An External Representation for the GHC Core Language (DRAFT for GHC5.02)

Andrew Tolmach (apt@cs.pdx.edu)
and The GHC Team

September 6, 2001

Abstract

This document provides a precise definition for the GHC Core language, so that it can be used to communicate between GHC and new stand-alone compilation tools such as back-ends or optimizers. The definition includes a formal grammar and an informal semantics. An executable typechecker and interpreter (in Haskell), which formally embody the static and dynamic semantics, are available separately.

Note: This is a draft document, which attempts to describe GHC's current behavior as precisely as possible. Working notes scattered throughout indicate areas where further work is needed. Constructive comments are very welcome, both on the presentation, and on ways in which GHC could be improved in order to simplify the Core story.

1 Introduction

The Glasgow Haskell Compiler (GHC) uses an intermediate language, called "Core," as its internal program representation during several key stages of compiling. Core resembles a subset of Haskell, but with explicit type annotations in the style of the polymorphic lambda calculus (F_ω). GHC's front end translates full Haskell 98 (plus some extensions) into well-typed Core, which is then repeatedly rewritten by the GHC optimizer. Ultimately, GHC translates Core into STG-machine code and then into C or native code. The rationale for the design of Core and its use are discussed in existing papers [Peyton Jones and Marlow, 1999, Peyton Jones and Santos, 1998], although the (two different) idealized versions of Core described therein differ in significant ways from the actual Core language in current GHC.

Researchers interested in writing just *part* of a Haskell compiler, such as a new back-end or a new optimizer pass, might like to make use of GHC to provide the other parts of the compiler. For example, they might like to use GHC's front end to parse, desugar, and type-check source Haskell, then feeding the resulting code to their own back-end tool. Currently, they can only do this by linking their code into the GHC executable, which is an arduous process (and essentially requires the new code to be written in Haskell). It would be much easier for external developers if GHC could be made to produce Core files in an agreed-upon external format. To allow the widest range of interoperability, the external format should be text-based; pragmatically, it should also be human-readable. (It may ultimately be desirable to use a standard interchange base format such as ASDL or XML.)

In the past, Core has had no rigorously defined external representation, although by setting certain compiler flags, one can get a (rather ad-hoc) textual representation to be printed at various points in the compilation process; this is usually done to help debug the compiler. To make Core fully useable a bi-directional communication format, it will be necssary to

1. define precisely the external format of Core;
2. modify GHC to produce external Core files, if so requested, at one or more useful points in the compilation sequence – e.g., just before optimization, or just after;

3. modify GHC to accept external Core files in place of Haskell source files, again at one or more useful points.

The first two facilities will let one couple GHC's front-end (parser, type-checker, etc.), and optionally its optimizer, with new back-end tools. Adding the last facility will let one implement new Core-to-Core transformations in an external tool and integrate them into GHC. It will also allow new front-ends to generate Core that can be fed into GHC's optimizer or back end; however, because there are many (undocumented) idiosynracies in the way GHC produces Core from source Haskell, it will be hard for an external tool to produce Core that can be integrated with GHC-produced core (e.g., for the Prelude), and we don't aim to support this.

This document addresses the first requirement, a formal Core definition, by proposing a formal grammar for an external representation of Core (Section 2, and an informal semantics (Section 3.

Beginning in GHC5.02, external Core (post-optimization) adhering to this definition can be generated using the compiler flag `-fext-core`.

Formal static and dynamic semantics in the form of an executable typechecker and interpreter are available separately in the GHC source tree under `fptools/ghc/utils/ext-core`.

2 External Grammar of Core

In designing the external grammar, we have tried to strike a balance among a number of competing goals, including easy parseability by machines, easy readability by humans, and adequate structural simplicity to allow straightforward presentations of the semantics. This has inevitably led to certain compromise. In particular:

- In order to avoid explosion of parentheses, various standard precedences and short-cuts are supported for expressions, types, and kinds; this led to the introduction of multiple non-terminals for each of these syntactic categories, which makes the concrete grammar longer and more complex than the underlying abstract syntax.
- On the other hand, the grammar has been kept simpler by avoiding special syntax for tuple types and terms; tuples (both boxed and unboxed) are treated as ordinary constructors.
- All type abstractions and applications are given in full, even though some of them (e.g., for tuples) could be reconstructed; this permits Core to be parsed without the necessity of performing any type reconstruction.
- The syntax of identifiers is heavily restricted (essentially to just alphanumerics); this again makes Core easier to parse but harder to read.

Working note: *These choices are certainly debatable. In particular, keeping type applications on tuples and case arms considerably increases the size of core files and makes them less human-readable, though it allows a Core parser to be simpler.*

We use the following notational conventions for syntax:

[pat]	optional
{ *pat* }	zero or more repetitions
{ *pat* }$^+$	one or more repetitions
$pat_1 \mid pat_2$	choice
`fibonacci`	terminal syntax in typewriter font

Module	*module*	→	`%module` *mident* { *tdef* `;` } { [`%local`] *vdefg* `;` }	
Type defn.	*tdef*	→	`%data` *qtycon* { *tbind* } `= {` *cdef* { `;` *cdef* } `}`	algebraic type
		|	`%newtype` *qtycon* { *tbind* } [`=` *ty*]	newtype
Constr. defn.	*cdef*	→	*qdcon* { `@` *tbind* } { *aty* }	
Value defn.	*vdefg*	→	`%rec {` *vdef* { `;` *vdef* } `}`	recursive
		|	*vdef*	non-recursive
	vdef	→	*qvar* `::` *ty* `=` *exp*	
Atomic expr.	*aexp*	→	*qvar*	variable
		|	*qdcon*	data constructor
		|	*lit*	literal
		|	`(` *exp* `)`	nested expr.
Expression	*exp*	→	*aexp*	atomic expresion
		|	*aexp* { *arg* }$^+$	application
		|	`\` { *binder* }$^+$ `->` *exp*	abstraction
		|	`%let` *vdefg* `%in` *exp*	local definition
		|	`%case` *exp* `%of` *vbind* `{` *alt* { `;` *alt* } `}`	case expression
		|	`%coerce` *aty* *exp*	type coercion
		|	`%note` `"` { *char* } `"` *exp*	expression note
		|	`%external` `"` { *char* } `"` *aty*	external reference
Argument	*arg*	→	`@` *aty*	type argument
		|	*aexp*	value argument
Case alt.	*alt*	→	*qdcon* { `@` *tbind* } { *vbind* } `->` *exp*	constructor alternative
		|	*lit* `->` *exp*	literal alternative
		|	`%_` `->` *exp*	default alternative
Binder	*binder*	→	`@` *tbind*	type binder
		|	*vbind*	value binder
Type binder	*tbind*	→	*tyvar*	implicitly of kind `*`
		|	`(` *tyvar* `::` *kind* `)`	explicitly kinded
Value binder	*vbind*	→	`(` *var* `::` *ty* `)`	
Literal	*lit*	→	`(` [`-`] { *digit* }$^+$ `::` *ty* `)`	integer
		|	`(` [`-`] { *digit* }$^+$ `.` { *digit* }$^+$ `::` *ty* `)`	rational
		|	`(` `'` *char* `'` `::` *ty* `)`	character
		|	`(` `"` { *char* } `"` `::` *ty* `)`	string
Character	*char*	→	*any ASCII character in range 0x20-0x7E except 0x22,0x27,0x5c*	
		|	`\x` *hex hex*	ASCII code escape sequence
	hex	→	`0` |...|`9` |`a` |...|`f`	

Atomic type	*aty*	→	*tyvar*	type variable
		|	*qtycon*	type constructor
		|	`(` *ty* `)`	nested type
Basic type	*bty*	→	*aty*	atomic type
		|	*bty aty*	type application
Type	*ty*	→	*bty*	basic type
		|	`%forall` { *tbind* }$^+$ `.` *ty*	type abstraction
		|	*bty* `->` *ty*	arrow type construction
Atomic kind	*akind*	→	`*`	lifted kind
		|	`#`	unlifted kind
		|	`?`	open kind
		|	`(` *kind* `)`	nested kind
Kind	*kind*	→	*akind*	atomic kind
		|	*akind* `->` *kind*	arrow kind
Identifier	*mident*	→	*uname*	module
	tycon	→	*uname*	type constr.
	qtycon	→	*mident* `.` *tycon*	qualified type constr.
	tyvar	→	*lname*	type variable
	dcon	→	*uname*	data constr.
	qdcon	→	*mident* `.` *dcon*	qualified data constr.
	var	→	*lname*	variable
	qvar	→	*[mident* `.` *] var*	optionally qualified variable
Name	*lname*	→	*lower* { *namechar* }	
	uname	→	*upper* { *namechar* }	
	namechar	→	*lower* | *upper* | *digit* | `'`	
	lower	→	`a` | `b` | ... | `z` | `_`	
	upper	→	`A` | `B` | ... | `Z`	
	digit	→	`0` | `1` | ... | `9`	

Working note: *Should add some provision for comments.*

3 Informal Semantics

Core resembles a explicitly-typed polymorphic lambda calculus (F_ω), with the addition of local `let` bindings, algebraic type definitions, constructors, and `case` expressions, and primitive types, literals and operators. It is hoped that this makes it easy to obtain an informal understanding of Core programs without elaborate description. This section therefore concentrates on the less obvious points.

3.1 Program Organization and Modules

Core programs are organized into *modules*, corresponding directly to source-level Haskell modules. Each module has a identifying name *mident*.

Each module may contain the following kinds of top-level declarations:

- Algebraic data type declarations, each defining a type constructor and one or more data constructors;
- Newtype declarations, corresponding to Haskell `newtype` declarations, each defining a type constructor; and
- Value declarations, defining the types and values of top-level variables.

No type constructor, data constructor, or top-level value may be declared more than once within a given module. All the type declarations are (potentially) mutually recursive. Value declarations must be in dependency order, with explicit grouping of mutually recursive declarations.

Identifiers defined in top-level declarations may be *external* or *internal*. External identifiers can be referenced from any other module in the program, using conventional dot notation (e.g., `PrelBase.Bool`, `PrelBase.True`). Internal identifiers are visible only within the defining module. All type and data constructors are external, and are always defined and referenced using fully qualified names (with dots). A top-level value is external if it is defined and referenced using a fully qualified name with a dot (e.g., `MyModule.foo = ...`); otherwise, it is internal (e.g., `bar = ...`). Note that the notion of external identifier does not necessarily coincide with that of "exported" identifier in a Haskell source module: all constructors are external, even if not exported, and non-exported values may be external if they are referenced from potentially in-lineable exported values. Core modules have no explicit import or export lists. Modules may be mutually recursive.

***Working note:** But in the presence of inter-module recursion, is there much point in keeping track of recursive groups within modules? Options: (1) don't worry about it; (2) put all declarations in module (indeed whole program) into one huge recursive pot; (3) abandon general module recursion, and introduce some kind of import declaration to define the types (only) of things from external modules that currently introduce module recursion.*

There is also an implicitly-defined module `PrelGHC`, which exports the "built-in" types and values that must be provided by any implementation of Core (including GHC). Details of this module are in Section 4.

A Core *program* is a collection of distinctly-named modules that includes a module called `Main` having an exported value called `main` of type `PrelIOBase.IO a` (for some type `a`).

Many modules of interest derive from library modules, such as `PrelBase`, which implement parts of the Haskell basis library. In principle, these modules have no special status. In practice, the requirement on the type of `Main.main` implies that every program will contain a large subset of the Prelude library modules.

3.2 Namespaces

There are five distinct name spaces:

1. module identifiers (`mident`),
2. type constructors (`tycon`),
3. type variables (`tyvar`),
4. data constructors (`dcon`),
5. term variables (`var`).

Spaces (1), (2+3), and (4+5) can be distinguished from each other by context. To distinguish (2) from (3) and (4) from (5), we require that (both sorts of) constructors begin with an upper-case character and that (both sorts of) variables begin with a lower-case character (or `_`). Primitive types and operators are not syntactically distinguished.

A given variable (type or term) may have multiple (local) definitions within a module. However, definitions never "shadow" one another; that is, the scope of the definition of a given variable never contains a redefinition of the same variable. The only exception to this is that (necessarily closed) types labelling `%external` expressions may contain `tyvar` bindings that shadow outer bindings.

Core generated by GHC makes heavy use of encoded names, in which the characters `Z` and `z` are used to introduce escape sequences for non-alphabetic characters such as dollar sign `$` (`zd`), hash `#` (`zh`), plus `+` (`zp`), etc. This is the same encoding used in `.hi` files and in the back-end of GHC itself, except that we sometimes change an initial `z` to `Z`, or vice-versa, in order to maintain case distinctions.

3.3 Types and Kinds

In Core, all type abstractions and applications are explicit. This make it easy to typecheck any (closed) fragment. An full executable typechecker is available separately.

Types are described by type expressions, which are built from named type constructors and type variables using type application and universal quantification. Each type constructor has a fixed arity ≥ 0. Because it is so widely used, there is special infix syntax for the fully-applied function type constructor (`->`). (The prefix identifier for this constructor is `PrelGHC.ZLzmzgZR`; this should only appear in unapplied or partially applied form.) There are also a number of other primitive type constructors (e.g., `Intzh`) that are predefined in the `PrelGHC` module, but have no special syntax. Additional type constructors are introduced by `%data` and `%newtype` declarations, as described below. Type constructors are distinguished solely by name.

As described in the Haskell definition, it is necessary to distinguish well-formed type-expressions by classifying them into different *kinds*. In particular, Core explicitly records the kind of every bound type variable. Base kinds (`*`,`#`, and `?`) represent actual types, i.e., those that can be assigned to term variables; all the nullary type constructors have one of these kinds. Non-nullary type constructors have higher kinds of the form k_1`->`k_2, where k_1 and k_2 are kinds. For example, the function type constructor `->` has kind `* -> (* -> *)`. Since Haskell allows abstracting over type constructors, it is possible for type variables to have higher kinds; however, it is much more common for them to have kind `*`, so this is the default if the kind is omitted in a type binder.

The three base kinds distinguish the *liftedness* of the types they classify: `*` represents lifted types; `#` represents unlifted types; and `?` represents "open" types, which may be either lifted or unlifted. Of these, only `*` ever appears in Core code generated from user code; the other two are needed to describe certain types in primitive (or otherwise specially-generated) code. Semantically, a type is lifted if and only if it has bottom as an element. Operationally, lifted types may be represented by closures; hence, any unboxed value is necessarily unlifted. In particular, no top-level identifier (except in `PrelGHC`) has a type of kind `#` or `?`. Currently, all the primitive types are unlifted (including a few boxed primitive types such as `ByteArrayzh`). The ideas behind the use of unboxed and unlifted types are described in [Peyton Jones and Launchbury, 1991].

There is no mechanism for defining type synonyms (corresponding to Haskell `type` declarations). Type equivalence is just syntactic equivalence on type expressions (of base kinds) modulo:

- alpha-renaming of variables bound in `%forall` types;
- the identity a `->` $b \equiv$ `PrelGHC.ZLzmzgZR` a b
- the substitution of representation types for *fully applied* instances of newtypes (see Section 3.5).

3.4 Algebraic data types

Each `data` declaration introduces a new type constructor and a set of one or more data constructors, normally corresponding directly to a source Haskell `data` declaration. For example, the source declaration

```
data Bintree a =
   Fork (Bintree a) (Bintree a)
|  Leaf a
```

might induce the following Core declaration

```
%data Bintree a = {
   Fork (Bintree a) (Bintree a);
   Leaf a)}
```

which introduces the unary type constructor `Bintree` of kind `*->*` and two data constructors with types

```
Fork :: %forall a . Bintree a -> Bintree a -> Bintree a
Leaf :: %forall a . a -> Bintree a
```

We define the *arity* of each data constructor to be the number of value arguments it takes; e.g. Fork has arity 2 and Leaf has arity 1.

For a less conventional example illustrating the possibility of higher-order kinds, the Haskell source declaration

```
data A f a = MkA (f a)
```

might induce the core declaration

```
%data A (f::*->*) (a::*) = { MkA (f a) }
```

which introduces the constructor

```
MkA :: %forall (f::*->*) (a::*) . (f a) -> (A f) a
```

GHC (like some other Haskell implementations) supports an extension to Haskell98 for existential types such as

```
data T = forall a . MkT a (a -> Bool)
```

This is represented by the Core declaration

```
%data T = {MkT @a a (a -> Bool)}
```

which introduces the nullary type constructor T and the data constructor

```
MkT :: %forall a . a -> (a -> Bool) -> T
```

In general, existentially quantified variables appear as extra univerally quantified variables in the data contructor types. An example of how to construct and deconstruct values of type T is shown in Section 3.6.

3.5 Newtypes

Each Core %newtype declaration introduces a new type constructor and (usually) an associated representation type, corresponding to a source Haskell newtype declaration. However, unlike in source Haskell, no data constructors are introduced. In fact, newtypes seldom appear in value types in Core programs, because GHC usually replaces them with their representation type. For example, the Haskell fragment

```
newtype U = MkU Bool
u = MkU True
v = case u of
  MkU b -> not b
```

might induce the Core fragment

```
%newtype U = Bool;
u :: Bool = True;
v :: Bool =
   %let b :: Bool = u
   %in not b;
```

The main purpose of including %newtype declarations in Core is to permit checking of type expressions in which partially-applied newtype constructors are used to instantiate higher-kinded type variables. For example:

```
newtype W a = MkW (Bool -> a)
data S k = MkS (k Bool)
a :: S W = MkS (MkW(\x -> not x))
```

might generate this Core:

```
%newtype W a = Bool -> a;
%data S (k::(*->*)) = MkS (k Bool);
a :: S W = MkS @ W (\(x::Bool) -> not x)
```

The type application (S W) cannot be checked without a definition for W.

Very rarely, source newtype declarations may be (directly or indirectly) recursive. In such cases, it is not possible to subsitute the representation type for the new type; in fact, the representation type is omitted from the corresponding Core %newtype declaration. Elements of the new type can only be created or examined by first explicitly coercing them from/to the representation type, using a %coerce expression. For example, the silly Haskell fragment

```
newtype U = MkU (U -> Bool)
u = MkU (\x -> True)
v = case u of
  MkU f -> f u
```

might induce the Core fragment

```
%newtype U;
u :: U = %coerce U (\ (x::U) -> True);
v :: Bool =
   %let f :: U -> Bool = %coerce (U -> Bool) u
   %in f u;
```

> ***Working note:*** *The treatment of newtypes is still very unattractive: acres of explanation for very rare phenomena.*

3.6 Expression Forms

Variables and data constructors are straightforward.

Literal (*lit*) expressions consist of a literal value, in one of four different formats, and a (primitive) type annotation. Only certain combinations of format and type are permitted; see Section 4. The character and string formats can describe only 8-bit ASCII characters. Moreover, because strings are interpreted as C-style null-terminated strings, they should not contain embedded nulls.

Both value applications and type applications are made explicit, and similarly for value and type abstractions. To tell them apart, type arguments in applications and formal type arguments in abstractions are preceded by an `@` symbol. (In abstractions, the `@` plays essentially the same role as the more usual Λ symbol.) For example, the Haskell source declaration

```
f x = Leaf (Leaf x)
```

might induce the Core declaration

```
f :: %forall a . a -> BinTree (BinTree a) =
  \ @a (x::a) -> Leaf @(Bintree a) (Leaf @a x)
```

Value applications may be of user-defined functions, data constructors, or primitives. None of these sorts of applications are necessarily saturated (although previously published variants of Core did require the latter two sorts to be).

Note that the arguments of type applications are not always of kind `*`. For example, given our previous definition of type `A`:

```
data A f a = MkA (f a)
```

the source code

```
MkA (Leaf True)
```

becomes

```
(MkA @Bintree @Bool) (Leaf @Bool True)
```

Local bindings, of a single variable or of a set of mutually recursive variables, are represented by `%let` expressions in the usual way.

By far the most complicated expression form is `%case`. `%case` expressions are permitted over values of any type, although they will normally be algebraic or primitive types (with literal values). Evaluating a `%case` forces the evaluation of the expression being tested (the "scrutinee"). The value of the scrutinee is bound to the variable following the `%of` keyword, which is in scope in all alternatives; this is useful when the scrutinee is a non-atomic expression (see next example).

In an algebraic `%case`, all the case alternatives must be labeled with distinct data constructors from the algebraic type, followed by any existential type variable bindings (see below), and typed term variable bindings corresponding to the data constructor's arguments. The number of variables must match the data constructor's arity.

For example, the following Haskell source expression

```
case g x of
  Fork l r -> Fork r l
  t@(Leaf v) -> Fork t t
```

might induce the Core expression

```
%case g x %of (t::Bintree a)
   Fork (l::Bintree a) (r::Bintree a) ->
      Fork @a r l
   Leaf (v::a) ->
      Fork @a t t
```

When performing a `%case` over a value of an existentially-quantified algebraic type, the alternative must include extra local type bindings for the existentially-quantified variables. For example, given

```
data T = forall a . MkT a (a -> Bool)
```

the source

```
case x of
  MkT w g -> g w
```

becomes

```
%case x %of (x'::T)
  MkT @b (w::b) (g::b->Bool) -> g w
```

In a `%case` over literal alternatives, all the case alternatives must be distinct literals of the same primitive type.

The list of alternatives may begin with a default alternative labeled with an underscore (`%_`), which will be chosen if none of the other alternative match. The default is optional except for a case over a primitive type, or when there are no other alternatives. If the case is over neither an algebraic type nor a primitive type, the default alternative is the *only* one that can appear. For algebraic cases, the set of alternatives need not be exhaustive, even if no default is given; if alternatives are missing, this implies that GHC has deduced that they cannot occur.

The `%coerce` expression is primarily used in conjunction with manipulation of newtypes, as described in Section 3.5. However, `%coerce` is sometimes used for other purposes, e.g. to coerce the return type of a function (such as `error`) that is guaranteed never to return. By their natures, uses of `%coerce` cannot be independently justified, and must be taken on faith by a type-checker for Core.

A `%note` expression is used to carry arbitrary internal information of interest to GHC. The information must be encoded as a string. Expression notes currently generated by GHC include the inlining pragma (`InlineMe`) and cost-center labels for profiling.

A `%external` expression denotes an external identifier, which has the indicated type (always expressed in terms of Haskell primitive types).

> ***Working note:*** *The present syntax is sufficient for describing C functions and labels. Interfacing to other languages may require additional information or a different interpretation of the name string.*

3.7 Expression Evaluation

The dynamic semantics of Core are defined on the type-erasure of the program; ie. we ignore all type abstractions and applications. The denotational semantics the resulting type-free program are just the conventional ones for a call-by-name language, in which expressions are only evaluated on demand. But Core is intended to be a call-by-*need* language, in which expressions are only evaluated *once*. To express the sharing behavior of call-by-need, we give an operational model in the style of Launchbury. This section describes the model informally; a more formal semantics is separately available in the form of an executable interpreter.

To simplify the semantics, we consider only "well-behaved" Core programs in which constructor and primitive applications are fully saturated, and in which non-trivial expresssions of unlifted kind (`#`) appear only as scrutinees in `%case` expressions. Any program can easily be put into this form; a separately available executable preprocessor illustrates how. In the remainder of this section, we use "Core" to mean "well-behaved" Core.

Evaluating a Core expression means reducing it to *weak-head normal form (WHNF)*, i.e., a primitive value, lambda abstraction, or fully-applied data constructor. Evaluation of a program is evaluation of the expression `Main.main`.

To make sure that expression evaluation is shared, we make use of a *heap*, which can contain

- *Thunks* representing suspended (i.e., as yet unevaluated) expressions.
- *WHNF*s representing the result of evaluating such thunks. Computations over primitive types are never suspended, so these results are always closures (representing lambda abstractions) or data constructions.

Thunks are allocated when it is necessary to suspend a computation whose result may be shared. This occurs when evaluating three different kinds of expressions:

- Value definitions at top-level or within a local `let` expression. Here, the defining expressions are suspended and the defined names are bound to heap pointers to the suspensions.
- User function applications. Here, the actual argument expression is suspended and the formal argument is bound to a heap pointer to the suspension.
- Constructor applications. Here, the actual argument expression is suspended and a heap pointer to the suspension is embedded in the constructed value.

As computation proceeds, copies of the heap pointer propagate. When the computation is eventually forced, the heap entry is overwritten with the resulting WHNF, so all copies of the pointer now point to this WHNF. Forcing occurs only in the context of

- evaluating the operator expression of an application;
- evaluating the "scrutinee" of a `case` expression; or
- evaluating an argument to a primitive or external function application

Ultimately, if there are no remaining pointers to the heap entry (whether suspended or evaluated), the entry can be garbage-collected; this is assumed to happen implicitly.

With the exception of functions, arrays, and mutable variables, the intention is that values of all primitive types should be held *unboxed*, i.e., not heap-allocated. This causes no problems for laziness because all primitive types are *unlifted*. Unboxed tuple types are not heap-allocated either.

Certain primitives and `%external` functions cause side-effects to state threads or to the real world. Where the ordering of these side-effects matters, Core already forces this order by means of data dependencies on the psuedo-values representing the threads.

The `raisezh` and `handlezh` primitives requires special support in an implementation, such as a handler stack; again, real-world threading guarantees that they will execute in the correct order.

4 Primitive Module

This section describes the contents and informal semantics of the primitive module `PrimGHC`. Nearly all the primitives are required in order to cover GHC's implementation of the Haskell98 standard prelude; the only operators that can be completely omitted are those supporting the byte-code interpreter, parallelism, and foreign objects. Some of the concurrency primitives are needed, but can be given degenerate implementations if it desired to target a purely sequential backend; see Section 4.1.1.

In addition to these primitives, a large number of C library functions are required to implement the full standard Prelude, particularly to handle I/O and arithmetic on less usual types.

4.1 Types

Type	Kind	Description
`ZLzmzgZR`	`* -> * -> *`	functions (`->`)
`Z1H`	`? -> #`	unboxed 1-tuple
`Z2H`	`? -> ? -> #`	unboxed 2-tuple
...	...	...
`Z100H`	`? -> ? -> ? -> ... -> ? -> #`	unboxed 100-tuple
`Addrzh`	`#`	machine address (pointer)
`Charzh`	`#`	unicode character (31 bits)
`Doublezh`	`#`	double-precision float
`Floatzh`	`#`	float
`Intzh`	`#`	int (30+ bits)
`Int32zh`	`#`	int (32 bits)
`Int64zh`	`#`	int (64 bits)
`Wordzh`	`#`	unsigned word (30+ bits)
`Word32zh`	`#`	unsigned word (32 bits)
`Word64zh`	`#`	unsigned word (64 bits)
`RealWorld`	`*`	pseudo-type for real world state
`Statezh`	`* -> #`	mutable state
`Arrayzh`	`* -> #`	immutable arrays
`ByteArrayzh`	`#`	immutable byte arrays
`MutableArrayzh`	`* -> * -> #`	mutable arrays
`MutableByteArrayzh`	`* -> #`	mutable byte arrays
`MutVarzh`	`* -> * -> #`	mutable variables
`MVarzh`	`* -> * -> #`	synchronized mutable variables
`Weakzh`	`* -> #`	weak pointers
`StablePtrzh`	`* -> #`	stable pointers
`ForeignObjzh`	`#`	foreign object
`ThreadIdzh`	`#`	thread id
`ZCTCCallable`	`? -> *`	dictionaries for CCallable pseudo-class
`ZCTCReturnable`	`? -> *`	dictionaries for CReturnable pseudo-class

In addition, the types `PrelBase.Bool` and `PrelBase.Unit`, which are non-primitive and are defined as ordinary algebraic types in module `PrelBase`, are used in the types of some operators in `PrelGHC`.

The unboxed tuple types are quite special: they hold sets of values in an unlifted context, i.e., to be manipulated directly rather than being stored in the heap. They can only appear in limited contexts in programs; in particular, they cannot be bound by a lambda abstraction or case alternative pattern. Note that they can hold either lifted or unlifted values. The limitation to 100-tuples is an arbitrary one set by GHC.

The type of arbitrary precision integers (`Integer`) is not primitive; it is made up of an ordinary primitive integer (`Intzh`) and a byte array (`ByteArrzh`). The components of an `Integer` are passed to primitive operators as two separate arguments and returned as an unboxed pair.

The `Statezh` type constructor takes a dummy type argument that is used only to distinguish different state *threads* [Launchbury and Peyton Jones, 1994]. The `RealWorld` type is used only as an argument to `Statezh`, and represents the thread of real-world state; it contains just the single value `realWorldzh`. The mutable data types `MutableArrayzh`,`MutableByteArrayzh`,`MutVarzh` take an initial type argument of the form (`Statezh` t) for some thread t. The synchronized mutable variable type constructor `MVarzh` always takes an argument of type `Statezh RealWorld`.

`Weakzh` is the type of weak pointers.

`StablePtrzh` is the type of stable pointers, which are guaranteed not to move during garbage collections; these are useful in connection with foreign functions.

`ForeignPtrzh` is the type of foreign pointers.

The dictionary types `ZCTCCallable` and `ZCTCReturnable` are just placeholders which can be represented by a void type; any code they appear in should be unreachable.

4.1.1 Non-concurrent Back End

The Haskell98 standard prelude doesn't include any concurrency support, but GHC's implementation of it relies on the existence of some concurrency primitives. However, it never actually forks multiple threads. Hence, the concurrency primitives can be given degenerate implementations that will work in a non-concurrent setting, as follows:

- `ThreadIdzh` can be represented by a singleton type, whose (unique) value is returned by `myThreadIdzh`.
- `forkzh` can just die with an "unimplemented" message.
- `killThreadzh` and `yieldzh` can also just die "unimplemented" since in a one-thread world, the only thread a thread can kill is itself, and if a thread yields the program hangs.
- `MVarzh a` can be represented by `MutVarzh (Maybe a)`; where a concurrent implementation would block, the sequential implementation can just die with a suitable message (since no other thread exists to unblock it).
- `waitReadzh` and `waitWritezh` can be implemented using a `select` with no timeout.

4.2 Literals

Only the following combination of literal forms and types are permitted:

Literal form	Type	Description
integer	`Intzh`	Int
	`Wordzh`	Word
	`Addrzh`	Address
	`Charzh`	Unicode character code
rational	`Floatzh`	Float
	`Doublezh`	Double
character	`Charzh`	Unicode character specified by ASCII character
string	`Addrzh`	Address of specified C-format string

4.3 Data Constructors

The only primitive data constructors are for unboxed tuples:

Constructor	Type	Description
`ZdwZ1H`	`%forall (a::?).a -> Z1H a`	unboxed 1-tuple
`ZdwZ2H`	`%forall (a1::?) (a2::?).a1 -> a2 -> Z2H a1 a2`	unboxed 2-tuple
...	...	...
`ZdwZ100H`	`%forall (a1::?) (a2::?)... (a100::?) .`	
	`a1 -> a2 -> ... -> a100 -> Z100H a1 a2 ... a100`	unboxed 100-tuple

4.4 Values

Operators are (roughly) divided into collections according to the primary type on which they operate.

Working note: *How do primitives fail, e.g., on division by zero or attempting an invalid narrowing coercion?*

Working note: *The following primop descriptions are automatically generated. The exact set of primops and their types presented here depends on the underlying word size at the time of generation; these were done for 32 bit words. This is a bit stupid. More importantly, the word size has a big impact on just what gets produced in a Core file, but this isn't documented anywhere in the file itself. Perhaps there should be a global flag in the file?*

Unless otherwise noted, each primop has the following default characteristics: Has no side effects. Implemented in line. Not commutable. Needs no wrapper. Cannot fail.

4.4.1 The word size story.

Haskell98 specifies that signed integers (type `Int`) must contain at least 30 bits. GHC always implements `Int` using the primitive type `Int#`, whose size equals the `MachDeps.h` constant `WORD_SIZE_IN_BITS`. This is normally set based on the `config.h` parameter `SIZEOF_LONG`, i.e., 32 bits on 32-bit machines, 64 bits on 64-bit machines. However, it can also be explicitly set to a smaller number, e.g., 31 bits, to allow the possibility of using tag bits. Currently GHC itself has only 32-bit and 64-bit variants, but 30 or 31-bit code can be exported as an external core file for use in other back ends.

GHC also implements a primitive unsigned integer type `Word#` which always has the same number of bits as `Int#`.

In addition, GHC supports families of explicit-sized integers and words at 8, 16, 32, and 64 bits, with the usual arithmetic operations, comparisons, and a range of conversions. The 8-bit and 16-bit sizes are always represented as `Int#` and `Word#`, and the operations implemented in terms of the the primops on these types, with suitable range restrictions on the results (using the `narrow`*n*`Int#` and `narrow`*n*`Word#` families of primops. The 32-bit sizes are represented using `Int#` and `Word#` when `WORD_SIZE_IN_BITS` ≥ 32; otherwise, these are represented using distinct primitive types `Int32#` and `Word32#`. These (when needed) have a complete set of corresponding operations; however, nearly all of these are implemented as external C functions rather than as primops. Exactly the same story applies to the 64-bit sizes. All of these details are hidden under the `PrelInt` and `PrelWord` modules, which use `#if`-defs to invoke the appropriate types and operators.

Word size also matters for the families of primops for indexing/reading/writing fixed-size quantities at offsets from an array base, address, or foreign pointer. Here, a slightly different approach is taken. The names of these primops are fixed, but their *types* vary according to the value of `WORD_SIZE_IN_BITS`. For example, if word size is at least 32 bits then an operator like `indexInt32Array#` has type `ByteArr# -> Int# -> Int#`; otherwise it has type `ByteArr# -> Int# -> Int32#`. This approach confines the necessary `#if`-defs to this file; no conditional compilation is needed in the files that expose these primops, namely `lib/std/PrelStorable.lhs`, `hslibs/lang/ArrayBase.hs`, and (in deprecated fashion) in `hslibs/lang/ForeignObj.lhs` and `hslibs/lang/Addr.lhs`.

Finally, there are strongly deprecated primops for coercing between `Addr#`, the primitive type of machine addresses, and `Int#`. These are pretty bogus anyway, but will work on existing 32-bit and 64-bit GHC targets; they are completely bogus when tag bits are used in `Int#`, so are not available in this case.

4.4.2 Char#

Operations on 31-bit characters.

```
gtCharzh ::  Charzh -> Charzh -> Bool
```

```
geCharzh ::  Charzh -> Charzh -> Bool
```

```
eqCharzh ::  Charzh -> Charzh -> Bool
```
Commutable.

```
neCharzh ::  Charzh -> Charzh -> Bool
```
Commutable.

```
ltCharzh ::  Charzh -> Charzh -> Bool
```

```
leCharzh ::  Charzh -> Charzh -> Bool
```

`ordzh :: Charzh -> Intzh`

4.4.3 Int#

Operations on native-size integers (30+ bits).

`zpzh :: Intzh -> Intzh -> Intzh`
Commutable.

`zmzh :: Intzh -> Intzh -> Intzh`

`ztzh :: Intzh -> Intzh -> Intzh`
Commutable.

`quotIntzh :: Intzh -> Intzh -> Intzh`
Rounds towards zero. Can fail.

`remIntzh :: Intzh -> Intzh -> Intzh`
Satisfies `(quotInt# x y) *# y +# (remInt# x y) == x`. Can fail.

`gcdIntzh :: Intzh -> Intzh -> Intzh`

`negateIntzh :: Intzh -> Intzh`

`addIntCzh :: Intzh -> Intzh -> Z2H Intzh Intzh`
Add with carry. First member of result is (wrapped) sum; second member is 0 iff no overflow occured.

`subIntCzh :: Intzh -> Intzh -> Z2H Intzh Intzh`
Subtract with carry. First member of result is (wrapped) difference; second member is 0 iff no overflow occured.

`mulIntCzh :: Intzh -> Intzh -> Z2H Intzh Intzh`
Multiply with carry. First member of result is (wrapped) product; second member is 0 iff no overflow occured.

`zgzh :: Intzh -> Intzh -> Bool`

`zgzezh :: Intzh -> Intzh -> Bool`

`zezezh :: Intzh -> Intzh -> Bool`
Commutable.

`zszezh :: Intzh -> Intzh -> Bool`
Commutable.

`zlzh :: Intzh -> Intzh -> Bool`

`zlzezh :: Intzh -> Intzh -> Bool`

`chrzh :: Intzh -> Charzh`

`int2Wordzh :: Intzh -> Wordzh`

`int2Floatzh :: Intzh -> Floatzh`

`int2Doublezh :: Intzh -> Doublezh`

`int2Integerzh :: Intzh -> Z2H Intzh ByteArrzh`
Implemented out of line.

`iShiftLzh :: Intzh -> Intzh -> Intzh`
Shift left. Return 0 if shifted by more than size of an Int#.

`iShiftRAzh :: Intzh -> Intzh -> Intzh`
Shift right arithemetic. Return 0 if shifted by more than size of an Int#.

`iShiftRLzh :: Intzh -> Intzh -> Intzh`
Shift right logical. Return 0 if shifted by more than size of an Int#.

4.4.4 Word#

Operations on native-sized unsigned words (30+ bits).

`plusWordzh :: Wordzh -> Wordzh -> Wordzh`
Commutable.

`minusWordzh :: Wordzh -> Wordzh -> Wordzh`

`timesWordzh :: Wordzh -> Wordzh -> Wordzh`
Commutable.

`quotWordzh :: Wordzh -> Wordzh -> Wordzh`
Can fail.

`remWordzh :: Wordzh -> Wordzh -> Wordzh`
Can fail.

`andzh :: Wordzh -> Wordzh -> Wordzh`
Commutable.

`orzh :: Wordzh -> Wordzh -> Wordzh`
Commutable.

`xorzh :: Wordzh -> Wordzh -> Wordzh`
Commutable.

`notzh :: Wordzh -> Wordzh`

`shiftLzh :: Wordzh -> Intzh -> Wordzh`
Shift left logical. Return 0 if shifted by more than number of bits in a Word#.

`shiftRLzh :: Wordzh -> Intzh -> Wordzh`
Shift right logical. Return 0 if shifted by more than number of bits in a Word#.

`word2Intzh :: Wordzh -> Intzh`

`word2Integerzh :: Wordzh -> Z2H Intzh ByteArrzh`
Implemented out of line.

`gtWordzh :: Wordzh -> Wordzh -> Bool`

`geWordzh :: Wordzh -> Wordzh -> Bool`

`eqWordzh :: Wordzh -> Wordzh -> Bool`

`neWordzh :: Wordzh -> Wordzh -> Bool`

`ltWordzh :: Wordzh -> Wordzh -> Bool`

`leWordzh :: Wordzh -> Wordzh -> Bool`

4.4.5 Narrowings

Explicit narrowing of native-sized ints or words.

`narrow8Intzh :: Intzh -> Intzh`

`narrow16Intzh :: Intzh -> Intzh`

`narrow32Intzh :: Intzh -> Intzh`

`narrow8Wordzh :: Wordzh -> Wordzh`

`narrow16Wordzh :: Wordzh -> Wordzh`

`narrow32Wordzh :: Wordzh -> Wordzh`

4.4.6 Int64#

Operations on 64-bit unsigned words. This type is only used if plain Int# has less than 64 bits. In any case, the operations are not primops; they are implemented (if needed) as ccalls instead.

`int64ToIntegerzh :: Int64zh -> Z2H Intzh ByteArrzh`
Implemented out of line.

4.4.7 Word64#

Operations on 64-bit unsigned words. This type is only used if plain Word# has less than 64 bits. In any case, the operations are not primops; they are implemented (if needed) as ccalls instead.

`word64ToIntegerzh ::  Word64zh -> Z2H Intzh ByteArrzh`
Implemented out of line.

4.4.8 Integer#

Operations on arbitrary-precision integers. These operations are implemented via the GMP package. An integer is represented as a pair consisting of an Int# representing the number of 'limbs' in use and the sign, and a ByteArr# containing the 'limbs' themselves. Such pairs are returned as unboxed pairs, but must be passed as separate components.

`plusIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Implemented out of line. Commutable.

`minusIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Implemented out of line.

`timesIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Implemented out of line. Commutable.

`gcdIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Greatest common divisor. Implemented out of line. Commutable.

`gcdIntegerIntzh ::  Intzh -> ByteArrzh -> Intzh -> Intzh`
Greatest common divisor, where second argument is an ordinary Int#.

`divExactIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Divisor is guaranteed to be a factor of dividend. Implemented out of line.

`quotIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Rounds towards zero. Implemented out of line.

`remIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Satisfies `plusInteger# (timesInteger# (quotInteger# x y) y) (remInteger# x y) == x`. Implemented out of line.

`cmpIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Intzh`
Returns -1,0,1 according as first argument is less than, equal to, or greater than second argument. Needs wrapper.

`cmpIntegerIntzh ::  Intzh -> ByteArrzh -> Intzh -> Intzh`
Returns -1,0,1 according as first argument is less than, equal to, or greater than second argument, which is an ordinary Int#. Needs wrapper.

`quotRemIntegerzh ::  Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z4H Intzh ByteArrzh Intzh ByteArrzh`

Compute quot and rem simulaneously. Implemented out of line. Can fail.

`divModIntegerzh :: Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z4H Intzh ByteArrzh Intzh ByteArrzh`
Compute div and mod simultaneously, where div rounds towards negative infinity and`(q,r) = divModInteger#(x,y)` implies `plusInteger# (timesInteger# q y) r = x`. Implemented out of line. Can fail.

`integer2Intzh :: Intzh -> ByteArrzh -> Intzh`
Needs wrapper.

`integer2Wordzh :: Intzh -> ByteArrzh -> Wordzh`
Needs wrapper.

`integerToInt64zh :: Intzh -> ByteArrzh -> Int64zh`

`integerToWord64zh :: Intzh -> ByteArrzh -> Word64zh`

`andIntegerzh :: Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Implemented out of line.

`orIntegerzh :: Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Implemented out of line.

`xorIntegerzh :: Intzh -> ByteArrzh -> Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Implemented out of line.

`complementIntegerzh :: Intzh -> ByteArrzh -> Z2H Intzh ByteArrzh`
Implemented out of line.

4.4.9 Double#

Operations on double-precision (64 bit) floating-point numbers.

`zgzhzh :: Doublezh -> Doublezh -> Bool`

`zgzezhzh :: Doublezh -> Doublezh -> Bool`

`zezezhzh :: Doublezh -> Doublezh -> Bool`
Commutable.

`zszezhzh :: Doublezh -> Doublezh -> Bool`
Commutable.

`zlzhzh :: Doublezh -> Doublezh -> Bool`

`zlzezhzh :: Doublezh -> Doublezh -> Bool`

`zpzhzh :: Doublezh -> Doublezh -> Doublezh`
Commutable.

`zmzhzh :: Doublezh -> Doublezh -> Doublezh`

`ztzhzh :: Doublezh -> Doublezh -> Doublezh`
Commutable.

`zszhzh :: Doublezh -> Doublezh -> Doublezh`
Can fail.

`negateDoublezh :: Doublezh -> Doublezh`

`double2Intzh :: Doublezh -> Intzh`

`double2Floatzh :: Doublezh -> Floatzh`

`expDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`logDoublezh :: Doublezh -> Doublezh`
Needs wrapper. Can fail.

`sqrtDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`sinDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`cosDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`tanDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`asinDoublezh :: Doublezh -> Doublezh`
Needs wrapper. Can fail.

`acosDoublezh :: Doublezh -> Doublezh`
Needs wrapper. Can fail.

`atanDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`sinhDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`coshDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`tanhDoublezh :: Doublezh -> Doublezh`
Needs wrapper.

`ztztzhzh :: Doublezh -> Doublezh -> Doublezh`
Exponentiation. Needs wrapper.

`decodeDoublezh :: Doublezh -> Z3H Intzh Intzh ByteArrzh`
Convert to arbitrary-precision integer. First Int# in result is the exponent; second Int# and ByteArr# represent an Integer# holding the mantissa. Implemented out of line.

4.4.10 Float#

Operations on single-precision (32-bit) floating-point numbers.

`gtFloatzh :: Floatzh -> Floatzh -> Bool`

`geFloatzh :: Floatzh -> Floatzh -> Bool`

`eqFloatzh :: Floatzh -> Floatzh -> Bool`
Commutable.

`neFloatzh :: Floatzh -> Floatzh -> Bool`
Commutable.

`ltFloatzh :: Floatzh -> Floatzh -> Bool`

`leFloatzh :: Floatzh -> Floatzh -> Bool`

`plusFloatzh :: Floatzh -> Floatzh -> Floatzh`
Commutable.

`minusFloatzh :: Floatzh -> Floatzh -> Floatzh`

`timesFloatzh :: Floatzh -> Floatzh -> Floatzh`
Commutable.

`divideFloatzh :: Floatzh -> Floatzh -> Floatzh`
Can fail.

`negateFloatzh :: Floatzh -> Floatzh`

`float2Intzh :: Floatzh -> Intzh`

`expFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`logFloatzh :: Floatzh -> Floatzh`
Needs wrapper. Can fail.

`sqrtFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`sinFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`cosFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`tanFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`asinFloatzh :: Floatzh -> Floatzh`
Needs wrapper. Can fail.

`acosFloatzh :: Floatzh -> Floatzh`
Needs wrapper. Can fail.

`atanFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`sinhFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`coshFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`tanhFloatzh :: Floatzh -> Floatzh`
Needs wrapper.

`powerFloatzh :: Floatzh -> Floatzh -> Floatzh`
Needs wrapper.

`float2Doublezh :: Floatzh -> Doublezh`

`decodeFloatzh :: Floatzh -> Z3H Intzh Intzh ByteArrzh`
Convert to arbitrary-precision integer. First Int# in result is the exponent; second Int# and ByteArr# represent an Integer# holding the mantissa. Implemented out of line.

4.4.11 Arrays

Operations on Array#.

`newArrayzh :: %forall a s . Intzh -> a -> Statezh s -> Z2H (Statezh s) (MutArrzh s a)`
Create a new mutable array of specified size (in bytes), in the specified state thread, with each element containing the specified initial value. Implemented out of line.

`sameMutableArrayzh :: %forall s a . MutArrzh s a -> MutArrzh s a -> Bool`

`readArrayzh :: %forall s a . MutArrzh s a -> Intzh -> Statezh s -> Z2H (Statezh s) a`
Read from specified index of mutable array. Result is not yet evaluated.

`writeArrayzh :: %forall s a . MutArrzh s a -> Intzh -> a -> Statezh s -> Statezh s`
Write to specified index of mutable array. Has side effects.

`indexArrayzh :: %forall a . Arrayzh a -> Intzh -> Z1H a`
Read from specified index of immutable array. Result is packaged into an unboxed singleton; the result itself is not yet evaluated.

`unsafeFreezzeArrayzh :: %forall s a . MutArrzh s a -> Statezh s -> Z2H (Statezh s) (Arrayzh a)`

Make a mutable array immutable, without copying. Has side effects.

`unsafeThawArrayzh :: %forall a s . Arrayzh a -> Statezh s -> Z2H (Statezh s) (MutArrzh s a)`
Make an immutable array mutable, without copying. Implemented out of line.

4.4.12 Byte Arrays

Operations on ByteArray#. A ByteArray# is a just a region of raw memory in the garbage-collected heap, which is not scanned for pointers. It carries its own size (in bytes). There are three sets of operations for accessing byte array contents: index for reading from immutable byte arrays, and read/write for mutable byte arrays. Each set contains operations for a range of useful primitive data types. Each operation takes an offset measured in terms of the size fo the primitive type being read or written.

`newByteArrayzh :: %forall s . Intzh -> Statezh s -> Z2H (Statezh s) (MutByteArrzh s)`
Create a new mutable byte array of specified size (in bytes), in the specified state thread. Implemented out of line.

`newPinnedByteArrayzh :: %forall s . Intzh -> Statezh s -> Z2H (Statezh s) (MutByteArrzh s)`
Create a mutable byte array that the GC guarantees not to move. Implemented out of line.

`byteArrayContentszh :: ByteArrzh -> Addrzh`
Intended for use with pinned arrays; otherwise very unsafe!

`sameMutableByteArrayzh :: %forall s . MutByteArrzh s -> MutByteArrzh s -> Bool`

`unsafeFreezzeByteArrayzh :: %forall s . MutByteArrzh s -> Statezh s -> Z2H (Statezh s) ByteArrzh`
Make a mutable byte array immutable, without copying. Has side effects.

`sizzeofByteArrayzh :: ByteArrzh -> Intzh`

`sizzeofMutableByteArrayzh :: %forall s . MutByteArrzh s -> Intzh`

`indexCharArrayzh :: ByteArrzh -> Intzh -> Charzh`
Read 8-bit character; offset in bytes.

`indexWideCharArrayzh :: ByteArrzh -> Intzh -> Charzh`
Read 31-bit character; offset in 4-byte words.

`indexIntArrayzh :: ByteArrzh -> Intzh -> Intzh`

`indexWordArrayzh :: ByteArrzh -> Intzh -> Wordzh`

`indexAddrArrayzh :: ByteArrzh -> Intzh -> Addrzh`

`indexFloatArrayzh :: ByteArrzh -> Intzh -> Floatzh` `indexDoubleArrayzh :: ByteArrzh -> Intzh -> Doublezh`

`indexStablePtrArrayzh :: %forall a . ByteArrzh -> Intzh -> StablePtrzh a`

`indexInt8Arrayzh :: ByteArrzh -> Intzh -> Intzh`

`indexInt16Arrayzh :: ByteArrzh -> Intzh -> Intzh`

`indexInt32Arrayzh :: ByteArrzh -> Intzh -> Intzh`

`indexInt64Arrayzh :: ByteArrzh -> Intzh -> Int64zh`

`indexWord8Arrayzh :: ByteArrzh -> Intzh -> Wordzh`

`indexWord16Arrayzh :: ByteArrzh -> Intzh -> Wordzh`

`indexWord32Arrayzh :: ByteArrzh -> Intzh -> Wordzh`

`indexWord64Arrayzh :: ByteArrzh -> Intzh -> Word64zh`

`readCharArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Charzh`
Read 8-bit character; offset in bytes.

`readWideCharArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Charzh`
Read 31-bit character; offset in 4-byte words.

`readIntArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Intzh`

`readWordArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Wordzh`

`readAddrArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Addrzh`

`readFloatArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Floatzh`

`readDoubleArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Doublezh` `readStablePtrArrayzh :: %forall s a . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) (StablePtrzh a)`

`readInt8Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Intzh`

`readInt16Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Intzh`

`readInt32Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Intzh`

`readInt64Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Int64zh`

`readWord8Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Wordzh`

`readWord16Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Wordzh`

`readWord32Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Wordzh`

`readWord64Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Statezh s -> Z2H (Statezh s) Word64zh`

`writeCharArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Charzh -> Statezh s -> Statezh s`
Write 8-bit character; offset in bytes. Has side effects.

`writeWideCharArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Charzh -> Statezh s -> Statezh s`
Write 31-bit character; offset in 4-byte words. Has side effects.

`writeIntArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeWordArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeAddrArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Addrzh -> Statezh s -> Statezh s`
Has side effects.

`writeFloatArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Floatzh -> Statezh s -> Statezh s`
Has side effects.

`writeDoubleArrayzh :: %forall s . MutByteArrzh s -> Intzh -> Doublezh -> Statezh s -> Statezh s`
Has side effects.

`writeStablePtrArrayzh :: %forall s a . MutByteArrzh s -> Intzh -> StablePtrzh a -> Statezh s -> Statezh s`
Has side effects.

`writeInt8Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeInt16Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeInt32Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeInt64Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Int64zh -> Statezh s -> Statezh s`
Has side effects.

`writeWord8Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeWord16Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeWord32Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeWord64Arrayzh :: %forall s . MutByteArrzh s -> Intzh -> Word64zh -> Statezh s -> Statezh s`
Has side effects.

4.4.13 Addr#

Addr# is an arbitrary machine address assumed to point outside the garbage-collected heap.

`nullAddrzh :: Intzh -> Addrzh`
Returns null address. Argument is ignored (nullary primops don't quite work!)

`plusAddrzh :: Addrzh -> Intzh -> Addrzh`

`minusAddrzh :: Addrzh -> Addrzh -> Intzh`
Result is meaningless if two Addr#s are so far apart that their difference doesn't fit in an Int#.

`remAddrzh :: Addrzh -> Intzh -> Intzh`
Return the remainder when the Addr# arg, treated like an Int#, is divided by the Int# arg.

`addr2Intzh :: Addrzh -> Intzh`
Coerce directly from address to int. Strongly deprecated.

`int2Addrzh :: Intzh -> Addrzh`
Coerce directly from int to address. Strongly deprecated.

`gtAddrzh :: Addrzh -> Addrzh -> Bool`

`geAddrzh :: Addrzh -> Addrzh -> Bool`

`eqAddrzh :: Addrzh -> Addrzh -> Bool`

`neAddrzh :: Addrzh -> Addrzh -> Bool`

`ltAddrzh :: Addrzh -> Addrzh -> Bool`

`leAddrzh :: Addrzh -> Addrzh -> Bool`

`indexCharOffAddrzh :: Addrzh -> Intzh -> Charzh`
Reads 8-bit character; offset in bytes.

`indexWideCharOffAddrzh :: Addrzh -> Intzh -> Charzh`
Reads 31-bit character; offset in 4-byte words.

`indexIntOffAddrzh :: Addrzh -> Intzh -> Intzh`

`indexWordOffAddrzh :: Addrzh -> Intzh -> Wordzh`

`indexAddrOffAddrzh :: Addrzh -> Intzh -> Addrzh`

`indexFloatOffAddrzh :: Addrzh -> Intzh -> Floatzh`

`indexDoubleOffAddrzh :: Addrzh -> Intzh -> Doublezh`

`indexStablePtrOffAddrzh :: %forall a . Addrzh -> Intzh -> StablePtrzh a`

`indexInt8OffAddrzh :: Addrzh -> Intzh -> Intzh`

`indexInt16OffAddrzh :: Addrzh -> Intzh -> Intzh`

```
indexInt32OffAddrzh :: Addrzh -> Intzh -> Intzh
```

```
indexInt64OffAddrzh :: Addrzh -> Intzh -> Int64zh
```

```
indexWord8OffAddrzh :: Addrzh -> Intzh -> Wordzh
```

```
indexWord16OffAddrzh :: Addrzh -> Intzh -> Wordzh
```

```
indexWord32OffAddrzh :: Addrzh -> Intzh -> Wordzh
```

```
indexWord64OffAddrzh :: Addrzh -> Intzh -> Word64zh
```

```
readCharOffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh
s) Charzh
```
Reads 8-bit character; offset in bytes.

```
readWideCharOffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H
(Statezh s) Charzh
```
Reads 31-bit character; offset in 4-byte words.

```
readIntOffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh
s) Intzh
```

```
readWordOffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh
s) Wordzh
```

```
readAddrOffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh
s) Addrzh
```

```
readFloatOffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H
(Statezh s) Floatzh
```

```
readDoubleOffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H
(Statezh s) Doublezh
```

```
readStablePtrOffAddrzh :: %forall s a . Addrzh -> Intzh -> Statezh s -> Z2H
(Statezh s) (StablePtrzh a)
```

```
readInt8OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh
s) Intzh
```

```
readInt16OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H
(Statezh s) Intzh
```

`readInt32OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh s) Intzh`

`readInt64OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh s) Int64zh`

`readWord8OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh s) Wordzh`

`readWord16OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh s) Wordzh`

`readWord32OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh s) Wordzh`

`readWord64OffAddrzh :: %forall s . Addrzh -> Intzh -> Statezh s -> Z2H (Statezh s) Word64zh`

`writeCharOffAddrzh :: %forall s . Addrzh -> Intzh -> Charzh -> Statezh s -> Statezh s`
Has side effects.

`writeWideCharOffAddrzh :: %forall s . Addrzh -> Intzh -> Charzh -> Statezh s -> Statezh s`
Has side effects.

`writeIntOffAddrzh :: %forall s . Addrzh -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeWordOffAddrzh :: %forall s . Addrzh -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeAddrOffAddrzh :: %forall s . Addrzh -> Intzh -> Addrzh -> Statezh s -> Statezh s`
Has side effects.

`writeForeignObjOffAddrzh :: %forall s . Addrzh -> Intzh -> ForeignObjzh -> Statezh s -> Statezh s`
Has side effects.

`writeFloatOffAddrzh :: %forall s . Addrzh -> Intzh -> Floatzh -> Statezh s -> Statezh s`
Has side effects.

`writeDoubleOffAddrzh :: %forall s . Addrzh -> Intzh -> Doublezh -> Statezh s -> Statezh s`
Has side effects.

`writeStablePtrOffAddrzh :: %forall a s . Addrzh -> Intzh -> StablePtrzh a -> Statezh s -> Statezh s`

Has side effects.

`writeInt8OffAddrzh :: %forall s . Addrzh -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeInt16OffAddrzh :: %forall s . Addrzh -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeInt32OffAddrzh :: %forall s . Addrzh -> Intzh -> Intzh -> Statezh s -> Statezh s`
Has side effects.

`writeInt64OffAddrzh :: %forall s . Addrzh -> Intzh -> Int64zh -> Statezh s -> Statezh s`
Has side effects.

`writeWord8OffAddrzh :: %forall s . Addrzh -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeWord16OffAddrzh :: %forall s . Addrzh -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeWord32OffAddrzh :: %forall s . Addrzh -> Intzh -> Wordzh -> Statezh s -> Statezh s`
Has side effects.

`writeWord64OffAddrzh :: %forall s . Addrzh -> Intzh -> Word64zh -> Statezh s -> Statezh s`
Has side effects.

4.4.14 ForeignObj#

Operations on ForeignObj#. The indexing operations are all deprecated.

`mkForeignObjzh :: Addrzh -> Statezh RealWorld -> Z2H (Statezh RealWorld) ForeignObjzh`
Has side effects. Implemented out of line.

`writeForeignObjzh :: %forall s . ForeignObjzh -> Addrzh -> Statezh s -> Statezh s`
Has side effects.

`foreignObjToAddrzh :: ForeignObjzh -> Addrzh`

`touchzh :: %forall (o::?) . o -> Statezh RealWorld -> Statezh RealWorld`
Has side effects.

`eqForeignObjzh :: ForeignObjzh -> ForeignObjzh -> Bool`
Commutable.

`indexCharOffForeignObjzh :: ForeignObjzh -> Intzh -> Charzh`
Read 8-bit character; offset in bytes.

`indexWideCharOffForeignObjzh :: ForeignObjzh -> Intzh -> Charzh`
Read 31-bit character; offset in 4-byte words.

`indexIntOffForeignObjzh :: ForeignObjzh -> Intzh -> Intzh`

`indexWordOffForeignObjzh :: ForeignObjzh -> Intzh -> Wordzh`

`indexAddrOffForeignObjzh :: ForeignObjzh -> Intzh -> Addrzh`

`indexFloatOffForeignObjzh :: ForeignObjzh -> Intzh -> Floatzh`

`indexDoubleOffForeignObjzh :: ForeignObjzh -> Intzh -> Doublezh`

`indexStablePtrOffForeignObjzh :: %forall a . ForeignObjzh -> Intzh -> StablePtrzh a`

`indexInt8OffForeignObjzh :: ForeignObjzh -> Intzh -> Intzh`

`indexInt16OffForeignObjzh :: ForeignObjzh -> Intzh -> Intzh`

`indexInt32OffForeignObjzh :: ForeignObjzh -> Intzh -> Intzh`

`indexInt64OffForeignObjzh :: ForeignObjzh -> Intzh -> Int64zh`

`indexWord8OffForeignObjzh :: ForeignObjzh -> Intzh -> Wordzh`

`indexWord16OffForeignObjzh :: ForeignObjzh -> Intzh -> Wordzh`

`indexWord32OffForeignObjzh :: ForeignObjzh -> Intzh -> Wordzh`

`indexWord64OffForeignObjzh :: ForeignObjzh -> Intzh -> Word64zh`

4.4.15 Mutable variables

Operations on MutVar#s, which behave like single-element mutable arrays.

`newMutVarzh :: %forall a s . a -> Statezh s -> Z2H (Statezh s) (MutVarzh s a)`
Create MutVar# with specified initial value in specified state thread. Implemented out of line.

`readMutVarzh :: %forall s a . MutVarzh s a -> Statezh s -> Z2H (Statezh s) a`
Read contents of MutVar#. Result is not yet evaluated.

```
writeMutVarzh :: %forall s a . MutVarzh s a -> a -> Statezh s -> Statezh s
```
Write contents of MutVar#. Has side effects.

```
sameMutVarzh :: %forall s a . MutVarzh s a -> MutVarzh s a -> Bool
```

4.4.16 Exceptions

```
catchzh :: %forall a b . (Statezh RealWorld -> Z2H (Statezh RealWorld) a) ->
(b -> Statezh RealWorld -> Z2H (Statezh RealWorld) a) -> Statezh RealWorld ->
Z2H (Statezh RealWorld) a
```
Implemented out of line.

```
raisezh :: %forall a b . a -> b
```
Implemented out of line.

```
blockAsyncExceptionszh :: %forall a . (Statezh RealWorld -> Z2H (Statezh
RealWorld) a) -> Statezh RealWorld -> Z2H (Statezh RealWorld) a
```
Implemented out of line.

```
unblockAsyncExceptionszh :: %forall a . (Statezh RealWorld -> Z2H (Statezh
RealWorld) a) -> Statezh RealWorld -> Z2H (Statezh RealWorld) a
```
Implemented out of line.

4.4.17 Synchronized Mutable Variables

Operations on MVar#s, which are shared mutable variables (*not* the same as MutVar#s!). (Note: in a non-concurrent implementation, (MVar# a) can be represented by (MutVar# (Maybe a)).)

```
newMVarzh :: %forall s a . Statezh s -> Z2H (Statezh s) (MVarzh s a)
```
Create new mvar; initially empty. Implemented out of line.

```
takeMVarzh :: %forall s a . MVarzh s a -> Statezh s -> Z2H (Statezh s) a
```
If mvar is empty, block until it becomes full. Then remove and return its contents, and set it empty. Has side effects. Implemented out of line.

```
tryTakeMVarzh :: %forall s a . MVarzh s a -> Statezh s -> Z3H (Statezh s)
Intzh a
```
If mvar is empty, immediately return with integer 0 and value undefined. Otherwise, return with integer 1 and contents of mvar, and set mvar empty. Has side effects. Implemented out of line.

```
putMVarzh :: %forall s a . MVarzh s a -> a -> Statezh s -> Statezh s
```
If mvar is full, block until it becomes empty. Then store value arg as its new contents. Has side effects. Implemented out of line.

```
tryPutMVarzh :: %forall s a . MVarzh s a -> a -> Statezh s -> Z2H (Statezh s)
Intzh
```
If mvar is full, immediately return with integer 0. Otherwise, store value arg as mvar's new contents, and return with integer 1. Has side effects. Implemented out of line.

```
sameMVarzh :: %forall s a . MVarzh s a -> MVarzh s a -> Bool
```

```
isEmptyMVarzh :: %forall s a . MVarzh s a -> Statezh s -> Z2H (Statezh s)
Intzh
```

Return 1 if mvar is empty; 0 otherwise.

4.4.18 Delay/wait operations

`delayzh :: %forall s . Intzh -> Statezh s -> Statezh s`
Sleep specified number of microseconds. Has side effects. Implemented out of line. Needs wrapper.

`waitReadzh :: %forall s . Intzh -> Statezh s -> Statezh s`
Block until input is available on specified file descriptor. Has side effects. Implemented out of line. Needs wrapper.

`waitWritezh :: %forall s . Intzh -> Statezh s -> Statezh s`
Block until output is possible on specified file descriptor. Has side effects. Implemented out of line. Needs wrapper.

4.4.19 Concurrency primitives

(In a non-concurrent implementation, ThreadId# can be as singleton type, whose (unique) value is returned by myThreadId#. The other operations can be omitted.)

`forkzh :: %forall a . a -> Statezh RealWorld -> Z2H (Statezh RealWorld) ThreadIdzh`
Has side effects. Implemented out of line.

`killThreadzh :: %forall a . ThreadIdzh -> a -> Statezh RealWorld -> Statezh RealWorld`
Has side effects. Implemented out of line.

`yieldzh :: Statezh RealWorld -> Statezh RealWorld`
Has side effects. Implemented out of line.

`myThreadIdzh :: Statezh RealWorld -> Z2H (Statezh RealWorld) ThreadIdzh`

4.4.20 Weak pointers

`mkWeakzh :: %forall (o::?) b c . o -> b -> c -> Statezh RealWorld -> Z2H (Statezh RealWorld) (Weakzh b)`
Has side effects. Implemented out of line.

`deRefWeakzh :: %forall a . Weakzh a -> Statezh RealWorld -> Z3H (Statezh RealWorld) Intzh a`
Has side effects.

`finalizzeWeakzh :: %forall a . Weakzh a -> Statezh RealWorld -> Z3H (Statezh RealWorld) Intzh (Statezh RealWorld -> Z2H (Statezh RealWorld) Unit)`
Has side effects. Implemented out of line.

4.4.21 Stable pointers and names

`makeStablePtrzh :: %forall a . a -> Statezh RealWorld -> Z2H (Statezh RealWorld) (StablePtrzh a)`

Has side effects.

`deRefStablePtrzh :: %forall a . StablePtrzh a -> Statezh RealWorld -> Z2H (Statezh RealWorld) a`
Has side effects. Needs wrapper.

`eqStablePtrzh :: %forall a . StablePtrzh a -> StablePtrzh a -> Intzh`
Has side effects.

`makeStableNamezh :: %forall a . a -> Statezh RealWorld -> Z2H (Statezh RealWorld) (StableNamezh a)`
Has side effects. Implemented out of line. Needs wrapper.

`eqStableNamezh :: %forall a . StableNamezh a -> StableNamezh a -> Intzh`

`stableNameToIntzh :: %forall a . StableNamezh a -> Intzh`

4.4.22 Unsafe pointer equality

`reallyUnsafePtrEqualityzh :: %forall a . a -> a -> Intzh`

4.4.23 Parallelism

`seqzh :: %forall a . a -> Intzh`
Has side effects.

`parzh :: %forall a . a -> Intzh`
Has side effects.

`parGlobalzh :: %forall a b . a -> Intzh -> Intzh -> Intzh -> Intzh -> b -> Intzh`
Has side effects.

`parLocalzh :: %forall a b . a -> Intzh -> Intzh -> Intzh -> Intzh -> b -> Intzh`
Has side effects.

`parAtzh :: %forall b a c . b -> a -> Intzh -> Intzh -> Intzh -> Intzh -> c -> Intzh`
Has side effects.

`parAtAbszh :: %forall a b . a -> Intzh -> Intzh -> Intzh -> Intzh -> Intzh -> b -> Intzh`
Has side effects.

`parAtRelzh :: %forall a b . a -> Intzh -> Intzh -> Intzh -> Intzh -> Intzh -> b -> Intzh`
Has side effects.

`parAtForNowzh :: %forall b a c . b -> a -> Intzh -> Intzh -> Intzh -> Intzh -> c -> Intzh`
Has side effects.

4.4.24 Tag to enum stuff

Convert back and forth between values of enumerated types and small integers.

```
dataToTagzh ::  %forall a .  a -> Intzh
```

```
tagToEnumzh ::  %forall a .  Intzh -> a
```

4.4.25 Bytecode operations

Support for the bytecode interpreter and linker.

```
addrToHValuezh ::  %forall a .  Addrzh -> Z1H a
```
Convert an Addr# to a followable type.

```
mkApUpd0zh ::  %forall a .  a -> Z1H a
```
Implemented out of line.

```
newBCOzh ::  %forall a s .  ByteArrzh -> ByteArrzh -> Arrayzh a -> ByteArrzh ->
Statezh s -> Z2H (Statezh s) BCOzh
```
Has side effects. Implemented out of line.

4.4.26 RealWorld

There is just one value of type `RealWorld`, namely `realWorldzh`. It is used only for dependency threading of side-effecting operations.

References

[Launchbury and Peyton Jones, 1994] Launchbury, J. and Peyton Jones, S. (1994). Lazy functional state threads. Technical report FP-94-05, Department of Computing Science, University of Glasgow.

[Peyton Jones and Launchbury, 1991] Peyton Jones, S. and Launchbury, J. (1991). Unboxed values as first class citizens. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, pages 636–666, Boston. ACM.

[Peyton Jones and Marlow, 1999] Peyton Jones, S. and Marlow, S. (1999). Secrets of the Glasgow Haskell Compiler inliner. In *Workshop on Implementing Declarative Languages*, Paris, France.

[Peyton Jones and Santos, 1998] Peyton Jones, S. and Santos, A. (1998). A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47.