstractions and programming constructs whenever possible because this can help to boost reuse and reliability as well as developer productivity. However, there are also some situations in systems programming that require a fine degree of control over low-level details such as performance, predictability, and data representation; in cases like these, systems developers accept the need for lower-level coding techniques in return for greater transparency and a more direct mapping between source code and the resulting behavior on an underlying hardware platform. The design of Habit is intended to encourage the use of high-level abstractions whenever possible, while also supporting low-level control whenever that is required.

The remaining sections of this report are as follows. In Section 2, we sketch the background and motivations for the design of Habit. This material is not required to understand the technical details in the rest of the report, but may be useful in providing context. Section 3 begins the technical portion of the report with a survey of the Habit language, much of which is presented in the form of an annotated grammar. This provides us with at least a brief opportunity to discuss each of the syntactic constructs of Habit. In Section 4, we discuss the standard environment for Habit, documenting the kinds, type classes, type constructors, and primitive operations that are used as the building blocks for Habit programs. In the terminology of Haskell, this amounts to a tour of the Habit standard prelude. Finally, in Section 5, we present an extended programming example using Habit. Instead of showcasing Habit's support for conventional functional programming techniques (such as high-

order functions, algebraic data types, polymorphism, type classes, and so on), much of which would already look very familiar to a Haskell programmer, we focus on a demonstration of the facilities that Habit provides for working with memory-based data structures. More specifically, the example provides an implementation for a priority set using memory-based arrays to support $O(1)$ identification of the maximum priority in the set, and logarithmic time operations for insertion and deletion. The example closely follows the structure of a previous implementation that was written in C as part of the timer interrupt handler for the `pork` implementation of L4; we include code for both the Habit and C implementations for the purposes of comparison. Because the timer interrupt is triggered many times a second, this is an example in which raw performance of compiled Habit code would be important, and in which dynamic memory allocation (and, most importantly, garbage collection) should be avoided in order to obtain predictable performance and low latency.

This version of the Habit language report is preliminary. We expect that aspects of the design will evolve, possibly in significant ways, as our implementation matures, and as we gain more experience using Habit to construct and reason about new systems programming artifacts.

2 Background

This section summarizes the background and previous work that has prompted the development of Habit. This information may help to provide some deeper insights into the motivations and goals for the language design; readers who are interested only in technical aspects of Habit may, however, prefer to skip directly to the next section. Much of the text in this section is derived from an earlier publication [1] where some of these issues were discussed in more detail and used to suggest the development of a new language with the placeholder name of *Systems Haskell*. Habit, of course, is the result of that development.

Development of systems software—including device drivers, operating systems, microkernels, and hypervisors—is particularly challenging when high levels of assurance about program behavior are required. On the one hand, programmers must deal with intricate low-level and performance-critical details of hardware such as fixed-width registers, bit-level data formats, direct memory access, I/O ports, data and instruction caches, and concurrency. On the other hand, to ensure correct behavior, including critical safety and security properties, the same code must also be related directly and precisely to high-level, abstract models that can be subjected to rigorous analysis, possibly including theorem proving and model checking. Failure of computer software can be a major problem in any application domain. However, the consequences of failure in systems software are especially severe: even simple errors or oversights—whether in handling low-level hardware correctly or in meeting the goals of high-level verification—can quickly compromise an entire system.

Despite the advances that have been made in programming language design, most real-world systems software today is still built using fairly low-level languages and tools such as C and assembly language. Use of such tools enables programmers to address important performance concerns but also makes it much harder to reason formally about the code. As a result, it can be much harder to obtain high confidence in the behavior of the resulting software. By comparison, modern functional languages, such as Haskell [14] and ML [13], support much higher levels of program abstraction than have traditionally been possible in this domain and offer software engineering benefits in the form of increased productivity and opportunities for code re-use. Such languages also provide strong guarantees about type and memory safety, automatically detecting and eliminating common sources of bugs at compile-time, and, because of their strong mathematical foundations, provide natural openings for mechanized formal verification and validation of software at the highest levels of assurance. Is it possible that languages like these might be better choices for building more reliable, secure, and trust-worthy systems software?

As part of our group's efforts to explore this question, we developed House [5], a prototype operating system that boots and runs on bare metal (IA32) and in which the kernel, a small collection of device drivers, and several sample applications, have all been implemented in the pure, lazy functional language Haskell. The House kernel supports protected execution of arbitrary user-level binaries and manages the virtual memory of each such process by direct manipulation of the page table data structures that are used by the hardware MMU. We have also developed two prototype implementations of the L4 microkernel [11], one in Haskell (called `Hank`) that builds on the H-interface foundation that was used in House, and, for comparison, a second (called `pork`) that was built using the traditional tools of C and assembly. L4 is interesting here because: (i) it is a microkernel design developed within the systems community, and hence reflects the priorities and goals of that community rather than those of programming language researchers; and (ii) there are informal but detailed specifications for several flavors of L4, as well as multiple working implementations that can provide a basis for comparison and evaluation.

The experience using Haskell was generally positive, and we found that several aspects of the language—particularly purity and strong typing—were very useful in both structuring and reasoning, albeit informally, about the code. Specifically, the pure semantics of Haskell makes information flow explicit (and hence more readily checked) via functional parameters instead of being hidden in global variables, while the expressive polymorphic type system promotes flexibility while also tracking the use of side-effects using monads. At the same time, however, we also encountered some problems in areas having to do with low-level operations, performance, run-time systems issues, and resource management. For example, functions involving manipulation of registers, I/O ports, or memory-based data structures or requiring use of special CPU instructions, were implemented in House by using the Haskell foreign function interface (FFI). Some of the required functions were already

provided by the FFI (for example, for `peek`ing or `poke`ing into memory), while others were handled by using the FFI to package low-level C or assembly code as Haskell primitives. Unfortunately, some of these functions violate the type- and memory-safety guarantees of Haskell, which enabled us to write and run buggy code with problems that potentially could have been prevented or caught at compile-time.

The design of Habit that is presented in this report is intended to preserve (or even enhance!) those aspects of Haskell that we found to be most useful in our previous work, but it also seeks to address the problems that we encountered. Some of the changes—such as the support that Habit provides for working with bitdata and strongly-typed memory areas [3, 2, 4]—were directly motivated by our previous experiences with Haskell. Others leverage ideas from past research—such as the work by Launchbury and Paterson on foundations of integrating *unpointed types* into Haskell [12], which we have developed into a full language design for Habit—or reflect engineering decisions and shifts in priorities—such as the move to a call-by-value semantics instead of the lazy semantics of Haskell—to better target the systems programming domain. One practical advantage of basing the design of Habit on Haskell [14] is that it avoids the need or temptation to develop fundamentally new syntactic notations or semantic foundations. As a result, from the low-level lexical structure of identifiers to the syntax and interpretation of type class declarations, many aspects of Habit will already be familiar to anyone with prior Haskell experience.

3 A Survey of the Habit Programming Language

This section provides a detailed, albeit informal survey of the Habit language in the form of an annotated grammar. In addition to documenting the basic syntax, the associated commentary also highlights technical aspects of the language design. Familiarity with grammars, type systems, and other aspects of programming language design is assumed throughout, especially in relation to the design of Haskell [14]. Although we include a few examples to illustrate the grammar constructs, we should note that this section is not intended as an introductory tutorial to programming in Habit.

We begin our tour of the language with summaries of notational conventions (Section 3.1) and basic lexical syntax (Section 3.2). With those foundations, we then follow up with details of the syntax of types (Section 3.3), expressions (Section 3.4), patterns (Section 3.5), and programs (Section 3.6).

3.1 Notational Conventions

Our presentation of the Habit grammar in this and subsequent sections uses the following notational conventions.

- Nonterminal symbols are written with an initial capital letter.
- Keywords are written in lowercase, exactly as they are written in Habit programs. The full list of keywords, in alphabetical order, is as follows:

```
area bitdata case class data deriving do else extends fails if
in infix infixl infixr instance let of struct then type where
```

- The five symbols `|`, `(`, `)`, `,`, and `=` have special roles in the notation that we use here, but they are also terminals in the grammar of Habit. To avoid confusion, we write these symbols between double quotes as `"|"`, `"("`, `")"`, `","`, and `"="`, respectively, to represent literal occurrences of those symbols as part of a production. All other symbols appearing in the grammar should be interpreted as part of the Habit language. The full list of reserved symbols, separated by spaces (and without any disambiguating double quotes), is as follows:

```
( ) | = , ‘ { ; } [ ] \ <- -> => :: # @ _ .
```

- Grammar rules are written using the notation:

```
N  =  rhs1
   |  ...
   |  rhsn
```

  where `N` is a nonterminal name and each right hand side (`rhs`) (after the first `=` symbol on the first line or the first `|` on subsequent lines) is a sequence of symbols corresponding to a production. Each production may span multiple lines so long as all of the symbols are indented to the right of the initial `=` or `|`.

- Fragments of grammar are annotated using Haskell commenting conventions: a `--` sequence introduces a one line comment that extends to the end of the line on which it appears and a `{- ... -}` pair provides a nested comment that may span multiple lines. In particular, for clarity, we write `{-empty-}` to indicate an empty right hand side of a production.
- As a notational convenience, we allow the use of parameterized grammar definitions in which the definition of a nonterminal may be annotated with one or more formal parameters, written with an initial upper case letter, enclosed in parentheses, and separated by commas. Each use of a parameterized nonterminal in the right hand side of a production should

specify a sequence of symbols to be substituted for the corresponding formal parameter. The following examples capture common patterns that are used throughout the grammar:

```
Opt(X)    =  {-empty-}      -- optional X
          |  X

List(X)   =  X              -- one or more Xs
          |  X List(X)

List0(X)  =  Opt(List(X))   -- zero or more Xs

Sep(X,S)  =  X              -- one or more Xs separated by S
          |  X S Sep(X,S)

Sep0(X,S) =  Opt(Sep(X,S))  -- zero or more Xs separated by S

Sep2(X,S) = X S Sep(X,S)    -- two or more Xs separated by S
```

Parameterized rules like these are interpreted as macros for generating productions, and the grammar is only valid if the process of macro expansion is guaranteed to terminate. For example, the following definition for `Inv` is not valid:

```
Inv(X)   = X Inv(Inv(X))    -- invalid!
```

On the other hand, the next definition, for `Inf`, is technically valid, but not useful in practice because it does not derive any finite strings:

```
Inf(X) = X Inf(X)           -- infinite sequences of X
```

- The productions in this report are written for clarity of presentation, and not for use with any specific parsing technologies or tools. For example, we make no attempt to identify or eliminate LR parsing conflicts.

3.2 Lexical Syntax

The lexical syntax of Habit follows the definition of Haskell [14, Chapter 2]:

- *Comments and whitespace*. Habit uses the same syntax and conventions for comments and whitespace as Haskell. In particular, a single line comment in Habit begins with the two characters `--` and a nested comment, which may span multiple lines, begins with `{-` and ends with `-}`.
- *Literate files*. Although it is not formally part of the language, Habit implementations are expected to support the use of literate files in which

lines of code are marked by a leading > character in the leftmost column and all other lines are considered to be comments. As in Haskell, comment and code lines must be separated by at least one blank line.

- *Filename suffix*. Although it is again not formally part of the language, we note that a Habit implementation can be expected to distinguish between literate and regular source files by using the suffix of the source filename. A suffix of `.lhb` indicates a literate Habit source file while a suffix of `.hb` indicates a regular Habit source file.

- *Identifier and symbol syntax*. Habit follows Haskell conventions for writing identifiers and symbols [14, Section 2.2]. Identifiers beginning with an upper case letter (represented by `Conid` in the grammar below) and symbols that begin with a leading colon (represented by `Consym`) are treated as constructors. Other identifiers (`Varid`) and symbols (`Varsym`) are typically used as variable or operator names. Symbols may be used in places where identifiers are expected by placing them between parentheses, as in `(+)`. Identifiers may be used in places where operators are expected by placing them between backticks, as in `` `div` ``. The following productions show how these alternatives are integrated in to the syntax for variable names and operator symbols that is used elsewhere in the grammar.

```
Var   = Varid                  -- Variables
      | "(" Varsym ")"
Varop = Varsym                 -- Variable operator symbols
      | ` Varid `
Con   = Conid                  -- Constructors
      | "(" Consym ")"
Conop = Consym                 -- Constructor operator symbols
      | ` Conid `
Op    = Varop                  -- Operator symbols
      | Conop
```

- *Integer literals*. Habit follows the basic Haskell syntax for integer literals (represented by `IntLiteral` in the following grammar) but also adopts the following three extensions.

  - *Binary literals*. In addition to decimal literals, hexadecimal literals (beginning with `0x` or `0X` and followed by a sequence of hexadecimal digits), and octal literals (beginning with `0o` or `0O` and followed by a sequence of octal digits), Habit allows binary literals that begin with either `0b` or `0B` and followed by a sequence of binary digits. For example, the tokens `11`, `0xB`, `0o13`, and `0b1011` represent the same value using decimal, hexadecimal, octal, and binary notation, respectively.
  - *Underscores*. Habit allows underscore characters to be included at any point in an integer literal (after the initial prefix that specifies a

radix, if present). Underscores can be used to increase the readability of long literals (such as `0b111_101_101`, `0x_ffff_0000_`, or `100_000`) but are otherwise ignored.

  - *Literal suffixes*. A single `K`, `M`, `G`, or `T` suffix may be added to a numeric literal to denote a multiplier of 2^{10} (kilo-), 2^{20} (mega-), 2^{30} (giga-), or 2^{40} (tera-). Such notations are common in systems programming, but programmers should note that Habit uses the binary interpretations for these suffixes and not the decimal versions that are commonly used in other scientific disciplines.

The syntax of Habit allows arbitrary length integer literals but uses a type class called `NumLit` to determine which literals can be used as values of which numeric types. For example, the integer literal `9` can be used in places where a value of type `Bit 4` is required, but not in places where a value of type `Bit 3` is required because the standard binary representation of the number 9 does not fit in 3 bits. Further details about the treatment of numeric literals in Habit are provided in Section 4.4.

- *Bit Vector literals*. Habit provides syntax for (fixed width) bit vector literals. Each of these tokens begins with a capital letter to specify a particular radix and includes a sequence of one or more digits of that radix, optionally interspersed with underscores to increase readability. The initial character may be `B` to specify binary notation (with one bit per digit), `O` to specify octal notation (with three bits per digit), and `X` to specify hexadecimal notation (with four bits per digit). For example, the tokens `XB`, `O13`, and `B1011` all represent the same numeric value, except that the second is a value of type `Bit 3` while the first and third are values of type `Bit 4`. Note that leading zeros are significant in bit vector literals. For example, the tokens `B_0000` and `B0` are not equivalent because the corresponding values have different types. Bit vector literals are represented by the nonterminal `BitLiteral` in the following grammar.

- *Other Literal Types*. Habit does not currently include syntax for floating point, character, or string literals because none of these types are included in the standard Habit environment. There are, for example, several design choices to be made in deciding how string literals might be supported, including details of character set encoding (as ASCII bytes, raw Unicode, UTF8, etc.) as well as representation (possibilities include: padded to some fixed length; null terminated; length prefixed; a list of characters, as in Haskell; or some combination of these, perhaps utilizing Habit's overloading mechanisms). Once we have gained sufficient experience to know which of these approaches will be most useful in practice, future versions of this report may extend the language to include support for these types. Of course, in that event, we would naturally expect to follow the Haskell syntax for literals.

- *Layout*. Habit uses a layout rule, as in Haskell, to allow automatic insertion of the punctuation that is used for lists of declarations, alternatives, statements, etc. The layout rule is triggered when the grammar calls for a `{` symbol at a point in the input where a different symbol appears. Writing n for the column at which this symbol appears, the compiler then behaves as if a `{` character had been inserted at column n, and then processes the rest of the input, inserting a `;` symbol each time it finds a new token on a subsequent line with this same indentation. This continues until the first symbol with an indentation less than n is found, at which point a closing `}` symbol is inserted and this instance of the layout rule concludes. A precise description of the layout rule is given in the Haskell report [14, Sections 2.7 and 9.3]. Habit uses the same approach, except that it does not insert a `;` symbol in front of the keywords `then`, `else`, `of`, or `in`; this allows a more natural syntax for conditionals and local definitions in do-notation (Section 3.4.1) and instance declarations (Section 3.6.5).

3.3 Kinds, Types, Predicates, and Type Functions

Habit is a strongly-typed language, requiring every expression to have an associated type that characterizes, at least approximately, the set of values that might be produced when it is evaluated. Using types, for example, a Habit compiler can distinguish between expressions that are expected to produce a Boolean result and expressions that are expected to evaluate to a function. From this information, it can report a compile-time error and reject any program that attempts to treat a Boolean as a function or vice versa. In this way, types serve both as a mechanism for detecting program bugs and as a way to document the intended use or purpose of an expression, function, or value. Habit uses a similar approach to enforce correct use of types, associating a unique *kind* with each type constant and type variable, and rejecting any type expression that is not well-formed (i.e., for which there is no valid kind). In addition to types and kinds, Habit uses (type class) *predicates* to identify sets of types with particular properties or to capture relationships between types.

In this section, we describe the Habit syntax for kinds, types, and predicates. These concepts provide the foundation for the Habit type system, just as they do for Haskell. Indeed, the Habit type system is not fundamentally different from the type system of Haskell, which also relies on the use of kinds, types, and predicates. Where the languages differ is in the set of primitive kinds, types, and predicates that are built in to the language; although some details of the Habit primitives are hinted at in this section, most of that information is deferred to the discussion of the standard Habit environment in Section 4.

3.3.1 Kinds

Every valid type expression in Habit, including type variables and type constants, has an associated kind that takes one of the following four forms:

- The kind `*` represents the set of nullary type constructors, including basic types like `Bool` and `Unsigned`.
- The kind `nat` represents the set of type-level numbers. In this report, we will typically use names beginning with the letter `n` for type variables of kind `nat` (or names beginning with the letter `l` in the case of type variables that represent memory area alignments). Specific types of kind `nat` are written as integer literals. For example, when they appear as part of a type expression, the integer literals `0`, `0x2a`, and `4K` are all valid type-level numbers of kind `nat`.
- The kind `area` represents the set of types that describe memory areas. In this report, we will typically use names beginning with the letter `a` for type variables of kind `area`. Informally, a type of kind `area` describes a particular layout of data in memory and, as such, creates a distinction between values (whose types are of kind `*`) and in-memory representations (whose types are of kind `area`). Habit programs cannot manipulate `area` types directly, but instead access them through references: if `a` is a type of kind `area`, then the type `Ref a`, which has kind `*`, represents references to memory areas with layout `a`.
- The kind `k -> k'` represents the set of type constructors that take an argument of kind `k` and produce a result of kind `k'`. For example, the `Ref` type constructor mentioned previously is a type constant of kind `area -> *`. The function type constructor, `->`, also has an associated kind: `* -> * -> *`. This indicates that a type expression of the form `d -> r` can be valid only if the subexpressions `d` and `r` have kind `*`, in which case the whole type expression also has kind `*`. This example also illustrates two small details about the syntax for kinds. First, note that the same symbol, `->`, is used to form both function kinds and function types. It is always possible, however, to distinguish between these two uses from the surrounding context. Second, in both cases, the `->` operator is assumed to group to the right. As a result, the kind expression `* -> * -> *` is really just a shorthand for `* -> (* -> *)`.

Readers familiar with Haskell will note that these kinds are the same as those of Haskell except for the addition of `nat` and `area`[1].

[1]Technical note: The definition of Haskell does not provide a concrete syntax for kinds. We include a syntax for kinds in Habit because it can be useful to annotate type parameters in datatype and class definitions with explicit kinds. In fact, similar extensions are already used in existing Haskell implementations.

In the grammar for kind expressions we distinguish between *kinds* (`Kind`) and *atomic kinds* (`AKind`); this enables us to capture right associativity of `->` as part of the grammar, but the distinction has no other, deeper significance.

```
Kind    = AKind -> Kind          -- function kinds
        | AKind                  -- atomic kinds

AKind   = *                      -- nullary type constructors
        | "(" Kind ")"           -- parentheses
        | nat                    -- type-level naturals
        | area                   -- memory areas
```

Note that `*`, `nat`, and `area` are not reserved symbols, and hence they may be used outside kind expressions as regular variable names. In addition, as a consequence of the lexical rules for constructing symbols, it is necessary to include spaces when writing a kind such as `* -> *`; without spaces, this string would instead be treated as a four character `Varop` symbol, `*->*`.

3.3.2 Types

The syntax of types in Habit is described by the following grammar:

```
Type        = TApp "::" Kind
            | TApp TyconOp Type       -- infix type constructor
            | TApp

TyConop     = Op                      -- constructor operator
            | ->                      -- function space constructor

TApp        = List(AType)             -- type application
            | struct [ Sep0(StructField, "|") ]

StructField = Sep(Varid, ",") :: Type -- structure field

AType       = "(" Type ")"            -- parentheses
            | "(" TyConop ")"         -- function space
            | Var                     -- type variable
            | Con                     -- type constant
            | IntLiteral              -- type-level numeric literal
```

The atomic types, `AType`, are just type variables, named constants (including type-level numbers of kind `nat` and the function space constructor `(->)` of kind `* -> * -> *`), and parenthesized type expressions.

From atomic types, we build type applications, `TApp`, as sequences of one or more atomic types. Application, which is denoted by juxtaposition, is treated

as a left associative operation, so an application of the form `t1 t2 t3` is treated in exactly the same way as a combination of two applications, `(t1 t2) t3`. Here, `t1` is first applied to `t2`, and then the resulting type, `t1 t2`, is applied to the third argument, `t3`. In a well-formed type application `t t1 ... tn` of an atomic type `t` to a sequence of arguments `t1`, ..., `tn`, the type `t` must have a kind of the form `k1 -> ... -> kn -> k`, where `k1` is the kind of `t1`, ..., `kn` is the kind of `tn`, and `k` is the kind of the resulting type expression.

The grammar for type applications also provides a syntax for structure types of the form `struct [ x1 :: a1 | ... | xn :: an ]`. Types like this describe the layout of a memory area and have kind `area`. The types `a1`, ..., `an` describe the layout of the individual memory components within the structure and are also required to have kind `area`. The corresponding labels, `l1`, ..., `ln`, can be used to reference individual fields. More precisely, if `S` is the above structure type, and `r` is a reference to a corresponding structure (i.e., a value of type `Ref S`), then the expressions `r.x1`, ..., `r.xn` will yield references to areas of type `a1`, ..., `an` (i.e., values of type `Ref a1`, ..., `Ref an`)[2].

Finally, the grammar for `Type` allows us to build complete type expressions as sequences of type applications separated by infix operators, including a special case for the function type constructor, `->`. This grammar reflects the fact that type application has higher precedence than any infix operator. For example, a type expression of the form `t a -> s b` is parsed as `(t a) -> (s b)` and not as `t (a -> s) b` or any other such variant. The grammar also suggests that all infix operators in type expressions have the same precedence and associativity (grouping to the right). In this case, however, Habit allows user declared fixity information (see Section 3.6.2) to determine whether a type expression of the form `t1 op1 t2 op2 t3` should be parsed as `(t1 op1 t2) op2 t3` or as `t1 op1 (t2 op2 t3)`. Despite differences in syntax, a type expression that is formed using infix operators is really just another way of writing a type application in which the operator is applied first to the left argument and then to the right. For example, the type expressions `s -> t` and `(->) s t` are different ways for writing the same type.

The standard types in Habit are summarized by the table in Figure 1; further details are provided in Section 4, and the constructs that Habit provides for user-defined types are described in Sections 3.6.7 and 3.6.8.

3.3.3 Predicates

Every top-level or locally defined variable in a Habit program has an associated *type signature*, which is calculated automatically by the compiler using a process of *type inference*. However, it is also possible (and generally recom-

[2]Technical note: Using the type function notation described in Section 3.3.4, the relationship between references to structures and references to individual components is captured by predicates of the form: `(Ref (struct [ x1 :: a1 | ... | xn :: an ])).xi = Ref ai`.

Type	Interpretation
`t -> t'`	functions from values of type `t` to values of type `t'`
`Bool`	Booleans, `False` and `True`
`WordSize`	a type level number giving the number of bits in a machine word
`Unsigned`	unsigned integers of width `WordSize`
`Signed`	signed integers of width `WordSize`
`()`	a unit type whose only element is also written as `()`
`Maybe t`	optional value of type `t`: either `Nothing` or `Just x` for some `x :: t`
`Nat n`	singleton types whose only element is the natural number `n`, introduced into programs solely via numeric literals
`Ix n`	index values (small, positive integers) in the range `0` to `(n-1)`
`Bit n`	bit vectors of length `n`
`ARef l a`	a reference to a memory area of type `a` with alignment `l`
`Ref a`	a reference to a memory area of type `a` with alignment `1`
`Array n a`	a memory area containing an array of `n` elements of type `a`
`struct [...]`	a memory area containing a structure with the described fields

Figure 1: Standard Habit Types

mended) for Habit programs to include explicit type signature declarations, which can serve as useful documentation and as an internal consistency check on the source code: if a declared type does not match the inferred type, then a compile-time diagnostic will be produced. The syntax for writing type signatures is described by the `SigType` nonterminal in the following grammar:

```
SigType = Opt(Preds =>) Type      -- (qualified) type signature

Preds   = "(" Sep0(Pred, ",") ")" -- predicate context
        | Pred                    -- singleton predicate context
```

A type signature that includes type variables represents a *polymorphic* type that can typically be *instantiated* in multiple ways within a single program[3], and a value or function with a polymorphic type is commonly referred to as a *polymorphic value* or a *polymorphic function*, respectively.

A standard example of a polymorphism is the identity function, `id`, which is defined as follows:

```
id  :: a -> a
id x = x
```

Here, the declared type includes the type variable `a`, indicating that we can apply the function `id` to an argument of any type, say `T`, to obtain a result with the same type, `T`. As a result, the `id` function may be treated as having any or all of the following types in a given program:

```
Unsigned -> Unsigned                       -- a is Unsigned
Bool -> Bool                               -- a is Bool
(Unsigned -> Bool) -> (Unsigned -> Bool)   -- a is (Unsigned -> Bool)
```

It is useful, in some cases, to restrict the ways in which type variables can be instantiated within a polymorphic type signature. This is accomplished by prefixing a type signature with a *context*, which is a list of zero or more *predicates*, represented in the preceding grammar by the `Preds` nonterminal. Type signatures of this form are sometimes referred to as *qualified types* because of the way that they restrict, or qualify the use of polymorphism [7].

The syntax for individual predicates is described by the following grammar:

```
Pred    = PredApp Opt("=" Type) Opt(fails)
                                  -- optional type function constraint
```

[3]Technical note: Habit supports only limited polymorphic recursion, and any program that potentially requires the use of a single variable at infinitely many distinct types will be rejected at compile-time. This restriction is designed to allow (although not require) implementations of Habit that use specialization rather than boxing to handle polymorphism.

```
                                -- and flag

PredApp = "(" PredApp ")"       -- parentheses
        | Con                   -- predicate name
        | PredApp AType         -- application
```

For example, the predicate `Eq t` (the result of applying the predicate name `Eq` to an argument type `t`) asserts that the type `t` must be a member of the set of equality types, written `Eq`, which includes precisely those types whose elements can be compared for equality using the `==` operator. More generally, a predicate with multiple parameters can be used to document relationships between types. As an example of this, a predicate `ValIn a t` (which again is an application, this time of a predicate name `ValIn` to two arguments, `a` and `t`) asserts that a memory region of type `a` can be used to store a value of type `t`.

Generalizing from these examples, a predicate of the form `C T1 ... Tn` can be interpreted as an assertion that the types `T1`, ..., `Tn` are related by the *type class* called `C`. The standard Habit environment includes several built-in type classes; these are summarized by the table in Figure 2 and further details are provided in Section 4. (In particular, the classes `Eq`, `Ord`, `Bounded`, `Num`, and `Monad` are very similar to classes with the same name in the Haskell standard prelude.) Habit also supports the introduction of user-defined classes (see Sections 3.6.4 and 3.6.5).

3.3.4 Type Functions

While type classes can be used to describe very general (for example, many-to-many) relations on types, many of the examples that we use in practice have more structure that can be documented by developers (by annotating the class declaration with one or more functional dependencies [9]) and then exploited by compilers (by using the information as part of the type inference process to obtain more accurate types [8]). In particular, it is often the case that one of the parameters is uniquely determined by the specific types that are used as the other parameters. A relation with this property can be viewed as a function on types. In particular, if the last parameter, `Tn`, of a predicate `C T1 ... Tn` is uniquely determined by (some subset of) the rest of the parameters, then we refer to `C` as a *type function* with n parameters. In such cases, we also allow (but do not require) the predicate to be written as `C T1 ... = Tn` with an `=` symbol before the last argument to emphasize the functional nature of `C`.

Habit also adopts a simple syntactic mechanism that allows type functions to be used within type signatures [10]. Specifically, if `C` is a type function with n parameters, then any type of the form `C T1 ...` with $n - 1$ arguments will be replaced by a new type variable, say `t`, together with the addition of an extra predicate, `C T1 ... = t` in the context of the type signature. For example, the standard Habit environment includes a `div` operator whose type indicates that

Predicate	Interpretation
`Eq t`	`t` is an equality type whose elements can be compared using `==`
`Ord t`	`t` is a totally ordered type with ordering tests (`<`, `<=`, `>`, `>=`) and `max`/`min` operators
`Bounded t`	`t` is a bounded type with maximum and minimum elements
`Num t`	`t` is a numeric type with basic operations of arithmetic
`NumLit n t`	a numeric literal for `n` can be used as a value of type `t`
`Boolean t`	`t` is a type whose elements support Boolean logical operators and, or, xor, and not
`Shift t`	`t` is a type that supports bitwise shift operations
`ToBits t`	values of type `t` are represented by bit vectors that can be extracted using the `toBits` function
`FromBits t`	values of type `t` can be constructed from a bit-level representation using the `fromBits` function
`BitManip t`	individual bits in a value of type `t` can be accessed and manipulated using indices of type `Ix (BitSize t)`
`ToUnsigned t`	values of type `t` can be converted into `Unsigned` values (by zero extending, if necessary)
`ToSigned t`	values of type `t` can be converted into `Signed` values (by sign extending, if necessary)
`Monad m`	`m` is a monad type constructor
`Index n`	`n` is a valid size for an `Ix` type
`Width n`	`n` is a valid width for bit-level operations
`Alignment l`	`l` is a valid memory area alignment
`n <= m`	type-level number comparison: `n` is less than or equal to `m`
`n < m`	type-level number comparison: `n` is less than `m`
`Pointed t`	`t` is a pointed type, which enables the use of recursive definitions at type `t`
`t =<= t'`	if `t` is pointed, then so is `t'`; this is required for a function type `t -> t'` to be valid

Figure 2: Standard Habit Type Classes

the divisor must not be zero, preventing the possibility of a division by zero exception at run time:

```
div :: t -> NonZero t -> t
```

In fact, `NonZero` is a two parameter type function and so the type of `div` can also be written as:

```
div :: (NonZero t t') => t -> t' -> t
```

The latter type signature indicates that the types `t` and `t'` for the dividend and the divisor, respectively, must be related by the `NonZero` class and hence hints (correctly) that the `div` operator is not completely polymorphic (because it cannot be applied in cases where there is no appropriate instance of the `NonZero` class). The original type signature, on the other hand, is more concise and may seem more natural to many programmers, although it could potentially be confusing to readers who do not realize that `NonZero` is a type function and assume (incorrectly) that `div` is fully polymorphic. In Habit, these two type signatures for `div` are completely interchangeable, leaving the programmer to choose when one is more appropriate than another on a case by case basis.

The standard Habit environment includes several built-in type functions, summarized by the table in Figure 3. Each of the predicates in this table has been written with an explicit `=` symbol to emphasize that we are working with type functions, but these characters can also be left out to help reduce syntactic clutter. Further details and explanations for each of these type functions are provided in Section 4. Of course, type functions and type classes in Habit are completely interchangeable, so the constructs that are used to define type classes can also be used to define new type functions (see Sections 3.6.4 and 3.6.5).

The first few type functions listed in Figure 3 are noteworthy because they provide a notation for expressing arithmetic relations within types. For example, the `(#)` operator, which concatenates a pair of bit vectors, has type:

```
(#) :: Bit m -> Bit n -> Bit (m + n)
```

This signature neatly captures the relationship between the lengths of the two inputs and the length of the result, but it is equivalent to the following type:

```
(#) :: (n + m = p) => Bit m -> Bit n -> Bit p
```

Predicate	Interpretation
`n + m = p`	`p` is the sum of `n` and `m`
`n - m = p`	`p` is the difference of `n` and `m`
`n * m = p`	`p` is the product of `n` and `m`
`n / m = p`	`p` is the result of dividing `n` by `m`, with no remainder
`GCD n m = p`	`p` is the greatest common divisor of `n` and `m`
`Exp2 n = p`	`p` is `2^n`
`NonZero t = t'`	nonzero values of type `t` are represented by values of type `t'`; in particular, values of type `t` can be divided by values of type `t'` (which can also be written as `NonZero t`) without triggering a divide by zero exception
`r.x = t`	`r` has a field called `x` (can be any `Varid`) of type `t`
`BitSize t = n`	values of type `t` are represented by bit vectors of width `n`
`ByteSize a = n`	a memory area of type `a` occupies `n` bytes in memory
`ValIn a = t`	a memory area of type `a` stores a value of type `t`
`LE t = a`	`a` holds a little-endian representation for values of type `t`
`BE t = a`	`a` holds a big-endian representation for values of type `t`
`Stored t = a`	`a` holds a value of type `t` in the platform's default format

Figure 3: Standard Habit Type Functions

3.4 Expressions

In this section, we focus on the syntax for expressions in Habit, as described by the following grammar:

```
Expr = Applic                       -- applicative expressions
     | LetExpr                      -- local definition
     | IfExpr                       -- conditional expression
     | CaseExpr                     -- case expressions
```

A parallel grammar is used for statements, which are expressions that appear in a known monadic context. We make a distinction between expressions and statements in the grammar because it allows us to use some notational shortcuts in writing monadic code (such as eliding the `do` keyword and omitting the `else` part of a conditional), but the distinction is purely syntactic and anything matching the `Expr` nonterminal can be used whenever a `Stmt` is expected.

```
Stmt = Applic                       -- applicative expressions
     | LetStmt                      -- local definition
     | IfStmt                       -- conditional statement
     | CaseStmt                     -- case statement
```

3.4.1 Applicative Expressions

The core expressions in Habit are either lambda terms (denoting anonymous functions), function applications (written using either prefix or infix syntax), `do` terms (providing syntactic sugar for monadic terms), and atomic expressions (selections, variables, literals, etc.), as shown in the following grammar:

```
Applic  = \ List(APat) -> Expr      -- anonymous function
        | do Block                  -- monadic expression
        | EInfix

EInfix  = EInfix Op EApp            -- infix application
        | EApp

EApp    = EApp AExpr                -- prefix application
        | AExpr                     -- atomic expression

AExpr   = ( Expr )                  -- parenthesized expression
        | Var                       -- variable name
        | Con                       -- constructor
        | Literal                   -- literal/constant
        | AExpr . Varid             -- selection

Literal = IntLiteral                -- natural number
        | BitLiteral                -- bit vector literal
```

Habit uses traditional dot-notation to indicate selection; intuitively, an expression of the form `r.x` returns the value of the component of the object `r` with label `x`. If `r` has type `t`, then the type of the expression `r.x` can be written as `t.x`. This syntax relies on Habit's notation for type functions (see Section 3.3.4), and it is equivalent to stating that the expression `r.x` has type `t'` subject to the constraint that `r.x = t'`. Note that the compiler automatically generates appropriate instances of the `.x` family of type functions to allow use of dot notation uniformly on both bitdata and structure types (in the first case, returning individual fields from a bitdata object, in the second computing references to individual components from a reference to the structure object).

The `Block` nonterminal describes a sequence of statements, separated by semicolons and enclosed in braces. This allows blocks to be written using layout instead of explicit punctuation.

```
Block   = { Stmts }
Stmts   = [ Var <- ] Stmt ; Stmts   -- monadic bind
        | let DeclBlock   ; Stmts   -- local definition
        | Stmt                      -- tail call
```

Note that the bind and local definition forms of `Stmts` can introduce new variables that scope over the rest of the list of statements. In the case of a bind, the result produced by the first statement call can be bound to a variable, but not to a pattern because there is no built-in mechanism in Habit for describing (pattern matching) failure in a monad.

Recall from Section 3.2 that the variant of the layout rule that is used in Habit does not insert a `;` in front of a `then`, `else`, `of`, or `in` keyword. This results in a slightly more relaxed (and frequently requested) version of the Haskell layout rule that admits code of the forms shown in the following examples:

```
do if ...           do case ...           do let f x = ...
   then ...            of p1 -> ...          in ... f ...
   else ...               p2 -> ...          s1
   s1                  s1
```

These program fragments are interpreted in exactly the same way as the following variants where the relevant keywords are indented by extra spaces:

```
do if ...           do case ...           do let f x = ...
     then ...            of p1 -> ...          in ... f ...
     else ...               p2 -> ...       s1
   s1                  s1
```

3.4.2 Local Declarations

A local declaration provides bindings for a set of variables (in the form of a semicolon-separated list of declarations) that scope over a particular subexpression or block.

```
LetExpr   = let Declblock in Expr   -- let expression
LetStmt   = let Declblock in Block  -- let statement
DeclBlock = { Sep0(Decl, ;) }       -- local declaration block
```

The difference between the `LetStmt` form shown here and the local definition construct for `Stmts` can be demonstrated by the following pair of examples:

```
do let decls                do let decls
   in s1                       s1
      s2                       s2
   s3                          s3
```

In the code on the left, a `LetStmt` is used and the variables introduced by `decls` scope over the code in statements `s1` and `s2` but not `s3`. By comparison, the same `decls` scope over all three statements in the code fragment on the right.

3.4.3 Conditionals (`if` and `case`)

Habit provides the familiar `if-then-else` construct for testing Booleans as well as the standard generalization to `case-of` for testing the values of a broader range of types. In addition, Habit also provides simple variants of each (the `if`-from and `case`-from constructs) for use in monadic code.

The syntax for `if-then-else` is standard, including a Boolean valued test and expressions for each of the `True` and `False` alternatives. Of course, the two branches must have the same type, which is then also the type of the conditional expression as a whole. For the monadic variants, a separate `Block` may be specified for each of the two branches, and the `False` branch may be omitted altogether. In the latter case, a default of `else return ()` is assumed and the type of the `True` branch must be of the form `m ()` for some monad `m`).

```
IfExpr  =  if Expr    then Expr  else Expr       -- if expression
        |  IfFrom
IfStmt  =  if Expr    then Block Opt(else Block) -- if statement
        |  IfFrom
IfFrom  =  if <- Stmt then Block Opt(else Block) -- if-from
```

The `if`-from variant, indicated by placing a `<-` immediately after the `if` keyword, can be used when the choice between two alternatives will be made as a result of the Boolean value that is returned by a statement of type `m Bool` for some monad `m`. The statement:

```
if<- e then s1 else s2
```

is syntactic sugar for the following expression that uses an extra, temporary variable name (`x` in this example):

```
do x <- e; if x then s1 else s2
```

The syntax for `case` expressions follows a similar pattern, providing an expression (the *scrutinee*) whose value is to be examined and a sequence of *alternatives* that use pattern matching and guards to distinguish between possible results.

```
CaseExpr = case Expr of Alts(Expr)              -- case expression
         | CaseFrom
```

```
CaseStmt = case Expr of Alts(Block)              -- case statement
         | CaseFrom
CaseFrom = case <- Stmt of Alts(Block)          -- case-from

Alts(E)  = { Sep(Alt(E), ;) }                   -- alternatives
Alt(E)   = Pat Rhs(->, E)                       -- alternative
```

Again, there are monadic variants for `case` statements and `case`-from statements, the latter allowing the choice between the alternatives to be made on the basis of the result returned by a statement of type `m t` for some monad `m`, where `t` is the type of the patterns in each of the alternatives. More specifically, the statement:

```
case<- e of alts
```

is syntactic sugar for the following expression that uses an extra, temporary variable name (`x` in this example):

```
do x <- e; case x of alts
```

3.5 Patterns

This section describes the syntax for patterns, which are used to describe the values that should be matched in function and pattern bindings, lambda expressions, and case statements. The complete grammar for patterns, represented by the nonterminal `Pat`, and atomic patterns, represented by the nonterminal `APat`, is as follows:

```
Pat    = Pat Conop PatApp        -- infix constructor
       | PatApp

PatApp = Con List0(APat)         -- constructor patterns
       | Con [ Sep0(BPat, "|") ] -- bitdata pattern
       | APat                    -- atomic patterns

BPat   = Var Opt("=" Pat)        -- bitdata field pattern

APat   = "(" Pat ")"             -- parentheses
       | "(" Sep2(APat, #) ")"   -- bit pattern
       | "(" Pat :: Type ")"     -- typed pattern
       | Var                     -- variable
       | Var @ APat              -- as-pattern
       | _                       -- wildcard
       | Con                     -- nullary constructor
       | Literal                 -- literal
```

Except for bit and bitdata patterns, all of the pattern forms shown here have the same interpretation in Habit as they do in Haskell. Different forms of pattern, of course, match against different values. For example:

- A variable, `x`, matches any value, binding the variable to that value within the scope of the pattern.
- A wildcard pattern, `_`, matches any value, without binding any variables.
- A literal pattern matches only the specified value, without introducing any variable bindings.
- An as-pattern, `x@p`, behaves like the pattern `p` but it also binds the variable `x` to the value that was matched against `p`.
- A pattern `C p1 ... pn`, where `C` is a constructor function of arity `n`, will match against any value of the form `C v1 ... vn` with the same constructor function so long as each component value `v1`, ..., `vn` matches the corresponding pattern `p1`, ..., `pn`.
- A pattern `B [ f1 | ... | fn ]`, where `B` is a constructor function of a bitdata type (see Section 3.6.8), will match any value that can be produced by the `B` constructor, so long as each of the individual field specifications, `f1`, ..., `fn`, are also matched. Each field specification is either a single field name, `x`, or a field name paired with a pattern, `x = p`. In the first case, the match always succeeds, binding the value of the `x` field to a variable of the same name within the scope of the pattern. (This is sometimes referred to as *punning*.) In the second case, the match only succeeds if the value of the `x` field matches the pattern `p`. In both cases, the name `x` must be a valid field name for the `B` constructor. A bitdata pattern may not name the same field more than once, but it is not necessary to list the fields in the same order as they appear in the original `bitdata` definition, and it is not necessary to list all of the fields for `B`; fields that are not mentioned explicitly will be treated as if they had been bound by a wildcard pattern.
- Bit patterns, which take the form `(p1 # ... # pn)`, match bit values of the form `(v1 # ... # vn)` so long as each of the bit vectors `v1`, ..., `vn` matches the corresponding pattern `p1`, ..., `pn`. The `#` operator used here is a function, mentioned previously at the end of Section 3.3.4, for concatenating bit vectors, and the pattern syntax is chosen to mirror the constructor syntax, just as it does for other forms of pattern. For example, `B11 # B0` yields the value `B110` of type `Bit 3`, and will match the pattern `(u # B10)` binding the variable `u` to the single bit value `B1`. It should be possible to infer the widths of the components in a bit from the context in which they are used; in some cases this may require the use of typed patterns or other type annotations.

3.6 Programs

A Habit program consists of a sequence of top-level declarations, each of which contributes in some way to the interpretation (or, in the case of a fixity declaration, parsing) of a named value or type constructor.

```
Prog    = Sep(TopDecl, ;)           -- program

Decl    = Equation                  -- equation in value definition
        | FixityDecl                -- fixity declaration
        | TypeSigDecl               -- type signature declaration

TopDecl = Decl
        | ClassDecl                 -- type class declaration
        | InstanceDecl              -- instance declaration
        | TypeDecl                  -- type synonym declaration
        | DataDecl                  -- data type declaration
        | BitdataDecl               -- bitdata type declaration
        | AreaDecl                  -- area declaration
```

Some forms of declaration (specifically, `Equation`, `FixityDecl`, and `TypeSigDecl`) can be used within local declarations as well as at the top-level; the remaining declaration forms within `TopDecl` can only be used at the top-level. The declaration of an entity in a Habit program may reference other entities defined later in the program, so there is no need for any explicit form of forward references. Tools processing Habit source code will use automated dependency analysis to determine program structure (e.g., to identify mutually recursive binding groups, and to determine an appropriate order for type checking).

3.6.1 Equations

User-defined values (including functions) are introduced by a sequence of one or more equations. There are two kinds of equations in a Habit program:

- The left hand side of a *function binding* comprises the name of the function and a sequence of patterns corresponding to a sequence of arguments. A function may be defined by more than one equation, but all of these equations must appear together in the source code, without intervening declarations, and all of the equations must have the same *arity* (i.e., the same number of argument patterns). These restrictions are inherited from the definition of Haskell, where they have proved to be useful as consistency checks that help to identify and avoid syntactic errors in input programs.
- A *pattern binding* is formed by an equation with a pattern on its left hand side. A pattern binding is evaluated by evaluating its right hand side

expression and then matching the result against the patter on its left hand side. Pattern bindings should be used with care because failure to match a pattern for a datatype that has multiple constructors could cause the matching process to fail, and abort further execution.

The grammar for equations is as follows:

```
Equation     = Lhs Rhs("=", Expr)             -- defining equation

Lhs          = Var List0(APat)                -- for function binding
             | Pat                            -- for pattern binding

Rhs(S,E)     = Rhs1(S,E) Opt(where DeclBlock) -- right hand side

Rhs1(S,E)    = S E                            -- unguarded rhs
             | List(Guarded(S,E))             -- guarded rhs

Guarded(S,E) = "|" Expr S E
```

The right hand side of each equation can be either a simple expression or else a sequence of Boolean guards that will be evaluated in turn until one of the returns `True`, at which point the value of the corresponding expression on the right of the `=` symbol will be used as the result of the equation. Note that we use a parameterized name here for `Rhs` so that we can reuse it as part of the syntax of `case` expressions where `->` is used in place of `=`.

3.6.2 Fixity Declarations

Fixity information, including both associativity and precedence, can be provided for user-defined operator symbols in both values and types using the `infix`, `infixr`, and `infix` declarations.

```
FixityDecl = Assoc      Prec Sep(Op, ",")      -- operator fixity
           | Assoc type Prec Sep(TyconOp, ",") -- type operator fixity
Assoc      = infixl                            -- left associative
           | infixr                            -- right associative
           | infix                             -- non associative
Prec       = Opt(IntLiteral)                   -- precedence
```

Fixity information is used to resolve ambiguities relating from adjacent occurrences of infix operators in value and type expressions. Specifically:

- `e1 + e2 * e3` will be parsed as `(e1 + e2) * e3` if either `+` has higher precedence than `*`, or if the two operators have equal precedence and both group to the left (`infixl`);

- `e1 + e2 * e3` will be parsed as `e1 + (e2 * e3)` if either `*` has higher precedence than `+`, or if the two operators have equal precedence and both group to the right (`infixr`);
- `e1 + e2 * e3` will be rejected as a syntax error if neither of the two cases above apply.

The precedence value (`Prec` in the grammar above) can be any numeric literal representing a number in the range 0 to 9, inclusive, with higher numbers corresponding to higher precedences. If `Prec` is omitted, then a precedence of 9 is assumed.

As in Haskell, any symbol that is used as an operator without an explicit fixity declaration is treated as if it had been declared `infixl 9`.

Any operator symbol that is named as part of a fixity declaration must have a corresponding definition elsewhere in the same binding group as the fixity declaration. An operator symbol may have multiple fixity declarations so long as the associativity and (implied) precedence is the same in all cases.

3.6.3 Type Signature Declarations

A type signature declaration provides an explicit type for one or more variables whose definitions appear elsewhere in the same binding group.

```
TypeSigDecl  = Sep(Var, ",") :: SigType        -- type signature
```

Type signature declarations are typically used as a form of documentation, but they can also be used to restrict the type of a particular variable in situations where a more general type might otherwise be inferred. We allow at most one type signature declaration for any single entity; it would not be difficult to lift this restriction, but would require the compiler to check that all of the declared types are equivalent. As described in the next section, type signature declarations are also used in `class` declarations to specify the names and types of class operations.

3.6.4 Class Declarations

A type class declaration introduces a new type class with a specified name, a list of parameters, and an associated list of members:

```
ClassDecl  = class ClassLhs              -- name and parameters
               Opt("|" Sep(Constraint, ","))
               Opt(where DeclBlock)      -- class operations
```

```
ClassLhs   = Con List0(TypeParam)        -- class left hand side
             Opt("=" TypeParam)         -- for type functions

TypeParam  = Var                        -- parameter (inferred kind)
           | ( Var :: Kind )            -- parameter (declared kind)

Constraint = Pred                       -- superclass
           | FunDep                     -- functional dependency

FunDep     = List0(Var) -> List(Var)    -- parameter dependency
```

Type classes share the same namespace as other type-level constants so it is not possible to use the same name simultaneously for both a class and a datatype, for example. (Indeed, any occurrence of a type class name within a type, other than as part of a predicate, will be interpreted as a use of type functions, as described in Section 3.3.4.)

A class may have zero or more parameters, each of which has an associated kind. The kinds of parameters may either be declared explicitly, or else will be inferred automatically from context (specifically, from the kinds of previously declared type-constants and from the structure of the type expressions in this and any other class or type declarations in the same binding group). No two parameters of a given class can share the same name.

A class with n parameters is interpreted as an n-place relation whose elements, referred to as the *instances* of the class, are tuples that define an appropriately kinded type constructor for each class parameter. We use the notation `C t1 ... tn` as an assertion that the types `t1`, ..., `tn` form a tuple that is included in the class `C`. If the assertion is true then we say that `C t1 ... tn` is a valid instance. The specifics of determining which instances of a class are valid is described independently via a collection of `instance` declarations.

Beyond determining the names and kinds of each parameter, there are two ways in which a `class` declaration may constrain the set of valid instances:

- By specifying a *superclass context*, which is a list of predicates introduced by the `extends` keyword. If the declaration of a class `C` begins as follows:

  ```
  class C a1 ... an extends P
    where ...
  ```

  then the compiler is responsible for ensuring that, if `C t1 ... tn` is a valid instance, then so are all of the predicates in `[t1/a1,...,tn/an]P`. Note that the parameters of the class `C` are the only variables that may occur in the superclass context `P`.

- By specifying one or more *functional dependencies*. A dependency annotation `a1 ... aj -> b1 ... bk` indicates that the choice of the parameters `b1`, ..., `bk` is uniquely determined by the choice of the parameters `a1`, ..., `aj`. For a class that has been annotated with such a dependency, the compiler must ensure that, if `C t1 ... tn` and `C s1 ... sn` are both valid instances whose components agree on the parameters `a1` though `aj`, then they must also agree on the parameters `b1` through `bk`.

Each of these features has been widely used in previous Haskell implementations to express invariants/relationships between class parameters, and for improving the precision of type inference. Note, however, that Habit and Haskell differ a little in details of the syntax that is used for superclasses. In Haskell, for example, the code fragment shown above would be written:

```
class P => C a1 ... an
  where ...
```

We have opted instead to use the notation described above for Habit because: (i) it places the name of the class that is being defined immediately after the opening `class` keyword so that it is easier to find; and (ii) it avoids a misleading use of the implication symbol, `=>`, in a context that, logically, corresponds to a reverse implication.

The (optional) `DeclBlock` at the end of a `class` declaration is used to specify names, types, and, in some cases, default implementations for the operations that are associated with the class. Names and types are specified using type signature declarations. The names of class operations have the same scope as any other top-level value/function and hence should not conflict with the names used for any other top-level entities. It is also possible to include fixity declarations and function definitions, comprising a sequence of one or more equations, as part of the list of declarations, but only if the associated functions are listed as class operations within the same list of declarations. A valid fixity declaration in a class can be moved to the top-level without changing the meaning of the program. If a class provides a default definition for a class operation, then that definition will be used if there is no explicit definition for that operation in a user provided instance declaration.

3.6.5 Instance Declarations

Instance declarations are used to determine the set of valid instances for each class in a Habit program. Because type-level programming via classes and instances is expected to be used quite heavily in Habit programs, it is important to have a flexible mechanism for defining instances. At the same time, in the interests of keeping the design as simple as possible, we would like to avoid

some of the complexities of type classes in Haskell. In particular, while Habit disallows the definition of overlapping instances (see below), it also introduces two new constructs (`else` and `fails`). These constructs can be used to deal with many of the examples that have previously been described using overlapping instances (as well as providing some new features altogether) while also avoiding some of the problems that they can cause.

The syntax of instance declarations is described by the following grammar.

```
InstanceDecl = instance Sep(Instance, else)

Instance     = Pred Opt(if Preds) Opt(where DeclBlock)
```

In essence, there are three basic forms of instance declaration, corresponding to the three forms of `InstanceBody` shown in the grammar. It is also possible to combine a sequence of basic instance declarations, separated from one another by the `else` keyword, into a single instance declaration; this is the only way to write overlapping instances in Habit, and it also provides a mechanism for defining closed classes.

The first basic form allows us to write instance declarations like the following without providing any definitions for class operations:

```
instance P => C t1 ... tn
```

In this example, `P` represents a list of predicates, and the declaration can be interpreted as an implication: if all of the predicates in `P` are valid instances, then `C t1 ... tn` will also be a valid instance. This form of instance declaration is useful in cases where the class `C` has no associated operations or where the definitions of those operations will be filled in by the default implementations provided in the definition of class `C`. More often, however, an instance declaration like this is used in documentation, such as this report, to describe a rule for generating class instances without giving details of its implementation.

The second basic form is similar to the first except that it includes a `DeclBlock` with definitions for the operations associated with the class `C`:

```
instance P => C t1 ... tn
  where ...
```

There are several restrictions on the declarations that can appear in an instance declaration like this that are not reflected in the grammar, some of which may be relaxed in the future. For example the list of declarations cannot include type signatures, fixity declarations, pattern bindings, or function definitions for names that are not operations of class `C`. As in the previous case, default

definitions provided in the definition of class `C` are used to supplement instance declarations that do not include explicit definitions for those operations.

The third basic form of instance declaration is used to provide information about predicates that are not valid instances of a class, and to prevent the definition of such instances in subsequent code. This kind of information can be useful in type-level programming and can also be used to detect and report some type errors more promptly. Specifically, a declaration of the following form specifies that, if all of the predicates in `P` are valid instances, then there cannot be any valid instance for the predicate `C t1 ... tn`, either in the current program, or in any future extension.

```
instance P => C t1 ... tn
  fails
```

One simple application for instance declarations like this is to implement the `never` form of type class directives [6] to specify, for example, that functions cannot be compared for equality:

```
instance Eq (a -> b) fails
```

If a declaration like this has been included in a program, then the compiler will report an error if the program attempts to use the equality operator, `==`, to compare functions. Without such a declaration, the compiler may, instead, infer types that include a predicate of the form `Eq (a -> b)`. The latter behavior is useful when the programmer envisions that it may be useful to add an instance for equality on functions in some later version of the program. In this particular case, however, there is no meaningful way to define equality on functions, and it is useful to be able to state this explicitly as part of the program's source code so that erroneous code that relies on such an instance will be detected more promptly. In this way, Habit provides general mechanisms that allow specific behaviors to be determined on a case-by-case basis.

Programs that include one or more pairs of *overlapping* instance declarations are not valid in Habit. For example, although each of the instance declarations in the following example would be valid on its own, the two declarations overlap (with common instance `IsBool Bool`) and cannot appear together:

```
class IsBool t
  where isBool :: t -> Bool

instance IsBool Bool     -- Matches only the predicate IsBool Bool
  where isBool x = True

instance IsBool t        -- Matches any predicate of the form IsBool t
  where isBool x = False
```

This restriction is necessary to ensure a well-defined semantics for the `isBool` operator. If we allowed programs like the one above, then there would be two possible interpretations for the expression `isBool True`: according to the first instance declaration, this should produce the Boolean result `True`, which is obviously not the same as the Boolean `False` that would be obtained by following the second declaration.

Some Haskell implementations allow examples like this by using the syntactic form of the instance predicates to an infer an implicit ordering between declarations. In examples like the one above, this would give priority to the first instance declaration, using that for `Bool` values and falling back on the second for any other type `t`. Habit, instead, requires orderings between instance declarations to be specified explicitly using an `instance...else...` construct:

```
instance IsBool Bool     -- Matches only the predicate IsBool Bool
  where isBool x = True
else IsBool t            -- Matches any predicate of the form IsBool t
  where isBool x = False
```

More generally, an instance declaration in Habit may take the form:

```
instance P1 => C t1 ...
else     ...
else     Pn => C tn ...
```

where `P1`, ..., `Pn` are arbitrary contexts, and all of the instance predicates have the same class `C`. In this case, it is permitted for the instance types `t1`, ..., `tn` to overlap, but any given clause can be used only if the all of the preceding clauses are guaranteed not to apply. The following instance declaration, for example, is valid, even though it has identical, and hence overlapping instance predicates:

```
instance Eq a => C a
  where ...
else     C a
  where ...
```

The first clause, however, can only be used with equality types, while the second has no context (i.e., it will work with any type). As a result, if `Eq Bool` is a valid instance, then `C Bool` follows from the first clause while `C (Bool -> Bool)` follows from the second. In the latter case, we have assumed that the program also includes the `fails` instance for `Eq (a -> b)` that was given previously, which can be used to confirm that the precondition, `Eq (Bool -> Bool)`, of the first clause does not hold.

Combinations of `else` and `fails` can be used to define a wide range of type class relations and type functions in a natural and concise manner. The following examples illustrate this with the definition of a closed class, `Closed` that is guaranteed to have only two valid instances, and the definition of a type function, `NumArgs`, that calculates the number of arguments in a function type.

```
class Closed (t :: *)
instance Closed Bool
else Closed Int
else Closed t fails

class NumArgs (t :: *) (n :: nat) | t -> n
instance NumArgs (d -> r) = 1 + NumArgs r
else     NumArgs t        = 0
```

3.6.6 Type Synonym Declarations

A type synonym declaration introduces a new name for an existing type.

```
TypeLhs   = Con List0(TypeParam)      -- type left hand side

TypeDecl  = type TypeLhs "=" Type      -- type synonym declaration
```

Type synonyms are typically used to provide convenient abbreviations for more complex type expressions, or to document intensions about how a particular value will be used, but they do not introduce new types.

To be well-formed, all of the variables appearing in the type on the right of the = symbol in a type synonym declaration must appear as a parameter in the `TypeLhs` on the left hand side, and no type variable may be listed more than once as a parameter on the left hand side.

In Habit, type synonym definitions are really just syntactic sugar for a special form of type function definition. In particular, a type synonym declaration:

```
type T p1 ... pn = t
```

is equivalent to the following combination of a class and instance declaration (for some fresh variable):

```
class T p1 ... pn = a | p1 ... pn -> a
instance T p1 ... pn t
else     T p1 ... pn a fails -- prevents other instances for T
```

where `a` is a fresh type variable. Some of the restrictions on type synonym definitions in Haskell are also implied by this formulation. For example:

- No partial applications: Because `T` is defined as a type function with $n+1$ arguments, it is not valid to use `T` in a type expression with fewer than n arguments. (See Section 3.3.4 for more details.)
- No recursion: Recursive type synonym definitions are not valid. Technically speaking, a recursive definition such as the following:

  ```
  type Stream = Pair Unsigned Stream
  ```

  could be expanded using the encoding described previously to obtain the following valid code:

  ```
  class Stream = s | -> s
  instance Stream = (Pair Unsigned Stream)
  else     Stream = s fails
  ```

  Expanding the use of the type function `Stream` on the right hand side of the instance declaration, however, we can see that this is equivalent to:

  ```
  instance Stream s => Stream (Pair Unsigned s)
  else     Stream s fails
  ```

  which does not define any valid instances.

3.6.7 Data Type Declarations

As in Haskell, datatype definitions, beginning with the keyword `data`, are used to introduce new algebraic datatypes in a Habit program. Each definition specifies a name for the new type, a sequence of parameters, a collection of zero or more constructor function definitions, and an optional list of classes. The syntax for datatype definitions is described by the following grammar:

```
DataDecl   = data TypeLhs                    -- name and parameters
               Opt("=" Sep(DataCon, "|"))    -- constructors
               Opt(deriving DeriveList)      -- deriving clause

DataCon    = Con List0(AType)                -- constructor function

DeriveList = Con                             -- deriving a single class
           | "(" Sep(Con, ",") ")"           -- or a list of classes
```

The optional `deriving` clause at the end of a `data` definition specifies a list of type class names and indicates that the compiler is expected to generate instances of those classes for the new datatype. The use of `deriving` is limited to certain built-in classes, and is subject to restrictions on the form of the data declaration.

- Instances of `Eq`, and `Ord` can be derived for any type so long as each of the component types is an instance of the corresponding class. This may result in a derived instance with a context that captures constraints on the parameters of the datatype.
- Instances of `BitSize`, `ToBits`, and `FromBits` can be derived for any type so long as each of the component types is an instance of the corresponding class. In these cases, each derived bit-level representations begins with $\lceil \log_2 n \rceil$ tag bits to distinguish between the different constructors of the datatype; the number n here is the total number of constructors. Note, in particular, that a datatype with only one constructor does not require any additional tag bits. After the initial tag bits, the the bit-level representation for each constructor is formed by concatenating the bit-level representations for each of the fields. Finally, the width of the resulting bit-level representation is computed as the maximum width over all constructors; note that this width must be a valid instance of the `Width` class.
- Instances of `Bounded`, `Num`, `BitManip`, `Boolean`, and `Shift` can be derived for any type that has a single constructor function with a single argument that is an instance of the corresponding class. In these cases, the datatype introduces a type that is isomorphic to an existing type and the `deriving` mechanism simply lifts the corresponding class structure to the new datatype.
- Instances of `Monad` can be derived for datatypes that have at least one parameter and exactly one constructor. In addition, the constructor can have only one field, which must be a type of the form `m a` where `a` is the last (i.e., rightmost) parameter of the datatype, and `m` is a type expression, not involving `a`, that is an instance of the `Monad` class. In this case, the new datatype is isomorphic to `m a` and the derived monad structure will be inherited directly from `m`.
- Instances of `Pointed` can be derived to specify that a pointed semantics should be used for the new type. This is necessary to allow the definition of general recursive functions over values of the new type.

3.6.8 Bitdata Type Declarations

Bitdata definitions are used to introduce names for new bitdata types, allowing fine-control over the bit-level representation of values as may be necessary to

match externally specified hardware or platform-oriented data structures [3, 4]. The syntax for bitdata definitions is described by the following grammar:

```
BitDataDecl  = bitdata Conid                -- name (no parameters)
               Opt([ Type ])                -- width specification
               Opt("=", Sep(BitdataCon, "|")) -- constructors
               Opt(deriving DeriveList)     -- deriving clause

BitdataCon   = Con [ Sep(BitdataField, "|") ]   -- bitdata constructor

BitdataField = Varid :: TApp Opt("=", Expr)  -- labeled field
             | IntLiteral                    -- constant field
             | BitLiteral                    -- tag bits
```

Although bitdata definitions look similar to `data` definitions, they differ in several important respects:

- Parameters are not permitted on the left hand side of a bitdata definition.
- Each constructor specifies a number of named fields, which may include optional default values.
- If extra tag bits are needed to distinguish between different constructors, these must be written explicitly as constant fields (i.e., as unlabeled integer or binary literals).
- All of the constructors must have the same width; this may be achieved, if necessary, by including additional padding as part of each constructor's layout.
- The width of a bitdata type may be specified explicitly by adding an annotation of the form `[n]` on the left hand side of the definition. The symbol `n` here denotes an arbitrary type expression of kind `nat` whose value can be determined at compile time. This annotation is used purely as documentation; if the declared width does not match the inferred width, then an error will be reported. In particular, no attempt will be made to pad or truncate bit-level representations to match the declared width. Of course, the width of a bitdata type, whether declared or inferred, must be a valid instance of the `Width` class.
- The assumptions of no junk and no confusion for algebraic datatypes are not guaranteed for bitdata types. In particular, this means that there may be bit patterns of the given width that cannot be produced using only the constructors of the bitdata type (these are the so-called *junk* values), and there may be bit patterns that match multiple patterns, even when the constructors are distinct (this is the source of the so-called *confusion*).

Definitions with either junk or confusion are valid in Habit programs, although it is likely that a typical compiler will provide a compile-time option to request the generation of appropriate warning diagnostics when such definitions are encountered. In general, however, programmers are expected to tackle issues arising from the presence of junk or confusion directly. For example, it is possible to deal with junk by using wildcard patterns or by using the `isJunk` operator, and it is possible to deal with confusion by selecting an appropriate ordering of constructors in a definition that uses pattern matching.

As with datatype definitions, it is possible for a programmer to request automatic generation of derived instances of standard type classes by attaching a `deriving` clause to the end of a bitdata definition. In addition, if a bitdata definition has a single constructor, then the compiler will automatically generate appropriate instances of the `.x` type functions for each field name `x`. For example, a PCI address can be specified by a 16 bit value using the following format:

```
bitdata PCI = PCI [ bus :: Bit 8 | dev :: Bit 5 | fun :: Bit 3 ]
```

Given this definition, the compiler will automatically generate the following instances:

```
instance PCI.bus = Bit 8
instance PCI.dev = Bit 5
instance PCI.fun = Bit 3
```

As a result, given an expression `addr` of type `PCI`, we can obtain the corresponding bus, device, and function numbers using the expressions `addr.bus`, `addr.dev`, and `addr.fun`.

3.6.9 Area Declarations

Area declarations are used to define regions of memory conforming to specified layout and alignment constraints. In essence, this provides a mechanism for defining static, memory-based data structures, including simple global variables, structures, and arrays. Note, however, that there are some significant restrictions on the use of areas defined in this way because they are described in terms of the types of kind `area`, and they cannot be used to store arbitrary values of types with kind `*`. In particular, there is no way to store a general function in a memory area.

The syntax of area declarations is very similar to the notation for type signature declarations except that it begins with the `area` keyword:

```
AreaDecl  =  area Sep(Var, ",") :: SigType
```

The type portion of an area declaration should be equivalent to a type of the form `ARef l a`, specifying both an alignment, `l`, and an area type, `a`, that is part of a valid instance for `ByteSize`. A Habit compiler can then determine appropriate locations for the declared memory areas subject to these constraints, taking account of platform specific details, and avoiding conflicts with any other memory areas that are being used either for code or for data. Habit programs can access these regions of memory using the variables listed in the area declaration as references of the declared type.

It is expected that future versions of Habit will extend the syntax for area declarations with mechanisms for specifying placement of allocated memory areas (for example, within certain portions of the address space, or even at specific addresses), and for providing custom initialization of the fields within each allocated area. For the time being, however, we simply require that allocated regions are initialized using the `memZero` operation from the `ByteSize` class.

4 Standard Environment

A programmer's view of a language is determined not only by details of the language syntax (our focus in the previous section) but also by the built-in types and functions that it offers; the latter is the main topic of this section.

The standard environment (i.e., the standard prelude or standard libraries) for Habit programs is collection of classes, type functions, types and operations that can be used in any Habit program. Many of the components of the standard environment have been mentioned briefly in the preceding text (see, for example, the tables of standard types, classes, and type functions in Figures 1, 2, and 3, respectively). In this section, we describe each of these in more detail, including information about standard operations. For conciseness, however, and to avoid over-specification, we focus on presenting an informal signature for the standard environment, eliding some of the implementation-level details that would have to be addressed in a practical system.

In particular, we often use syntax like the following to document the details of built-in type constructors and operations, providing the appropriate kind and type information in each case:

```
primitive type Con :: Kind
primitive Var :: SigType
```

We also use `primitive` as a prefix for some class and type function declarations to distinguish classes that are built-in to the system (i.e., that do not admit user-

defined instances) from those that do allow user-defined instances, subject to the normal rules. Also, in some cases, we write instances for type functions using an underscore in place of the result type:

```
instance LE Signed = _    -- memory area holding little-endian
                          -- representation of a Signed value
```

The intention here is to signal that there is no way to refer to the result type directly by name; the only way that we can write a range type like this explicitly as part of a Habit program is by using the functional notation and writing it as `LE Signed`. Some readers may prefer to think of `_` here as a placeholder for some (potentially implementation-defined) primitive type whose name is not exported from the standard prelude.

Note that neither of the notations mentioned here—either using `primitive` or underscores in the range of a type function—is valid Habit syntax; these are just notations that we use for the purposes of documentation.

4.1 Basic Types

Function types in Habit are constructed using the symbol `->`, which is typically written as a right associative, infix operator:

```
infixr type 5 ->
primitive type (->) :: * -> * -> *
```

The `Bool` type, with constructors `False` and `True`, is defined as follows:

```
bitdata Bool = False [B0] | True  [B1]
               deriving (Eq, Ord, Bounded, BitSize, ToBits, FromBits)
```

Because Habit is a call-by-value language, we need to provide special treatment for the familiar `(&&)` and `(||)` operators to obtain the expected lazy/short-circuit semantics. Specifically, we parse and type check these two symbols as infix operators using the following fixities and types:

```
infixr 3 &&
infixr 2 ||
(&&), (||) :: Bool -> Bool -> Bool
```

but we interpret calls to these functions via macro-expansion (treating partial applications as syntax errors) using:

```
(e1 && e2)   =   if e1 then e2 else False
(e1 || e2)   =   if e1 then True else e2
```

The unit type has only one value and is defined as follows:

```
data Unit = Unit
            deriving (Eq, Ord)
```

For convenience, and because of its familiarity to Haskell programmers, we also use the notation `()` for both the unit type and its only value.

The `Maybe` type is most commonly used as the return type of a function that might fail if the inputs do not satisfy some appropriate condition. A successful call is represented by a result of the form `Just x`, while a result of `Nothing` indicates failure:

```
data Maybe t = Nothing | Just t
               deriving (Eq, Ord)
```

4.2 Type Level Numbers

Type-level numbers, which are just types of kind `nat`, are widely used in the standard environment Habit to describe, among other things, the widths of bit vectors, the alignments and sizes of memory areas, and the limits on valid array indices. As mentioned in Section 3.3.4, we use a small collection of type functions to describe basic arithmetic operations or constraints on type-level numbers, most of which are written using infix notation with the following associativities and precedences:

```
infixl type 6 +, -
infixl type 7 *, /
infix  type 4 <=, <
```

The type functions for addition and multiplication are defined as follows (note that we use prefix syntax here to match the grammar for class declarations, but note also that predicates like `(+) m n p` can (and usually are) written in the form `m + n = p` when they appear as part of a type signature.):

```
primitive class (+) (m :: nat) (n :: nat) (p :: nat)
  | m n -> p, m p -> n, n p -> m
```

```
primitive class (*) (m :: nat) (n :: nat) (p :: nat)
  | m n -> p
```

Note that there are three functional dependencies on the addition predicate: if any two of the types in `m + n = p` is known, then the third is uniquely determined. This property does not hold for multiplication—the value for `m` in `m * 0 = 0` is not uniquely determined, even though the second and third arguments of the predicate are known—and so there is only one functional dependency in this case. Details about the instances of these two classes are built-in to the Habit compiler, including general rules such as the following:

```
instance 0 + n = n
instance n + 0 = n
instance 0 * n = 0
instance n * 0 = 0
instance 1 * n = n
instance n * 1 = n
```

as well as an infinite set of basic arithmetic axioms, like the following, which are effectively generated on demand during type checking:

```
instance 1 + 1 = 2
instance 2 + 1 = 3
...
instance n + 1 = 0 fails
...
instance 1 * 2 = 2
instance 2 * 2 = 4
...
instance n * 2 = 3 fails
...
```

Axioms like these, together with the declared functional dependencies, are sufficient to allow a Habit compiler to simplify a predicate like `n + 1 = 3` by unifying `n` with `2`, and to recognize that predicates like `n + 1 = 0` and `n * 2 = 3` have no solutions at all, in which case the compiler can report an immediate error diagnostic.

Predicates for subtraction and comparison of type-level numbers can be defined in terms of addition[4]):

```
class (-) (m :: nat) (n :: nat) (p :: nat)
  | m n -> p, m p -> n, n p -> m
```

[4]There is no formal requirement for a Habit compiler to implement these operations in this way, however. Indeed, it might be preferable to build them in to the Habit compiler like addition and multiplication in the interests of obtaining better error diagnostics

```
instance (z + y = x) => (x - y = z)
else     (x - y = z) fails

class (<=) (x :: nat) (y :: nat)

instance x <= x+n   -- equivalent to (x + n = y => x < y)
else     x <= y fails

class (<) (x :: nat) (y :: nat)

instance  (x + 1 <= y) => (x < y)
else     x < y fails
```

In a similar way, we can define division on type-level numbers in terms of multiplication.

```
class (/) (m :: nat) (n :: nat) (p :: nat)
  | m n -> p, n p -> m   -- note extra fundep

instance (m / 0 = n) fails
else     (p * n = m, 0 < n) => (m / n = p)
else     (m / n = p) fails
```

Note that the second functional dependency for division is valid because we explicitly exclude the possibility of division by zero.

There are two additional primitive type functions on type-level numbers that provide a means for computing powers of two (writing `Exp2 n = p` if `p = 2^n`) and greatest common divisors (writing `GCD m n = p` if `p` is the greatest common divisor of both `m` and `n`). The definitions of these classes are as follows:

```
primitive class Exp2 (n :: nat) (p :: nat)
  | n -> p, p -> n

primitive class GCD (m :: nat) (n :: nat) (p :: nat)
  | m n -> p
```

Again, instance rules of the form illustrated below can be generated on demand by a Habit compiler to define a formal interpretation of these two classes:

```
instance Exp2 0 = 1
instance Exp2 1 = 2
instance Exp2 n  = 3 fails
...
instance Exp2 12 = 4K
```

```
...
instance GCD 1 n = n
instance GCD n 1 = n
instance GCD 2 3 = 1
instance GCD 6 10 = 2
...
```

Of course, these rules, and their combination with functional dependency information, fall far short of providing a complete algorithm for deciding satisfiability or for solving arbitrary arithmetic formulas, but they are actually quite effective in many of the simple cases that occur in systems programming.

4.3 Standard Classes

The Habit standard environment includes simplified versions of the most common type classes in Haskell for describing equality (class `Eq`), ordering (classes `Ord` and `Bounded`), and basic arithmetic (class `Num`). In this section, we provide the definitions of these classes, including the type signatures (and, where appropriate, fixities) of associated class members. Details of specific instances for these classes appear in subsequent sections as we discuss each of Habit's primitive types.

We start with the definition of the set of equality types, `Eq`, which also includes the equality test operation, `==`, as well as its logical complement, `/=`. A default definition is provided for the latter, which is useful both as documentation and because it means that a programmer need only supply a definition for `==` when they define a new instance of the `Eq` class. As described previously, we also include an explicit `fails` instance to exclude function types from `Eq`.

```
infix 4 ==, /=

class Eq t where
  (==), (/=) :: t -> t -> Bool
  x /= y      = not (x == y)   -- default definition

instance Eq (a -> b) fails
```

Although programmers can, in principle, provide an arbitrary semantics for the definition of equality on new, user-defined types, the intention is that `==` should always correspond strongly to a notion of *structural equality*, modulo details of representation. Note also that all derived instances of `Eq` assume structural equality.

The `Ord` class represents the set of types whose elements admit a total, structural ordering relation. An instance of `Ord` can be specified by providing definitions for the `<` and `<=` ordering functions, and then default definitions are used

to provide implementations for the symmetric > and >= variants as well as operations for computing the minimum and the maximum of a pair of values:

```
infix  4 <=, <, >, >=

class Ord t extends Eq t where
  (<), (<=), (>), (>=) :: t -> t -> Bool
  min, max             :: t -> t -> t

  -- default definitions:
  x > y   = y < x
  x >= y  = y <= x
  min x y = if x <= y then x else y
  max x y = if y <= x then x else y
```

Note that `Ord` lists `Eq` as a superclass, so the ordering functions such as `<=` should only be defined for equality types and should produce results that are consistent with the underlying equality.

Like Haskell, Habit also provides a `Bounded` class for describing types that have minimal and maximal elements:

```
class Bounded t extends Ord t where
  minBound, maxBound :: t
```

Support for basic arithmetic on numeric types is provided by the `Num` class, which includes operations for addition, subtraction, multiplication, and unary minus (the `negate` operator).

```
infixl 7 *
infixl 6 +, -

class Num t where
  (+), (-), (*) :: t -> t -> t
  negate        :: t -> t

  x - y = x + negate y   -- default definition
```

Even for types like `Unsigned` that do not include any negative elements, the `negate` operator still makes sense as a function for computing additive inverses. It is not actually necessary to include a definition of subtraction as part of an instance of `Num` because a default implementation using only `negate` and addition is provided. (However, it may be appropriate to include a specific definition in cases where a more implementation is possible; for example, many machines allow subtraction of word values using a single machine instruction.)

4.4 Numeric Literals

One detail of the `Num` class in Haskell that is conspicuously absent from the Habit version of `Num` is the `fromInteger` function that is used to support the handling of numeric literals in Haskell. One reason that we do not include the same function here is that there is no built-in, arbitrary precision `Integer` type in the Habit standard environment. A more compelling reason, however, is that Habit uses a different approach for handling numeric literals that leverages the type system to provide stronger coupling between types and values.

Specifically, any occurrence of a numeric (integer) literal, `n`, in Habit source code is treated as the application of the `fromLiteral` function to a value of the singleton type `Nat n`. These primitives are defined as follows:

```
primitive type Nat :: nat -> *

class NumLit n t where
  fromLiteral :: Nat n -> t
```

For example, an occurrence of the literal `42` in the source of a Habit program will behave initially (i.e., before considering the context in which it appears) as a value of type `NumLit 42 t => t`. Now, by providing appropriate instance declarations, a simple literal like this can be treated as having many different types, subject to constraints imposed by the corresponding instances of the `NumLit` class. For example, it makes sense to consider `42` as a value of type `Bit 6`, but not as a value of type `Bit 5` because the largest value of that type is `31`. The following declaration captures a general rule that allows numeric literals to be used as bit vector literals so long as the bit vector width is large enough to represent the specified literal value:

```
instance (n < Exp2 m) => NumLit n (Bit m)
```

This provides a flexible and extensible mechanism for handling numeric literals. With simple variations, for example, it is possible to define a type that allows only nonzero literals, a type in which only even literals are valid, or a type whose literals are always powers of two! In this way, the `NumLit` class provides a connection between the numeric literals that appear as values in Habit source programs and the corresponding type-level numbers that can be used to enforce data or system invariants as a result of type checking.

4.5 Division

Although operations like addition, subtraction, and multiplication are typically used more frequently in systems programming, there are also situations

where it is necessary to perform a division. But introducing a simple division operator, `div :: t -> t -> t`, to support this functionality is problematic because it does not account for the possibility of an implicit attempt to divide by zero, which, on many machines, triggers a hardware exception that will typically need to be trapped and handled in some manner by the operating system.

In Habit, we use types instead to ensure, at compile-time, that the second argument of a division will never be zero, this is accomplished by treating division as an operation with type, `div :: t -> NonZero t -> t`. Here, `NonZero t` is a special type that represents all nonzero values of type t. In fact `NonZero` is actually a two parameter type function with the following definition:

```
infixl 7 `quot`, `rem`, `div`, `mod`

class NonZero t = t' | t -> t' extends Num t where
  nonZero             :: t -> Maybe (NonZero t)
  div, mod, quot, rem :: t -> NonZero t -> t
```

A key detail here is that there are only two ways to construct values of type `NonZero t` to use as a divisor. The first is to use the `nonZero` method, which will fail (non-catastrophically) by returning `Nothing` if the input is zero. The second is by writing a literal, and using types to check for a zero at compile time:

```
instance (NumLit n t, 0 < n) => NumLit n (NonZero t)
```

The overall effect is to ensure that every dividend has been checked before attempting to perform a division, preventing the possibility of a divide by zero exception. In the special case of division by a constant, however, the check is performed at compile time, without run time overhead.

4.6 Index Types

Habit provides a family of types of the form `Ix n` each of which represents the natural numbers from `0` up to but not including `n`. We refer to these as *index types* because their values can be used to give the index of a component in a larger structure such as a bit vector (Section 4.7) or an array (Section 4.13). Moreover, if we can be sure that the larger structure has (at least) `n` elements, then we can use values of type `Ix n` to index into the structure efficiently and safely without a run-time range check.

The following definitions introduce the `Ix` type constructor as well as a class, `Index`, to specify which type-level numbers can be used as arguments to `Ix`[5].

[5]Technical note: These definitions are sufficient to ensure that a value of an index type will fit

```
primitive type Ix :: nat -> *

class Index n extends 0 < n where
  incIx, decIx :: Ix n -> Maybe (Ix n)
  maybeIx      :: Unsigned -> Maybe (Ix n)
  modIx        :: Unsigned -> Ix n
  (<=?)        :: Unsigned -> Ix n -> Maybe (Ix n)
  relaxIx      :: (Index m, n < m) => Ix n -> Ix m

instance  (n < WordSize) => Index (Exp2 n)
  -- implementation can use bit-oriented operations (e.g., masking)
else (n < Exp2 WordSize) => Index n
  -- implementation uses modulo arithmetic
```

The `incIx` and `decIx` operations can be used to increment or decrement an index value, returning either `Nothing` if the input is already at the limit of its range, or else a value `Just i` for some new index value `i`. The `maybeIx` function works in a similar way but takes an arbitrary `Unsigned` input, while `modIx` uses modulo arithmetic to ensure a valid index. In practice, however, the checked comparison primitive, `<=?`, is often most flexible in code that iterates over a sequence of index values because it uses a comparison with some programmer-specified upper bound to implement an appropriate range check. (The `maybeIx` operator, for example, is really just a special case with `maybeIx u = u <=? maxBound`.)

Index types support the usual operations for equality and ordering. In addition, an instance of `NumLit` for index types allows numeric literals to be used as index values, subject to a compile time range check. Note, however, that we do not allow index arithmetic and hence there is no `Num` instance for index types:

```
instance Eq (Ix n)
instance Ord (Ix n)
instance Bounded (Ix n)
instance Num (Ix n) fails
instance (Index n, i < n) => NumLit i (Ix n)
```

4.7 Bit Vector Types

Habit provides a family of bit vector types. Specifically, a value of type `Bit n` is a bit vector with `n` bits:

```
primitive type Bit :: nat -> *
```

within a single machine word. It is permitted for an implementation to provide more instances of `Index` than this, but we do not require that because it seems likely that it would complicate a typical implementation and unlikely that it would be useful in practice.

As mentioned previously (Section 3.2), Habit provides special syntax for bit literals, such as `B0 :: Bit 1`, `B101 :: Bit 3`, etc., as well as a primitive (Section 3.3.4) for concatenating bit vectors:

```
primitive (#) :: Bit m -> Bit n -> Bit (m + n)
```

The reverse operation, breaking a bit vector into two (or more) constituent pieces, can be performed using bit patterns (Section 3.5).

Basic operations on bit vectors are provided by the following built-in instances:

```
instance Width n => Eq (Bit n)
instance Width n => Ord (Bit n)
instance Width n => Bounded (Bit n)
instance Width n => Num (Bit n)
instance Width n => NonZero (Bit n) = _
instance (Width n, i < Exp2 n) => NumLit i (Bit n)
```

In particular, numeric literals for values of type `Bit n` are allowed only for literals that are less than $2^{\mathtt{n}}$ (i.e., literals that are representable in `n` bits).

The instances above include a `Width n` constraint that potentially restricts the set of valid bit vector widths to which the class operations can be applied.

```
primitive class Width (n::Nat) extends Index n
```

All values of `n` that are less than or equal to the width of a word on the underlying machine must be valid instances of `Width`, so the above operations can be used on any bit vector that fits within a single machine word. A particular implementation may provide additional instances of `Width`, but this is not required. Note also that every instance of `Width` is also required to be an instance of `Index`; this is used in the `BitManip` class in Section 4.8 where index values are used to reference individual bits within a bit vector.

4.8 Bit-Level Representation Classes

For some programming tasks, it is necessary to inspect, and perhaps even manipulate bit-level representations of data. Habit reflects this with the definition of a primitive type function and three type classes. The type function, called `BitSize`, identifies the set of types `t` for which a bit-level representation has been exposed, and specifies the associated bit vector width.

```
primitive class BitSize (t :: *) = (n :: nat) | t -> n
```

The definition of `BitSize` as a type function should make it clear that this mechanism is intended only for types that are represented by fixed-width bit vectors, and not for higher-level aggregates that might require variable width representations or parsing of potentially unbounded bit streams.

The first two type classes, called `ToBits` and `FromBits`, provide functions for inspecting the underlying bit representation of a given input value, and for constructing values from a bit-level representation. It is necessary to separate these two roles because there are some types where it is useful to have the functionality of `ToBits`, but unsafe to provide the functionality of `FromBits`. It can be useful to inspect the bit representation of a memory area reference, for example, but we should not allow the construction of a reference from an arbitrary bit vector because this would make it possible to create invalid references and to compromise memory safety. The definitions of these classes are as follows:

```
primitive class ToBits t where
  toBits :: t -> Bit (BitSize t)

primitive class FromBits t extends ToBits t where
  fromBits :: Bit (BitSize t) -> t
  isJunk   :: t -> Bool
```

Note that `FromBits` includes `ToBits` as a superclass; this can sometimes lead to simpler types, and we have not yet encountered any examples where it us useful to be able to construct values from a given bit-level representation without also being able to inspect those representations.

Habit also provides a collection of operations for manipulating individual bits within a bit vector, which we capture with a third class:

```
class BitManip t extends (FromBits t, Ix (BitSize t)) where
  bit                       :: Ix (BitSize t) -> t
  setBit, clearBit, flipBit :: t -> Ix (BitSize t) -> t
  bitSize                   :: t -> Ix (BitSize t)
  testBit                   :: t -> Ix (BitSize t) -> Bool
```

The intention here is that `bit i` returns a value of type `t` in which the $\texttt{i}^{th}$ bit has been set; `setBit x i`, `clearBit x i`, and `flipBit x i` return a modified copy of the value `t` with the $\texttt{i}^{th}$ bit set, cleared, or flipped, respectively; `bitSize x` returns the index of the most significant bit in `x`; and `testBit x i` tests to see if the $\texttt{i}^{th}$ bit of `x` is set.

Of course, the bit vector types from Section 4.7 provide prototypical instances for each of the classes that we have described in this section:

```
instance Width n => BitSize (Bit n) = n
instance Width n => ToBits (Bit n)
```

```
instance Width n => FromBits (Bit n)
instance Width n => BitManip (Bit n)
```

4.9 Boolean and Shift Classes

Strict versions of Boolean operations—including and, or, xor, and complement—are meaningful on a range of different types including both `Bool` and `Bit n` types, so we describe these operations in more general form using a type class with appropriate instances:

```
infixl 7 .&.    -- bitwise and
infixl 6 .^.    -- bitwise exclusive or
infixl 5 .|.    -- bitwise inclusive or

class Boolean t where
  (.&.), (.|.), (.^.) :: t -> t -> t
  not                 :: t -> t

instance Boolean Bool
instance Width n => Boolean (Bit n)
instance (Index p, Exp2 n p) => Boolean (Ix p)
```

Note that we also include an instance of `Boolean` for index types of the form `Ix p`, but only in the special case where `p` is a power of two.

Shift operations are not included in `Boolean` but instead packaged in a subclass because they are not particularly meaningful for all `Boolean` types, such as `Bool`:

```
infixl 8 shiftL, shiftR  -- shift left (/2), shift right (*2)

class Shift t extends Boolean t where
  shiftL, shiftR  :: t -> Unsigned -> t

instance Width n => Shift (Bit n)
instance (Index p, Exp2 n p) => Shift (Ix p)
```

4.10 Words

The `Unsigned` and `Signed` primitive types represent unsigned and signed word values, respectively, in the underlying machine's natural word size. These types can be used for general and efficient arithmetic in the absence of specific size or representation requirements.

```
primitive type Unsigned :: *
primitive type Signed   :: *
```

Both types are instances of the `Eq`, `Ord`, `Num`, `Bounded`, `Boolean`, `Shift`, `ToBits`, `FromBits`, and `BitManip` classes. The appropriate `BitSize` instances are:

```
instance BitSize Unsigned = WordSize
instance BitSize Signed   = WordSize
```

The type `WordSize` used here is a primitive type-level number that is defined in the standard environment:

```
primitive type WordSize :: nat  -- architecture specific
```

It is, of course, convenient to allow numeric literals to be interpreted as `Unsigned` or `Signed values`

```
instance (i < Exp2 WordSize)       => NumLit i Unsigned
instance (i < Exp2 (WordSize - 1)) => NumLit i Signed
```

Habit also provides classes (`ToUnsigned` and `ToSigned`), with member functions (`unsigned` and `signed`), to support conversion of word values, bit vectors and index values into corresponding (`Unsigned` or `Signed`) word values:

```
class ToUnsigned t where
  unsigned :: t -> Unsigned
instance ToUnsigned Unsigned
instance ToUnsigned Signed
instance Width n => ToUnsigned (Bit n)
instance Index n => ToUnsigned (Ix n)

class ToSigned t where
  signed :: t -> Signed
instance ToSigned Unsigned
instance ToSigned Signed
instance Width n => ToSigned (Bit n)
instance Index n => ToSigned (Ix n)
```

The `unsigned` function converts values to `Unsigned` words using zero extension if the input has fewer than `WordSize` bits or truncation if the input has more than `WordSize` bits. In a similar way, the `signed` function converts values to `Signed` words using sign extension if the input has fewer than `WordSize` bits or

truncation if the input has more than `WordSize` bits. In practice, `unsigned` and `signed` are likely to be implemented as identity functions, at least in common cases, reflecting a change of type, but not a change of value.

4.11 Pointed Types

Many of the types that arise naturally in systems programming do not fit the model of pointed types in Haskell where every type, without exception, has a so-called *bottom* value representing failure to terminate in addition to any other elements. The advantage of the Haskell approach is that the presence of bottom elements is sufficient to guarantee that every recursive definition has a (least) solution, which means that recursion can be used freely within Haskell programs. A downside, however, is that the extra bottom elements result in clutter that complicates reasoning and reduces the precision of the type system.

Habit supports the use of *pointed types* as in Haskell, but also allows the definition and used of *unpointed types*. The latter do not include a bottom element, and hence it is not possible to define values of an unpointed type using general recursion. As a result, however, it is possible to obtain strong termination guarantees for programs that manipulate only unpointed types.

The basic approach that is used to support combinations of both pointed and unpointed types together in Habit was first suggested by Launchbury and Paterson [12]. In particular, it relies on the use of a type class that includes all of the `Pointed` types and provides the foundations that are necessary to support recursion (captured here by operations that return the `bottom` value and an associated fixpoint combinator, `fix`, for each such type):

```
primitive class Pointed t where
  bottom :: t
  fix    :: (t -> t) -> t
```

The following instance declaration provides an important component of the definition of the `Pointed` class, indicating that a function type with a pointed range type is itself pointed:

```
instance Pointed t' => Pointed (t -> t')
```

Instances of the `Pointed` class for other types are generated by the compiler as necessary. In particular, types defined as `bitdata` are not included (the compiler can generate appropriate `instance ... fails` declarations for these cases), while types defined using `data` are included only if the `Pointed` class is listed explicitly as part of the `deriving` clause. (Note that pointedness constraints are required for some forms of datatype definition to ensure that the definition is

valid; in these cases, it is an error for the programmer to omit the `Pointed` from the `deriving` clause.)

Because Habit is a call-by-value language, it is not possible to define a function `f :: P -> U` where `P` is pointed and `U` is unpointed. If such a function could be defined, then it could be applied to the bottom value of type `P`, producing a bottom value of type `U` as a result, which contradicts the assumption that `U` is unpointed. The Habit type system enforces this restriction by requiring that a predicate of the form `t =<= u` holds for every function type `t -> u` that is used in a program. This predicate, which captures an informal intuition that `u` is at least as pointed as `t`, can be defined as follows:

```
instance          (a =<= a)         -- reflexivity
else Pointed b => (a =<= b)         -- pointed on right
else Pointed a => (a =<= b) fails   -- pointed on left, unpointed right
else              (a =<= b)         -- unpointed on left
```

For the purposes of simplifying constraints of the form `t =<= u`, it is useful to add some extra rules that are consistent with the above definition, but not necessarily easy to extract and apply automatically:

```
a =<= (b -> c)   <=>   a =<= c      -- function space on right
(a -> b) =<= c   <=>   b =<= c      -- function space on left
```

To reduce clutter, constraints of the form `t =<= u` can be omitted from Habit type signatures if they are implied by the structure of the rest of the type. For example, the function `\x y -> x` has type `a -> b -> a`, which includes two function arrows, and hence requires two `=<=` constraints to ensure validity:

```
(a =<= (b -> a), b =<= a) => a -> b -> a
```

Using the rules above, this can be simplified to either of the following forms:

```
(a =<= a, b =<= a) => a -> b -> a   -- function space on right
(b =<= a) => a -> b -> a            -- reflexivity
```

In a Habit program, however, we can use the following simple form, leaving out the constraints altogether because they are implied by the form of the type:

```
const :: a -> b -> a
const  = \x y -> x
```

It is important to note, however, that this is just a syntactic abbreviation (i.e., a matter of presentation). Even though it may not be not written down explicitly, the `b =<= a` constraint is still part of the formal type for `const`, and an expression of the form `const u p` will trigger a type error if `u` has an pointed type while `p` is unpointed.

An alternative way to ensure validity of the type signature for `const` would be to add a pointedness constraint, as in:

```
const :: Pointed a => a -> b -> a
const  = \x y -> x
```

In this case, no additional `=<=` constraints are required (because they are implied by the `Pointed a` constraint), but the resulting version of `const` is less general because it can only be used in cases where `a` is a pointed type.

The treatment of pointed and unpointed types in Habit is one of the most unusual aspects of the language design and will be described in more detail in future versions of this report.

4.12 Monads

As described in Section 3.4.1, Habit provides special syntactic support for programming with monads. This allows the definition and use of functions that work over a range of different monads so long as they have all been defined as instances of the following class:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Instances of the `Monad` class are generally expected (but not required) to satisfy the standard monad laws:

```
return e >>= f   =   f e                        -- left identity
e >>= return     =   e                          -- right identity
(e >>= f) >>= g  =   e >>= (\x -> f x >>= g)    -- associative
```

The operators of the `Monad` class are used to provide a semantics for `do` expressions by repeated use of the following rewrites:

```
do { x <- e; s }  =  e >>= \x -> do { s }
do {      e; s }  =  e >>= \_ -> do { s }
do { let ds; s }  =  let ds in do { s }
do { e }          =  e
```

4.13 Memory areas, References and Alignments

Habit provides direct support for manipulating memory-based data structures using a combination of area and references types [2].

An area type (i.e., a type of kind `area`) describes the layout of a block of memory. Habit includes primitives for area types that can hold basic values (such as `Unsigned` and `Signed` words) as well as primitives for defining (statically-sized) arrays/tables. In addition, structure types (Section 3.3.2) can be used to describe the layout of record-like blocks of memory whose individual components that can be accessed by name. Memory areas cannot be manipulated as first-class values because they have kind `area` rather than kind `*`. Instead, memory areas are accessed and manipulated via references using operations that make reads and writes to memory explicit.

Reference types, whose values correspond to machine addresses, are used as pointers to specific regions of memory. (The necessary storage space can be reserved using the `area` declarations described in Section 3.6.9.) Reference types of the form `ARef l a`, for example, include a specification of memory layout, given by a type `a` of kind `area`, as well as an alignment, given by a type `l` of kind `nat`. The latter indicates that the associated machine address must be a multiple of `l`. Alignment specifications are sometimes used to enforce hardware constraints (for example, to ensure positioning of data on word, cache-line, or page boundaries). Alignments can also be used to reduce the number of bits that are needed to store a reference. For example, a `4K` aligned reference in a 32 bit machine can be represented using only 20 bits; there is no need to store the lower bits explicitly because every multiple of `4K` has zeros in its least significant 12 bits. A simpler reference type of the form `Ref a` can be used in cases where alignment is not important, in which case a default (minimal) alignment is assumed:

```
primitive type ARef :: nat -> area -> *
primitive type MinAlign :: nat
type Ref = ARef MinAlign

instance Alignment l => Eq (ARef l t)
```

The `MinAlign` constant here reflects the minimum alignment that is allowed on the underlying platform. On machines that require word alignment, for example, we would have `MinAlign = 4`. On a machine that allows arbitrary alignment, we would have `MinAlign = 1`. There is also a class, `Alignment`, whose instances are the type-level numbers corresponding to legal alignment values on a particular platform.

```
primitive class Alignment (l :: nat)
```

Of course, `MinAlign` must be an instance of `Alignment`, but the details beyond that are architecture specific. The following instances suggest some possibilities, and we hope to standardize on a specific choice that will be usable on a broad range of platforms in future versions of this report:

```
instance Width n => Alignment (Exp2 n)   -- MinAlign = 1
instance (n > 0) => Alignment (4*n)      -- MinAlign = 4
instance (n > 0) => Alignment n          -- MinAlign = 1
```

Basic area types, `LE t` and `BE t`, are provided for types `t` with a bit-level representation that takes some whole number of bytes, that is for types that are instances of `FromBits` with `BitSize t` a multiple of eight. Area types of the form `BE t` use big-endian representations (i.e., the most significant byte is stored first/at the lowest address) while those of the form `LE t` use little-endian representations (i.e., the least significant byte is stored first/at the lowest address).

```
primitive class BE (t :: *) = (a :: area) | t -> a
instance (ToBits t, BitSize t = 8 * n) => BE t = _

primitive class LE (t :: *) = (a :: area) | t -> a
instance (ToBits t, BitSize t = 8 * n) => LE t = _
```

As these (pseudo) declarations suggest, `BE t` and `LE t` are implemented as type functions, mapping types `t` to appropriate (but unnamed) primitive area types. The underlying names of these area types are not visible in user programs (i.e., they are not exported from the standard environment) so they can only be referenced indirectly using the names `BE t` and `LE t`.

For practical purposes, the distinction between `BE t` and `LE t` is only likely to be significant in situations where the precise structure of a memory area is required to match some external specification such as an operating system API or a hardware data sheet. In other situations, it will normally be preferable to use areas of type `Stored t` which use the native (and typically most efficient) representation for the underlying platform. (In practice, `Stored t` is likely to be a synonym for either `BE t` or `LE t`, but this is not guaranteed.)

```
primitive class Stored (t :: *) = (a :: area) | t -> a
instance (ToBits t, BitSize t = 8 * n) => Stored t = _
```

In addition to the primitive area types—`LE t`, `BE t` and `Stored t`— and the `struct` area types described in Section 3.3.2, Habit also provides support for memory based array or table structures. Specifically, a type `Array n a` describes a memory area that contains a contiguous block of `n` component areas each with layout `a`. The only special function for working with arrays of this kind is the

array indexing operation, `@@`, which takes a reference to an array and an index (guaranteed, as a value of type `Ix n` to be in the correct range) and returns a reference to the corresponding component area:

```
primitive type Array :: nat -> area -> area

primitive (@@) :: ARef l (Array n a)
                    -> Ix n
                       -> ARef (GCD l (ByteSize a)) a
```

The return type of the `@@` shown here is a little complicated because it includes the arithmetic that is needed to compute the alignment of the resulting pointer. As a special case, `@@` can also be treated as a function of the simpler and more intuitive type: `Ref (Array n a) -> Ix n -> Ref a`.

The type `ByteSize a` is an application of the following primitive type function, which returns the number of bytes in an arbitrary memory area.

```
primitive class ByteSize (a :: area)  (n :: nat) | a -> n

instance ByteSize (BE t)      = BitSize t / 8
instance ByteSize (LE t)      = BitSize t / 8
instance ByteSize (Stored t)  = BitSize t / 8
instance ByteSize (Array n t) = n * ByteSize t
```

These instances cover the cases for primitive memory areas and arrays. Additional instances are generated automatically for `struct` types in the obvious way: the `ByteSize` of a `struct` type is just the sum of the `ByteSize` values of its components.

Values that are stored in memory areas are accessed via a small set of monadic primitives that are captured by the following class declaration[6]:

```
class MemMonad m extends Monad m where
  memZero  :: ARef l a -> m ()
  memCopy  :: ARef l a -> ARef l' a -> m ()
  readRef  :: ARef l a -> m (ValIn a)
  writeRef :: ARef l a -> ValIn a -> m ()
```

The `memZero` and `memCopy` operations are used to initialize or copy the contents of one memory area to another area of the same type. The `readRef` and `writeRef`

[6]By defining a class of monads that support these operations instead of hardwiring them to a fixed monad, we hope to avoid the feature creep that has occurred as ever more functionality (and complexity) has been added to the `IO` monad in Haskell.

operations are used to read and write the values stored in the referenced memory regions. The types of these operations use the `ValIn` type function to determine the type of value that is stored by a given area type:

```
primitive class ValIn a = t | a -> t

instance ValIn (BE t)     = t
instance ValIn (LE t)     = t
instance ValIn (Stored t) = t
```

Note that Habit only provides instances of `ValIn` for basic area types, so it is not possible to read (or write) a complete array or `struct` area using a single `readRef` (or `writeRef`) call.

5 Extended Example: Memory-based Arrays

This section describes an extended example of programming in Habit—an implementation of a memory-based, maximum heap data structure of thread priorities. From a functional programming perspective, this is an unusual choice because it does not make heavy use of Habit's conventional functional programming features such as algebraic datatypes and higher-order functions. We have chosen this example, however, to demonstrate how some of the other, less familiar features of Habit might be used in a systems programming context. In fact this particular example was originally implemented in C as part of `pork`, a prototype implementation of an L4 microkernel, and the Habit implementation is written in a very similar style. For the purposes of comparison, we include both the C and Habit versions of the code in the following text.

To provide more background, we begin with a summary of how this example fits in to the implementation of `pork` (Section 5.1). We then describe the main data structures that are used (Section 5.2), and the algorithms for inserting a priority (Section 5.3), removing a priority (Section 5.4), and determining the highest priority (Section 5.5) from the priority set. We end with some reflections and conclusions based on the example (Section 5.6).

5.1 Background

As an implementation of the L4 microkernel, `pork` includes code for managing multiple address spaces and multiple threads of execution, including context switching code to move between different threads and code for handling system calls, machine exceptions, and hardware interrupts. In particular, `pork` configures the machine hardware to generate periodic timer interrupts that

interrupt the execution of user level code. As each interrupt occurs, the kernel updates an internal counter recording the amount of time that the current thread has been running and, if its timeslice has expired, determines which thread should be executed next. In L4, scheduling decisions like this are made on the basis of the priority values that are assigned to each thread.

The implementation of `pork` maintains a data structure, referred to in the source code as the *priority set*, that stores the priorities of all runnable threads. The priority set is implemented as a maximum heap data structure, which enables the kernel to determine the priority of the highest runnable thread in constant time. This, in turn, allows the scheduler to find the highest-priority runnable thread in constant time by indexing into an array of runqueues, one for each possible priority. On the downside, insertion and deletion into the priority set are $O(log(p))$ operations, where p is the size of the set of all distinct priorities. In practice, however, we expect that p is likely to be quite small (because many threads have the same priority), and that insertion and deletion are relatively uncommon, being necessary only when we add the first thread or remove the last thread at a given priority. And even if there are many active threads, there are only 256 possible priority levels in L4, so we know that $p < 256$. Although `pork` has yet to be heavily stress tested, these arguments support the choice of a heap data structure and suggest that the $O(log(p))$ costs for insertion and deletion will not be a problem in practice.

5.2 Data Structures

In this section, we turn our attention to concrete details of the implementation of the priority set. In the C code, the underlying data structures are as follows:

```
#define PRIOBITS   8             // Priorities are 8 bit values
#define PRIORITIES (1<<PRIOBITS) // Total number of priorities (256)

// Max Heap: children of i are 2i+1, 2i+2; parent of i is (i-1)/2
static unsigned prioset[PRIORITIES]; // A heap of active priorities
static unsigned prioidx[PRIORITIES]; // Index priorities in prioset
static unsigned priosetSize = 0;     // Number of entries in prioset
```

The `prioset` array stores the main heap structure with the relationship between parent and children indices that is described in the comments. The `priosetSize` variable records the number of distinct elements that are stored in the priority set; we start with an empty set, and hence the initial value of `priosetSize` is set to zero. The `prioidx` array records the index at which each priority value occurs within `prioset` and is used to help in the implementation of the delete operation. For example, if `prio` is a member of the current priority set, then `prioset[prioidx[prio]]` will be equal to `prio`. In general, the code maintains the relationship between `prioset` and `prioidx` by using a pair of lines like the

following every time that it writes a value, `prio`, to an index, `i`, of `prioset`:

```
prioset[i] = prio;
prioidx[prio] = i;
```

This pattern appears four times in the `pork` source code; it could have been abstracted as a function (perhaps marked to be automatically inlined) or as a macro, but either the possibility was not noticed, or else it was not considered to be worth the trouble.

In Habit, we can define the same data structures as memory areas using the following declarations (for stylistic reasons, we rename `PRIORITIES` as `NumPrio` and introduce a name, `Priority`, for the type of priority values:

```
type NumPrio      = 256  -- Number of priority levels
type Priority     = Ix NumPrio

area prioset     :: Ref (Array NumPrio (Stored Priority))
area prioidx     :: Ref (Array NumPrio (Stored (Ix NumPrio)))
area priosetSize :: Ref (Stored Unsigned)
```

Habit code to save a single value in the priority set looks very similar to the C code shown above, except for the addition of more precise types (which, in this case, could have been inferred from the body if we had not chosen to include the type as documentation)[7]:

```
-- Update priority set to save priority value prio at index i
prioSet       :: Ix NumPrio -> Priority -> M ()
prioSet i prio = do writeRef (prioset @ i) prio
                    writeRef (prioidx @ prio) i
```

With this definition, a call `prioSet i prio` updates the heap data structures to indicate that priority `prio` is stored at index `i` in the heap. (Note that we use the types `Ix NumPrio` and `Priority` to indicate the primary role for the two argument types. The fact that the types are synonyms of one another means that we can use both as array indices.)

In moving from C to Habit, we have taken the opportunity to use more precise types for the elements of the `prioset` and `prioidx` arrays. Why then are we still using an unsigned integer for `priosetSize`? Given that there are 256 different values of type `Priority`, and that we will allow each distinct priority to be included in the priority set at most once, it follows that the value of `priosetSize`

[7] In this section, we write `M` for some fixed monad that can be chosen arbitrarily except for the restriction that it must be an instance of the `MemMonad` class described in Section 4.13.

can only take values between 0 and 256. As such, it might seem natural to treat `priosetSize` as a value of type `Ix (NumPrio+1)`. However, if we use a value of this index type to record the size of the priority set then we will need to use a corresponding checked increment or decrement operation each time that we insert or remove a priority, which is awkward and redundant. Moreover, we do not know how to reflect the intuitions that we have relied upon in the argument above within the the Habit type system. For example, it is not easy see how we could arrange for an attempt to insert the same priority twice to be treated as a type error. After experimenting with several implementation choices here, our experience suggests that using a simple `Unsigned` value for `priosetSize` is the most practical choice. The consequences of this decision will be discussed again in the following text as we encounter uses of `priosetSize`.

5.3 Inserting a Priority

There is only one place in the `pork` source code where a value is inserted into the priority set: this is at the point where we add an element to an empty runqueue. For this reason, the C implementation uses the following code fragment inline as part of the body of `insertRunnable()` instead of defining a separate `insertPriority()` function:

```
// insert priority value "prio" into the priority set
heapRepairUp(prio, priosetSize++);
```

This code follows an increment of `priosetSize` with a call to an auxiliary function, `heapRepairUp()`, whose purpose is to restore the heap structure after a value has been inserted. As the name suggests, `heapRepairUp` works by percolating a possibly misplaced value from the end of the heap towards the root until it finds a position in which that value is greater than all of its children in the tree. The C implementation of this function is as follows:

```
/*-------------------------------------------------------------------
 * Insert "prio" into "prioset" given that (a) there is a gap at
 * index "i"; and (b) the rest of the structure, excluding "i" is
 * a valid heap.
 */
static void heapRepairUp(unsigned prio, unsigned i) {
  while (i>0) {
    unsigned parent = (i-1)>>1;
    unsigned pprio  = prioset[parent];
    if (pprio<prio) {
      prioset[i] = pprio;
      prioidx[pprio] = i;
      i = parent;
    } else {
```

```
      break;
    }
  }
  prioset[i] = prio;
  prioidx[prio] = i;
}
```

Following the same structure, we code these operations in Habit using a top-level `insertPriority` function and an associated `heapRepairUp` worker function:

```
insertPriority     :: Priority -> M ()
insertPriority prio = do s <- readRef priosetSize
                         writeRef priosetSize (s+1)
                         heapRepairUp (modIx s) prio

heapRepairUp :: Ix NumPrio -> Priority -> M ()
heapRepairUp i prio
  = case dec i of
      Nothing -> prioSet 0 prio  -- at the root
      Just j  -> do let parent = j>>1
                    pprio <- readRef (prioset @ parent)
                    if pprio < prio then
                      prioSet i pprio
                      heapRepairUp parent prio
                    else
                      prioSet i prio
```

An implicit precondition for the insert operation in the original C code, which we carry over directly to the Habit code, is that the priority value we are inserting is not already included in the priority set. Among other things, this precondition should be enough to ensure that the value stored in `priosetSize` will always be less than or equal to 256, and that the value of `s` in the body of `insertPriority` will always be less than or equal to 255. These properties, however, are not captured in the type system, and so we have used the `modIx` function to provide an explicit guarantee that a valid `Ix NumPrio` will be passed in to `heapRepairUp`. In this particular context, the call to `modIx` might be implemented by a single bitwise and instruction. However, if we are sure that the precondition is always satisfied, then that instruction is redundant and it will have no effect on the computation. To put it another way, the type of `insertPriority` is not strong enough to ensure that the precondition is satisfied, so additional steps (i.e., the call to `modIx`) must be taken to map the size of the set safely to a corresponding index.

5.4 Removing a Priority

Like the code for inserting a priority, there is only one place in the `pork` source code where it is necessary to remove a priority from the priority set: this is the point at which we remove the last runnable thread from a given priority queue. As a result, the C code for removing a priority is inlined into the `removeRunnable()` function that removes a runnable process from its runqueue. Again, there is an implicit precondition that the specified `prio` is a member of the priority set.

```
// remove priority value prio from the priority set
unsigned rprio = prioset[--priosetSize]; // remove last entry on heap
if (rprio!=prio) {         // we wanted to remove a different element
  unsigned i = prioidx[prio];
  heapRepairDown(rprio, i);
  heapRepairUp(prioset[i], i);
}
// The following is needed only if we want an O(1) membership test
prioidx[prio] = PRIORITIES;
```

The general algorithm for removing an element from the priority set is to shrink the heap by one element, reinserting the priority, `rprio`, that was previously stored at the end of the heap array in place of the priority, `prio`, that we are deleting. In the special case where these two priorities are the same, there is nothing for us to do beyond decrementing `priosetSize`. More generally, however, we must find the index of the priority that we are deleting (using `i = prioidx[prio]`), reinsert the removed priority into the subtree of the heap at that node (using `heapRepairDown(rprio, i)`, and then blend that subtree into the rest of the heap (using `heapRepairUp(prioset[i], i)`). The last line in the C code above inserts a value into the `prioidx` array that is technically out of range. As the comment indicates, this was intended to provide a mechanism for determining, in constant time, whether any given priority value was included in the priority set. In the end, however, we did not use this feature elsewhere in the `pork` code, so we have chosen not to replicate it in the Habit code below. In any case, if we wanted to reintroduce this kind of functionality later on in the Habit code, it would probably be better to do so using a separate array/bitmap of Booleans instead of extending the array to admit out of range values. (Indeed, this would not even require any additional space: the Habit version of `prioidx` requires only one byte for each possible priority, while the C version requires at least 9 bits (and, in fact, currently takes 32 bits) per priority in order to represent the `PRIORITIES` value.)

We have already seen the `heapRepairUp()` operation used in the code above, but `heapRepairDown()` is a second auxiliary function that is needed only for the remove operation. Its role is to move down the tree, comparing the value at each node with the values at each of its children to ensure that the (maximum)

heap property is satisfied. The trickiest part of implementing this function is to ensure that we only look at valid children as we descend the tree. This requires checking that the index values we compute for the left (`2i+1`) and right (`2i+2`) children of a given node `i` are not just valid indices for `prioset`, but also that they are less than `priosetSize`. The C implementation below calculates a candidate child index in the variable `c` and uses comparisons with `priosetSize` to distinguish between heap nodes with 2, 1, or no children:

```
/*---------------------------------------------------------------
 * Insert "prio" into "prioset" by replacing the maximum element
 * at "i".  Assumes that the left and right children of "i" (if they
 * exist) both satisfy the heap property.
 */
static void heapRepairDown(unsigned prio, unsigned i) {
  for (;;) {// move bigger elements up until we find a place for prio
    unsigned c = 2*i+1;
    if (c+1<priosetSize) {      // two children
      if (prio>prioset[c] && prio>prioset[c+1]) {
        break;
      } else if (prioset[c+1] > prioset[c]) {
        c = c+1;
      }
    } else if (c<priosetSize) { // one child
      if (prio>prioset[c]) {
        break;
      }
    } else {                    // no children
      break;
    }
    prioset[i] = prioset[c];
    prioidx[prioset[c]] = i;
    i                   = c;
  }
  prioset[i] = prio;
  prioidx[prio] = i;
}
```

Turning to Habit, we can code the top-level remove operation as follows, following the same basic pattern as in the C implementation.

```
removePriority     :: Priority -> M ()
removePriority prio = do s <- readRef priosetSize
                         writeRef priosetSize (s-1)
                         rprio <- readRef (prioset @ modIx (s-1))
                         if prio/=rprio then
                           i <- readRef (prioidx @ prio)
                           heapRepairDown i rprio (modIx (s-2))
```

```
                              nprio <- readRef (prioset @ i)
                              heapRepairUp i nprio
```

Unlike the C version, we have added a third parameter to the `heapRepairDown` function that provides the index of the last remaining element in the priority set. Among other things, this means that `heapRepairDown` can be written without having to read the value of `priosetSize` on each iteration.

The code for `removePriority` includes two calls to `modIx`; both of which we would, ideally, prefer to omit. The first is used to compute the index of the priority that had previously been stored in the last active slot of the heap. Given the precondition, we can assume that this code will only be executed when the set contains at least one element, so the index `s-1` is always valid. By a similar but slightly more complicated argument, the second call to `modIx` will only be needed when the set contains at least two elements (the priority, `prio`, that is being removed and the distinct priority, `rprio`, that will replace it), so the index `s-2` used here is also valid. It is reasonable to assume that theorem proving tools could be used to formalize these arguments and so justify removing the modulo arithmetic (or bitwise and) operations that are suggested by the `modIx` calls if the preconditions were guaranteed. If we are forced to rely only on the type system, however, then these two conversions remain as minor concessions to pragmatism in the Habit code.

Other than the addition of an extra argument, our Habit implementation of `heapRepairDown` follows a similar structure to the C code except that, instead of calculating a candidate child node `c`, we calculate index values `l` and `r` for left and right children, respectively, where they exist, using the `(<=?)` operator.

```
heapRepairDown :: Ix NumPrio -> Priority -> Ix NumPrio -> M ()
heapRepairDown i prio last
  = let u = unsigned i in
    case (2*u+1) <=? last of         -- Look for a left child
     Nothing -> prioSet i prio       --   i has no children
     Just l  ->                      --   i has a left child
       do lprio <- readRef (prioset @ l)
          case (2*u+2) <=? last of   -- Look for a right child
            Nothing ->               --   i has no right child
              if lprio > prio then
                prioSet i lprio
                prioSet l prio
              else
                prioSet i prio
            Just r  ->               --   i has two children
              rprio <- readRef (prioset @ r)
              if prio > lprio && prio > rprio then
                prioSet i prio
              else if (lprio > rprio) then
                prioSet i lprio      --   left is higher
```

```
                heapRepairDown l prio last
              else                    --   right is higher
                prioSet i rprio
                heapRepairDown r prio last
```

This example nicely illustrates the flexibility that we have to navigate an array in a non-linear manner using the `(<=?)` operator. Note that we can safely avoid any array bounds checks when we read the priorities `lprio` and `rprio` of the left and right children, respectively, because of the way in which we obtained the corresponding indices `l` and `r`.

5.5 Finding the Highest Priority

The effort that we invest in building and maintaining the priority set data structures pays off when we want to find the highest priority value for which there are runnable threads. This feature is used in the `pork` scheduler to allow selection of the next runnable thread in constant time in the `reschedule()` function shown below:

```
/*-------------------------------------------------------------------
 * Select a new thread to execute.  We pick the next runnable thread
 * with the highest priority.
 */
void reschedule() {
  switchTo(holder = priosetSize ? runqueue[prioset[0]] : idleTCB);
}
```

This code examines the value of `priosetSize` to decide if there are any runnable threads in the system, and then switches context, either to the next runnable thread at the highest priority, or else to the idle thread if the priority set is empty. (Note that the idle thread is only scheduled when there are no other runnable threads at any priority level so that it does not take time from any other thread. In effect, the idle thread runs at a reserved priority level below the lowest value that is permitted for any other thread.)

In the interests of providing a standalone priority set abstraction that is independent of details of context switching (`switchTo`), `runqueues`, or the current time slice `holder`, we will provide a Habit function to return either `Nothing` if the priority set is empty, or else `Just prio` where `prio` is the highest priority of a runnable thread. The code is straightforward:

```
highestPriority :: M (Maybe Priority)
highestPriority  = do s <- readRef priosetSize
                      if s==0 then
                        return Nothing
```

```
                      else
                        prio <- readRef (prioset @ 0)
                        return (Just prio)
```

Note that in this case we do not need a `modIx` operation because, so long as the set is non-empty, we can be sure that the `0` index is valid.

Using `highestPriority`, the original code for `reschedule` might be translated into code like the following:

```
reschedule :: M a
reschedule  = pickThread >>= switchHolderTo

pickThread :: M (Ref TCB)
pickThread  = case<- highestPriority of
                Nothing -> return idleTCB
                Just p  -> readRef (runqueue @ prio)

switchHolderTo    :: Ref TCB -> M a
switchHolderTo tcb = do writeRef holder tcb
                        switchTo tcb
```

While it is appealing to separate out the `highestPriority` operation like this, it could lead to some unnecessary work. In the code above, `highestPriority` is used to construct a result of type `Maybe Priority`, based on an internal test of the value of `priosetSize`, but then that value is subjected to pattern matching and immediately discarded by the code in `pickThread`. If we assume that the `Maybe` values involved here will be represented as unboxed values without the need for dynamic memory allocation, then the only real problem here is the overhead of an unnecessary test. This is probably not too significant from a performance perspective, but it was avoided completely in the original C program because of the way that two operations were fused together (a translation carried out by hand and blurring the abstraction boundary around the priority set implementation in the process). Fortunately, However, if the compiler performs some reasonable (whole-program) inlining and optimization, then it should be possible to obtain the same effect automatically, translating `reschedule` into the following code that avoids the use of intermediate `Maybe` values, just like the original C version:

```
reschedule = do s   <- readRef priosetSize
                tcb <- if s==0 then
                         return idleTCB
                       else
                         prio <- readRef (prioset @ 0)
                         readRef (runqueue @ prio)
                switchHolderTo tcb
```

There is, in fact, one other use of the priority set in `pork`, which appears at the end of the timer interrupt handler. By the time the kernel reaches this point in the code, it has acknowledged and re-enabled the timer interrupt, updated the system clock, performed basic timeslice accounting, and is preparing to return to the current timeslice holder having determined that its timeslice has not yet expired. (Timeslice periods can be set on a per thread basis in L4 and will typically span multiple clock ticks/timer interrupts.) A final step is needed to determine whether some higher-priority thread has become runnable since the last timer interrupt; this could occur, for example, as the result of an intervening hardware interrupt or system call. If a higher-priority thread has become runnable, then we switch to that instead of returning to the current timeslice holder. (The preempted holder remains at the front of the runqueue for its lower level priority so that it will still get the rest of its timeslice once the work of higher-priority threads has been done.)

```
ENTRY timerInterrupt() {
  ...

  // Here if infinite timeslice or current timeslice has not finished
  if (priosetSize && prioset[0] > holder->prio) {
    reschedule();               // preempt by higher priority thread?
  }
  resume();
}
```

This code can also be translated into Habit using `highestPriority`:

```
timerInterrupt
  = do ...
       ...
       case<- highestPriority of
         Just prio -> hprio <- readRef (holder.prio)
                       if prio > hprio then
                         reschedule
       resume
```

Considering the implementations for `reschedule` given above, we can see that a naive compilation of the code at the end of `timerInterrupt` will involve two tests of `priosetSize` along the path to preempting the current timeslice holder. Unless we somehow expect the value of `priosetSize` to change as a result of some external behavior/concurrency in the system, a possibility that can be captured explicitly in C by marking the variable as `volatile`, the second test of `priosetSize` is redundant. In the C version, absent a `volatile` annotation, we can expect that a reasonable optimizing compiler will automatically produce code that omits the second test. A clarification of the semantics of memory

areas will be required to determine whether the same result could be obtained in Habit. Alternatively, if this proved to be a real problem in practice, then it might just be better to rewrite the code by hand to eliminate the redundancy:

```
case<- highestPriority of
  Just prio -> hprio <- readRef (holder.prio)
               if prio > hprio then
                 readRef (runqueue @ hprio) >>= switchHolderTo
resume
```

Although examples like these will probably not be significant sources of problems in practical programs, they do suggest some possible goals for the design of an optimizing Habit compiler.

5.6 Conclusions

In this section, we have described a non-trivial example of Habit programming that uses memory-based arrays to re-implement a portion of the timer interrupt handler in `pork`. Among other features, this example shows how Habit types can be used to ensure safety for array access operations that are implemented without array bounds checking.

As far as code size or clarity is concerned, there is little to distinguish between the Habit version of the program and the original that was written in C. Given that the former was specifically written to follow the structure of the latter, this is probably not too surprising.

In terms of performance, it also seems reasonable to expect that a compiler for Habit could reasonably be expected to generate code of the same quality that we can obtain via C, at least if we assume the use of unboxed/unpointed types for unsigned, index, and reference values. In particular, provided that the compilation of monadic expressions and related primitives is handled in an appropriate manner, there is no need for heap allocation of either data structures or function closures. In addition, the only recursive calls in the Habit code are tail calls, which could be compiled directly into simple loops. The only places where it seems likely that we would not be able to obtain essentially the same machine code as we might get for the C version is in the three calls to `modIx`, one in `insertPriority` and two in `removePriority`. Apart from the (probably negligible) overhead of additional modulo arithmetic or bitwise and instructions, these are also, subjectively, the ugliest and most difficult to justify sections of the code. In their defense, the purpose of those calls is to establish invariants that are already implied by preconditions of the functions in whose definitions they appear. The real villain of the piece, perhaps, is our inability to express and enforce those preconditions from within the type system.

One advantage of the Habit code over the C version is that the former uses only

safe operations. While this does not remove the need for verification of algorithmic properties, it does mean that we can be sure of memory safety for all of the Habit code. By comparison, the C version uses unchecked array indexing operations and would require careful and detailed analysis of every line of code just to establish memory safety. It seems very likely, for example, that this would require us to establish a global invariant on the value of `priosetSize`. In addition, we would probably need to make an even broader assumption that no other part of the complete program, beyond the fragments of C shown here, could somehow use buggy address arithmetic to modify, either inadvertently or maliciously, the contents of any of the `prioset`, `prioidx`, or `priosetSize` global variables.

One possible criticism of the Habit code in this section is that it is written in a very C-like style, without leveraging many of the higher-level tools of functional programming. In principle, we might expect that properties relating to algorithmic or functional correctness would be more easily established for an implementation written in a more functional style. It would certainly be possible to write versions of the code that we have shown here in the higher-level style, for example using algebraic datatypes, higher-order functions, and perhaps even lazy evaluation. It is much harder to determine, however, what we would necessarily have to sacrifice in terms of performance and predictability in such an implementation. And, finally, although the functional code in this example may seem fairly low-level and imperative, the fact that it is written in Habit should mean that it can be called fairly easily from other, higher-level Habit code without having to resort to (sometimes fragile) inter-language working.

References

[1] Iavor S. Diatchki, Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. Writing systems software in a functional language: An experience report. In *Proceedings of the Fourth Workshop on Programming Languages and Operating Systems (PLOS 2007)*, Stevenson, WA, USA, October 2007.

[2] Iavor S. Diatchki and Mark P. Jones. Strongly typed memory areas. In *Proceedings of ACM SIGPLAN 2006 Haskell Workshop*, pages 72–83, Portland, Oregon, September 2006.

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

[4] Iavor Sotirov Diatchki. *High-Level Abstractions for Low-Level Programming*. PhD thesis, OGI School of Science & Engineering at Oregon Health & Science University, May 2007.

[5] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A Principled Approach to Operating System Construction in Haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, Tallinn, Estonia, September 2005.

[6] Bastiaan Heeren and Jurriaan Hage. Type Class Directives. In *Seventh International Symposium on Practical Aspects of Declarative Languages (PADL 05)*, Long Beach, California, USA, January 2005. Springer Verlag, Lecture Notes in Computer Science (LNCS 3350).

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

[8] Mark P. Jones. Simplifying and Improving Qualified Types. Technical Report YALEU/DCS/RR-1040, Yale University, New Haven, CT, USA, June 1994.

[9] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the Ninth European Symposium on Programming*, pages 230–244, Berlin, Germany, March 2000.

[10] Mark P. Jones and Iavor Diatchki. Language and program design for functional dependencies. In *Proceedings of the ACM Haskell Symposium (Haskell '08)*, Victoria, British Columbia, Canada, September 2008.

[11] L4Ka Team. L4 eXperimental Kernel Reference Manual, Version X.2. Technical report, System Architecture Group, Department of Computer Science, Universität Karlsruhe, May 2009. `http://www.l4ka.org/`.

[12] John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming (ESOP 1996)*, Linköping, Sweden, April 1996. Springer Verlag, Lecture Notes in Computer Science (LNCS 1058).

[13] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML—Revised*. MIT Press, May 1997.

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