

BH™ (Bluespec Haskell/Classic)

Language Reference Guide

Revision: 17 February 2024

Copyright © 2000 – January 2020: Bluespec, Inc.
January 2020 onwards: various open-source contributors

Trademarks and copyrights

Verilog is a trademark of IEEE (the Institute of Electrical and Electronics Engineers). The Verilog standard is copyrighted, owned and maintained by IEEE.

VHDL is a trademark of IEEE (the Institute of Electrical and Electronics Engineers). The VHDL standard is copyrighted, owned and maintained by IEEE.

SystemVerilog is a trademark of IEEE. The SystemVerilog standard is owned and maintained by IEEE.

SystemC is a trademark of IEEE. The SystemC standard is owned and maintained by IEEE.

Bluespec is a trademark of Bluespec, Inc.

Contents

Table of Contents 3

1 Introduction 6

2 Types 7

2.1 Type classes and overloading . . . 9

2.1.1 Context-qualified types . . . 9

3 Packages 10

3.1 Name clashes and qualified names . . . 11

4 Top level definitions 11

4.1 `data` . . . 11

4.2 `struct` . . . 13

4.2.1 Tuples . . . 13

4.3 `type` . . . 14

4.4 `interface` . . . 14

4.5 `class` declarations . . . 15

4.6 `instance` declarations . . . 16

4.6.1 Deriving `Bits` . . . 16

4.6.2 Deriving `Eq` . . . 16

4.6.3 Deriving `Bounded` . . . 17

4.6.4 Deriving for isomorphic types . . . 17

4.7 Value definitions . . . 17

4.8 Calling foreign functions . . . 18

5 Expressions 19

5.1 Applications . . . 19

5.2 Variables . . . 21

5.3 Constructors and literal constants . . . 21

5.3.1 Integer literals . . . 21

5.3.2 String literals . . . 22

5.4 `case` and `if` . . . 22

5.5 `let` . . . 23

5.6 Structs and Tuples . . . 23

5.7 Struct field selection . . . 24

5.8 Struct “update” . . . 24

5.9 `interface` expressions . . . 24
5.10 "Don't care" expressions . . . 25
5.11 Actions . . . 25
5.11.1 `ActionValue` . . . 26
5.12 `rules` . . . 26
5.12.1 Nested rule guards . . . 27
5.12.2 Aggregating and prioritizing rules . . . 28
5.13 Modules . . . 28

6 Patterns **30**
6.1 Variable and wild-card patterns . . . 30
6.2 Constructor and constant patterns . . . 30
6.3 Struct and Tuple patterns . . . 31

7 Guards **31**

8 Important Primitives **31**
8.1 The "size" types . . . 32
8.2 The type `Bit` . . . 33
8.3 The `Bits` class . . . 33
8.4 UInt, Int . . . 34
8.5 Registers . . . 34
8.6 FIFOs . . . 35
8.7 FIFOFs . . . 35

9 Interfacing to Verilog **36**
9.1 Verilog modules . . . 36
9.1.1 Method definitions . . . 37
9.1.2 Parameters and arguments . . . 37
9.1.3 Scheduling information . . . 39
9.1.4 Multiple methods . . . 39
9.2 Generated Verilog . . . 39
9.2.1 Verilog code generation properties . . . 41

10 Interfacing to C **42**
10.1 C modules . . . 43
10.2 Generated C . . . 43

11 Guiding the compiler **44**
11.1 Pragmas . . . 44
11.1.1 Pragma `verilog` . . . 44
11.1.2 Pragma `noinline` . . . 44
11.2 Rule assertions . . . 45
11.2.1 Assertion `fire when enabled` . . . 45
11.2.2 Assertion `no implicit conditions` . . . 45

A Advanced topics **46**
A.1 Lambda expressions . . . 46
A.2 `do` . . . 46
A.2.1 Creating modules with `do` . . . 46
A.2.2 Recursive bindings in module . . . 46
A.3 `IsModule` . . . 47

B Syntax **49**
B.1 Reserved words . . . 49

C Semantics of primitive operations **50**

D The Standard Prelude and Additional Libraries **52**

Index **53**

1 Introduction

BH (Bluespec Haskell/Classic) is a language for hardware design. The language borrows its notation, type and package system from an existing general-purpose functional programming language called Haskell [HPJWe92] where those constructs have been well tested for over a decade. Unlike Haskell, BH is meant solely for hardware design— a BH program represents a circuit. The abstract model for these circuits is a Term Rewriting System (TRS); details about using TRSs for describing circuits, and compiling these descriptions to real hardware, may be found in James Hoe's thesis [Hoe00]. BH has several restrictions and extensions relative to Haskell, arising out of this hardware focus.

This document is not meant as a tutorial on BH (separate documents exist for that purpose). Nevertheless, this document has numerous small examples to explicate BH notation.

Meta notation

The grammar rules in the presentation below mostly follow the usual EBNF (Extended BNF) structure. Grammar alternatives are separated by "|". Items enclosed in [ ] are optional. Items enclosed in { } can be repeated zero or more times. The last piece of notation is used sloppily; sometimes there must be at least one item, and also, the last terminal inside the { } is sometimes a separator rather than terminator.

Identifiers and the rôle of upper and lower case

An identifier in BH consists of a letter followed by zero or more letters, digits, underscores and single quotes. Identifiers are case sensitive: `glurph`, `gluRph` and `Glurph` are three distinct identifiers.

The case of the first letter in an identifier is very important. If the first letter is lower case, the identifier is a "variable identifier", referred to in the grammar rules as a *varId*. If the first letter is upper case, the identifier is a "constructor identifier", referred to in the grammar rules as a *conId*.

In BH, package names (*packageId*), type names (*tycon*) and value constructor names are all constructor identifiers. (Ordinary) variables, field names and type variables are all variable identifiers.

A lone underscore, "`_`", is treated as a special identifier— it is used as a "don't care" pattern or expression (more details in Sections 5.10 and 6.1).

The Standard Prelude

The Standard Prelude is a predefined package that is imported implicitly into every BH package. It contains a number of useful predefined entities (types, values/functions, classes, instances, etc.). It is somewhat analogous to the combination of various ".h" files and standard libraries in C, except that in BH no special action is needed to import the prelude or to link it in. We will refer to the prelude periodically in the following sections, and there are more details in appendix D.

Lexical syntax/layout

In BH, there are various syntactic constructs that involve zero or more items enclosed in braces and separated by semicolons:

`{` *item* `;` *item* `;` ⋯ `;` *item* `}`

These braces and semicolons can be omitted entirely if the components are laid out with proper indentation.

Suppose the parser discovers a missing open brace (e.g., after the keywords `where`, `let`, `do` and `of`). Then, the indentation of the next lexical element is remembered (and the missing open brace is implicitly inserted before it). For each subsequent line, if it contains only whitespace or is indented more, then it is treated as a continuation of the current item. If it is indented the same amount, it is treated as the beginning of the next item (*i.e.,* a semicolon is inserted implicitly before the item). If it is indented less, then the list of items is considered to be complete (*i.e.,* a closing brace is implicitly inserted). An explicit brace is never matched against an implicit one. Thus, while using the layout rule, if the parser encounters an explicit open brace, then it does not resume using the layout rule for this list of items until it has "emerged" past the explicit corresponding closing brace (a construct nested inside this list of items may still use the layout rule).

Comments in BH programs

In a BH program, a *comment* is legal as whitespace, and may be introduced in two ways. An *ordinary comment* is introduced by a lexical token consisting of two or more consecutive dashes followed by a non-symbol, and extends up to and including the end of the line. (See Section 5.1 for the list of symbols.) Note: the lexical token `-->` is a legal token in BH, and since it contains three consecutive dashes followed by a symbol, it does not begin a comment.

A *nested comment* is introduced by the lexeme "`{-`" and extends until the next matching "`-}`", possibly spanning multiple lines. A nested comment can itself contain another nested comment; this nesting can be repeated to any depth.

In an ordinary comment, the character sequences "`{-`" and "`-}`" have no special significance, and, in a nested comment, a sequence of dashes has no special significance.

General organization of this document

A concept that is pervasive in BH is the notion of a *type*. Every value expression in BH, even a basic value identifier, has a type, and the compiler does extensive static type checking to rule out absurd use of values (such as taking the square root of an IP address). Types are discussed in section 2.

A BH program consists of one or more packages. These outermost constructs are described in section 3. As explained later, a BH package is a linguistic namespace-management mechanism and does not have any direct correlation with any hardware module being described by the program. Hardware modules correspond to *modules*, a particular type of value in BH.

Within each package is a collection of top-level definitions. These are described in section 4.

Amongst the top-level definitions are *value definitions* (section 4.7), which constitute the actual meat of the code. Value definitions are built around *expressions*, which are described in section 5.

2 Types

Every value expression and, in particular, every value identifier in BH has a *type*. In some cases the programmer must supply a *type signature* specifying this and in many cases the compiler infers it automatically. The BH programmer should be aware of types at all times.

type	::=	*btype* [`->` *type*]
btype	::=	[*btype*] *atype*
atype	::=	*tycon* | *tyvar* | `(` { *type* `,` } `)`
tycon	::=	*conId*

Most type expressions have the form:

TypeConstructor $t_1 \cdots t_n$

where $t_1 \cdots t_n$ are themselves type expressions, and $n \geq 0$. The $t_1 \cdots t_n$ are referred to as the *type arguments* to the type constructor. n is also called the *arity* of the type constructor.

Familiar basic types have zero-arity type constructors (no type arguments, $n = 0$). Examples:

```
Integer
Bool
String
Action
```

Other type constructors have arity $n > 0$; these are also known as *parameterized types*. Examples:

```
List   Bool
List  (List Bool)
Array  Integer  String
Maybe  Integer
```

These represent the types of lists of Booleans, lists of lists of Booleans, arrays indexed by integers and containing strings, and an optional result possibly containing an integer.

A type can be *polymorphic*, indicated using type variables. Examples:

```
List   a
List  (List b)
Array  i  (List  String)
```

These represent lists of things of some unknown type "`a`", lists of lists of things of some unknown type "`b`", and arrays indexed by some unknown type "`i`" and containing lists of strings.

One type constructor is given special status in the syntax. The type of functions from arguments of type t_1 to results of type t_2 could have been written as:

`Function` t_1 t_2

but in BH we write the constructor as an infix arrow:

t_1 `->` t_2

These associate to the right, *i.e.,*

t_1 `->` $\cdots$ `->` t_{n-1} `->` t_n $\equiv$ t_1 `->` (`$\cdots$` `->` (t_{n-1} `->` t_n))

There is one particular set of niladic type constructors that look like numbers. These are used to represent certain "sizes". For example, the type:

```
Bit 16
```

consists of the unary type constructor `Bit` applied to type represented by the niladic type constructor "`16`". The type as a whole represents bit vectors of length 16 bits. Similarly the type

```
UInt 32
```

represents the type of unsigned integers that can be represented in 32 bits. These numeric types are said to have kind `#`, rather than kind `*` for value types.

Strings can also be used as type, having kind `$`. This is less common, but string types are quite useful in the generics library, described in the *Libraries Reference Guide*. Examples:

```
MetaData#("Prelude","Maybe",PrimUnit,2)
MetaConsNamed#("Valid",1,1)
```

2.1 Type classes and overloading

BH's `class` and `instance` mechanisms form a systematic way to do *overloading* (the approach has been well tested in Haskell).

Overloading is a way to use a common name to refer to a set of operations at different types. For example, we may want to use the "`<`" operator name for the integer comparison operation, the floating-point comparison operation, the vector comparison operation and the matrix comparison operation. Note that this is not the same as polymorphism: a polymorphic function is a *single* function that is meaningful at an infinity of types (*i.e.,* at every possible instantiation of the type variables in its type). An overloaded identifier, on the other hand, usually uses a common name to refer to a (usually) small set of distinct operations.

Further, it may make sense to have "`<=`", "`>`" and "`>=`" operations wherever there is a "`<`" operation, on integers, floating points numbers, vectors and matrices. Rather than handle these separately, we say:

- there is class of types which we will call `Ord` (for "ordered types"),
- that the integer, floating point, vector and matrix types are members (or "instances") of this class, and
- that all types that are members of this class have appropriate definitions for the "`<`", "`<=`", "`>`" and "`>=`" operations. We also say that these operations are *overloaded* across these instance types, and we refer to these operations as the *methods* of this class.

Another example: we could use a class `Hashable` with an operation called `hash` to represent those types T for which we can and do define a hashing function. Each such type T has to specify how to compute the `hash` function at that type.

Classes, and the membership of a type in a class, do not come into existence by magic. Every class is created explicitly using a class declaration, described in section 4.5. A type must explicitly be made an instance of a class and the corresponding class methods have to be provided explicitly; this is described in 4.6.

2.1.1 Context-qualified types

Consider the following type declaration:

```
sort  ::  (Ord a) => List a -> List a
```

It expresses the idea that a sorting function takes an (unsorted) input list of items and produces a (sorted) output list of items, but it is only meaningful for those types of items ("`a`") for which the ordering functions (such as "`<`") are defined. Thus, it is ok to apply `sort` to lists of `Integer`'s or lists of `Bool`'s, because those types are instances of `Ord`, but it is not ok to apply `sort` to a list of, say, `Counter`'s (assuming `Counter` is not an instance of the `Ord` class).

In the type of `sort` above, the part before "`=>`" is called a *context*. A context expresses constraints on one or more type variables— in the above example, the constraint is that any actual type "`a`" must be an instance of the `Ord` class.

A context-qualified type has the following grammar:

ctxType	::=	[*context* `=>`] *type*
context	::=	`(` { *classId* { *varId* } `,` } `)`
classId	::=	*conId*

In the above example, the class `Ord` had only one type parameter (*i.e.,* it constrains a single type) but, in general, a type class can have multiple type parameters. For example, in BH we frequently

use the class "`Bits a n`" which constrains the type represented by `a` to be representable in bit strings of length represented by the type `n`.

Note

When using an overloaded identifier `x` there is always a question of whether or not there is enough type information available to the compiler to determine which of the overloaded `x`'s you mean. For example, if `read` is an overloaded function that takes strings to integers or Booleans, and `show` is an overloaded function that takes integers or Booleans to strings, then the expression `show (read s)` is ambiguous— is the thing to be read an integer or a Boolean?
In such ambiguous situations, the compiler will so notify you, and you may need to give it a little help by inserting an explicit type signature, e.g.,

```
show ((read s) :: Bool)
```

End of Note

3 Packages

Packages are the outermost constructs in BH— all BH code must be inside packages. There should be one package per file. A BH package is a linguistic device for namespace control, and is particularly useful for programming-in-the-large. A package does not directly correspond to hardware modules. (Hardware modules correspond to BH modules, described in section 5.13.)

A BH package consists of the package header, import declarations, and top level definitions. The package header indicates which names defined in this package are exported, *i.e.,* available for import into other packages.

packageDefn ::= `package` *packageId* `(` *exportDecl* `) where {`
{ *importDecl* `;` }
{ *fixityDecl* `;` }
{ *topDefn* `;` }
`}`

exportDecl ::= *varId* | *typeId* [*conList*]

conList ::= `(..)`

importDecl ::= `import` [`qualified`] *packageId*

fixityDecl ::= *fixity integer varId*

fixity ::= `infix` | `infixl` | `infixr`

packageId ::= *conId*

Example:

```
package Foo (x, y) where

import Bar
import Glurph

... top level definition ...
... top level definition ...
... top level definition ...
```

Here, `Foo` is the name of this package, `x` and `y` are names exported from this package (they will be defined amongst the top level definitions in this package), and `Bar` and `Glurph` are the names of package being imported (for use in this package).

The export list is a list of identifiers, each optionally followed by `(..)`. Each identifier in the list will be visible outside the package. If the exported identifier is the name of `data`, `struct`, or `interface`, then the constructors or fields of the type will be visible only if `(..)` is used. Otherwise, if you export only the name of a type without the `(..)` suffix, the type is an abstract (opaque) data type outside the package. The list of identifiers may include identifiers defined in the package as well as identifiers imported from other packages.

If the keyword `qualified` is present in the import declaration all the imported entities from that package must be referred to by a qualified name.

The fixity declaration can be used to give a precedence level to a user-defined infix operator. The `infixl` specifies a left associative operator, `infixr` a right associative operator, and `infix` a non-associative operator.

3.1 Name clashes and qualified names

When used in any scope, a name must have an unambiguous meaning. If there is name clash for a name x because it is defined in the current package and/or it is available from one or more imported packages, then the ambiguity can be resolved by using a qualified name of the form $M.x$ to refer to the version of x contained in package M.

4 Top level definitions

Top level definitions can be used only on the top level within a package.

4.1 `data`

A `data` definition defines a brand new type, which is different from every primitive type and every other type defined using a `data` definition, even if they look structurally similar. The new type defined by a `data` definition is a "sum of products", or a "union of products".

topDefn	::=	`data` *typeId* { *tyVarId* } `=` { *summand* `|` } [*derive*]
summand	::=	*conId* { *type* }
summand	::=	*conId* `{` { *fieldDef* `;` } `}`
derive	::=	`deriving (` { *classId* `,` } `)`
fieldDef	::=	*fieldId* `::` *type*

The *typeId* is the name of this new type. If the *tyVarId*'s exist, they are type parameters, thereby making this new type polymorphic. In each *summand*, the *conId* is called a "constructor". You can think of them as unique *tag*'s that identify each summand. Each *conId* is followed by a specification for the fields involved in that summand (*i.e.,* the fields are the "product" within the summand). In the first way of specifying a summand, the fields are just identified by position, hence we only specify the types of the fields. In the second way of specifying a summand, the fields are named, hence we specify the field names (*fieldId*'s) and their types.

The same constructor name may occur in more than one type. The same field name can occur in more than one type. The same field name can occur in more than one summand within the same type, but the type of the field must be the same in each summand.

The optional *derive* clause is used as a shorthand to make this new type an instance of the *classId*'s, instead of using a separate, full-blown `instance` declaration. This can only be done for certain predefined *classId*'s: `Bits`, `Eq`, and `Bounded`. The compiler automatically derives the operations

corresponding to those classes (such as `pack` and `unpack` for the `Bits` class). Type classes, instances, and `deriving` are described in more detail in sections 2.1, 4.5 and 4.6.

To construct a value corresponding to some `data` definition T, one simply applies the constructor to the appropriate number of arguments (see section 5.3); the values of those arguments become the components/fields of the data structure.

To extract a component/field from such a value, one uses pattern matching (see section 6).

Example:

```
data Bool = False | True
```

This is a "trivial" case of a `data` definition. The type is not polymorphic (no type parameters); there are two summands with constructors `False` and `True`, and neither constructor has any fields. It is a 2-way sum of empty products. A value of type `Bool` is either the value `False` or the value `True` Definitions like these correspond to an "enum" definition in C.

Example:

```
data Operand = Register (Bit 5)
             | Literal  (Bit 22)
             | Indexed  (Bit 5) (Bit 5)
```

Here, the first two summands have one field each; the third has two fields. The fields are positional (no field names). The field of a `Register` value must have type Bit 5. A value of type `Operand` is either a `Register` containing a 5-bit value, or a `Literal` containing a 22-bit value, or an `Indexed` containing two 5-bit values.

Example:

```
data Maybe a = Nothing | Just a
             deriving (Eq, Bits)
```

This is a very useful and commonly used type. Consider a function that, given a key, looks up a table and returns some value associated with that key. Such a function can return either `Nothing`, if the table does not contain an entry for the given key, of `Just` v, if the table contains v associated with the key. The type is polymorphic (type parameter "`a`") because it may be used with lookup functions for integer tables, string tables, IP address tables, etc., *i.e.,* we do not want here to over-specify the type of the value v at which it may be used.

Example:

```
data Instruction = Immediate { op::Op; rs::Reg; rt::CPUReg; imm::UInt16; }
                 | Jump      { op::Op; target::UInt26; }
```

An `Instruction` is either an `Immediate` or a `Jump`. In the former case, it contains a field called `op` containing a value of type `Op`, a field called `rs` containing a value of type `Reg`, a field called `rt` containing a value of type `CPUReg`, and a field called `imm` containing a value of type `UInt16`. In the latter case, it contains a field called `op` containing a value of type `Op`, and a field called `target` containing a value of type `UInt26`.

Note

Error messages involving data type definitions sometimes show traces of how they are handled internally. Data type definitions are translated into a data type where each constructor has exactly one argument. The types above translate to:

```
data Bool = False PrimUnit | True PrimUnit

data Operand = Register (Bit 5)
             | Literal (Bit 22)
             | Indexed Operand_$Indexed
struct Operand_$Indexed = { _1 :: Reg 5; _2 :: Reg 5 }

data Maybe a = Nothing PrimUnit | Just a

data Instruction = Immediate Instruction_$Immediate
                 | Register Instruction_$Register
struct Instruction_$Immediate = { op::Op; rs::Reg; rt::CPUReg; imm::UInt16; }
struct Instruction_$Register = { op::Op; target::UInt26; }
```

End of Note

4.2 `struct`

Defines a record type (a "pure product"). This is a specialized form of a `data` definition. The same field name may occur in more than one type.

topDefn ::= `struct` *typeId* `{` *tyVarId* `}` `=` `{` `{` *fieldDef* `;` `}` `}` [*derive*]

fieldDef ::= *fieldId* `::` *type*

Example:

```
struct Proc = { pc :: Addr; rf :: RegFile; mem :: Memory }
struct Coord = { x :: Int; y :: Int }
```

Section 5.6 describes how to construct values of a `struct` type. A field of a `struct` type can be extracted either directly using "dot" notation (section 5.7) or using pattern matching (section 6.3).

4.2.1 Tuples

One way to group multiple values together is to use a `data` definition in which a constructor has multiple fields.

However, there is a built-in notation for a common form of grouping, called "tuples". To group two (or more) values together the Prelude contains a type, `PrimPair`, for which there is syntactic sugar for type expressions, value expressions, and patterns.

The type has the following definition

```
struct PrimPair a b = { fst :: a; snd :: b } deriving (Eq, Bits, Bounded)
```

For type expressions the following shorthand can be used:

`(a, b)` ≡ `PrimPair a b`

Or, more generally,

`(`t_1`, `t_2`, ⋯, `t_n`)` ≡ `PrimPair` t_1 `(PrimPair` t_2 `(⋯` t_n`))`

There is a corresponding shorthand for value expressions and patterns:

`(a, b)` ≡ `PrimPair { fst = a; snd = b }`

There is also special syntax for the empty tuple. It is written "`()`" for types, expressions, and patterns. The real type has the following definition

```
struct PrimUnit = { } deriving (Eq, Bits, Bounded)
```

4.3 type

Defines a type synonym. These are used purely for readability, *i.e.,* a type synonym can always be "expanded out" to its definition at any time.

topDefn ::= `type` *typeId* { *tyVarId* } `=` *type*

Examples:

```
type Byte     = Bit  8
type Word     = Bit 16
type LongWord = Bit 32
```

These provide commonly used names for certain bit lengths. In a specification of a processor:

```
data RegName = R0 | R1 | ... | R31
type Rdest = RegName
type Rsrc  = RegName
data ArithInstr = Add Rdest Rsrc Rsrc
                | Sub Rdest Rsrc Rsrc
```

the last two lines suggest the roles of the registers in the instructions, and is more readable than:

```
data ArithInstr = Add RegName RegName RegName
                | Sub RegName RegName RegName
```

4.4 interface

Defines an interface for a hardware module (see section 5.13). An interface is essentially a `struct`, but its components are restricted to those things that have a physical interpretation as wires in and out of a circuit. The types of fields in an interface are more likely to involve `Action`'s (see section 5.11), which are typically interpreted as "enable signals" into a circuit. The fields of an interface are also known as *methods* (not to be confused with methods of a class, described in Sections 2.1 and 4.5).

topDefn ::= `interface` *typeId* { *tyVarId* } `=` `{` { *fieldDef* `;` } `}`

Example:

```
interface Stack a =
    push :: a -> Action
    pop :: Action
    top :: Maybe a
```

This describes a circuit that implements a stack (a LIFO) of items. This polymorphic definition does not specify the type of the contents of the stack, just that they have some type "`a`". Corresponding to the `push` method, the circuit will have input wires to carry a value of type "`a`", and a "push-enable" input wire that specifies when the value present on the input wires should be pushed on the stack. Corresponding to the `pop` component, the circuit will have a "pop-enable" input wire that specifies when a value should be popped off the stack. Corresponding to the `top` component, the circuit will have a set of output wires: if the stack is empty, the wires will represent the value `Nothing`, and if the stack is non-empty and v is the value at the top of the stack, the wires will represent `Maybe` v.

4.5 `class` declarations

The general concepts behind classes, instances, overloading etc. were introduced in section 2.1. A new class is declared using the following:

topDefn ::= `class` [*context* `=>`] *classId* { *tyVarId* } [`|` *funDep*] `where {`
{ *varId* `::` *ctxType* `;` }
`}`

classId is the newly declared class. It can be polymorphic, if *tyVarId*'s exist; these are called the *parameters* of the type class. The *tyVarId*'s may themselves be constrained by *context*, in which case the classes named in *context* are called the "super-classes" of this class. The "*varId*`::`*ctxType*" list declares the class method names and their types.

Example (from the Prelude):

```
class Literal a where
    fromInteger :: Integer -> a
```

This defines the class `Literal`. It says that any type `a` in this class must have a method (a function) called `fromInteger` that converts an `Integer` value into the type `a`. In fact, this is the mechanism the BH uses to interpret literal constants, e.g., to resolve whether a literal like `6847` is to be interpreted as a signed integer, an unsigned integer, a floating point number, a bit value of 10 bits, a bit value of 8 bits, etc. (This is described in more detail in Section 5.3.)

Example (from the Prelude):

```
class (Literal a) => Arith a where
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    negate :: a -> a
    (*) :: a -> a -> a
```

This defines the class `Arith` with super-class `Literal`. It says that for any type `a` that is a member of the class `Arith`, it must also be a member of the class `Literal`, and it must have four methods with the given names and types. Said another way, an `Arith` type must have a way to convert integer literals into that type, and it must have addition, subtraction, negation and multiplication defined on it.

The optional *funDep* section specifies *functional dependencies* between the parameters of the type class:

funDep ::= { { *tyVarId* } `->` { *tyVarId* } `,` }

These declarations specify that a type parameter may be determined uniquely by certain other type parameters. For example:

```
class Add x y z | x y -> z, y z -> x, z x -> y
```

Here, the class declaration says that for any triple of types `x`, `y` and `z` that are in the class `Add`, any two of the types uniquely determines the remaining type, *i.e.,*

- `x` and `y` uniquely determine `z`,
- `y` and `z` uniquely determine `x`, and
- `z` and `z` uniquely determine `y`.

See section 8.1 for more detailed insights into the use of functional dependencies.

Note

Functional dependencies are not currently checked by the compiler.

End of Note

4.6 `instance` declarations

A type can be declared as an instance of a class in two ways. The general mechanism is the `instance` declaration; a convenient shortcut that can sometimes be used is the `deriving` mechanism.

The general `instance` declaration grammar is the following:

topDefn ::= `instance` *context* `=>` *classId* { *type* } `where`
`{` { *localDefn* `;` } `}`

This can be read as saying that the type *type* is an instance of class *classId*, provided the constraints of *context* hold, and where the *localDefn*'s specify the implementation of the methods of the class.

Sometimes, when a new type is defined using a `data` declaration, it can simultaneously be made a member of certain useful, predefined classes, allowing the compiler to choose the "obvious" implementation of the class methods. This is done using the `deriving` qualification to a `data` declaration (described in section 4.1) or to a `struct` declaration (described in section 4.2). The only classes for which `deriving` can be used for general types are `Bits`, `Eq` and `Bounded`. Furthermore, `deriving` can be used for any class if the type is a data type that is isomorphic to a type that has an instance for the derived class.

4.6.1 Deriving `Bits`

The instances derived for the `Bits` class can be described as follows:

- For a `struct` type it is simply the the concatenation of the bits for all the fields. The first field is in the leftmost (most significant) bits, and so on.
- For a `data` type, all values of the type occupy the same number of bits, regardless of which disjunct (constructor) it belongs to. This size is determined by the largest disjunct. The leftmost (most significant) bits are a code (a tag) for the constructor. As few bits as possible are used for this. The first constructor in the definition is coded 0, the next constructor is coded 1, and so on. The size of the rest of the bits is determined by the largest numbers of bits needed to encode the fields for the constructors. For each constructor, the fields are laid out left to right, and the concatenated bits are stored right justified (*i.e.,* at the least significant bits). For disjuncts that are smaller than the largest one, the bits between the constructor code and the field bits, if any, are "don't care" bits.

Examples: The type

```
data Bool = False | True
```

uses one bit. `False` is represented by 0 and `True` by 1.

```
struct Två = { första ::  Bit 8; andra ::  Bit 16 }
```

uses 24 bits with `första` in the upper 8 bits and `andra` in the lower 16.

```
data Maybe a = Nothing | Just a
```

will use $1 + n$ bits, where n bits are needed to represent values of type `a`. The extra bit will be the most significant bit and it will be 0 (followed by n unspecified bits) for `Nothing` and 1 (followed by the n bits for `a`) for `Just`.

4.6.2 Deriving `Eq`

The instances derived for the `Eq` class is the natural equality for the type. For a struct all fields have to be equal, for a data type the constructors have to be equal and then all their parts.

4.6.3 Deriving `Bounded`

An instance for `Bounded` can be derived for an enumeration type, *i.e.,* a data type where all constructors are niladic. The `minBound` will be the first constructor and the `maxBound` will be the last.

`Bounded` can also be derived for a `struct` type if all the field types of the struct are `Bounded`. The `minBound` will be the struct with all fields having their respective `minBound`, and correspondingly for `maxBound`.

4.6.4 Deriving for isomorphic types

A data type with one constructor and one argument is isomorphic to its type argument. For such a type any one-parameter class can be used, in a `deriving`, for which there is an instance for the underlying type.

Example:

```
data Apples = Apple (UInt 32) deriving (Literal, Arith)
five ::  Apples
five = 5
eatApple ::  Apples -> Apples
eatApple n = n - 1
```

4.7 Value definitions

A value definition defines the value of an identifier (which could be a function). Value definitions are the meat of a BH program.

Value definitions consist of a type signature followed immediately by one or more defining clauses:

topDefn	::=	*valueDefn*
valueDefn	::=	*varId* `::` *ctxType* `;` { *clause* `;` }
clause	::=	*varId* { *apat* } [`when` *guard*] `=` *exp*

The first line of a value definition is the type signature— it simply specifies that the identifier *varId* has the type *ctxType*. Subsequent lines define the value, one clause at a time. The *varId*'s on the left-hand side of the type signature and on the left-hand side of each clause must all be the same, *i.e.,* they collectively define a single *varId*.

Each clause defines part of the value, using pattern matching and guards. If there are patterns (*apat*'s) present, then the *varId* being defined is a function, and the patterns represent arguments to the function. The *guard* is a list of arbitrary predicates that may use identifiers bound in the patterns (see Section 7). The clause should be read as follows: if the function *varId* is applied to arguments that match the corresponding *apat*'s (in which case, identifiers in the *apat*'s are bound to the corresponding components of the arguments), and if the predicates in the *guard* are true, then the function returns the value of the expression *exp*.

Example:

```
wordSize :: Integer
wordSize = 16
```

This simply defines the identifier `wordSize` to have type `Integer` and value 16.

Example:

```
not :: Bool -> Bool
not True  = False
not False = True
```

This defines the classical Boolean negation function. The type signature specifies that `not` is a function with argument type `Bool` and result type `Bool`. After that, the first clause specifies that if the argument matches the value `True` (*i.e.,* it *is* the value `True`), then it returns `False`. The final clause specifies that if the argument is `False` it returns `True`.

Example:

```
f :: Maybe Int -> Int -> Int
f (Just x) y when x > 10, Just y' <- g y  = x + y'
f _        _                              = 0
```

(If necessary, please first remember the definition of the `Maybe` type, introduced in section 4.1). The first line specifies that `f` is a function of two arguments, of type `Maybe Int` and `Int`, respectively, and that its result has type `Int`. The second line specifies that if the first argument has the form `Just x` (in which case let us call its component `x`), if the second argument is anything (let us call it `y`), if `x`'s value is greater than 10, if the result of applying `g` to `y` has the form `Just y'` (in which case let us call the component `y'`), then the result is the value of `x + y'`. In all other cases, the result is the value 0. The bare underscores in the second line are *wild-card* patterns that match anything (described in section 6.1).

Clauses are attempted in order, from top to bottom, proceeding to the next clause only if the pattern matching and guard evaluation fail. Within each clause, pattern matching and guard evaluation are attempted from left to right. If no clause succeeds, then the system will raise a "pattern matching error".

4.8 Calling foreign functions

A function can be declared to be foreign which means that its implementation is not in BH.

topDefn ::= `foreign` *varId* `::` *type* [`=` *string*] [`,` `(` { *string* } `)`]

The optional string gives the name of the external "function" to use. If no string is given the same name as the BH name is used. The optional strings in parentheses are the port names of the Verilog module that implements the function. Without port names positional arguments will be used.

Example:

```
foreign countOnes :: Bit n -> Bit 32 = "pop_count"
```

A call to `countOnes` will instantiate the Verilog `pop_count` module. It should have the same number of arguments (with the same type) as the BH function, *and* an additional trailing argument which is the result. If the function is (size) polymorphic the instantiated types will be used as Verilog parameters.

Example: using the declaration above an action, with the type of `x` being `Bit 5`,

```
y := countOnes x
```

will translate to something like

```
pop_count #(5) ires1(R_x, I_y);
```

5 Expressions

As described in 4.7, expressions appear in the right-hand sides of value definitions.

In the following *exp* stands for an arbitrary expression and *aexp* for an atomic expression, *i.e.,* one that is syntactically delimited.

5.1 Applications

Function application (a.k.a. a function call) is expressed just by the juxtaposition of two expressions. The first expression should evaluate to a function value, and that function is applied to the value of the second expression.

exp ::= *exp aexp*

Parentheses can be used freely for grouping. By default, if parentheses are omitted, function application associates to the left:

```
f  x  y  z        ≡        ((f  x)  y)  z
```

i.e., `f` is applied to `x`, producing a function which is applied to `y` which produces a function which, in turn, is applied to `z`.

Infix applications

Infix operators (binary functions written between their arguments) can be used for convenience.

exp ::= exp_1 *binop* exp_2
binop ::= *symbol* { *symbol* }
symbol ::= ! | # | $ | % | & | * | + | . | / | < | = | > | ? | | | \ | ^ | | | - | ~ |
¡ | ■ | ■ | ¨ | © | ª | ■ | ¬ | ® | ¯ | ° | ± |
² | ³ | ´ | µ | ¶ | · | ¸ | ¹ | º | ■ | ¼ | ½ | ¾ | ¿ | × | ÷

The following table lists the predefined operators with their precedence and associativity (see the Standard Prelude in section D for an explanation of what these operators do):

operator	precedence	associativity
`$`	0	Right
`:=`	1	Right
`||`	2	Right
`&&`	3	Right
`|`	4	Right
`&`	5	Right
`==`	6	n/a
`/=`	6	n/a
`<=`	6	n/a
`>=`	6	n/a
`<`	6	n/a
`>`	6	n/a
`<<`	7	n/a
`>>`	7	n/a
`++`	8	Right
`:>`	8	Right
`+`	10	Left
`-`	10	Left
`*`	11	Left
`/`	11	Left
·	13	Right
user-defined	15	Left

The last line indicates that any user-defined infix operator has higher precedence than any predefined operator, and it always associates to the left. Function application by juxtaposition always has higher precedence than all operators, and associates to the left. Constructs that do not have any closing lexeme (such as `if-then-else` or `let-in`) have lowest precedence so that, for example,

```
if ... then ... else e1 + e2
```

parenthesizes as follows:

```
if ... then ... else (e1 + e2)
```

and not as follows:

```
(if ... then ... else e1) + e2
```

The user can define new infix operators by following the above syntax. For example, here is a new infix operator `|-|` that "clips" a value to a [−limit, +limit] range:

```
|-| :: Int -> Int ->Int

x  |-|  lim when x < 0-lim = 0-lim
x  |-|  lim when x >   lim =   lim
x  |-|  _                  =   x
```

An infix operator can be converted into an ordinary identifier (eliminating its special syntactic role) by enclosing it in parentheses. Conversely, an ordinary identifier representing a binary function can be used in infix position by enclosing it in back-quotes.

Examples:

```
f x y
x + y
f x `max` g y
(+) 1 2
```

5.2 Variables

A variable in an expression simply represents its value.

aexp ::= *varId*

Remember that variable names start with a lower case letter.

5.3 Constructors and literal constants

Each value constructor, introduced in some `data` declaration (section 4.1), is a constant.

aexp ::= *conId*

Suppose a constructor `C` is declared as follows:

```
data T a b = ... | C t0 t1 t2 | ...
```

Then, the constructor identifier `C` is a constant whose value is a function of type:

```
C :: t0 -> t1 -> t2 -> T a b
```

If `C` had no parameters, then it represents a traditional (non-functional) constant value.

Remember that constructor names start with an upper case letter.

Literal constants are constants supported with special syntax and with overloading support. BH has support for integer and string literals.

5.3.1 Integer literals

Integer literals are written in the usual way as a sequence of decimal digits (0-9), or `0x` followed by a sequence of hexadecimal digits (0-9, a-f, A-F), or `0b` followed by a sequence of binary digits (0,1).

There is no direct notation for negative integer literals— use the expression `(negate` <literal>`)` instead.

Note

Constants with base 2 must have a type of the form $t\ n$ where t is a type like `Bit` or `UInt` and n is the number of digits in the literal.

End of Note

aexp ::= *int*

Example:

```
125
0x48454a
0b101010
```

Since BH has several integer-like numeric types (of various bit widths), a numeric literal i is really shorthand for `fromInteger(`i`)`, where i is treated as belonging to `Integer`, the type of (arbitrary precision) integers. The `fromInteger` method belongs to the class `Literal`:

```
class Literal a where
    fromInteger :: Integer -> a
```

The normal overloading-resolution mechanism (see section 2.1) is used by the compiler to figure out what type the literal should be converted into. As usual, if necessary you can insert a type signature to help the compiler resolve this.

5.3.2 String literals

String literals are written enclosed in double quotes `"`. Special characters may be inserted in string literals with the the following backslash notations:

`\n`	newline
`\t`	tab
`\\`	backslash
`\"`	double quote
`\x`HH	any hexadecimal character code HH

5.4 `case` and `if`

Case expressions can be used to scrutinize values of a data type— to discover which disjunct it conforms to, and to bind names to the components of that disjunct.

exp ::= `case` *exp* `of {`
`{` *caseArm* `}`
`}`

caseArm ::= *pat* [`when` *guard*] `->` *exp* `;`

The value of the expression (first *exp*) is tested against the patterns and guards of each case arm in succession, top to bottom. At each case arm, the value is matched against the pattern; if it succeeds, the pattern identifiers are bound to the respective components, and the guard expressions are evaluated, left to right. If they are all true, then the case arm is successful, and the value of the right-hand side expression is returned as the value of the entire `case` expression. If none of the case arms succeed, the result is unspecified.

Example (uses the `Maybe` type definition of section 4.1):

```
case f a of {
    Just x when x  < 0 -> negate x;
    Just x when x >= 0 -> x;
    Nothing            -> 0;
}
```

First, the value of `f a` is obtained. In the first arm, the value is checked to see if it has the form `Just x`, in which case we let `x` refer to the component. Then, we check if this `x` is less than zero. If so, then the case arm succeeds and we return `negate x` as the value. Otherwise, we fall through to the second case arm, and so on.

Booleans can be tested with an `if` expression (also known as conditional expressions):

exp ::= `if` exp_1 `then` exp_2 `else` exp_3

This is just a convenient and familiar shorthand for:

```
case exp1 of {
True  -> exp2;
False -> exp3;
}
```

5.5 let

Local definitions are introduced by the `let` expression. The definitions look like function definitions on the top level, but the type signature is optional.

exp	::=	`let` { *letDefn* } `in` *exp*
letDefn	::=	*localDefn* | *patternBind* | *shortDefn*
localDefn	::=	[*varId* `::` *ctxType* `;`] { *clause* }
patternBind	::=	*pat* `=` *exp*
shortDefn	::=	*varId* `::` *ctxType* `=` *exp*

Example:

```
let   x2 = x * x
      y2 = y * y
in
      x2 + x2 - y2
```

A `let` definition can bind a single identifier, but it can also bind one or more identifier through a pattern binding. Pattern bindings are only allowed for patterns that cannot fail.

Example:

```
let (x, y) = foo z
in  x + y
```

which is equivalent to saying

```
let x = (foo z).fst
    y = (foo z).snd
in  x + y
```

Note

Currently only `struct` patterns are allowed in pattern bindings.

End of Note

5.6 Structs and Tuples

Section 4.2 describes how to define a `struct` type. To produce a value of such a type, we write the type name followed by values for the fields.

aexp	::=	*typeId* `{` { *field* } `}`
field	::=	*fieldId* `=` *exp* `;`

Example:

```
Proc { pc = 0; cc = EQ }
```

Section 4.2.1 describes the `PrimPair` struct type. This "tuple type" may be expressed using the special syntactic shorthand involving parentheses and commas. Similarly, this notation can be used for value expressions as well, *i.e.,* the expression on the left is a shorthand for for the expression on the right:

```
(a, b)        ≡        PrimPair { fst = a; snd = b }
```

5.7 Struct field selection

A field of a struct value can be selected with dot notation.

exp ::= *exp* `.` *fieldId*

Example

```
r.pc
```

Note

Since the same field-name can occur in multiple types, the compiler uses type information to resolve which field-name you mean when you do a field selection. Occasionally, you may need to add a type signature to help the compiler resolve this.

End of Note

There is a shorthand for making a field selection into a function:

aexp ::= `(` `.` *fieldId* `)`

The expression "`(.name)`" is equivalent to "`\x -> x.name`".

5.8 Struct "update"

A struct value can be constructed from another struct value by changing some of the fields.

exp ::= *aexp* `{` { *field* } `}`

Example

```
s { x = 77; y = 88 }
```

Here "`s`" is an expression of `struct` type. The entire expression has the same type as "`s`" and all fields expect "`x`" and "`y`" also have the same values.

5.9 `interface` expressions

An interface expression defines a value of interface type.

exp ::= `interface` *typeId* `{` { *ifcDefn* `;` } `}`

ifcDefn ::= *localDefn* [`when` *guard*]

Example:

```
interface Stack
    push x = r := Just x    when r == Nothing
    pop    = r := Nothing   when r /= Nothing
    top    = r._read
```

The `when` clause in a method specifies the condition, called the *implicit condition*, which must hold when this method is called. The compiler will make a scheduler that fulfills the condition. Using implicit conditions, it is possible to write client code that is not cluttered with conditionals that test whether the method is applicable. For example, a client of a FIFO module can just call the "enqueue" or the "dequeue" method without having explicitly to test whether the FIFO is full or empty, respectively; those predicates are usually specified in implicit conditions inside the FIFO module interface definition itself.

The `when` clause can refer to any variables from the surrounding scope. In particular, note that the interface method arguments are *not* available in the `when` clause.

Note

There are good implementation (hardware) reasons for not allowing the interface arguments to be used in the implicit condition.

End of Note

5.10 "Don't care" expressions

When the value of an expression does not matter a "don't care" expression can be used. It is written as an underscore and has any type. The compiler will pick some suitable value.

If a "don't care" value is part of any computation (such as an argument to an addition function) the result will be a new "don't care" value.

aexp ::= `_`

Note that this is a distinct (but related) use of the underscore from its use as a "don't care" pattern (section 6.1).

The programmer is encouraged to use "don't care" values where possible, both because it is useful documentation and because the compiler can often exploit this to produce better circuits.

5.11 Actions

Any expression which is intended to act on state is called an *action* and has type `Action`. Primitive actions are provided as fields of the interfaces to objects provided by the compiler (such as registers or arrays). The programmer can create new actions only by building on these primitives, or by using Verilog modules.

Example:

```
interface Reg a =
    _write :: a -> Action
    _read  :: a
```

Actions are combined by the keyword `action` followed by a sequence of actions.

exp ::= `action {` { *stmt* `;` } `}`

Example:

```
action { x := x+1; y := z }
```

The Standard Prelude defines the "empty" action:

```
noAction :: Action
```

which is equivalent to the expression: `action {}`.

The `Action` type is actually a special case of the more general type `ActionValue`, described in the next section:

```
type Action = ActionValue ()
```

5.11.1 `ActionValue`

The `ActionValue` is an abstract type:

```
interface ActionValue a
instance Monad ActionValue
```

Values of `ActionValue` type should be thought of as performing an action as well as returning a value.

The `ActionValue` type is a monad and the `action` syntax allows variable bindings for the value as well as performing actions with no return value.

Example:

```
interface IntStack =
    push :: Int -> Action
    pop  :: ActionValue Int




...
    s1 :: IntStack
...
    s2 :: IntStack
...
    action
        x <- s1.pop        -- A
        s2.push (x+1)      -- B
```

In line A, we perform a `pop` action on stack `s1`, and the returned value is bound to `x`. If we were not interested in the returned value, we could have omitted the "`x <-`" part. In line B, we perform a `push` action on `s2`, and the returned value `()` is discarded (not bound to anything).

5.12 `rules`

The `rules` expression introduces rewrite rules for the Term Rewriting System. It appears inside a module and specifies part of the behavior of the module. A `rules` expression has type `Rules` and consists of a list of rewrite rules.

Each rewrite rule has a left hand side and a right hand side. The left hand side is a guard; the rule only applies if the guard is valid. The right hand side of a rule is an *action*. Commonly, it is a composite of many actions to be performed when the rule is applied. An entire rule may optionally be prefixed with a *rule label* which is useful primarily in debugging, *i.e.,* when simulating/executing the hardware description produced by the BH compiler, the execution engine may be able to inform you about when a particular rule fires using these rule labels. A rule label is a string valued expression.

exp ::= `rules { {` *rule* `; } }`

rule ::= [*ruleLabel* `:`] `when` *guard* `==>` *exp*

ruleLabel ::= *aexp*

Example:

```
let
  instr :: Word
  instr = mem[pc]
in
  rules
    when Add r1 r2 r3 <- instr
    ==> action {
          pc := pc+1;
          rf[r1] := rf[r2] + rf[r3]
        }

    when Jz r1 r2 <- instr, rf[r1] == 0
    ==> action { pc := r2 }
```

5.12.1 Nested rule guards

Sometimes in a series of rules, each rule may have a conjunction of conditions, and all the rules share one of those conditions. In such a situation, it is useful to be able to factor out the common condition, and this can be done by nesting `when` clauses.

rule ::= [*ruleLabel* `:`] `when` *guard* `rules { {` *rule* `; } }`

Example,

```
rules
    when c, c1 ==> action1
    when c, c2 ==> action2
    when c, c3 ==> action3
    when d     ==> action4
```

can be written more clearly as:

```
rules
    when c
    rules
        when  c1 ==> action1
        when  c2 ==> action2
        when  c3 ==> action3

    when d  ==> action4
```

5.12.2 Aggregating and prioritizing rules

Rules are first class objects. The following operators allow rule sets to be combined:

```
(<+>) :: Rules -> Rules -> Rules
(<+)  :: Rules -> Rules -> Rules
(+>)  :: Rules -> Rules -> Rules
```

The "`<+>`" operator makes a symmetric union of two rule sets. The "`<+`" operator makes a directed union, *i.e.,* the rules on the right may fire only when none of the rules on the left are enabled. The "`+>`" operator makes a directed union in which rules on the left may fire only when none of the rules on the right are enabled.

5.13 Modules

Modules are the heart of BH. Modules turn into actual hardware, and correspond roughly to Verilog modules. State can exist only inside a module. Modules also incorporate the rules which act on their state.

A module consists of three things: state, rules on that state, and an interface to the outside world. This information is given in a `module` expression which has type "`Module a`",[1] where `a` is the type of the interface.

There is a strong analogy between BH modules and *objects* in object-oriented programming languages, particularly objects that represent processes. A `module` expression of type "`Module a`" defines an object constructor, *i.e.,* something that allocates and initializes an object. The constructor returns an object reference of type "`a`", *i.e.,* a handle on which you can call the interface methods. Each invocation of the constructor produces a new object, and returns the handle to that new object, so it is easy to make multiple copies of an object. The state elements of the object correspond to private variables inside the object. They cannot directly be manipulated or accessed by any other object; this can only be done *via* interface methods of the object. The rules in an object specify the internal, free-running behavior of the object, *i.e.,* the "process" that the object represents.

Here is the grammar for `module` expressions:

exp	::=	`module {` { *mstmt* `;` } `}`
mstmt	::=	*stmt* | *mrules* | *minterface*
stmt	::=	*varId* `::` *ctxType* `;` *varId* `<-` *exp*
	|	*pat* `::` *ctxType* `<-` *exp*
	|	`let {` *letDefn* `}`
mrules	::=	`rules {` { *rule* `;` } `}`
minterface	::=	`interface {` { *fieldDef* `;` } `}`
	|	`interface (` { *exp* `,` } `)`

The state-creation statements look like this: x `::` t `;` x `<-` e. An equivalent way of writing this is x `::` t `<-` e. The first part is a type signature, as usual. The second part is called a *monadic* binding. The right hand side expression e allocates some state, which is just another module, either a module defined elsewhere or a primitive module like a register, array, or FIFO. The right-hand side e also returns a value, which is bound to the left-hand side identifier x. Thus, the right-hand side e must have type `Module` t, and x will be bound to a value of type t, *i.e.,* to the interface of the module.

[1] Actually, the type is more general, see A.3.

If you do not want to use the value returned by *exp*, you can use *exp* as a statement by itself (no need for the "*varId* <-" part).

Statements can also be `let` statements, which are used for ordinary value bindings (the left-hand side identifier and the right-hand side expression can be of any type).

Note

The advanced user will recognize that module statements are similar to the body of a do-statement (see section A.2). Modules are monads. Thus, an expression which is bound to a variable of type a by means of the <- syntax must have type `(Module a)`. The module constructed by the expression will have its own internal state and rules, which will be incorporated into the rules and state of the containing module, although they will be hidden behind an abstraction barrier. Only the interface of the module is accessible and it is that which is bound to the variable.
While the body of a do-statement has a final expression which provides the value for the whole expression, the `rules` and `interface` section of a module form an implicit final expression which adds the rules to the monad and returns the interface.

End of Note

Example: A register is primitive module whose interfaces is defined as follows:

```
interface Reg a =
    set :: a -> Action
    get :: a
```

and with the following module constructor function:

```
-- takes an initial value for the register
mkReg :: (Bits a sa) => a -> Module (Reg a)
```

A module built on these primitives would look like:

```
interface ArithIO a =
    input :: a -> a -> Action
    output :: a

mkGCD :: Module (ArithIO (Bit 32))
mkGCD = module
             x :: Reg (Bit 32)
             x <- mkReg _

             y :: Reg (Bit 32)
             y <- mkReg _

             done :: Reg Bool
             done <- mkReg True
             interface
                 input a b = action { x._write a; y._write b; done._write False }
                      when done._read
                 output = x._read
                      when done._read
             rules
                 when not done._read, x._read > y._read, y._read /= 0
                   ==> action { x._write y; y._write x }
```

```
when not done._read, y._read == 0
  ==> action { done._write True }

when not done._read, x._read <= y._read, y._read /= 0
  ==> action { y._write (y._read - x._read) }
```

Note how the two methods in the interface can only be applied when the computation is done, ensuring that the result is not read too early and that new arguments do not overwrite an ongoing computation.

Because registers are the most common state elements, a special notation is available to relieve the programmer from having to type `._write` and `._read` everywhere. This is described in Section 8.5.

6 Patterns

Patterns are used in value definitions (section 4.7), in `case` expressions (section 5.4), in λ-expressions (section A.1) and in guards (section 7). A pattern is always *matched* against an actual value and, in the process, it plays two roles:

- First, a pattern acts as a Boolean filter— it succeeds only if the actual value against which it is matched has the same form as the pattern, *i.e.,* (a) the value is built out of the same constructor, and (b), the corresponding components of the constructor in the pattern and actual value also match.
- Second, assuming the pattern does match, then the variables in the pattern are bound to the corresponding components in the actual value.

Thus, a pattern is used both as a predicate ("does it match?") and as a binding mechanism to name components of an actual value.

The variables used in a pattern may not be repeated, *i.e.,* any given variable can occur at most once in a pattern.

6.1 Variable and wild-card patterns

pat ::= *varId* | _

A variable or a wild-card (bare underscore) is a trivial pattern that matches any actual value. If it is a variable, it also binds the variable name to that value.

(Of course, when we say *any actual value* here, we mean any value that could possibly be supplied for matching here. Static type checking will ensure that the only actual values supplied here will have the correct type.)

Note that this use of an underscore is distinct (but related to) its use as a "don't care" expression (section 5.10).

6.2 Constructor and constant patterns

pat ::= *conId* { *apat* }

In a constructor pattern, there must always be as many *apat* arguments as the number of arguments in the constructor *conId*'s original declaration.

Such a pattern matches an actual value that is constructed out of the same constructor, and where (recursively) each *apat* argument matches its corresponding component in the actual value. The variable bindings produced by the match is the union of the variable bindings of the individual *apat* matches.

6.3 Struct and Tuple patterns

Struct patterns are used to match struct values. A struct pattern has a number of field patterns. Not all fields in a struct need be present in the field patterns (if a field patterns is missing, the corresponding component in the actual value is is ignored). A field pattern can be abbreviated, or *punned*, if the bound variable has the same name as the field.

apat	::=	*typeId* `{` { *fieldPat* `;` } `}`
fieldPat	::=	*fieldId* `=` *pat* | *fieldId*

Example (see section 4.2 for corresponding type definition):

```
Proc { pc = pc; rf = regfile }
```

This matches any value constructed using the `Proc` constructor, and binds the identifiers `pc` and `regfile` to the `pc` and `rf` field values, respectively. The `mem` field is ignored.

Note that in the phrase `pc = pc`, the left-hand occurrence of `pc` is the fieldname, whereas the right-hand occurrence is a variable that happens to be spelt the same. Using the "punning" abbreviation described above, we could also write this as:

```
Proc { pc; rf = regfile }
```

Section 4.2.1 describes the `PrimPair` struct type, and section 5.6 describes corresponding value expressions. In both cases, these "tuple" types and objects may be expressed using the special syntactic shorthand involving parentheses and commas. Similarly, this notation can be used in pattern matching as well, *i.e.,* the pattern on the left is a syntactic shorthand for the pattern on the right:

```
(a, b)        ≡        PrimPair { fst = a; snd = b }
```

7 Guards

Guards are used as extra conditions in a pattern match to limit when a certain case should be used. They are used in value definitions (section 4.7), in `case` expressions (section 5.4), and in rules (section 5.12).

A pattern together with a guard is considered to match only if both the pattern and the guard match. A guard consists of a list of zero or more parts. A guard matches if all its parts match. The parts are tested from left to right. Identifiers bound in one part may be used in subsequent parts to its right.

guard	::=	{ *qual* `,` }
qual	::=	*exp* | *pat* `<-` *exp*

A Boolean guard *exp* is an arbitrary expression. It is considered to match if the expression evaluates to `True`.

A pattern guard *pat* `<-` *exp* has a pattern and an expression. It is considered to match if the pattern matches the value of the expression. The variables in the pattern get bound to the corresponding components.

8 Important Primitives

These primitives are available *via* the standard prelude and other standard libraries. See also Appendix D for more useful libraries.

8.1 The "size" types

As described in section 2, there is a collection of types representing "sizes" that are written as numbers. Typically, the only place these types are/can be used are as arguments to other parameterized types. For example, the type:

```
Bit 16
```

consists of the unary type constructor `Bit` applied to the type "`16`". The type as a whole represents bit vectors of length 16 bits.

Collections of size types are also instances of certain predefined classes that can be used to express size constraints:

```
class Add x y z | x y -> z, y z -> x, z x -> y

class Max x y z | x y -> z

class Log x y | x -> y, y -> x
```

The `Add` class has instances for all size types x, y, and z such that $x + y = z$. The `Max` class has instances for all size types x, y, and z such that $max(x, y) = z$. The `Log` class has instances for all size types x and y such that $ceil(log_2 x) = y$. These functional dependencies enable the type checker to do some limited forms of arithmetic.

Example:

```
pad0101 :: (Add n 4 m) => Bit n -> Bit m
pad0101 x = x ++ 0b0101
```

The second line defines the function `pad0101` as taking a bit vector and padding it to the right with the bits "0101" using the bit-concatenation operator "`++`". The type signature on the first line expresses the idea that the function takes a bit vector of length n and returns a bit vector of length m, where $n + 4 = m$.

To get the value that corresponds to a size there is a special "function", `valueOf`, that takes a size type and gives the corresponding `Integer` value.

```
type Five = 5
x :: Integer
x = valueOf Five  -- x will have the value 5
```

In the first line, the symbol "5" represents the size type "5", not the integer value 5. The type synonym is there just for readability. In the last line, `x` gets the corresponding integer value of 5.

In a pinch, this mechanism can be used to do arithmetic for you!

```
type WordSize = 32

logW :: (Log WordSize k) => Integer
logW = valueOf k
```

The type synonym is there just for readability. The type signature says that `logW` has an integer value, provided that `WordSize` and `k` are instances of the `Log` class, *i.e.,* provided the values corresponding to `WordSize` and `k` are in the logarithm relation. Then, the last line binds `logW` to the integer value corresponding to `k`

```
logWPlusOne :: (Log WordSize m, Add m 1 n) => Integer
logWPlusOne = valueOf n
```

The type signature establishes that (the integers corresponding to) `WordSize` and `m` are in the log relation; that (the integers corresponding to) `m` and `1` and `n` are in the addition relation, and that `logWPlusOne` has Integer type. The second line binds `logWPlusOne` to the integer value corresponding to `n`.

A "function" `stringOf` is also available to get the string value of a type-level string.

```
type MyString = "Hello, world!"
x :: String
x = stringOf MyString  -- x will have the value "Hello, world!"
```

8.2 The type `Bit`

A very important built-in unary type constructor is "`Bit`". It represents bit vectors of a certain size.

Example:

```
zero :: Bit 16
zero = 0

type BurroughsWord = Bit 51
```

To extract a sub-vector from a bit-vector there is a special notation taken from Verilog.

exp ::= *exp*`[`*exp*`:`*exp*`]`

The expression `e[h:l]` extracts bits from `l` (low index) to `h` (high index) inclusively.

Note

The type system is not powerful enough to express the exact type of bit extraction, so the extracted bit field can be used as a bit vector of any width. To adjust it to the right size, it is either truncated from the left or extended with zeros to the left, as necessary (most significant bit side).

End of Note

To concatenate bit vectors the `++` operator can be used. The type of this operator expresses its type exactly.

```
(++) :: (Add m n mn) => Bit m -> Bit n -> Bit mn
```

There is also a function to split bit fields

```
split :: (Add m n mn) => Bit mn -> (Bit m, Bit n)
```

8.3 The `Bits` class

The type class `Bits` contains the types that are convertible to bit strings of a certain size. For a type to be an instance of this class is a prerequisite for a number of things, such as putting it in a register, array, or fifo.

```
class Bits a n | a -> n where
    pack   :: a -> Bit n
    unpack :: Bit n -> a
```

Here, "a" represents the type that can be converted to/from bits, and "n" is always instantiated by a size type representing the number of bits needed.

The most trivial instance declaration is that a bit vector can be converted to a bit vector:

```
instance Bits (Bit k) k where
    pack x = x
    unpack x = x
```

Another example:

```
data Color = Red | Green | Blue
instance Bits Color 2 where
    pack Red    = 0b00
    pack Green  = 0b01
    pack Blue   = 0b10
    unpack 0b00 = Red
    unpack 0b01 = Green
    unpack 0b10 = Blue
```

Instances of the `Bits` class can be derived by the compiler by using the `deriving` directive. Example:

```
struct Coord = { x :: Int; y :: Int } deriving(Bits)
```

This defines a new struct type `Coord` with two `Int` fields. The `deriving` clause registers `Coord` as an instance of the `Bits` class and automatically produces the required class methods `pack` and `unpack` to convert from `Coord`'s to bit vectors and vice versa (the mapping algorithm is described in more detail in Section 4.6.1).

There is a type "function," `SizeOf`, that can be applied to a type to get its corresponding bit size.

8.4 UInt, Int

`UInt` n and `Int` n define an unsigned and a signed integer data type, respectively, of n bits.

These types are instances of the classes `Bits`, `Literal`, `Eq`, `Arith`, `Ord`, `Bounded`, and `Bitwise` (see Appendix D for the operations that come with these classes).

Note

The `UInt` and `Int` types are not really primitive; they are defined completely in BH.

8.5 Registers

The most elementary form of state available in BH is the register. Registers can be created with the Verilog function `mkReg` which is a module with interface `Reg`. The argument to `mkReg` is the initial value of the register. A function `mkRegU` exists which creates a register whose initial value we don't care about.

```
interface Reg a =
    _write :: a -> Action
    _read  :: a

mkReg :: (Bits a sa) => a -> Module (Reg a)

mkRegU :: (Bits a sa) => Module (Reg a)
```

With this interface, it is necessary to use the functions `_read` and `_write` to retrieve and set values in a register. To save the programmer some keystrokes and to improve readability of programs, mechanisms have been introduced to allow the functions to be dropped. First, the `._read` can be dropped from most variable names and the compiler will add it implicitly if it is needed. The compiler will not be able to add `._read` to expressions, only to identifiers. In some cases, the compiler might accidentally insert `._read` where the programmer really intended to refer to the register and not its contents. If this happens, simply apply the function `asReg` to the variable name, thereby turning it into an expression, and the compiler will not insert the `._read`.

```
asReg :: Reg a -> Reg a
asReg r = r
```

The symbol `:=` can be used as syntactic shorthand for setting a register:

exp ::= *exp* `:=` *exp*

Example:

`pc := pc + 1` *is shorthand for* `pc._write (pc._read + 1)`

8.6 FIFOs

Package `FIFO` defines several useful interfaces and modules for FIFOs.

```
interface FIFO a =
    enq      :: a -> Action
    deq      :: Action
    first    :: a
    clear    :: Action

-- Make a FIFO
mkFIFO :: (Bits a as) => Module (FIFO a)
mkSizedFIFO :: (Bits a as) => Integer -> Module (FIFO a)
```

The constructor `mkFIFO` leaves the capacity of the FIFO unspecified (the number of entries in the FIFO before it becomes full). The constructor `mkSizedFIFO` takes the desired capacity of the FIFO as an argument.

8.7 FIFOFs

Package `FIFOF` defines several useful interfaces and modules for FIFOs. The `FIFOF` interface is like a `FIFO`, but it also has methods to test if the FIFO is full or empty.

```
interface FIFOF a =
    enq      :: a -> Action
    deq      :: Action
    first    :: a
    clear    :: Action
    notFull  :: Bool
    notEmpty :: Bool

-- Make a FIFOF
mkFIFOF :: (Bits a as) => Module (FIFOF a)
mkSizedFIFOF :: (Bits a as) => Integer -> Module (FIFOF a)
```

The constructor `mkFIFOF` leaves the capacity of the FIFO unspecified (the number of entries in the FIFO before it becomes full). The constructor `mkSizedFIFOF` takes the desired capacity of the FIFO as an argument.

9 Interfacing to Verilog

BH programs can include components that are written in Verilog, and BH also generates Verilog, so there are two (related) mechanisms to consider.

9.1 Verilog modules

Modules written in Verilog are an important part of BH since this is where state elements are defined. A Verilog module definition specifies the naming of all the signals that are needed to implement the interface methods as well as some standard signals that all modules have. Verilog modules have the following syntax.

exp ::= `module verilog` *vModName* [*vParams*] *clkNames rstNames* [*vArgs*]
`{` { *fieldId* [*mult*] `=` *portSpec* { *portSpec* } `;` } `}`
[*schInfo*]

vModName ::= *aexp*
clkNames ::= { *portName* `,` }
rstNames ::= { *portName* `,` }
portSpec ::= *portName*[{ *portProp* `,` }]
portName ::= *string*
portProp ::= `reg` |`const` |`unused` |`inhigh`
vParams ::= `(` { *exp* `,` } `)` `,`
vArgs ::= `(` { `(` *portName* `,` *exp* `)` } `)`
mult ::= `[` *int* `]`

schInfo ::= `[` { *schMeths schOp schMeths* `,` } `]`
schMeths ::= *fieldId*
| `[` { *fieldId* `,` } `]`
schOp ::= `<>` |`<` |`<<`

A Verilog module has many parts because a lot of information needs to be conveyed to the BH compiler. Many of the parts are optional, so most definitions look less formidable than the grammar suggests.

A basic Verilog module definition gives the name of the Verilog module (*vModName*), the name of the clock signal (*clkName*) and then a number of definitions of the methods of the interface.

Example:

```
interface Counter =
    up     :: PrimAction
    preset :: Bit 4 -> PrimAction
    value  :: Bit 4

vCount :: Module Counter
vCount = module verilog "count4" "clk" {
    up     = "enable";
    preset = "inp" "set";
    value  = "outp";
}
```

The name of the Verilog module is `count4` and it is clocked by the port `clk`. It has three input ports: `enable`, `inp`, and `set`, and one output port: `outp`.

Beware, the compiler has no way of checking that the definition of a Verilog module really corresponds to what the Verilog code actually does so it will just believe you.

The names of the Verilog ports (the quoted names) do not have to be unique in a Verilog module description. If the same port name is used more than once the compiler will assume that the methods in which the names occur share a port and it will insert a multiplexer accordingly.

Following a port name there can be port a property, which is one of the following:

`reg` specifies that the port is directly connected to the input or output of a register. This property is informational only and propagated by the compiler to the generated top level module.
`const` *not used at the moment*
`ununsed` *not used at the moment*
`inhigh` specifies that this enable signal (which is the only place where it is allowed) is always high, i.e., the method executes on every clock cycle. There will be no Verilog port corresponding to this enable signal.

Example:

```
interface VSyncSRAM adrs dtas =
    exec  :: Bit adrs -> Bit dtas -> Bit 1 -> Bit 1 -> PrimAction
    rdata :: Bit dtas

mkSPSRAM_V :: Integer -> Module (VSyncSRAM adrs dtas)
mkSPSRAM_V nwords = do
    module verilog name "CLK" {
        exec  = "ADR" "DI" "WE" "EN" "?"{inhigh};
        rdata = "DO";
    } [ [exec,rdata] <> [exec,rdata] ]
```

Since there will be no wire for the `exec` enable signal the name does not matter, as long as it does not conflict with any other port names. All port names must be unique including the inhigh placeholders.

9.1.1 Method definitions

Each method definition consists of a number of port names. There must be one name for each part of the type of the interface. A type of a method defined in a Verilog interface must be of the form

t_1 `->` t_2 $\cdots$ `->` t_n

where each of the t_i must be of type `Bit` n, except for the final type, t_n, which can also be of type `PrimAction`. If the last type is `PrimAction` all the ports will be input ports and the last port is the enable signal for the action. If the last type is not `PrimAction` the last port will be an output port and the others will be inputs.

9.1.2 Parameters and arguments

A Verilog module may require parameters to be instantiated. Parameters must be compile time constants and to ensure this they must be of type `Integer` in BH. The parameters are given right after module name.

```
module verilog "foo" (2,16)  ···
```

Additional arguments (ports) can be given values by the argument part of the definition. The arguments are given by specifying a Verilog port name and the value that should be on this port. A typical use for this feature is to provide an initial value for a state element that is to be used when the reset signal is asserted. A expression used as an argument to a Verilog module cannot have an implicit condition (the compiler checks this).

Example, (part of) the definition of the `mkReg` module:

```
interface VReg n =
    set :: Bit n -> PrimAction
    get :: Bit n

vMkReg :: Bit n -> Module (VReg n)
vMkReg v =
    module verilog "RegN" (valueOf n) "CLK" "RST" (("init", v)) {
        get = "get"{reg};
        set = "val"{reg} "SET";
    } [ get <> get, get < set, set < set ]
```

The Verilog module is named `RegN` and is given one parameter, namely the size of the register to create. In addition it is passed an additional value `v` on the `init` port which is used to set the initial value when the reset signal (`RST`) is asserted.

The definition of `mkReg` is completed by wrapping the `vMkReg` module in some packing and unpacking.

```
mkReg :: (Bits a sa) => a -> Module (Reg a)
mkReg v =
    module
        r :: VReg sa
        r <- vMkReg (pack v)
        interface
            _read    = unpack r.get
            _write x = fromPrimAction (r.set (pack x))
```

The following Verilog code is one possible implementation of `RegN`:

```
module RegN(CLK, RSTN, init, get, val, SET);
  parameter width = 1;
  input CLK;
  input RSTN;
  input [width - 1 : 0] init;
  input SET;
  input [width - 1 : 0] val;
  output [width - 1 : 0] get;
  reg [width - 1 : 0] get;
  always@(posedge CLK or negedge RSTN) begin
    if (!RSTN)
      get <= init;
    else if (SET)
      get <= val;
  end
endmodule
```

9.1.3 Scheduling information

The scheduling information is used to describe what operations can be performed at the same time. Currently, three relations can be described: Conflict Free (<>), Sequentially Composable (<), and Restricted Sequentially Composable (<<) (which means sequentially composable, but not parallelly composable). These relations are simply given by enumerating the elements of the set (of method name pairs) that make up the relation. A shorthand is provided for generating sets where the left or right component is the same.

In the absence of scheduling information both relations are considered to be empty, which is always a safe approximation.

9.1.4 Multiple methods

For some Verilog modules several ports with identical operation may be available. An example is a multiported memory where there are several read ports available that can be used simultaneously. This can, of course, be described by an interface that has several similar methods and the use of these can then be determined by the BH code. But the Verilog module definitions also offers a more convenient alternative; a port multiplicity can be specified. This is done by a "`[n]`" following the field name. This informs the compiler how many similar ports are available and the compiler will make sure to use them appropriately.

Example:

```
interface SRAM =
    rd :: Addr -> Data
    wr :: Addr -> Data -> PrimAction

mkSRAM :: Module SRAM
mkSRAM = module verilog "SRAM" "CLK" {
    rd[3] = "raddr" "rdata";
    wr    = "waddr" "wdata" "we";
}
```

This specifies that there are 3 read ports. The names of the port wires are `raddr_1/rdata_1`, `raddr_2/rdata_2`, and `raddr_3/rdata_3`.

TBD: The naming of the multiple ports may not be the best.

9.2 Generated Verilog

The BH compiler can generate Verilog code (a module) for a BH module definition. The type of the interface for the module has to obey certain restrictions so that it can be converted to wires.

TBD: Accurately describe restrictions.

The interface type (of the designated module) will be "mangled" by the BH compiler to generate an interface that obeys the restriction that Verilog modules have, see section 9.1.1. The definition of this interface type is available in the generated signature ("`.bi`") file for informational purposes. The "mangled" interface will contain one extra method for each of the original methods (beginning in `RDY_`) which is a handshake signal indicating that the method is ready to be used.

Example:

```
package Cube(mkCube, mkCube16, Cube) where
import UInt
import Mult

interface Cube n =
    start  :: UInt n -> Action    -- An input causing an action
    result :: UInt n              -- The output

data State = Idle | Working deriving (Eq, Bits)

mkCube :: Module (Cube n)
mkCube =
    module
        state :: Reg State
        state <- mkReg Idle
        x :: Reg (UInt n)
        x <- mkRegU
        r :: Reg (UInt n)
        r <- mkRegU
        m :: Mult n
        m <- mkMult
        let (*) = m.mul
        rules
            when state == Working
             ==> action { r := r * x; state := Idle }
        interface
            start n = action { x := n; r := x*x; state := Working }
                when state == Idle
            result  = r
                when state == Idle

mkCube16 :: Module (Cube 16)
mkCube16 = mkCube
```

If code generation for `mkCube16` is requested the generated signature file will contain this:

```
signature Cube where {
type (Cube.Cube :: # -> *) n;

Cube.mkCube :: Prelude.Module (Cube.Cube n);

Cube.mkCube16 :: Prelude.Module (Cube.Cube 16);

interface (Cube.Cube_16_ :: *) = {
    Cube.start :: Prelude.Bit 16 -> Prelude.Action;
    Cube.RDY_start :: Prelude.Bit 1;
    Cube.result :: Prelude.Bit 16;
    Cube.RDY_result :: Prelude.Bit 1
};

Cube.mkCube16_ :: Prelude.Module Cube.Cube_16_
}
```

The generated Verilog module header:

```
module mkCube16_(CLK,
                 RST,
                 RDY_start,   // output, asserted when start can accept
                 result,
                 RDY_result,  // output, asserted when result signal is valid
                 start_1,     // corresponds to first argument of start interface method
                 EN_start);   // input, assert when start method has valid data
  input CLK, RST;
  output RDY_start;
  output [15 : 0] result;
  output RDY_result;
  input [15 : 0] start_1;
  input EN_start;
  ...
```

The naming conventions for the ports is to take the method name (of the mangled interface) and suffix it with _*n* for the *n*th argument. The output of a method will have the method name. The enable signal (for actions) will have `EN_` prefixed to the method name.

Note

The mangled interface is only there for informational purposes; it cannot be used. Perhaps there would be a better way to convey this information?

End of Note

9.2.1 Verilog code generation properties

A number of properties can be specified for a module which affects Verilog code generation.

pragma ::= `{-# properties` *varId* `= {` { *cgprop* `,` } `} #-}`

cgprop ::=
- `verilog`
- `alwaysReady`
- `alwaysEnabled`
- `scanInsert =` *int*
- `bitBlast`
- `CLK =` *varName*
- `RSTN =` *varName*
- `options = {` { *string* `,` } `}`

varName ::= *varId* | *conId* | *string*

The `properties` pragma is given for a specific module and the listed properties affect code generation.

`verilog`	generate Verilog for this module. Without this, or the command line option, no code is generated for this module, instead it will be inlined where used.
`alwaysReady`	all methods in the module interface should be continuously ready, i.e., there is no need to us the BH ready signalling protocol so those wires are left out. The compiler verifies that the methods are indeed always ready.
`alwaysEnabled`	all methods that are actions (i.e., where the type ends with `PrimAction`) are assumed to be continuously enabled, i.e., they execute in every cycle. There is thus no need for the enable signal for the method and it is omitted. The compiler generates code as if the enable wire was always high and verifies that the method will fire in every clock cycle.

`scanInsert`	put extra ports used for scan insertion into the generated Verilog code. The number specifices the number of scan chains to insert.
`bitBlast`	do "bit blasting" of the generated ports, i.e., split ports that consist of multiple bits into the individual bits, and also make all port names upper case.
`CLK`	specify the name of the clock, the default is `CLK`.
`RSTN`	specify the name of the reset, the default is `RST_N`.
`options`	specify additional compiler flags that override the current compiler flags (as given on the command line).

The `alwaysReady` and `alwaysEnabled` properties are useful when the generated code will be connected to other Verilog modules that are not written in BH, and where these modules assume a synchronous signalling protocol.

Example: A module which connects to an external synchronous SRAM

```
{-# properties useSRAM = {
       alwaysReady,
       alwaysEnabled,
} #-}
useSRAM :: Module (SyncSRAMC 1 (Bit 20) (Bit 32))
useSRAM =
   module
       (extram, ram) <- wrapSRAM
       ...
       interface (extram)
```

Example: Do not perform ATS optimization for the module `slow`

```
{-# properties slow = { options = { "-no-opt-ATS" } } #-}
slow :: Module ...
```

10 Interfacing to C

The C code generated and used by the BH compiler is structured similarly to the Verilog modules generated and used by the compiler. Each Verilog module corresponds to a "class" and its instances to "objects". Since C is not object oriented the notion of an object has to be simulated.[2]

Each "class" definition is a struct. It first always contains certain fields, a "`struct obj`", which are explained below. Following these are function pointers that implement all the methods in the interface. Each of these functions takes a pointer to the "object" itself as the first argument (the "`self`" pointer, which is the standard way of implementing object oriented languages). If the interface method is an action (*i.e.,* its type ends in `Action`) the function will have one argument for each of the methods arguments. If the method returns a value it will have an additional argument, the second, where this value will be stored. All arguments are passed by reference; the type "`varp`" is used for updatable values and "`varcp`" for constant value. Each of these arguments represents something which in BH has type "`Bit` *n*" and in Verilog is a bunch of wires.

The initial part of each "class" (*i.e.,* its base class) has the following definition:

```
typedef struct obj *obj;
struct obj {
    obj parent;
```

[2] Generating C++ would have been slightly easier since it has objects.

```
    const struct varinfo *vinfo;
    updfun update;
    dumpfun dump;
    uInt nrules;
    varp *preds;
    ruleinfo *rules;
    uInt nobjs;
    obj *objs;
};
```

None of these fields are needed for a user of an "object", but we will explain them for completeness.

- `parent` points to the object which this state element is a part of.
- `vinfo` contains name etc. for this state variable.
- `update` is the function that must be called when any action has been performed on the object. It will recompute all the private fields in the object, and of all sub-objects.
- `dump` will, if called, print the state of the object.
- `nrules` the number of rules in this object.
- `preds` pointers (`nrules` of them) point to the predicates for all the rules. Use the `GETBOOL()` macro to get the value of one of these.
- `rules` points to an array of information for each rule. The rule information contains the name of the rule and the function to call to execute the rule.
- `nobjs` the number of sub-objects contained in this object.
- `objs` pointers to all the sub-objects.

10.1 C modules

Wherever a Verilog module is used when generating Verilog a corresponding C module is needed for generating C. If the Verilog module, with interface type "*ifc*", is named "*mod*" the compiler will assume that there is a corresponding C header file named "*mod*`.h`". This header file should contain a type definition (a `typedef`) for the type "`B`*ifc*" and a function called "`new_`*mod*" returning a "`B`*ifc*" object.

Example: The Verilog register module, see section 9.1.2, has a corresponding C header file, named "`RegN.h`", with these contents:

```
#if !defined(REG)
#define REG
typedef struct BReg *BReg;
struct BReg {
    struct obj hdr;
    void (*Bget)(BReg, varp);
    void (*Bset)(BReg, varcp);
};
#endif

BReg new_RegN(obj, const struct varinfo *, uInt, varcp);
```

10.2 Generated C

The generated C code follows the conventions described in the preceding sections. If C code is generated for a module named "*templ*", the compiler will generate a header file "*templ*.h" and a code file "*templ*.c".

11 Guiding the compiler

11.1 Pragmas

To guide the compiler to do the right thing there are a number of pragmas. Pragmas can be used where top level definitions are valid. Pragmas have the following general form:

topDefn ::= *pragma*

pragma ::= `{-#` *pragmaId* $\cdots$ `#-}`

pragmaId ::= *varId*

Syntactically, pragmas are comments because they are enclosed in `{-` and `-}` brackets.

11.1.1 Pragma `verilog`

When the compiler generates code for a module it normally tries to integrate all definitions into one big Verilog module. If this is not desirable for some reason you can use the `verilog` pragma to instruct the compiler to generate Verilog modules for parts of the design.

The syntax is:

pragma ::= `{-# verilog` *varId* [{ *veriProp* `,` }] `#-}`

veriProp ::= `noReady` | `alwaysEnabled`

This will tell the compiler to generate Verilog modules for the named module when it is doing code generation.

Some properties of the generated code can be specified as well:

`noReady` specifies that no ready signals should be generated. The compiler verifies that all the methods in the interface are permanently ready.

`alwaysEnabled` specifies that there should be no enable signal for action methods. The method will be executed on every clock cycle, and the compiler verifies that the caller does this.

Note

It is currently not possible to give these properties for individual method, just for the whole interface.

11.1.2 Pragma `noinline`

The `noinline` pragma can be given for functions, it tells the compiler not to inline the function, but to generate code for it directly. The function has same type restrictions as for interface methods that are involved in code generation.

The syntax is:

pragma ::= `{-# noinline` { *varId* } `#-}`

Example:

```
{-# noinline cswap #-}
cswap :: Bool -> (Int 32, Int 32) -> (Int 32, Int 32)
cswap True (x, y) = (y, x)
cswap False xy = xy
```

11.2 Rule assertions

Rule assertions instruct the compiler to abort compilation unless it can verify that a rule satisfies a particular condition. Each assertion affects the rule that immediately follows it and all rules nested within.

rule ::= *ruleAssert* [;] *rule*

Rule assertions are not triggered until the generation of Verilog or C code for the module that includes them.

11.2.1 Assertion `fire when enabled`

This asserts that a rule is scheduled to fire whenever its predicate and its implicit conditions are true, *i.e.,* when they are true, there are no scheduling conflicts that will prevent it from firing.

ruleAssert ::= `{-# ASSERT fire when enabled #-}`

11.2.2 Assertion `no implicit conditions`

This asserts that interface methods called within the rule do not have implicit conditions that contribute to its enabling, *i.e.,* only the explicit rule predicate controls whether it is enabled or not.

ruleAssert ::= `{-# ASSERT no implicit conditions #-}`

References

[Hoe00] J. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2000.

[HPJWe92] P. Hudak, S. L. Peyton Jones, and P. Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[Wad] P. Wadler. Monads. `http://www.cs.bell-labs.com/who/wadler/topics/monads.html`

[Wad90] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–77, Nice, France, 1990.

A Advanced topics

This section contains topics that are not necessary for the beginning BH programmer.

A.1 Lambda expressions

Value definitions (section 4.7) enable definition of functions, but it is bundled with also binding the function value to a name. It is possible to define a function value independently of giving it a name, *i.e.,* to define a function value "anonymously" using so-called "λ-expressions":

exp ::= `\` { *varId* } `->` *exp*

Example:

```
\x -> x * x
```

This defines the "squaring function", *i.e.,* a function of one argument that returns the product of that argument with itself.

TBD: Allow irrefutable patterns instead of variables for λ.

A.2 `do`

exp ::= `do {` { *stmt* `;` } *exp* `}`

The `do` expression in BH provides a convenient syntax for programming with *monads* [Wad, Wad90]. A translation of the `do` expression into simpler expressions is given in the Haskell report [HPJWe92].

The value of a `do` statement is the value of the very last expression. This last expression is commonly a call to the method `return` which is defined for any monad. `return` takes a value and returns a monad type with that value.

A.2.1 Creating modules with `do`

To further illustrate how the `module` syntax is nearly just a `do` expression and the interface nearly a `struct`, consider the following example which creates a module without the special syntax:

```
struct Two =
    a :: Reg (Bit 5)
    b :: Reg (Bit 10)

mkTwo :: Module Two
mkTwo = do
    a <- mkReg 0
    b <- mkReg 0
    return (Two { a = a; b = b })
```

A.2.2 Recursive bindings in module

Normally (i.e., in Haskell) the bindings in a `do` expression come into scope in order, but BH also allows forward references to variables bound in a do expression.

Example: The following is legal

```
do
    x <- foo y
    y <- bar x
    return (x, y)
```

Such recursive bindings does not make sense in all monads so there is a type restriction to capture this. Normally a `do` expression has type `(Monad m) => m` *t*, but a `do` expression with as forward reference has type `(MonadFix m) => m` *t*. The class `MonadFix` is defined as

```
class (Monad m) => MonadFix m where
    mfix :: (a -> m a) -> m a
```

A `do` with forward references is transformed into an ordinary `do` as follows: Let `x1` $\cdots$ `xk` be the identifiers that are referenced forwardly.

```
do
    s1
    ...
    sn
    e
```

transforms to (where `n`, `t`, and `r` are fresh identifiers)

```
do
    n <- mfix (\ t -> let (r, x1, ... xk) = t
                      in  do
                              s1
                              ...
                              sn
                              r <- e
                              return (r, x1, ... xk)
              )
    return n.fst
```

Both the `Module` and `ActionValue` monads belong to the class `MonadFix`.

A.3 IsModule

The type constructor `Module` is a primitive type that the compiler knows about. It is possible to build variations of this type within BH. To express that a type is related to `Module` we use the class `IsModule`. The special `module` syntax is available for all types that belong to `IsModule`.

```
class (Monad m) => IsModule m where
    liftModule :: Module a -> m a
```

The `liftModule` function is the conversion from a standard module into the augmented module type.

Naturally, the `Module` type trivially belongs to the `IsModule` class.

```
instance IsModule Module where
    liftModule  m  = m
```

All the primitive state generators, e.g., `mkReg`, have a type that is general enough that they can be used in any module variation.

```
mkReg :: (IsModule m, Bits a sa) => a -> m (Reg a)
```

It is very easy to make a value with such a type; just apply `liftModule` to an ordinary `Module` value.

B Syntax

B.1 Reserved words

The following words are reserved in BH:

```
action as
case class
data default data deriving do
else
foreign
hiding
if import in infix infixl infixr instance interface
let
module
newtype
of
package prefix primitive
qualified
return rules
signature struct
then type
valueOf verilog
when where
```

C Semantics of primitive operations

The BH compiler internally translates a BH program until it is entirely defined in terms of primitive operations and external modules. This section describes the semantics of the primitive operations.

All primitive operations are defined for all sizes (including 0) given the type constraints. All numbers are interpreted in two's complement representation where applicable.

`primitive primAdd :: Bit n -> Bit n -> Bit n`
: Adds two n bit numbers and returns the lower n bits of the result.
Generates a Verilog "`+`".

`primitive primSub :: Bit n -> Bit n -> Bit n`
: Subtracts two n bit numbers and returns the lower n bits of the result.
Generates a Verilog "`-`".

`primitive primMul :: Bit n -> Bit n -> Bit n`
: Multiplies two n bit numbers and returns the lower n bits of the result.
Generates a Verilog "`*`".

`primitive primNeg :: Bit n -> Bit n`
: Negation of an n bit number.
Generates a Verilog "`-`".

`primitive primAnd :: Bit n -> Bit n -> Bit n`
: Bitwise AND of two n bit numbers.
Generates a Verilog "`&`".

`primitive primOr :: Bit n -> Bit n -> Bit n`
: Bitwise OR of two n bit numbers.
Generates a Verilog "`|`".

`primitive primXor :: Bit n -> Bit n -> Bit n`
: Bitwise XOR of two n bit numbers.
Generates a Verilog "`^`".

`primitive primSL :: Bit n -> Nat -> Bit n`
: Shift the first argument left by the number of bits given by the second argument. Vacated positions are filled with 0. The behaviour is undefined if the shift count is equal or larger than the width of the shifted word.
Generates a Verilog "`<<`" if the shift count is non-constant, otherwise just bit selection.

`primitive primSRL :: Bit n -> Nat -> Bit n`
: Shift the first argument right by the number of bits given by the second argument. Vacated positions are filled with 0. The behaviour is undefined if the shift count is equal or larger than the width of the shifted word.
Generates a Verilog "`>>`" if the shift count is non-constant, otherwise just bit selection.

`primitive primSRA :: Bit n -> Nat -> Bit n`
: Shift the first argument right by the number of bits given by the second argument. Vacated positions are filled with the most significant bit. The behaviour is undefined if the shift count is equal or larger than the width of the shifted word.
Generates a Verilog "`>>`" and replication of the sign if the shift count is non-constant, otherwise just bit selection.

`primitive primInv :: Bit n -> Bit n`
: Bitwise inverting of an n bit number.
Generates a Verilog "`~`".

`primitive primEQ :: Bit n -> Bit n -> Bit 1`
: Comparison of two n bit numbers, returns 1 if they are equal otherwise 0.
Generates a Verilog "`==`".

`primitive primULE :: Bit n -> Bit n -> Bit 1`
: Unsigned comparison of two n bit numbers, returns 1 if the first one is less than or equal to the second, otherwise 0.
Generates a Verilog "`<=`".

`primitive primULT :: Bit n -> Bit n -> Bit 1`
: Unsigned comparison of two n bit numbers, returns 1 if the first one is less than the second, otherwise 0.
Generates a Verilog "`<`".

`primitive primSLE :: Bit n -> Bit n -> Bit 1`
: Signed comparison of two n bit numbers, returns 1 if the first one is less than or equal to the second, otherwise 0.
Generates a Verilog "`<=`" with the sign bits inverted.

`primitive primSLT :: Bit n -> Bit n -> Bit 1`
: Signed comparison of two n bit numbers, returns 1 if the first one is less than the second, otherwise 0.
Generates a Verilog "`<`" with the sign bits inverted.

`primitive primZeroExt :: (Add n k m) => Bit n -> Bit m`
: Extend the argument with 0 on the left to make it the right size.
Generates a Verilog bit concatenation.

`primitive primSignExt :: (Add n k m) => Bit n -> Bit m`
: Extend the argument with the sign bit replicated on the left to make it the right size. An argument of size 0 is assumed to have a 0 sign bit.
Generates a Verilog bit concatenation.

`primitive primTrunc :: (Add k m n) => Bit n -> Bit m`
: Truncate bits on the left. Generates a Verilog bit extraction.

`primitive primBNot :: Bit 1-> Bit 1`
: Boolean NOT.
Generates a Verilog "`!`".

`primitive primBAnd :: Bit 1 -> Bit 1 -> Bit 1`
: Boolean AND. During compile time this is a "short circuit" operator that avoids evaluating the second operand if possible.
Generates a Verilog "`&&`".

`primitive primBOr :: Bit 1 -> Bit 1 -> Bit 1`
: Boolean OR. During compile time this is a "short circuit" operator that avoids evaluating the second operand if possible.
Generates a Verilog "`||`".

D The Standard Prelude and Additional Libraries

Please see the separate document *Libraries Reference Guide* for the Standard Prelude and Additional Libraries.

Index

‘, *see* infix operators, converting from ordinary identifiers
`:=` (`Reg` assignment), **35**
`.`, *see* structs, field selection
`=>` (in context-qualified types), **9**
`{-` (open nested comment), 7
`(..)` (exporting constructors and field names), 10
`<-` (in guards), 31
`<-` (in statements), 28
`<+` (`Rules` aggregation operator), **28**
`<+>` (`Rules` aggregation operator), **28**
`->` (in lambda expressions), 46
`->` (infix function type constructor), **8**
`--` (ordinary comment), 7
`-}` (close nested comment), 7
`+>` (`Rules` aggregation operator), **28**
`++` (`Bit` concatenation operator), **33**
’ (character, in identifiers), 6

`_`
- “don’t care” expression, 6, 18, **25**
- “don’t care” pattern, 6, 18, **30**

`_`
- in identifiers, 6

actions
- `Action` (type), **25**
- `action` (keyword), **25**
- combining, 25

`ActionValue` (type), 26
`Add` (type class, of size types), 32
application
- associativity, 19
- infix, **19**
- of functions to arguments, **19**

arity, of type constructor, 8
arrow types, *see* function types
`asReg` (dummy `Reg` function), **35**

`Bit` (type), **33**
`Bits` (type class), **33**
- `deriving`, 16
- representation of data types, 16

braces and semicolons, *see* layout

`case` (keyword), 22
case expression, 22
class, *see* type class
`clear` (`FIFOF` interface method), **35**
`clear` (`FIFO` interface method), **35**
comment
- nested, 7
- ordinary, 7

conditional expressions, *see* if-then-else expressions
conId (grammar non-terminal), **6**
constants, 21
context-qualified types, **9**
contexts, *see* context-qualified types

`deq` (`FIFOF` interface method), **35**
`deq` (`FIFO` interface method), **35**
`deriving` (keyword), **11**
`do` (keyword), 46
do expressions, 46
don’t care (patterns and expressions), *see* “`_`”

`else` (keyword), 22
`enq` (`FIFOF` interface method), **35**
`enq` (`FIFO` interface method), **35**
exp (grammar non-terminal), **19**
export, identifiers from a package, **10**

field names, 6
`FIFO` (interface type), **35**
`FIFOF` (interface type), **35**
`fire when enabled` (rule assertion), 45
`first` (`FIFOF` interface method), **35**
`first` (`FIFO` interface method), **35**
`foreign` (keyword), **18**
foreign functions, calling, 18
`fromInteger` (`Literal` class method), **22**
function types, **8**

`get` (`Reg` interface method), **34**
grammar, 6
guards
- with clauses of top-level definitions, 17
- with pattern matching, **31**

Haskell, 6

identifiers, **6**
- case of first letter, 6
- constructor, 6
- converting to infix operators, 20
- export from a package, 10
- qualified, **11**
- variable, 6

`if` (keyword), 22
if-then-else expressions, **22**
implicit conditions, **25**

asserting that a rule has none, 45
disallowed in arguments to Verilog modules, 38
on interface methods, 25
`import` (keyword), 10
`in` (keyword), 23
indentation, *see* layout
`infix` (keyword), 10
infix operators
associativity, 19
converting from ordinary identifiers, 20
converting to ordinary identifiers, 20
defining new, 20
precedence, 19
predefined, 19
`infixl` (keyword), 10
`infixr` (keyword), 10
instance (of type class), **9**
`Int` (type), **34**
Integer literals, 21
`interface` (keyword)
in interface definitions, **14**
in interface expressions, 24
interfaces, **14**
`IsModule` (typeclass), 47

`Just`, *see* `Maybe`

lambda expressions, **46**
layout, 6
`let` (keyword), 23, 28
let expressions, **23**
`liftModule` (`IsModule` method), 47
Literals, 21
Integer, 21
String, 22
`Log` (type class, of size types), 32

`Max` (type class, of size types), 32
`Maybe` (type), **12**
meta notation, *see* grammar
methods
of a type class, **9**
of an interface, **14**
`mkFIFO` (`FIFO` function), **35**
`mkFIFOF` (`FIFOF` function), **35**
`mkReg` (`Reg` function), **34**
`mkRegU` (`Reg` function), **34**
`mkSizedFIFO` (`FIFO` function), **35**
`mkSizedFIFOF` (`FIFOF` function), **35**
modules
analogy with objects in object-oriented programming languages, 28
`Module` (type), **28**
`module` (keyword), **28**

`no implicit conditions` (rule assertion), 45
`noAction` (empty action), 26
`noinline` (pragma), 44
`Nothing`, *see* `Maybe`

`of` (keyword), 22
overloading, of type, 9

`pack` (`Bits` class method), **34**
package, **10**
`package` (keyword), 10
packageId (grammar non-terminal), 6
pattern binding, **23**
pattern matching, **30**
error, 18, 22
patterns, **30**
`_` (underscore, don't care), **30**
constructor, **30**
struct, **31**
variable, **30**
`pragma` (keyword), 44
`Prelude`, 6
product types, *see* structs

records, *see* structs
`Reg` (type), **34**
rule assertions, **45**
`fire when enabled`, 45
`no implicit conditions`, 45
rules
aggregating, **28**
expression, **26**
nested rule guards, 27
prioritizing, 28
`Rules` (type), **26**
`rules` (keyword), 27

`set` (`Reg` interface method), **34**
size types, 8, **32**
type classes for constraints, 32
`SizeOf` (pseudo-function on types), **34**
`split` (`Bit` function), **33**
String literals, 22
string types, 8
`stringOf` ("function" of string types), 33
`struct` (keyword), 13
structs
construction, **23**
field selection, **24**
type definition, **13**
update, **24**

`then` (keyword), 22
tuples
patterns, 13, **31**

type definition, **13**
values, 13, **24**
tycon (grammar non-terminal), 6, **7**
type class, **9**
type constructor, 7
type signature, 7
type variables, 6, 8
types, **7**
context-qualified, *see* context-qualified types
parameterized, 8
polymorphic, 8

`UInt` (type), **34**
underscore, *see* "_"
`unpack` (`Bits` class method), **34**

value constructor names, 6, 21
value definitions, **17**
`valueOf` ("function" of "size" types), 32
variables, 6, 21
`varId` (grammar terminal), **6**
`verilog` (pragma), 44

`when` (keyword)
in clauses of top-level definitions, 17
in interface expression, 24
in rules expressions, 27
`where` (keyword), 10