The Habit Programming Language: The Revised Preliminary Report

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

November 2010

Contents

1 Introduction **3**

1.1 Document History 5

2 Background **6**

3 A Survey of the Habit Programming Language **8**

3.1 Notational Conventions 8

3.2 Lexical Syntax 10

3.3 Kinds, Types, and Predicates 13

3.3.1 Kinds 13

3.3.2 Types 15

3.3.3 Type Signatures 17

3.3.4 Predicates 18

3.3.5 Type Functions 20

3.4 Expressions 22

3.4.1 Applicative Expressions 22

3.4.2 Blocks 23

3.4.3 Local Declarations 24

3.4.4 Conditionals (`if` and `case`) 25
3.5 Patterns 26
3.6 Programs 28
3.6.1 Equations 29
3.6.2 Fixity Declarations 30
3.6.3 Type Signature Declarations 31
3.6.4 Class Declarations 31
3.6.5 Instance Declarations 33
3.6.6 Type Synonym Declarations 37
3.6.7 Datatype Declarations 38
3.6.8 Bitdata Type Declarations 39
3.6.9 Structure Declarations 44
3.6.10 Area Declarations 49

4 Standard Environment **50**
4.1 Basic Types 51
4.2 Type Level Numbers 52
4.3 Dot Notation: the Select and Update Classes 55
4.4 Standard Classes 57
4.5 Numeric Literals 59
4.6 Division 60
4.7 Index Types 61
4.8 Bit Vector Types 62
4.9 Bit-Level Representation Classes 63
4.10 Boolean and Shift Classes 65
4.11 Words 65
4.12 Pointed Types 67
4.13 Monads 69
4.14 Memory Areas, References and Alignments 70
4.15 Memory Area Initialization 74
4.15.1 Initialization of Stored Data 76

4.15.2 Null Initialization 76
4.15.3 No Initialization 77
4.15.4 Default Initialization 78

5 Extended Example: Memory-based Arrays 78
5.1 Background 79
5.2 Data Structures 80
5.3 Inserting a Priority 82
5.4 Removing a Priority 84
5.5 Finding the Highest Priority 87
5.6 Conclusions 90

1 Introduction

This report presents a preliminary design for the programming language Habit, a dialect of Haskell [14] that supports the 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.

1.1 Document History

This is the second version of the Habit language report, replacing the version dated August 2009, and incorporating a small number of changes and clarifications. In addition to many minor adjustments and tweaks, the most notable changes in this version of the report are as follows:

- Clarification of the syntax for type expressions (Section 3.3.2).
- Clarification of the dot notation mechanisms that Habit uses to describe selection of fields in bitdata and memory area structures (Sections 3.6.8, 3.6.9, and 4.3).
- New details about the syntax for constructing, updating, and matching against bitdata values (Section 3.6.8).
- New mechanisms for specifying memory area initialization (Section 4.15).
- A change from anonymous `struct [ ... ]` types to named structure types introduced by top-level `struct` declarations (Section 3.6.9); this change enables the definition of recursive structure types and allows the specification of default initializers.
- A small change to allow the keyword `type` as a synonym for `*` in kind expressions (Section 3.3.1).

We still consider this version of the report to be preliminary, and we expect that aspects of the design may still 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 use of side-effects using monads. At the same time, however, we 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, enabling us to write and run buggy code with problems that potentially could have been prevented or caught at compile-time.

The design of Habit is intended to preserve (or enhance) those aspects of Haskell that we found to be most useful in our previous work, but also seeks to address the problems that we encountered. Some of the changes—such as support for 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—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] with some extensions for the syntax of literals:

- *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.

  In practice, we expect that Habit implementations will distinguish between literate and regular source code by using the suffix of the source filename. The default convention is to use an `.lhb` suffix to indicate a literate source file or an `.hb` suffix to indicate a regular 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) 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 operators
      | ` Varid `
Con   = Conid                 -- Constructors
      | "(" Consym ")"
Conop = Consym                -- Constructor operators
      | ` Conid `
Op    = Varop                 -- Operators
      | Conop
Id    = Varid                 -- Identifiers
      | Conid
```

  A slightly different set of conventions is used in Habit type expressions (Section 3.3.2) to include the function type constructor `->`, allow only

`Varids` as type variables, and interpret `Varsyms` as type constructors. This allows the use of operator symbols like `+`, `*`, and `<` in predicates from the standard Habit environment, and so enables the use of traditional notation for type-level arithmetic (Section 3.3.4).

```
Tyvar   = Varid                  -- Type variables
Tyvarop = ‘ Varid ‘              -- Type variable operators
Tycon   = Con                    -- Type constructors
        | "(" Varsym ")"
        | "(" -> ")"
Tyconop = Conop                  -- Type constructor operators
        | Varsym
        | ->
Tyop    = Tyvarop                -- Type operators
        | Tyconop
```

- *Integer literals*. The syntax for integer literals in Habit—represented by `IntLiteral` in the grammar—mostly follows Haskell, 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 are followed by a sequence of one or more 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.

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 4.5.

- *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 `BitLiteral` terminal in the 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. 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, and Predicates

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, a compiler can report an 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, type signatures, predicates, and type functions. 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 five forms:

- The kind `*`, which can also be written as `type`, 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` for type variables that represent 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 `* -> (* -> *)`.

- The kind `lab` represents field labels that can be used to identify components in bitdata and structure types. In this report, we will typically use names beginning with the letter `f` (a mnemonic for *field label*) for type variables of kind `lab`. For each identifier, `x` (either a `Varid` or a `Conid`), there is a corresponding type `#.x` of kind `lab` as well as a unique value, also written `#.x`, of type `Lab #.x`; the `Lab` type constructor is a primitive with kind `lab -> *`. These mechanisms, described in more detail in Section 4.3, are intended to support a flexible approach to dot notation in the implementation of core Habit features and high-level library code and are not expected to be used heavily in application code.

These kinds are the same as those of Haskell but with the addition of `nat`, `area`, and `lab`, and the introduction of `type` as a synonym for `*`. In the following 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[1].

```
Kind    = AKind -> Kind          -- function kinds
        | AKind                  -- atomic kinds
AKind   = *                      -- nullary type constructors
        | type                   -- a synonym for *
        | nat                    -- type-level naturals
        | area                   -- memory areas
```

[1]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.

```
        | lab                   -- field labels
        | "(" Kind ")"          -- parentheses
```

Note that `type` and `area` are Habit keywords, but `*`, `nat`, and `lab` are not reserved symbols and hence they can be used outside kind expressions as regular variable/operator 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 would instead be treated as a four character `Varsym` symbol, `*->*`. In this particular example, the same kind can also be written as `type->type` without having to worry about inserting extra spaces.

3.3.2 Types

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

```
Type        = AppType List0(Tyop AppType) -- infix syntax
Tyop        = Tyconop                     -- type constructor operator
            | Tyvarop                     -- type variable operator
AppType     = List(AType)                 -- applicative syntax
AType       = Tyvar                       -- type variable
            | Tycon                       -- type constructor
            | "(" ")"                     -- the unit type
            | IntLiteral                  -- type-level numeric literal
            | #. Id                       -- type-level label literal
            | AType . Id                  -- selection
            | "(" Tyop ")"                -- type operator
            | "(" Type Opt(:: Kind) ")"   -- parentheses
```

The atomic types, represented in the grammar by `AType`, are type variables, constants (including named type constructors, the unit type, type-level numbers of kind `nat`, and labels of the form `#.x`), selections (described further in Section 4.3), and parenthesized type expressions.

Type applications are written 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.

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, `->`, which has kind `* -> * -> *`. 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. As written, the grammar for `Type` does not specify 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)`, but these ambiguities can be eliminated in practice using declared fixity information (see Section 3.6.2). For example, the function type constructor, `->`, has fixity `infixr 5`, and hence a type expression of the form `a -> b -> c` is parsed as `a -> (b -> c)`. 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, 3.6.8, and 3.6.9.

Type	Interpretation
`t -> t'`	functions from values of type `t` to values of type `t'`
`Bool`	Booleans, `False` and `True`
`WordSize`	the number of bits in a machine word as a type-level number
`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, introduced into programs solely via numeric literals; the only element of `Nat n` is the natural `n`
`Ix n`	index values (integers) in the range `0` to `(n-1)`
`Bit n`	bit vectors of length `n`
`ARef l a`	references to memory areas of type `a` with alignment `l`
`Ref a`	references with default alignment
`APtr l a`	pointers to memory areas of type `a` with alignment `l`
`Ptr a`	pointers with default alignment
`Array n a`	memory areas containing an array of `n` elements of type `a`
`Pad n a`	an inaccessible area of padding, taking the same space as an `Array n a`.
`Init a`	initializers for memory areas of type `a`
`Lab f`	field label types; for each `#.x` of kind `lab`, there is a corresponding field label value, also written `#.x`, of type `Lab #.x`

Figure 1: Standard Habit Types

3.3.3 Type Signatures

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 recommended) 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[2], 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

[2]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.

that they restrict, or qualify the use of polymorphism [7]. Details of the syntax for predicates, as well as a survey of the predicates that are defined in the standard Habit environment are provided in the next Section.

3.3.4 Predicates

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

```
Pred    = AppPred Opt("=" Type) Opt(fails)
        | SelPred     "=" Type  Opt(fails)
        | "(" Pred ")"
AppPred = Type Tyconop Type  -- predicate with applicative syntax
        | PrePred
PrePred = Tycon              -- predicate with prefix syntax
        | PrePred AType
        | "(" AppPred ")"
SelPred = AType . Id         -- selection
        | "(" SelPred ")"
```

For practical purposes, however, predicates are likely to be parsed as expressions of the form `Type Opt("=" Type) Opt(fails)`, with subsequent checks to ensure that the result can be interpreted as an application `C t1 ... tn`, where `C` is a `Tycon` that names an n-parameter *type class* and `t1`, ..., `tn` are appropriately kinded types. (The optional `"=" Type` portion of a predicate is used for type functions when the class `C` has an appropriate functional dependency while an optional `fails` suffix indicates the negation of a predicate; these details are discussed more fully in Sections 3.3.5 and 3.6.5, respectively.)

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 `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 names in the Haskell standard prelude.) Habit also supports the introduction of user-defined classes (see Sections 3.6.4 and 3.6.5).

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)`
`Update r f`	values of type `r` have a field labeled `f` (of type `Select r f`) that can be updated
`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
`MemMonad m`	`m` is a monad that supports memory operations
`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
`Initable a`	areas of type `a` have a default initializer
`NoInit a`	areas of type `a` do not require initialization
`NullInit a`	areas of type `a` can be null-initialized

Figure 2: Standard Habit Type Classes

3.3.5 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`. For a three parameter type function like `+` whose name is an operator symbol, we can also use the name as an infix operator on the left of the `=` symbol. For example, the predicate `(+) x y z` can also be written as `(+) x y = z` or as `x + y = z`.

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 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 is written with an explicit = symbol to emphasize the use of a type function. 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).

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
`n ^ m = p`	`p` is the result of raising `n` to the power `m`
`GCD n m = p`	`p` is the greatest common divisor of `n` and `m`
`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
`Select r f = t`	`r` has a component with field label `f` 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

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 the following type (simplified for this example; see Section 4.8 for the full 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 can also be written in either of the following equivalent forms that expose the underlying type class:

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

3.4 Expressions

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

```
Expr   = EInfix Opt(:: Type)         -- typed expression
EInfix = EApp List0(Op EApp)         -- infix operators
EApp   = LetExpr                     -- local definition
       | IfExpr                      -- conditional expression
       | CaseExpr                    -- case expression
       | Applic                      -- applicative expression
```

Habit broadly follows the syntax of Haskell, including function applications (using prefix or infix syntax, but without Haskell's special case for unary negation). The grammar shown here is ambiguous regarding the extent of local definitions, conditionals, and case statements (and also lambda expressions and `do` notation, as described in the next section); as in Haskell, these ambiguities are resolved by the rule that each of these constructs extends as far to the right as possible.

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   = SInfix Opt(:: Type)         -- typed statement
SInfix = SApp List0(Op SApp)         -- infix operators
SApp   = LetStmt                     -- local definition
       | IfStmt                      -- conditional statement
       | CaseStmt                    -- case statement
       | Applic                      -- applicative expression
```

3.4.1 Applicative Expressions

The core of the expression syntax for Habit is shown in the following grammar:

```
Applic  = \ List(APat) -> Expr              -- anonymous function
        | do Block                          -- monadic expression
        | List(AExpr)                       -- prefix application
AExpr   = Var                               -- variable name
        | Con                               -- constructor
        | Literal                           -- literal/constant
        | AExpr . Id                        -- selection
        | AExpr [ Fields ]                  -- update expression
```

```
         | "(" ")"                          -- unit
         | "(" Expr ")"                     -- parenthesized expression
         | "(" AExpr Op ")"                 -- left section
         | "(" Op AExpr ")"                 -- right section
         | "(" Sep2(Expr, ",") ")"          -- tuple
         | Conid  [ Sep (Id <- Expr, "|") ] -- structure initializer
Fields   = Sep(Id Opt("=" Expr), "|")       -- field bindings
Literal = IntLiteral                        -- natural number
         | BitLiteral                       -- bit vector literal
         | #. Id                            -- field label literals
```

Again, Habit follows the syntax of Haskell for lambda terms, `do` notation (syntactic sugar for monadic terms), tuples, sections, and atomic expressions. Note, however, that Habit does not provide special syntax for Haskell-style list comprehensions, arithmetic list ranges, or list enumerations.

Dot notation (Section 4.3) is used to indicate selection in Habit expressions; intuitively, an expression of the form `r.x` returns the value of the `x` component of the object `r`. In particular, dot notation can be used to access all of the fields of bitdata (Section 3.6.8) and structure types (Section 3.6.9).

Update expressions of the form `r[x=e]` provide a syntax for constructing a value that is almost the same as the target, `r`, but with the `x` field replaced by the value of `e`. Multiple updates can be combined in a single field list, with an expression like `r[x=e1|y=e2]` describing a simultaneous pair of updates. In particular, update expressions are supported for all bitdata types (Section 3.6.8). Of course, expressions like these are only valid if the value described by `r` includes the referenced fields. The mechanisms that are used to ensure correct usage of field names in Habit are described in Section 4.3. As a special case, if the target in an update expression is a bitdata constructor name, then the update is interpreted instead as bitdata construction. For example, `BC[x=0|y=1]` constructs a bitdata value, assuming that `BC` has been declared elsewhere in the program as a bitdata constructor with appropriately typed fields `x` and `y`.

Finally, Habit provides special syntax for describing structure initializers using expressions like `S[x<-0]`; this is described in more detail in Section 3.6.9.

3.4.2 Blocks

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    = Opt(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
```

3.4.3 Local Declarations

A local declaration provides bindings (a semicolon-separated list of declarations) for a set of variables 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.

3.4.4 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 `Bool`-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 `Pat`, and atomic patterns, represented by `APat`, is as follows:

```
Pat       = AppPat List0(Op AppPat)    -- infix syntax
AppPat    = List(APat)                 -- application pattern
APat      = Var                        -- variable
          | _                          -- wildcard
          | Var @ APat                 -- as-pattern
          | Con                        -- constructor
          | Con [ PatFields ]          -- bitdata pattern
          | "(" Sep2(Pat, ",") ")"     -- tuple pattern
          | "(" Pat :: Type ")"        -- typed pattern
          | "(" Pat ")"                -- parentheses
          | Literal                    -- literal pattern
PatFields = Sep0(Id Opt("=" Pat), "|") -- field patterns
```

Different patterns, of course, match against different values, and most of the pattern forms in Habit have the same basic interpretation as they do in Haskell:

- A variable, `x`, matches any value, binding the variable to that value within the scope of the pattern. Note that all patterns in Habit are required to be *linear*, meaning that no variable name can appear more than once in a given pattern.

- 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 wildcard pattern, `_`, matches any value, without binding any variables.

- 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`. In general, fixity information is required to identify valid constructor expressions. For example, until we have determined whether an input like `` (x `C` y `D` z) t `` should be treated as `(C x (D y z) t)` or as `(D (C x y) z t)`, we cannot check that each of `C` and `D` has the correct number of arguments.

- A tuple pattern `(p1, ..., pn)` matches against tuple values of the form `(v1, ..., vn)` so long as each component value `v1, ..., vn` matches the corresponding pattern `p1, ..., pn`. (Of course, the type system ensures that there are exactly the same number of pattern and value components in any such match.)

- A pattern `BC [ f1 | ... | fn ]`, where `BC` is a constructor function of a bitdata type (see Section 3.6.8), will match any value that can be produced by the `BC` constructor, so long as each of the individual field specifications, `f1`, ..., `fn`, are also matched. Each field specification is either a field name paired with a pattern, `x = p`, or just a single field name, `x`. In the first case, the match only succeeds if the value of the `x` field matches the pattern `p`. The second case is treated as an abbreviation for `x = x`. For example, if `x` is a `Varid`, then 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* because the same identifer string is used as both a field label and a variable name.) In both cases, the name `x` must be a valid field name for the `BC` 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, or to list all of the fields for `BC`; any fields that are not mentioned 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)` (for any type that is an instance of the `ToBits` class via an implicit application of `toBits`; see Section 4.9) so long as each of the bit vectors `v1, ..., vn` matches the corresponding pattern `p1, ..., pn`. The `:#` operator used here is a constructor function, mentioned previously at the end of Section 3.3.5, 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.

- A literal pattern matches only the specified value, without introducing any variable bindings.

There are some additional syntactic restrictions on patterns that are hard to capture in the grammar for `Pat` because of the interaction between parsing and fixity resolution. In particular, we do not allow as-, tuple, bitdata, wildcard, or literal patterns as the first item of an application pattern `p1 ... pn` with two or more `APat` components. Uses of variables names at the beginning of patterns like this (or equivalent forms using infix operators) are also restricted. In the following text, we distinguish between two general pattern forms:

- A *data pattern* does not contain any subterms of the form `f x` or ``x `g` y`` (equivalent to `g x y`) in which a variable symbol (either a `Var` or, as part of an infix expression, a `Varop`) is applied to one or more arguments.
- A *function pattern* is a pattern in which a variable symbol is applied to one or more data patterns. Infix notation is permitted. For example, ``x `g` y`` is a function pattern in which `g` is applied to two data patterns `x` and `y`.

Almost all uses of patterns in Habit require data patterns. The only exception is on the left hand side of an equation (Section 3.6.1) where a function pattern can be used as part of a function definition. Terms matching the `Pat` grammar that do not meet the requirements for either data or function patterns are not valid in Habit.

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
         | StructDecl              -- structure type declaration
         | AreaDecl                -- area declaration
```

Some forms of declaration (specifically, `Equation`, `FixityDecl`, and `TypeSigDecl`, which make up the alternatives for `Decl`) 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 pattern 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(Tyop, ",")  -- 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 signatures are typically used as a form of documentation, but they can also be used 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 too 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, ",")) -- constraints
              Opt(where DeclBlock)          -- operations
ClassLhs   = TypeLhs Opt("=" TypeParam)     -- class header
           | "(" ClassLhs ")"
TypeLhs    = TypeParam Tyconop TypeParam    -- type header
           | PreTypeLhs
PreTypeLhs = Tycon                          -- prefix type header
           | PreTypeLhs TypeParam
           | "(" TypeLhs ")"
TypeParam  = Var                            -- type parameter
           | "(" TypeParam Opt(:: Kind) ")"
Constraint = Pred                           -- superclass
           | FunDep                         -- functional dependency
FunDep     = List0(Var) -> List(Var)
```

Although we give an explicit grammar, the `ClassLhs` portion of a class declaration is likely to be parsed as a predicate, with subsequent checks to ensure

that the result can be interpreted as a an application `C t1 ... tn`, where `C` is a `Tycon` that names a class (not defined elsewhere) and `t1`, ..., `tn` are (zero or more) distinct type parameters. Each parameter has an associated kind that can 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).

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 3.3.5.)

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 (Section 3.6.5).

Beyond specifying the names and types of each parameter, there are two ways in which a `class` declaration can be annotated to constrain the set of instances:

- By specifying *superclasses*, each of which is represented by a predicate in the list of constraints. If the declaration of a class `C` begins as follows:

  ```
  class C a1 ... an | ..., P, ...
    where ...
  ```

  where `P` is some predicate, 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 predicate `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. Note also that, if the `ClassLhs` portion of a class declaration is written in the form `C t1 ... = tn` and if there is no explict functional dependency to indicate that `tn` is uniquely determined by the first $n - 1$ parameters, then a dependency `t1 ... -> tn` is automatically added to class `C`. This allows functional notation to be indicated by writing an `=` symbol without having to write an explicit dependency.

The (optional) `DeclBlock` in a `class` declaration specifies 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 top-level values/functions 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 in 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 default definitions, they will be used if there is no explicit definition for the corresponding operations in user-provided instance declarations.

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 effect, there are three basic forms of instance declaration, and these can be combined into a single instance declaration, separated from one another by the `else` keyword. 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 C t1 ... tn if P
```

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 C t1 ... tn if P
  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. Although it is not reflected in the gram-

mar, it is not permitted (and would not make sense) to include a `DeclBlock` of definitions in this case.

```
instance C t1 ... tn fails if P
```

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 practical way to define general equality on functions, and it is useful to state this explicitly so that erroneous code that assumes such an instance will be detected more promptly.

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 `isBool`. 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 other predicates of the form IsBool t
  where isBool x = False
```

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

```
instance C t1 if P1 ...
else     ...
else     C tn if Pn ...
```

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 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 C a if Eq 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)
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.

```
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
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 3.3.5 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
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 (Pair Unsigned s) if Stream s
else     Stream s fails
```

which does not define any valid instances.

3.6.7 Datatype 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    = Type Conop Type                 -- infix syntax
           | PreDataCon
PreDataCon = Con                             -- prefix syntax
           | PreDataCon AType
           | "(" DataCon ")"
DeriveList = Con                             -- deriving a single class
           | "(" Sep(Con, ",") ")"           -- or a list of classes
```

The optional `deriving` clause 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 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(BitdataRegion, "|") ]   -- bitdata constructor
BitdataRegion = Sep(BitdataField, ",") :: TApp    -- labeled field
              | Expr                              -- tag bits
BitdataField  = Varid Opt("=" Expr)               -- field with default
```

Bitdata definitions are similar to data definitions (Section 3.6.7) because they introduce a new type name (the Conid that appears after the bitdata keyword) as well as a collection of constructor names (the Con symbols at the front of each BitdataCon). However, there are also some important differences:

- Parameters are not permitted on the left hand side of a bitdata definition.

- The description of each constructor specifies a collection of zero or more component fields (each of which has an associated name, type, and an optional default value) as well as a concrete, bit-level layout for constructed

values. If extra tag bits are needed to distinguish between different constructors, then these must be written explicitly as (unlabeled) expressions of type `Bit N` for some (uniquely determined) width `N`. The representation for values corresponding to a particular constructor is determined by concatenating the values of the component fields and tag bits in the order they appear in the definition from left to right (i.e., from most to least significant bit). For example, the following definition indicates that a `PCI` address is described by a sixteen bit value whose most significant eight bits identify a particular hardware `bus`. The remaining bits specify a five bit `dev`ice id and, in the three least significant bits, a device `fun`ction.

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

- No constructor layout can include a component, either directly or indirectly, of the type that is being defined. For example, the following definitions break this restriction and hence are not valid.

```
bitdata U = X [ x :: U ]       -- invalid
bitdata V = Y [ y :: W | B1 ]  -- invalid
bitdata W = Z [ z :: V ]
```

- Each constructor has an associated width (which must be an instance of the `Width` class) that is calculated as the sum of the widths of its components, including data fields and tag bits. All of the constructors in a given bitdata definition must have the same width, and this will be used as the `BitSize` of the type that is being defined. If extra padding is required in one or more constructors to satisfy this property, then it must be written explicitly as part of the definition. In all cases, the `BitSize` of the each bitdata type must be uniquely determined.

- The `BitSize` 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. (As described in Section 3.6.9, the same notation can be used to specify the width of a structure type; in that case, however, the width specifies a number of bytes rather than a number of bits.) If the declared width does not match the width that can be inferred from the rest of the definition, then an error will be reported. In particular, no attempt will be made to pad or truncate bit-level representations automatically to match the declared width.

- An expression that is used to specify tag bits does not require an explicit type annotation if the associated width can be inferred from the context in which it appears. In the following examples, it is clear that the `0` value in the definitions of `R` and `S` must be treated as a constant of type `Bit 4` given the width annotation in the definition of `R` and the requirement that all constructors have the same width in the definition of `S`. On the other

hand, the definition of `T` is not valid because the `0` and `1` literals can be interpreted as having many different widths, and hence the width of `T` is not uniquely determined.

```
bitdata R/8 = A [ x :: Bit 4 | 0 ]    -- valid, BitSize R = 8
bitdata S   = B [ 0 ] | C [ B1111 ]   -- valid, BitSize S = 4
bitdata T   = D [ 0 ] | E [ 1 ]       -- invalid, BitSize T = ?
```

- The assumptions of no junk and no confusion for algebraic datatypes are not guaranteed for bitdata types [4]. 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 a compiler will typically provide an 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 and alternatives in a definition that uses pattern matching.

Bitdata Construction. The basic notation for constructing a bitdata value, described previously in Section 3.4.1, reflects the syntax of a corresponding `BitdataCon` except that the fields may be listed in any order and that fields with a specified default do not need to be mentioned at all. Tag bits are not required (or permitted) because they are already implied by the choice of a particular constructor name. The following code, for example, defines a bitdata type, `Perms`, that is three bits wide and represents a set of Unix-style read (most significant bit), write, and execute (least significant bit) permissions.

```
bitdata Perms/3 = Perms [ r=B0, w=B0, x=B0 :: Bit 1 ]

nilPerms :: Perms
nilPerms  = Perms [ r=B0 | w=B0 | x=B0 ]
```

The definition of `nilPerms` specifies a particular value of the `Perms` type in which all three bits are set to zero. However, because the definition of `Perms` already specifies `B0` as the default value for each of the three components, any of the following alternative definitions for `nilPerms` would have the same effect:

```
nilPerms  = Perms [ x=B0 | r=B0 | w=B0 ]  -- reorders fields
nilPerms  = Perms [ w=B0 | x=B0 ]         -- omits the r field
```

```
nilPerms  = Perms [ r=B0 ]                -- omits the w & x fields
nilPerms  = Perms [ ]                     -- omits all fields
nilPerms  = Perms                         -- omits field list
```

Note that the last of these examples omits the field list altogether. This is permitted only for bitdata constructors whose layout specifies a default value for every field (which, as a special case, includes constructors with no data fields, and whose layout contains only tag bits).

Bitdata Constructor Types. For each constructor `C` of a bitdata type `T`, there is an associated type written `T.C` whose values are the subset of bit patterns in `T` that can be produced using the constructor `C`. Values of type `T.C` can be obtained by pattern matching against values of type `T` and can be converted back into values of type `T` by applying the constructor function `C`, which is treated as a function of type `T.C -> C`.

The compiler generates instances of `Select` (see Section 4.3) to define each of these component types and to provide operations for selecting the values of the fields associated with each constructor. The compiler will also generates instances of `Update` for each bitdata field. To illustrate how this works in practice, consider the following `bitdata` definition.

```
bitdata T = X [ B1 | x :: Bit 3 | y :: Bit 4 ]
          | Y [ B0 | x :: Bit 7 ]
```

By considering the lists of constructors, and the list of field names in each structure, the compiler can generate the following collection of primitive instances:

```
instance T.X = _             -- Component types of T
else     T.Y = _
else     Select T f = t fails

instance T.X.x = Bit 3       -- Selectors for fields of T.X
else     T.X.y = Bit 4
else     Select (T.X) f = t fails

instance Update T.X #.x      -- Update functions for fields of T.X
else     Update T.X #.y
else     Update T.X t fails

instance T.Y.x = Bit 7       -- Selectors for fields of T.Y
else     Select (T.Y) f = t fails

instance Update T.Y #.x      -- Update functions for fields of T.Y
else     Update T.Y t fails
```

These definitions are sufficient to enable the use of intuitive dot and update notation for working with values of the `T` type. Indeed, many programmers will be able to work with bitdata types like this, as in the following examples, without needing to understand all the details of the associated instances.

```
sumT      :: T -> Bit 7   -- use dot notation to select field values
sumT (X r) = (0 :# r.x) + (0 :# r.y)
sumT (Y r) = r.x

incT      :: T -> T       -- use update to increment or set fields
incT (X r) = X r[x = r.x+1]
incT (Y r) = Y r[x = 0]
```

Single Constructor Bitdata Types. In the special case of a `bitdata` type, `T`, with only a single constructor, `C`, the compiler will replicate the instances that it produces for `T.C` so that the dot and update notations can be used directly with values of type `T`, avoiding the need for an initial match against the `C` constructor. For the `PCI` type defined previously, for example, the compiler will generate the following instances:

```
instance PCI.PCI = _          -- PCI has only one constructor
else     PCI.bus = Bit 8      -- Selectors for fields of PCI
else     PCI.dev = Bit 5
else     PCI.dev = Bit 3
else     Select PCI f = t fails

instance Update PCI #.bus     -- Update functions for fields of PCI
else     Update PCI #.dev
else     Update PCI #.fun
else     Update PCI f fails

instance PCI.PCI.bus = Bit 8  -- Selectors for fields of PCI.PCI
else     PCI.PCI.dev = Bit 5
else     PCI.PCI.dev = Bit 3
else     Select (PCI.PCI) f = t fails

instance Update PCI.PCI #.bus -- Update functions for fields of PCI.PCI
else     Update PCI.PCI #.dev
else     Update PCI.PCI #.fun
else     Update PCI.PCI f fails
```

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`. Definitions for component types (in this case, `PCI.PCI` in the third and fourth instance chains above) are retained for consistency, although we expect that they are unlikely to be used very much in practice. The

following definitions show some simple examples that use selection and updates involving values of the single constructor `PCI` type:

```
onZeroBus     :: PCI -> Bool
onZeroBus addr = addr.bus == 0

incFun        :: PCI -> PCI
incFun addr    = addr[fun = addr.fun + 1]
```

Bitdata Deriving. 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. The rules for deriving instances of `Eq`, `Ord`, `Bounded`, `Num`, `BitManip`, `Boolean`, `Shift`, and `Pointed`, and are exactly the same as those for datatypes, as described in Section 3.6.7, and, for `bitdata` types, they are also extended to requests for derived instances of `ToBits`, and `FromBits`. Note however, that it is not permitted (or possible) to derive an instance of `Monad` for a `bitdata` type.

3.6.9 Structure Declarations

Structure declarations are used to name and describe the layout of memory areas that are constructed by combining a sequence of and individually labeled, adjacent memory blocks into a single memory area. The syntax for structure declarations is described by the following grammar:

```
StructDecl   = struct Conid                  -- name (no parameters)
               Opt(/ Type)                   -- size specification
               [ Sep0(StructRegion, "|") ]   -- structure regions
               Opt(deriving DeriveList)      -- deriving clause
StructRegion = Opt(Sep(StructField, ",") ::) Type  -- list of fields
StructField  = Id Opt(<- Expr)               -- name & initializer
```

Each structure declaration introduces a new type-constant name of kind `area`, which cannot be the same as the name of any other type constant (including type classes, type functions, bitdata, or data names) because they share the same namespace. Note that a structure declaration serves only to define a type; separate `area` declarations (Section 3.6.10) must be used to reserve one or more memory areas with the associated layout.

Structure Layout. Each structure type is organized as a collection of region specifications, each of which has (at least) an associated type (which must have kind `area` as well as an associated `ByteSize` instance). The resulting memory

layout will include one component of the specified type for each field that is declared within the region. If a region is specified by a type without any fields, then a single (inaccessible) block of memory corresponding to that type will be allocated within the structure. A Habit compiler guarantees that the regions and fields of a structure will be arranged in memory using the same left-to-right/lower-to-higher address order in which they are listed in the declaration, without inserting any padding. For example, the following structure describes an area of memory that takes the same space as an `Array 4 (Stored Unsigned)`, with the `x` field at index 0, the `y` field at index 1, and the `z` field at index 3:

```
struct S [ x, y :: Stored Unsigned | Stored Unsigned
                                   | z :: Stored Unsigned ]
```

Unlike an array, however, there is no way to access the index 2 component of an `S` structure because it does not have any named fields.

Structure Size. For every declared structure type, the compiler will automatically generate a corresonding `ByteSize` instance (see Section 4.14), summing the sizes of its constituent regions to compute the structure size. Given the `S` structure defined above, for example, with four `Stored Unsigned` components, each of which takes `(WordSize/8)` bytes, the compiler will generate an instance equivalent to the following:

```
instance ByteSize S = 4 * (WordSize/8)
```

A structure declaration can include an optional `/n` annotation, immediately after the structure name, for some type expression `n` of kind `nat`, to document the expected size of the structure. This behaves much like the width annotation for bitdata declarations except that the size of a structure is measured in bytes, while the width of a bitdata type is measured in bits. As in the case of bitdata, an error will be reported if the inferred size of the structure does not match the declared size `n`; no attempt will be made to pad or truncate a structure type automatically to reach the specified size. For example, the previous definition of structure `S` could be modified to include a size annotation as follows:

```
struct S/16 [ x, y :: Stored Unsigned | Stored Unsigned
                                      | z :: Stored Unsigned ]
```

This definition will be accepted on a platform with `WordSize = 32` (where four words, each of which takes four bytes, can be stored in sixteen bytes), but it will trigger an error if compiled for a machine with a different `WordSize`. Unportable behavior like this may be considered undesirable in some circumstances, but such details can be quite important in some systems code, and this mechanism

allows a programmer to document platform-specific assumptions and enables violations of those assumptions to be detected at compile-time.

Recursion. The layout of a structure type `S` cannot include a region, either directly or indirectly, of the same type that is being defined. This prohibits recursive definitions like the following in which the layout of the structure being defined is not completely specified (as in the examples `Bad1`, `Bad2`, and `Bad3`), or else for which there is no valid layout (as in the example `Bad4`, requiring `ByteSize Bad4 = ByteSize Bad4 + (WordSize/8)`, which is clearly not possible).

```
struct Bad1 [ x :: Bad1 ]                          -- direct recursion
struct Bad2 [ x :: Bad3 ]                          -- mutual recursion
struct Bad3 [ x :: Bad2 ]
struct Bad4 [ x :: Bad4 | y :: Stored Unsigned ] -- impossible layout
```

Of course, the `ByteSize` of a stored reference does not depend on the type of the region that it points to, so recursion through reference types is permitted:

```
struct Okay [ x :: Stored (Ref Okay) ]  -- Stored Ref has fixed size
```

Field Access. No field name may be used more than once within a single structure, but the same field name may appear in multiple distinct structure types (or in bitdata types) without ambiguity. For each declared structure, the compiler generates a corresponding instance of `Select` (Section 4.3) to provide functions for accessing the structure's named components. For the example structure given above, this instance declaration takes the following form:

```
instance (Ref S).x = Ref (Stored Unsigned) where ...
else     (Ref S).y = Ref (Stored Unsigned) where ...
else     (Ref S).z = Ref (Stored Unsigned) where ...
else     Select (Ref S) f = t fails
```

Note that these definitions work with references rather than direct values. For example, if `r` is a reference to a structure of type `S`, then `r.y` will produce a reference to a `Stored Unsigned`, and not the `Unsigned` value that might be held at that address. In concrete terms, this means that selection of a component within a structure is a pure operation, performing address arithmetic but no memory access, which must instead be captured by a separate action (e.g., `readRef r.y`). Note also that the final line in the preceding instance declaration rules out the possibility of accessing any field other than the `x`, `y`, or `z` fields of an `S` structure mentioned in the preceding clauses. This ensures that a compiler can report an error if a programmer writes an expression of the form `r.t`, assuming that `r` is an expression of type `Ref S`, because there is no `t` field in an `S` structure.

Field Update. Individual fields in a structure that is identified by a reference `r` can be updated using statements like `writeRef r.y e`[3]. Update expressions, such as `r[y=e]`, do not make sense for structures (it is not possible to change the address of a field within a structure), so the Habit compiler will also generate an instance of the following form for each declared structure type (see Section 4.3 for details of the `Update` class):

```
instance Update (Ref S) f fails
```

Structure Initialization. As described in Section 4.15, proper initialization of structures and other memory areas in Habit programs is important to guarantee type correctness and well-defined behavior. Habit provides special syntax as part of the grammar for `AExpr` (see Section 3.4.1) for writing structure initializers in the form `Conid [ Sep(Id <- Expr, "|") ]`. Each expression like this identifies a particular structure type and specifies an initializer for each field in that structure. The fields may be listed in any order, but repetition is not permitted, and only field names that are defined as part of the named structure are allowed. It is not necessary (although it is permitted) to include an entry for a field when an initializer has been specified as part of the structure definition (see the `SField` nonterminal in the earlier grammar) or if a default initializer is available as an instance of `Initable`. An initializer specified in a `StructInit` takes priority over an initializer specified in a `StructDecl`, which, in turn, takes priority over a default initializer specified in an instance of `Initable`. It is an error if this process fails to specify an initializer for any field in the structure.

The following examples show four initializers (i.e., values of type `Init S`) for the `S` structure that was declared previously:

```
init1 = S[ x <- initStored 0 | y <- initStored 1 | z <- initStored 2 ]
init2 = S[ z <- initStored 2 | x <- initStored 0 | y <- initStored 1 ]
init3 = S[ x <- 0 | y <- 1 | z <- 2 ]
init4 = S[ y <- 1 | z <- 2 ]
```

Although they look different, each of these initializers has the same overall effect. The mapping between field names and initializers in examples `init1` and `init2` is the same, even though the fields are listed in a different order. The `init3` example achieves the same result, relying on the overloading of numeric literals as initializers. The `init4` example takes this a step further by omitting an explicit initializer for the `x` field and relying instead on the `initNull` initializer that will be used as the default in this case.

[3]Note that `writeRef r.y e` is parsed as `writeRef (r.y) e`, following the grammar in Section 3.4, and not as `(writeRef r).(y e)` as it would be in Haskell. This is because Habit treats the period, (`.`), as a reserved symbol for describing selection and not as an infix composition operator or as special syntax for working with modules.

Derived Initializers. If all of the field types in a given structure are instances of `NullInit`, meaning that they can all be null-initialized, then the structure itself can be treated as an `NullInit` instance by including that class name in the `deriving` portion of a structure declaration. Conversely, if `NullInit` is not mentioned in the `deriving` clause, then the compiler will generate a `fails` instance of the class instead; this mechanism prevents the introduction of user-defined null-initializers for structures, which could otherwise compromise type safety. Instances of the `NoInit` class for structure types, corresponding to regions of memory that do not require initialization, are handled in the same way by including (or omitting) the name `NoInit` from the `deriving` list. Note that any initializers specified as part of a structure declaration are ignored when generating a derived instances of either `NullInit` or `NoInit`.

As an example, because we did not include a `deriving` clause in the original definition of the structure `S`, the compiler will generate two instances:

```
instance NullInit S fails -- S cannot be null initialized
instance NoInit S fails   -- ... or left uninitialized
```

If we modified the definition of `S` by appending `deriving NullInit`, however, then the compiler would generate a different pair of instances:

```
instance NullInit S       -- S can be null-initialized
instance NoInit S fails   -- ... but not left uninitialized
```

Note that the compiler will report an error if the classes listed in a `deriving` clause cannot be applied to all of the structure fields. For example, it would be an error to add `deriving (NullInit, NoInit)` to the definition of the `Okay` structure given previously because stored references must be properly initialized with a valid pointer, and cannot be either null-initialized or left uninitialized.

If the deriving clause in a structure declaration includes `Initable`, then the compiler will attempt to generate a default initializer for the structure as an instance of the `Initable` class using the initializers that are specified for each field. Given the following declaration, for example, the compiler will generate a default initializer for `Point` structures that sets both the `x` and `y` components of the structure to `0`:

```
struct Point [ x <- 0, y <- 0 :: Stored Unsigned ] deriving Initable
```

It is also possible to use `deriving Initable` if the structure declaration does not include explicit initializers for all of the fields, provided that default initializers (i.e., other instances of `Initable`) are available in those cases. As a result, the preceding definition of `Point` would still be accepted if we omitted the two `<- 0` clauses because there is a default (null-)initializer for `Stored Unsigned` values.

An explicit Initable instance can be used in cases where either null- or no-initialization is required as the default for a particular structure type. For example, we could make null-initialization the default for the S structure type defined previously by adding deriving NullInit to the original declaration and then adding the following Initable instance:

```
instance Initable S where initialize = nullInit
```

Providing an explicit Initable instance is also appropriate, of course, if some other default initialization semantics is required.

3.6.10 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 a variant of the notation for type signature declarations except that it begins with the area keyword, and includes extra syntax for specifying initializers:

```
AreaDecl = area Sep(AreaVar, ",")  -- area names and initializers
             :: Type               -- area type
             Opt(where decls)      -- local definitions
AreaVar  = Var Opt(<- EInfix)      -- area name and initializer
```

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.

The syntax for area declarations also allows the specification of an initializer expression, introduced by an <- symbol, for each named area. The initializer expression can be omitted if a default is available (i.e., if the area type, A, is an instance of Initable). For example, the following declaration:

```
area r1 <- init1, r2 <- init2, r3 :: Ref A
```

can also be written as three separate area declarations:

```
area r1 <- init1 :: Ref A
area r2 <- init2 :: Ref A
area r3          :: Ref A
```

Each of these options introduces three constants named `r1`, `r2`, and `r3` with type `Ref A` at the top-level of the program. Of course, these declarations are only acceptable if `init1` and `init2` are valid initializers for `A` (i.e., if these expressions have type `Init A`), and if there is a default initializer for areas of type `A` (i.e., an instance of `Initable A`) to ensure that all three areas have a well-defined initialization semantics.

The optional `where` clause at the end of an area declaration provides an opportunity for including local definitions that scope across all of the area initializers in the declaration. This can be used for defining complex initializer expressions without introducing new symbols at the top-level, as in the following example, which shows how an array can be initialized so that each stored `Unsigned` contains the square of its index:

```
area squares <- initArray square :: Ref (Array 10 (Stored Unsigned))
 where square i = let n = unsigned i in initStored (n * n)
```

As a final note, we mention that future versions of Habit are expected to extend the syntax for area declarations with mechanisms for specifying memory area placement. This may be useful, for example, to specify locations within certain portions of the address space, or even at specific addresses.

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 or type information:

```
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 here 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, 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 used in the standard environment to describe, among other things, bit vector widths, alignments and sizes of memory areas, and limits on valid array indices. As mentioned in Section 3.3.5, 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 *, /
```

```
infixl type 8 ^
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 as well as an infinite set of basic arithmetic facts, like the following, which are effectively generated on demand during type checking:

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

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

instance x - y = z  if z + y = x
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 < y  if x + 1 <= 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     m / n = p  if p * n = m, 0 < n
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 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 (^) (m :: nat) (n :: nat) (p :: nat)
  | m n -> p, m p -> n
```

[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

```
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 2^0 = 1
instance 2^1 = 2
instance 2^n = 3 fails
...
instance 2^12 = 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 Dot Notation: the Select and Update Classes

The syntax of Habit allows components of bitdata and structures to be accessed using dot notation: the `x` component of a value `e` is accessed by writing `e.x`. In this section, we describe the features of the standard environment that support use of dot notation in both types and expressions as well as the `e[x=e']` update syntax in expressions. These details are likely to be of most interest to programmers who are building high-level library code or investigating internals of Habit; they are not expected to be used heavily in application code.

Label Types and Values. As mentioned in Section 3.3.1, Habit includes a kind, `lab`, whose elements represent field labels. More specifically, for each identifier, which could be either a `Varid` like `x` or a `Conid` like `X`, there are corresponding types, written `#.x` and `#.X`, of kind `lab`. Using a kind, `lab`, allows us to separate types representing labels from types of kind `*` like `Bool` or `Unsigned` that do not. To make a connection between label types and values, however, the Habit type system includes the following type constructor as a primitive:

```
primitive type Lab :: lab -> *
```

In particular, we will interpret each of the resulting `Lab l` types as a singleton, and write `#.x` for the unique label value in the the type `Lab #.x`.

Selection. The primary role of a field label is to identify a component within a (typically) larger data structure. We will use expressions `select r l` to denote the value of the component of `r` that is associated with the label value `l`. This only makes sense for certain combinations of `r` and `l`, which suggests that the type of `select` will involve some form of overloading. It is also reasonable to expect that different combinations of inputs will result in different types of output. These observations are reflected in the following class definition:

```
class Select (r :: *) (f :: lab) = (t :: *) where
  select :: r -> Lab f -> t
```

Thanks to the dependency, we can also use functional notation to rewrite the type of `select` in the following form:

```
select :: r -> Lab f -> Select r f
```

This type confirms the intuition that taking the type of a selection from an object gives the same result as selecting from the type of the object.

In practice, we will often use the `select` function and the associated `Select` type function with a specific label value, `#.x` that is known at compile-time. Recognizing this common case, the syntax of Habit interprets any expression of the form `e.x` as an abbreviation for `select e #.x`. and any type expression `r.x` as an abbreviation for `Select r #.x`. In particular, if `e` has type `r`, then `e.x` has type `r.x`, where the latter, thanks to the use of functional notation, is actually an abbreviation for `Select r #.x t => t`.

The Habit compiler automatically generates instances of `Select` for bitdata types (Section 3.6.8) and structure types (Section 3.6.9) so that the components of these types can be accessed using dot notation. In principle, however, it is also possible to define instances of `Select` for other types. The following example illustrates this by defining a type `Temperature` with values that can be read in either Fahrenheit or Celsius by using an appropriate projection:

```
data Temperature = Temp Signed
instance Temperature.celsius = Signed where
  (Temp c).celsius = c
instance Temperature.fahrenheit = Signed where
  (Temp c).fahrenheit = 32 + (c*9)/5
```

This code uses the dot notation abbreviations described previously, but the same program can also be written directly in terms of `select` and `Select`:

```
instance Select Temperature #celsius = Signed where
  select (Temp c) #celsius = c
instance Select Temperature #fahrenheit = Signed where
  select (Temp c) #fahrenheit = 32 + (c*9)/5
```

A more compelling reason to use `Select` explicitly is to define generic operations that will work for any label type. For example, the following definition allows us to select the value of a 'field' `f` from an arbitrary monadic computation, so long as the result that it produces has an `f` field:

```
instance Select (m r) f = m (Select r f) if Monad m where
   select c f = do r <- c; return (select r f)
```

In particular, this definition includes instances `(m r).x = m (r.x)` as a special case for each identifier `x`.

Update. As mentioned in Section 3.4.1, in addition to the dot notation for selecting components of an object, Habit also provides a notation, `e1[x=e2]` for updating the values of those components. (More accurately, this is a pure operation, so the expression `e1[x=e2]` actually constructs a *new* value that is like `e1` except that the `x` field has been *replaced* with the value of `e2`.) Also like the dot notation, the update syntax `e1[x=e2]` is implemented as syntactic sugar for the expression `update e1 #.x e2`, which uses the `update` function that is defined in the following class:

```
class Update (r :: *) (f :: lab) where
  update :: r -> Lab f -> Select r f -> r
```

Updates involving multiple fields can be implemented using a nested sequence of `update` calls, as in the following example:

```
e[x=e1|y=e2]   =   update (update e #.x e1) #.y e2
```

Once again, the compiler will automatically generate appropriate instances of `Update` for each bitdata type (Section 3.6.8) and structure type (Section 3.6.9) in a program. However, there is nothing to prevent a programmer from defining other instances of `Update` for user-defined types where that seems appropriate.

4.4 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 | 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 | 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.)

4.5 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 (a preview from Section 4.8) 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 NumLit n (Bit m) if Width m, (n < 2^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.

4.6 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 type function with the following definition:

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

class NonZero t = t' | Num t, t -> 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 (NonZero t) if NumLit n t, 0 < n
```

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.

4.7 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 4.8) or an array (Section 4.14). 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].

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

class Index n | 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  Index (2^n) if n < WordSize where
  -- implementation can use bit-oriented operations (e.g., masking)
else Index n if n < 2^WordSize where
  -- implementation uses modulo arithmetic
```

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

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)          if Index n
instance Ord (Ix n)         if Index n
instance Bounded (Ix n)     if Index n
instance Num (Ix n) fails
instance NumLit i (Ix n)    if Index n, i < n
```

4.8 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 -> *
```

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 3.3.5) for concatenating bit vectors:

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

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 Eq (Bit n)          if Width n
instance Ord (Bit n)         if Width n
instance Bounded (Bit n)     if Width n
instance Num (Bit n)         if Width n
```

```
instance NonZero (Bit n) = _ if Width n
instance NumLit n (Bit m)    if Width m, n < 2^m
```

In particular, numeric literals for values of type `Bit n` are allowed only for literals that are less than 2^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) | 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 4.9 where index values are used to reference individual bits within a bit vector.

4.9 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 | 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 | FromBits t, Index (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 4.8 provide prototypical instances for each of the classes that we have described in this section. There are also instances of these classes for index types, but only in those special cases where the size of the index type is a power of two:

```
instance BitSize (Bit n) = n if Width n
instance ToBits (Bit n)      if Width n
instance FromBits (Bit n)    if Width n
instance BitManip (Bit n)    if Width n

instance BitSize (Ix p) = n  if Index p, 2^n = p
instance ToBits  (Ix p)      if Index p, 2^n = p  -- OVERLY RESTRICTIVE
instance FromBits (Ix p)     if Index p, 2^n = p
instance BitManip (Ix p)     if Index p, 2^n = p
```

4.10 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 Boolean (Bit n) if Width n
instance Boolean (Ix p)  if Index p, 2^n = 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 are instead packaged in a subclass because they are not particularly useful for all `Boolean` types (such as `Bool`, for example):

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

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

instance Shift (Bit n) if Width n
instance Shift (Ix p)  if Index p, 2^n = p
```

4.11 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 expected classes:

```
instance Eq Unsigned           ;   instance Eq Signed
instance Ord Unsigned          ;   instance Ord Signed
instance Num Unsigned          ;   instance Num Signed
instance NonZero Unsigned = _  ;   instance NonZero Signed = _
instance Bounded Unsigned      ;   instance Bounded Signed
instance Boolean Unsigned      ;   instance Boolean Signed
instance Shift Unsigned        ;   instance Shift Signed
instance ToBits Unsigned       ;   instance ToBits Signed
instance FromBits Unsigned     ;   instance FromBits Signed
instance BitManip Unsigned     ;   instance BitManip Signed
```

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 NumLit i Unsigned if i < 2^WordSize
instance NumLit i Signed   if i < 2^(WordSize - 1)
```

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 ToUnsigned (Bit n) if Width n
instance ToUnsigned (Ix n)  if Index n

class ToSigned t where
  signed :: t -> Signed

instance ToSigned Unsigned
```

```
instance ToSigned Signed
instance ToSigned (Bit n) if Width n
instance ToSigned (Ix n)  if Index 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.

4.12 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 -> t') if Pointed 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:

```
class (a :: *) =<= (b :: *)
instance a =<= b fails if Pointed a, Pointed b fails
else     a =<= b
```

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 remains as one of the most unusual aspects of the language design and as a topic that will be described in more detail in future versions of this report.

4.13 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` notatation 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
```

4.14 Memory Areas, References and Alignments

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

Area Types. 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. 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 3.6.10.) 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.

```
primitive type ARef :: nat -> area -> *

instance Eq (ARef l a)                       if Alignment l
instance BitSize (ARef l a) = WordSize - n  if Alignment l, 2^n = l
instance ToBits (ARef l a)                   if Alignment l
instance FromBits (ARef l a) fails
```

The instances listed here show that references may be tested for equality using Eq and converted to bit vectors using ToBits. The last line, however, is probably the most important part of this code because it ensures that the fromBits function cannot be used to fake an invalid reference value from an arbitrary bit vector; this restriction is essential, of course, to ensure memory safety.

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 MinAlign :: nat

type Ref = ARef MinAlign
```

The MinAlign constant reflects the minimum valid alignment on the underlying platform. For example, we might have MinAlign = 4 on machines that require word alignment, or MinAlign = 1 on machines that allow arbitrary alignment. There is also a class, Alignment, whose instances are the type-level numbers corresponding to legal alignment values on the target 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 Alignment (2^n) if Width n      -- MinAlign = 1
instance Alignment (4*n) if (n > 0)      -- MinAlign = 4
instance Alignment n     if (n > 0)      -- MinAlign = 1
```

Pointer Types. In addition to reference types like Ref a, whose values are guaranteed to hold valid addresses to appropriately typed memory areas, Habit also supports pointer types whose values are guaranteed either to be a valid reference or else to be Null. As such, any program that works with a pointer type must perform extra Null pointer checks to obtain valid references into memory. As with references, pointer types come in two flavors, depending on whether an explicit or a default alignment is required:

```
primitive type APtr :: nat -> area -> *
type Ptr = APtr MinAlign

instance Eq (APtr l a)                          if Alignment l
instance BitSize (APtr (2^n) a) = WordSize - n  if Alignment l
```

```
instance ToBits (APtr l a)                    if Alignment l
instance FromBits (APtr l a) fails
```

From a programmer's perspective, however, pointer types behave much as if they were introduced by a datatype definition of the following form:

```
data APtr l a = Ref (ARef l a)
              | Null
```

In particular, Habit programs can build pointer values using `Null` and `Ref` as constructor functions, and they can use pattern matching over the same constructors to implement `Null` pointer tests.

Stored Data. 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 `ToBits` 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 BE t = _ if ToBits t, BitSize t = 8 * n

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

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 Stored t = _ if ToBits t, BitSize t = 8 * n
```

Arrays and Padding. In addition to the primitive area types—`LE t`, `BE t` and `Stored t`—and user defined structure types (Section 3.6.9), Habit also provides support for memory based array or table structures. Specifically, the `Array n a` and `Pad n a` types both describe a memory area that contains a contiguous block of `n` component areas each with layout `a`. The difference between these two types is that Habit does not provide any operations for accessing any part of a `Pad` area, so a `Pad n a` type is useful only for describing padding. On the other hand, the components of an `Array n a` area can be accessed using the 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 type Pad   :: nat -> area -> area

primitive (@@) :: (Index n) =>
                   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`.

Area Size. 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.

Data Access. 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]:

[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

```
class MemMonad m | 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` 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.

4.15 Memory Area Initialization

Memory allocated by `area` declarations (Section 3.6.10) must, in general, be properly initialized at startup time. Failing to initialize a `Stored (Ix n)` with an appropriate index, or a `Stored (Ref a)` with a valid address, for example, could compromise both type- and memory-safety guarantees for Habit programs. But even in cases where it is not strictly required—such as when dealing with a `Stored Unsigned` field—some form of initialization is still likely to be needed for algorithmic purposes. Suitable initialization code could be included as part of a `main` routine elsewhere in the code, but separating the definition of the area and the code for initialization in this way is awkward and error-prone.

In this section, we describe the types, functions, and classes that are provided in the standard environment to support the definition of initializers for memory areas. The resulting initializers can then be used within individual `area` declarations to ensure valid and predictable initialization. The central idea is to introduce an abstract type for initializers: a value of type `Init a` captures a method for initializing a memory area of the type described by the `a` parameter:

complexity) has been added to the `IO` monad in Haskell.

```
primitive type Init :: area -> *
instance Pointed (Init a) fails
```

Initializers are pure values, but they correspond to methods that, when executed, can write to memory, but not read or perform other side-effecting computations. As a result, a collection of initializers, operating on disjoint memory areas, can be executed in any order with no observable difference in semantics.

We describe `Init` as an abstract type because the implementation of `Init a` values is not exposed to the programmer. Instead, the range of initializers that can be specified is limited by the set of operations that are provided by the languge and its standard environment for constructing `Init a` values. The `Pointed (Init a) fails` instance guarantees that the construction of initializers will terminate, but terminatation of the initializers themselves follows independently by a case analysis of the operations on the abstract type. Those operations include the special syntax for structure initialization that is described in Section 3.6.9, as well as a general primitive for initialization of arrays:

```
primitive initArray :: Index n => (Ix n -> Init a) -> Init (Array n a)
```

Note that `initArray` is a higher-order function; the argument—a function from index values to initializers—allows us to specify a potentially different initialization strategy for each array element.

The Habit standard environment also includes another higher-order function, `initSelf`, for defining initializers on structures that use the address of the object being initialized as an input to the initialization process:

```
primitive initSelf :: (Ref a -> Init a) -> Init a
```

To illustrate this, consider the following simple structure that might be used to implement a doubly-linked list or a union-find algorithm:

```
struct DLL [ prev, next :: Stored (Ref DLL) | val :: Stored Unsigned ]
```

Each node of a `DLL` structure contains a stored value as well as pointers to the previous and next items in the list. Using `initSelf`, we can define an initializer for `DLL` that sets `prev` and `next` to point to the object that is being initialized:

```
initDLL  :: Unsigned -> Init DLL
initDLL v = initSelf (\self -> DLL [ prev <- initStored self
                                   | next <- initStored self
                                   | val  <- initStored v ])
```

In the following subsections, we describe the primitives that support initialization of stored data (Section 4.15.1); the `NullInit` and `NoInit` classes that capture common patterns for null- and no-initialization (in Sections 4.15.2 and 4.15.3, respectively); and the `Initable` class that allows the definition of a default initializer for each memory area type (Section 4.15.4).

4.15.1 Initialization of Stored Data

Memory areas of the form `Stored t` (as well as the `LE t` and `BE t` variants) can be initialized by specifying an appropriate initial value of type `t`, as described by the following three primitives[7]:

```
initStored :: t -> Init (Stored t)
initLE     :: t -> Init (LE t)
initBE     :: t -> Init (BE t)
```

Because numeric literals in Habit are overloaded, they can also be used as initializers defined implicitly in terms of `initStored` via the following instance (with very similar instances for `LE` and `BE`):

```
instance NumLit n (Init (Stored t)) if NumLit n t where
  fromLiteral n = initStored (fromLiteral n)
```

4.15.2 Null Initialization

A large class of memory areas can be initialized safely by setting their contents to zero/null; the area types for which this is possible are captured as instances of the `NullInit` class:

```
class NullInit a where
  nullInit :: Init a
```

Mechanisms for enabling null-initialization of structure types are described in Section 3.6.9, while null-initialization of an array is permitted whenever the type of the elements in the array supports null-initialization:

```
instance NullInit (Array n a) if Index n, NullInit a where
  nullInit = initArray (\i -> nullInit)
```

[7] Written in the form shown here, the types of these primitives suggest that they are fully polymorphic in `t`. Note, however, that use of the `Stored`, `LE`, or `BE` type functions actually implies a restriction to choices of `t` that are instances of the corresponding class. Indeed, these primitives will typically be implemented as members of those type function classes.

An area of padding, however, cannot be null-initialized because that would require writing data into an inaccessible region of memory:

```
instance NullInit (Pad n a) fails
```

We can also null-initialize a range of `Stored` values, with special cases for pointer and index types and a general case for stored values that can be represented by a vector of zero bits (again, there are very similar instances for `LE` and `BE`):

```
instance NullInit (Stored (APtr l a)) if Alignment l where
  nullInit = initStored Null
else     NullInit (Stored (Ix n)) if Index n where
  nullInit = initStored 0
else     NullInit (Stored t) if FromBits t where
  nullInit = initStored (fromBits 0)
else     NullInit (Stored t) fails
```

4.15.3 No Initialization

Some memory areas can be used immediately without any explicit initialization steps. This approach, which is referred to as *no-initialization* in Habit, is available only in cases where type safety is assured, but it does not guarantee that an area will contain useful or predictable values. As such, no-initialization may be useful in special situations—for example, to support lazy initialization of large arrays—but further initialization steps will typically still be required elsewhere in the program before the memory is actually used to ensure correct algorithmic behavior. The area types that support no-initialization are captured as instances of the `NoInit` class:

```
class NoInit a where
   noInit :: Init a

instance NoInit (Pad n a)    if NoInit a
instance NoInit (Array n a)  if NoInit a
instance NoInit (Stored t)   if FromBits t
instance NoInit (LE t)       if FromBits t
instance NoInit (BE t)       if FromBits t
```

Note that the last three instances (for stored data) allow no-initialization only for types that permit construction from an arbitrary bit-level representation. In particular, this excludes reference, pointer, and index types (except, in the latter case, when the number of valid index values is a power of two).

Structure types whose components can all be no-initialized can also be included as instances of `NoInit`, as described in Section 3.6.9.

4.15.4 Default Initialization

Habit allows programmers to associate a default initialization strategy with each memory area type by defining an instance of the `Initable` class:

```
class Initable a where
  initialize :: Init a
```

The default initializers that are defined in this way are used for initializing structure fields when no explicit initializer has been specified, or for handling the declaration of a memory area that does not include an explicit initializer. (Conversely, it is an error to declare a memory area with a type of the form `ARef L A` for some area type `A` without specifying an explicit initializer unless `A` has been declared as an instance of `Initable`.)

The selection of a default initialization strategy must be made carefully. Using null-initialization as a default is likely to result in better predictability, while no-initialization might result in better performance. There are also cases where neither of those approaches is applicable, or where other, application-specific behavior is required. The set of predefined instances of `Initable` in Habit is described by the following declarations, using null-initialization for stored data (with similar instances for `LE` and `BE`), element-wise initialization for arrays, and no-initialization for padding:

```
instance Initable (Stored t) if NullInit (Stored t) where
  initialize = nullInit

instance Initable (Array n a) if Initable a where
  initialize = initArray (\ix -> initialize)

instance Initable (Pad n a) if NoInit a where
  initialize = noInit
```

Default initializers for a structure type, using a null- or a no-initialization strategy, for example, can be generated automatically by including an appropriate `deriving` clause as part of the `struct` declaration for that type (see Section 3.6.9), but it is also possible to specify a default initialization behavior for a structure type using a handwritten instance of `Initable`.

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 $p \leq 256$. 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 usually 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`.

To illustrate how this works in practice, the diagram in Figure 4 shows a specific configuration that represents the set `{3,4,5,8}`. The left portion of this

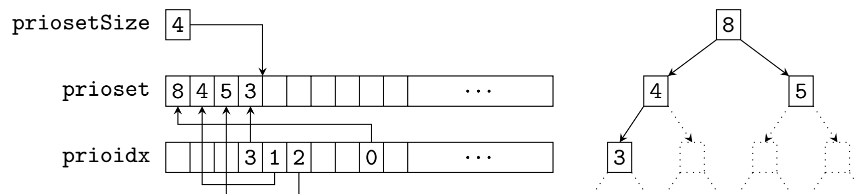

Figure 4: Priority set data structures for `{3,4,5,8}`

diagram shows the concrete data structures, including the `priosetSize` variable that records a total of four elements in the set, and the two arrays `prioset` and `prioidx`. The arrows from `prioidx` to `prioset` show the mapping from individual priorities, `p`, to corresponding positions, `i`, in the main priority set array. Many of the array elements have been left blank; their contents are not important because they will not be used in any of the computations that are performed using the priority set. The right portion of the diagram shows the maximum heap structure that is encoded within the arrays, marking as-yet unused portions of the tree with dotted lines. The maximum heap structure is visible here with each element greater than the elements in its children, and the maximum priority (in this case, `8`) at the root of the tree (and hence at the

beginning of the prioset array).

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 priority set structures as memory areas using the following declarations. For stylistic reasons, however, 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 priosetSize <- 0  :: Ref (Stored Unsigned)
area prioset <- noInit :: Ref (Array NumPrio (Stored Priority))
area prioidx <- noInit :: Ref (Array NumPrio (Stored (Ix NumPrio)))
```

The initializers here ensure that the priority set is initially empty. In this particular case, we could simply have relied on the default initializers, which would have achieved exactly the same effect for priosetSize and performed redundant but harmless null-initialization of the prioset and prioidx arrays. Once again, however, for stylistic reasons, we prefer to make the initializers explicit.

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)[8]:

```
-- 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

[8]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 4.14.

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 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 decIx 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.

Index

&&, 52
() (type), 15, 16, 25, 52
*, 59
* on types, 21, 53
+, 59
+ on types, 21, 53
-, 59
- on types, 21, 54
-> function types, 10, 14, 51, 58, 67
.&., 65
.^., 65
/ on types, 21, 54
/=, 58
:#, 21, 27, 43, 62
<, 58
< on types, 19, 54
<=, 58
<= on types, 19, 54
<=?, 61, 62, 86, 87
=<= (type class), 19, 68, 69
==, 58
>, 58
>=, 58
>>=, 69
^ on types, 21, 54

Alignment (type class), 19, 70, 71, 77
APtr (type constructor), 16, 71, 77
area (kind), 13, 44, 70
area declarations, 44, 49
ARef (type constructor), 16, 49, 70, 72, 73
Array (type constructor), 16, 50, 73, 75–78, 81

BE (type function), 21, 72–74, 76–78
Bit (type constructor), 11, 12, 16, 28, 40, 60, 62–66
BitManip (type class), 19, 39, 44, 63–65
BitSize (type function), 19, 21, 51, 63, 64, 66, 70–73
bitSize, 64
Bool (type), 16, 25, 51, 65
Boolean (type class), 19, 39, 44, 65
bottom, 67
Bounded (type class), 18, 19, 39, 44, 51, 58, 59, 62, 65
ByteSize (type function), 21, 44, 45, 49, 73

case, 25
case-from, 26
class declarations, 31
ClassDecl, 31
ClassLhs, 31
clearBit, 64
Comments, 9, 10
Con, 10
Conid, 10, 14, 39, 47, 55
Conop, 10
Consym, 10

decIx, 61, 83
default definitions in a class, 33, 34
div, 20, 60
Division, 60
do notation, 12, 23, 24, 69
dot notation, 5, 14, 23, 55–57

Eq (type class), 18, 19, 35, 36, 38, 44, 51, 52, 57, 58, 62, 65, 70, 71

False (constructor function), 25, 51
fix, 67
flipBit, 64
FromBits (type class), 19, 44, 51, 63–65, 70, 71, 77
fromBits, 19, 63, 71, 77
fromLiteral, 59, 76
Functional dependency, 18, 20, 32, 33, 53–56

GCD (type function), 21, 54

if, 25
if-from, 25
incIx, 61

Index (type class), 19, 61–66, 73, 75
Index Types, 61
Init (type constructor), 16, 47, 50, 74, 75
Initable (type class), 19, 47–50, 76, 78
initArray, 50, 75, 76, 78
initBE, 76
initialize, 78
initLE, 76
initStored, 47, 50, 75–77
instance declarations, 32, 33
isJunk, 41, 63
Ix (type constructor), 16, 19, 61, 62, 64–66, 73–75, 77, 81

Just (constructor function), 16, 52, 62, 83, 87

Keywords, 8
Kinds, 5, 13, 14
- *, 13
- area, 13, 70
- Function kinds, 14
- lab, 14, 16, 55
- nat, 13
- type, 13

Lab (type constructor), 14, 16, 55–57
lab, 14, 16, 55
Labels, 14
Layout, 12, 24
LE (type function), 21, 51, 72–74, 76–78
let, 23, 24
Literals, 11, 59
- Binary, 11
- Bit Vector, 12
- Integer, 11, 13

Literate files, 10

max, 58
maxBound, 59
Maybe (type constructor), 16, 52, 60, 61, 87
maybeIx, 61
memCopy, 73
MemMonad (type class), 19, 73, 81
Memory Areas, 13
memZero, 73
min, 58
minBound, 59
mod, 60
modIx, 61, 62, 83, 85, 86, 88, 90
Monad (type class), 18, 19, 39, 44, 57, 69, 73

Nat (type), 16, 59
nat, 15, 52
negate, 59
NoInit (type class), 19, 48, 76–78
noInit, 77, 78, 81
NonZero (type function), 20, 21, 60–62, 65
nonZero, 60
Nothing (constructor function), 16, 52, 61, 62, 83, 87
Null (constructor function), 71, 72, 77
NullInit (type class), 19, 48, 49, 76–78
nullInit, 49, 76–78
Num (type class), 18, 19, 39, 44, 58–60, 62, 65
NumLit (type class), 11, 19, 59–62, 66, 76

Ord (type class), 18, 19, 38, 44, 51, 52, 58, 59, 62, 65
overlapping instances, 33–36

Pad (type constructor), 16, 73, 77, 78
Patterns, 26
Pointed (type class), 19, 39, 44, 67, 69, 74
Predicates, 13
Ptr (type constructor), 16, 71

quot, 60

readRef, 46, 73, 74, 83, 85–90
Ref (constructor function), 72
Ref (type constructor), 14, 16, 46, 47, 49, 71–74, 78, 81
relaxIx, 61
rem, 60
return, 69

`Select` (type function), 21, 42, 46, 56, 57
`select`, 56, 57
`setBit`, 64
`Shift` (type class), 19, 39, 44, 65
`shiftL`, 65
`shiftR`, 65
`Signed` (type), 16, 65, 66
`signed`, 66, 67
`Stored` (type function), 21, 45, 46, 48, 50, 72–78
`struct` declarations, 44
superclass, 32

`testBit`, 64
`ToBits` (type class), 19, 27, 44, 51, 63–65, 70–72
`toBits`, 19, 27, 63
`ToSigned` (type class), 19, 66
`ToUnsigned` (type class), 19, 66
`True` (constructor function), 25, 51
`type`, 5, 13, 37
type synonyms, 37
Type-level Numbers, 13
`TypeLhs`, 31
`TypeParam`, 31
Types, 13, 15
    Function Types, 15

Unit type, 16, 25, 52
`Unsigned` (type), 16, 65, 66
`unsigned`, 66, 67
`Update` (type class), 19, 42, 47, 57
`update`, 57

`ValIn` (type function), 18, 21, 73, 74
`Var`, 10
`Varid`, 10, 11, 14, 27, 55
`Varop`, 10
`Varsym`, 10, 11

`Width` (type class), 19, 40, 62–66, 71
`WordSize` (type), 16, 45, 61, 66, 67, 70, 71
`writeRef`, 47, 73, 74, 81, 83, 85, 88