DRAFT: An External Representation for the GHC Core Language

Andrew Tolmach
apt@cs.pdx.edu

June 4, 2001

Abstract

This document proposes a precise definition for the GHC Core language, so that it can be used to communicate between GHC and new stand-along compilation tools such as back-ends or optimizers. The definition includes a formal grammar, an informal semantics, a set of typing rules, and a formal operational semantics.

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, would 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 van 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.)

Currently Core has 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;
- An informal semantics (Section 3;
- A formal static semantics in the form of typing rules (Section 5);
- A formal operational semantics (Section 6).

2 External Grammar of Core

In designing the external grammar, I 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, I have kept the grammar 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.

> ***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 | pat_2	choice
`fibonacci`	terminal syntax in typewriter font

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

```
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                 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.

By default, all identifiers defined in top-level declarations can be referenced from any other module in the program, using conventional dot notation (e.g., `PrelBase.Bool`, `PrelBase.True`). However, top-level values can optionally be tagged as `%local`, in which case they are only visible within the defining module. There are 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.

3.3 Types and Kinds

In Core, all type abstractions and applications are explicit. This make it easy to typecheck any (closed) fragment. Full typing rules are given in Section 5.

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., `Int#`) 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 modules generated from user code; the other two are needed to describe certain types in primitive (or otherwise specially-generated) modules. A type is lifted if and only if it has bottom as an element. In particular, any value that is to be operationally represented by a closure has lifted type; any unboxed value is unlifted. Currently, the unlifted types are coextensive with the primitive types (including a few boxed primitive types such as `ByteArray#`). 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, and modulo the identity

a `->` $b \equiv$ `PrelGHC.ZLzmzgZR` a b

Any uses of type constructors introduced by `newtypes` are marked in terms by explicit coercions (see Section 3.5).

3.4 Algebraic data types

Each `data` declaration introduces a new distinct 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 an associated representation type, normally corresponding to a source Haskell `newtype` declaration. However, unlike in source Haskell, no data constructors are introduced; 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 Haskell fragments

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

might induce the Core fragments

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

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.

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. 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) x
```

Value applications may be of user-defined functions, data constructors, or primitives. Applications of the latter two sorts need *not* be saturated.

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.

In any sort of `%case`, the list of alternatives may conclude with a default alternative labeled with an underscore (`%_`), which will be chosen if none of the other alternative match. At least one alternative must always be given. 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 and literal cases, the set of alternatives need not be exhaustive, even if no default is given; it is a disasterous run-time error if a needed case arm is missing.

The `%coerce` expression is primarily used in conjunction with manipulation of newtypes, as described in Section 3.5. Coercions of this sort can be checked for type correctness by reference to the right-hand side of the relevant `%newtype` declaration. 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. Uses of this sort cannot be type-checked 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. Notes currently supported include the inlining pragma (`InlineMe`) cost-center labels for profiling.

A `%ccall` expression denotes an external call to a named C function, which has indicated type (always expressed in terms of Haskell primitive types).

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 given in Section 6

Evaluating a Core expression means reducing it to *weak-head normal form (WHNF)*, i.e., a primitive value, lambda abstraction, primitive function (unapplied or partially applied), or data construction (partially or fully applied). 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.
- *WHNFs* representing the result of evaluating such thunks.

All elements in the heap are initially thunks; they are allocated when it is necessary to suspend a computation whose result may be shared. Allocation 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 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*.

Certain primitives and `%ccall`'s 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 `raise#` and `handle#` 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`. We include only those GHC primitives that are needed to cover GHC's implementation of the Haskell98 standard prelude; due to the organization of GHC's libraries, there are more of these than you might expect. (Whenever one member of a set of similar operations was required, we have chosen to include the entire set.) The only sets of primitives we omit are those for supporting parallel runtime systems and for stable pointers. The primitives for supporting concurrent runtime systems *are* required, but can be given degenerate definitions, as described below.

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. We do not list these separately.

4.1 Types

Type	Kind	Description
`ZLzmzgZR`	`* -> * -> *`	functions (`->`)
`ZL#z32U#ZR`	`? -> #`	unboxed 1-tuple
`Z1U`	`? -> ? -> #`	unboxed 2-tuple
`Z2U`	`? -> ? -> ? -> #`	unboxed 3-tuple
...	...	...
`Z99U`	`? -> ? -> ? -> ... -> ? -> #`	unboxed 100-tuple
`Addr#`	`#`	machine address (pointer)
`Char#`	`#`	unicode character (31+ bits)
`Double#`	`#`	double-precision float
`Float#`	`#`	float
`Int#`	`#`	int (32+ bits)
`Int64#`	`#`	int (64+ bits)
`Word#`	`#`	unsigned word (32+ bits)
`Word64#`	`#`	unsigned word (64+ bits)
`RealWorld`	`*`	pseudo-type for real world state
`State#`	`* -> #`	mutable state
`Array#`	`* -> #`	immutable arrays
`ByteArray#`	`#`	immutable byte arrays
`MutableArray#`	`* -> * -> #`	mutable arrays
`MutableByteArray#`	`* -> #`	mutable byte arrays
`MutVar#`	`* -> * -> #`	mutable variables
`Mvar#`	`* -> * -> #`	synchronized mutable variables
`Weak#`	`* -> #`	weak pointers
`ForeignObj#`	`#`	foreign object pointer
`ThreadId#`	`#`	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 (as specified by the type-checking rules of Section 5), notably as the return type of functions. Note that they can hold either lifted or unlifted values. The limitation to 100-tuples is an arbitrary one set by GHC.

> ***Working note:*** *The naming of tuples is unfortunate. The number in the name counts represents the number of commas in the tuple representation rather than the arity of the tuple, so appears to be "off by one." The odd name of the 1-tuple is due to a GHC bug.*

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

The `State#` 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 `State#`, and represents the thread of real-world state; it contains just the single value `RealWorld`. The various mutable data types (`MutableArray#`,`MutableByteArray#`,`MutVar#`, and `MVar#`) take an initial type argument of the form (`State#`t) for some thread t.

In a non-concurrent setting, `MVar#`'s can be implemented as `MutVar#`'s, and `ThreadId#` can be represented by a singleton type.

`Weak#` is the type of weak pointers.

`ForeignObj#` is the type of foreign object 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.

4.2 Literals

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

Literal form	Type	Description
integer	`Int#`	Int
	`Int64#`	Int64
	`Word#`	Word
	`Word64#`	Word64
	`Addr#`	Address
rational	`Float#`	Float
	`Double#`	Double
character	`Char#`	Unicode character
string	`Addr#`	Address of specified C-format string

4.3 Data Constructors

The only primitive data constructors are for unboxed tuples:

Constructor	Type	Description
`ZDWZL#z43U#ZR`	`%forall (a::?).a -> ZL#z32U#ZR a`	unboxed 1-tuple
`ZDWZ1H`	`%forall (a1::?) (a2::?).a1 -> a2 -> Z1H a1 a2`	unboxed 2-tuple
...	...	...
`ZDWZ99H`	`%forall (a1::?) (a2::?)... (a100::?) . a1 -> a2 -> ... -> a100 -> Z99H 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?*

4.4.1 General

Operation	Type	Meaning
`realWorld#`	`State# RealWorld`	the sole value of type `RealWorld`
`seq#`	`%forall a . a -> Int#`	just return any non-0 value!
`dataToTag#`	`%forall a . a -> Int#`	??
`tagToEnum#`	`%forall a . Int# -> a`	??
`getTag#`	`%forall a . a -> Int#`	??
`unsafeCoerce#`	`%forall a b . a -> b`	very unsafe!

> ***Working note:*** *Explain data/tag/enum operators.*

4.4.2 Addr#

Operation	Type	Meaning
`gtAddr#`	`Addr# -> Addr# -> PrelBase.Bool`	greater
`geAddr#`	`Addr# -> Addr# -> PrelBase.Bool`	greater or equal
`eqAddr#`	`Addr# -> Addr# -> PrelBase.Bool`	equal
`neAddr#`	`Addr# -> Addr# -> PrelBase.Bool`	not equal
`ltAddr#`	`Addr# -> Addr# -> PrelBase.Bool`	less
`leAddr#`	`Addr# -> Addr# -> PrelBase.Bool`	less or equal
`addr2Int#`	`Addr# -> Int#`	coerce

4.4.3 Char#

Operation	Type	Meaning
`gtChar#`	`Char# -> Char# -> PrelBase.Bool`	greater
`geChar#`	`Char# -> Char# -> PrelBase.Bool`	greater or equal
`eqChar#`	`Char# -> Char# -> PrelBase.Bool`	equal
`neChar#`	`Char# -> Char# -> PrelBase.Bool`	not equal
`ltChar#`	`Char# -> Char# -> PrelBase.Bool`	less
`leChar#`	`Char# -> Char# -> PrelBase.Bool`	less or equal
`ord#`	`Char# -> Int#`	convert

4.4.4 Double#

Operation	Type	Meaning
`zg##`	`Double# -> Double# -> PrelBase.Bool`	greater
`zgze##`	`Double# -> Double# -> PrelBase.Bool`	greater or equal
`zeze##`	`Double# -> Double# -> PrelBase.Bool`	equal
`zsze##`	`Double# -> Double# -> PrelBase.Bool`	not equal
`zl##`	`Double# -> Double# -> PrelBase.Bool`	less
`zlze##`	`Double# -> Double# -> PrelBase.Bool`	less or equal
`zp##`	`Double# -> Double# -> Double#`	add
`zm##`	`Double# -> Double# -> Double#`	subtract
`zt##`	`Double# -> Double# -> Double#`	multiply
`zs##`	`Double# -> Double# -> Double#`	divide
`negateDouble#`	`Double# -> Double#`	negate
`double2Int#`	`Double# -> Int#`	convert
`double2Float#`	`Double# -> Float#`	convert
`expDouble#`	`Double# -> Double#`	exp
`logDouble#`	`Double# -> Double#`	log
`sqrtDouble#`	`Double# -> Double#`	sqrt
`sinDouble#`	`Double# -> Double#`	sin
`cosDouble#`	`Double# -> Double#`	cos
`tanDouble#`	`Double# -> Double#`	tan
`asinDouble#`	`Double# -> Double#`	asin
`acosDouble#`	`Double# -> Double#`	acos
`atanDouble#`	`Double# -> Double#`	atan
`sinhDouble#`	`Double# -> Double#`	sinh
`coshDouble#`	`Double# -> Double#`	cosh
`tanhDouble#`	`Double# -> Double#`	tanh
`ztzt##`	`Double# -> Double# -> Double#`	exponentiation
`decodeDouble#`	`Double# -> Z2U Int# Int# ByteArray#`	conversion

4.4.5 Float#

Operation	Type	Meaning
`gtFloat#`	`Float# -> Float# -> PrelBase.Bool`	greater
`geFloat#`	`Float# -> Float# -> PrelBase.Bool`	greater or equal
`eqFloat#`	`Float# -> Float# -> PrelBase.Bool`	equal
`neFloat#`	`Float# -> Float# -> PrelBase.Bool`	not equal
`ltFloat#`	`Float# -> Float# -> PrelBase.Bool`	less
`leFloat#`	`Float# -> Float# -> PrelBase.Bool`	less or equal
`plusFloat#`	`Float# -> Float# -> Float#`	add
`minusFloat#`	`Float# -> Float# -> Float#`	subtract
`timesFloat#`	`Float# -> Float# -> Float#`	multiply
`divideFloat#`	`Float# -> Float# -> Float#`	divide
`negateFloat#`	`Float# -> Float#`	negate
`float2Int#`	`Float# -> Int#`	convert
`expFloat#`	`Float# -> Float#`	exp
`logFloat#`	`Float# -> Float#`	log
`sqrtFloat#`	`Float# -> Float#`	sqrt
`sinFloat#`	`Float# -> Float#`	sin
`cosFloat#`	`Float# -> Float#`	cos
`tanFloat#`	`Float# -> Float#`	tan
`asinFloat#`	`Float# -> Float#`	asin
`acosFloat#`	`Float# -> Float#`	acos
`atanFloat#`	`Float# -> Float#`	atan
`sinhFloat#`	`Float# -> Float#`	asinh
`coshFloat#`	`Float# -> Float#`	acosh
`tanhFloat#`	`Float# -> Float#`	atanh
`powerFloat#`	`Float# -> Float# -> Float#`	exponentiation
`float2Double#`	`Float# -> Double#`	convert
`decodeFloat#`	`Float# -> Z2U Int# Int# ByteArr#`	decode??

Working note: *What does decode do?*

4.4.6 Int#

Operation	Type	Meaning
`zp#`	`Int# -> Int# -> Int#`	add
`zm#`	`Int# -> Int# -> Int#`	subtract
`zt#`	`Int# -> Int# -> Int#`	multiply
`quotInt#`	`Int# -> Int# -> Int#`	divide
`remInt#`	`Int# -> Int# -> Int#`	remainder
`gcdInt#`	`Int# -> Int# -> Int#`	greatest common divisor
`negateInt#`	`Int# -> Int#`	negate
`addIntC#`	`Int# -> Int# -> Z1U Int# Int#`	??
`subIntC#`	`Int# -> Int# -> Z1U Int# Int#`	??
`mulIntC#`	`Int# -> Int# -> Z1U Int# Int#`	??
`zg#`	`Int# -> Int# -> PrelBase.Bool`	greater
`zgze#`	`Int# -> Int# -> PrelBase.Bool`	greater or equal
`zeze#`	`Int# -> Int# -> PrelBase.Bool`	equal
`zsze#`	`Int# -> Int# -> PrelBase.Bool`	not equal
`zl#`	`Int# -> Int# -> PrelBase.Bool`	less
`zlze#`	`Int# -> Int# -> PrelBase.Bool`	less or equal
`chr#`	`Int# -> Char#`	convert
`int2Word#`	`Int# -> Word#`	convert
`int2Addr#`	`Int# -> Addr#`	convert
`int2Float#`	`Int# -> Float#`	convert
`int2Double#`	`Int# -> Double#`	convert
`int2Integer#`	`Int# -> Z1U Int# ByteArr#`	convert
`iShiftL#`	`Int# -> Int# -> Int#`	shift left
`iShiftRA#`	`Int# -> Int# -> Int#`	shift right arithmetic
`iShiftRL#`	`Int# -> Int# -> Int#`	shift right logical

Working note: *What do the* `IntC` *operators do?*

4.4.7 Int64#

Operation	Type	Meaning
`int64ToInteger#`	`Int64# -> Z1U Int# ByteArr#`	convert

Other `Int64#` operations are accomplished using `%ccall`.

4.4.8 Integer

Operation	Type	Meaning
plusInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	add
minusInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	subtract
timesInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	multiply
gcdInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	greatest common divisor
gcdIntegerInt#	Int# -> ByteArr# -> Int# -> Int#	greatest common divisor
divExactInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	divide
quotInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	divide
remInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	remainder
cmpInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Int#	compare: return -1,0,1
cmpIntegerInt#	Int# -> ByteArr# -> Int# -> Int#	compare: return -1,0,1
quotRemInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z3U Int# ByteArr# Int# ByteArr#	quotient and remainder
divModInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z3U Int# ByteArr# Int# ByteArr#	div and mod
integer2Int#	Int# -> ByteArr# -> Int#	convert
integer2Word#	Int# -> ByteArr# -> Word#	convert
integerToInt64#	Int# -> ByteArr# -> Int64#	convert
integerToWord64#	Int# -> ByteArr# -> Word64#	convert
andInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	bitwise and
orInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	bitwise or
xorInteger#	Int# -> ByteArr# -> Int# -> ByteArr# -> Z1U Int# ByteArr#	bitwise xor
complementInteger#	Int# -> ByteArr# -> Z1U Int# ByteArr#	bitwise complement

> ***Working note:*** *What are the relationships between quotient,remainder, div, mod?*

4.4.9 Word#

Operation	Type	Meaning
plusWord#	Word# -> Word# -> Word#	add
minusWord#	Word# -> Word# -> Word#	subtract
timesWord#	Word# -> Word# -> Word#	multiply
quotWord#	Word# -> Word# -> Word#	divide
remWord#	Word# -> Word# -> Word#	remainder
and#	Word# -> Word# -> Word#	bitwise and
or#	Word# -> Word# -> Word#	bitwise or
xor#	Word# -> Word# -> Word#	bitwise xor
not#	Word# -> Word#	bitwise complement
shiftL#	Word# -> Int# -> Word#	shift left
shiftRL#	Word# -> Int# -> Word#	shift right logical
word2Int#	Word# -> Int#	convert
word2Integer#	Word# -> Z1U Int# ByteArr#	convert
gtWord#	Word# -> Word# -> PrelBase.Bool	greater
geWord#	Word# -> Word# -> PrelBase.Bool	greater or equal
eqWord#	Word# -> Word# -> PrelBase.Bool	equal
neWord#	Word# -> Word# -> PrelBase.Bool	not equal
ltWord#	Word# -> Word# -> PrelBase.Bool	less
leWord#	Word# -> Word# -> PrelBase.Bool	less or equal

4.4.10 Word64#

Operation	Type	Meaning
word64ToInteger#	Word64# -> Z1U Int# ByteArr#	convert

Other `Word64#` operations are accomplished using `%ccall`.

4.4.11 Explicitly sized Int# and Word#

Operation	Type	Meaning
intToInt8#	Int# -> Int#	truncate
intToInt16#	Int# -> Int#	truncate
intToInt32#	Int# -> Int#	truncate
wordToWord8#	Word# -> Word#	truncate
wordToWord16#	Word# -> Word#	truncate
wordToWord32#	Word# -> Word#	truncate

> ***Working note:*** *What do these operators do?*

4.4.12 Arrays

Operation	Type	Meaning
newArray#	%forall a s. Int# -> a -> State# s -> Z1U (State# s) (MutArr# s a)	create mutable array
newByteArray#	%forall s Int# -> State# s -> Z1U (State# s) (MutByteArr# s)	create mutable byte array
sameMutableArray#	%forall s a . MutArr# s a -> MutArr# s a -> PrelBase.Bool	same array?
sameMutableByteArray#	%forall s . MutByteArr# s -> MutByteArr# s -> PrelBase.Bool	same byte array?
readArray#	%forall s a . MutArr# s a -> Int# -> State# s -> Z1U (State# s) a	read
writeArray#	%forall s a . MutArr# s a -> Int# -> a -> State# s -> State# s	write
indexArray#	%forall a . Array# a -> Int# -> ZL#z32U#ZR a	fetch but don't evaluate
unsafeFreezeArray#	%forall s a . MutArr# s a -> State# s -> Z1U (State# s) (Array# a)	coerce
unsafeFreezeByteArray#	%forall s . MutByteArr# s -> State# s -> Z1U (State# s) ByteArr#	coerce
unsafeThawArray#	%forall a s Array# a -> State# s -> Z1U (State# s) (MutArr# s a)	coerce
sizeofByteArray#	ByteArr# -> Int#	length
sizeofMutableByteArray#	%forall s . MutByteArr# s -> Int#	length

> ***Working note:*** *What does* `indexArray#` *do?*

The following are all reads from immutable byte arrays.

> ***Working note:*** *For these and similar operations below: presumably we read from 1-8 bytes, as needed by return type? Must offets be aligned?*

Operation	Type
indexCharArray#	ByteArr# -> Int# -> Char#
indexWideCharArray#	ByteArr# -> Int# -> Char#
indexIntArray#	ByteArr# -> Int# -> Int#
indexWordArray#	ByteArr# -> Int# -> Word#
indexAddrArray#	ByteArr# -> Int# -> Addr#
indexFloatArray#	ByteArr# -> Int# -> Float#
indexDoubleArray#	ByteArr# -> Int# -> Double#
indexInt8Array#	ByteArr# -> Int# -> Int#
indexInt16Array#	ByteArr# -> Int# -> Int#
indexInt32Array#	ByteArr# -> Int# -> Int#
indexInt64Array#	ByteArr# -> Int# -> Int64#
indexWord8Array#	ByteArr# -> Int# -> Word#
indexWord16Array#	ByteArr# -> Int# -> Word#
indexWord32Array#	ByteArr# -> Int# -> Word#
indexWord64Array#	ByteArr# -> Int# -> Word64#

The following are all reads from mutable byte arrays.

Operation	Type
readCharArray#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Char#
readWideCharArray#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Char#
readIntArray#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Int#
readWordArray#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Word#
readAddrArray#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Addr#
readFloatArray#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Float#
readDoubleArray#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Double#
readInt8Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Int#
readInt16Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Int#
readInt32Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Int#
readInt64Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Int64#
readWord8Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Word#
readWord16Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Word#
readWord32Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Word#
readWord64Array#	%forall s . MutByteArr# s -> Int# -> State# s -> Z1U (State# s) Word64#

The following are all store operations on mutable byte arrays.

Operation	Type
writeCharArray#	%forall s . MutByteArr# s -> Int# -> Char# -> State# s -> State# s
writeWideCharArray#	%forall s . MutByteArr# s -> Int# -> Char# -> State# s -> State# s
writeIntArray#	%forall s . MutByteArr# s -> Int# -> Int# -> State# s -> State# s
writeWordArray#	%forall s . MutByteArr# s -> Int# -> Word# -> State# s -> State# s
writeAddrArray#	%forall s . MutByteArr# s -> Int# -> Addr# -> State# s -> State# s
writeFloatArray#	%forall s . MutByteArr# s -> Int# -> Float# -> State# s -> State# s
writeDoubleArray#	%forall s . MutByteArr# s -> Int# -> Double# -> State# s -> State# s
writeInt8Array#	%forall s . MutByteArr# s -> Int# -> Int# -> State# s -> State# s
writeInt16Array#	%forall s . MutByteArr# s -> Int# -> Int# -> State# s -> State# s
writeInt32Array#	%forall s . MutByteArr# s -> Int# -> Int# -> State# s -> State# s
writeInt64Array#	%forall s . MutByteArr# s -> Int# -> Int64# -> State# s -> State# s
writeWord8Array#	%forall s . MutByteArr# s -> Int# -> Word# -> State# s -> State# s
writeWord16Array#	%forall s . MutByteArr# s -> Int# -> Word# -> State# s -> State# s
writeWord32Array#	%forall s . MutByteArr# s -> Int# -> Word# -> State# s -> State# s
writeWord64Array#	%forall s . MutByteArr# s -> Int# -> Word64# -> State# s -> State# s

The following are all reads of immutable locations relative to machine addresses.

Operation	Type
indexCharOffAddr#	Addr# -> Int# -> Char#
indexWideCharOffAddr#	Addr# -> Int# -> Char#
indexIntOffAddr#	Addr# -> Int# -> Int#
indexWordOffAddr#	Addr# -> Int# -> Word#
indexAddrOffAddr#	Addr# -> Int# -> Addr#
indexFloatOffAddr#	Addr# -> Int# -> Float#
indexDoubleOffAddr#	Addr# -> Int# -> Double#
indexInt8OffAddr#	Addr# -> Int# -> Int#
indexInt16OffAddr#	Addr# -> Int# -> Int#
indexInt32OffAddr#	Addr# -> Int# -> Int#
indexInt64OffAddr#	Addr# -> Int# -> Int64#
indexWord8OffAddr#	Addr# -> Int# -> Word#
indexWord16OffAddr#	Addr# -> Int# -> Word#
indexWord32OffAddr#	Addr# -> Int# -> Word#
indexWord64OffAddr#	Addr# -> Int# -> Word64#

The following are all reads from immutable locations operations relative to foreign object addresses.

Operation	Type
indexCharOffForeignObj#	ForeignObj# -> Int# -> Char#
indexWideCharOffForeignObj#	ForeignObj# -> Int# -> Char#
indexIntOffForeignObj#	ForeignObj# -> Int# -> Int#
indexWordOffForeignObj#	ForeignObj# -> Int# -> Word#
indexAddrOffForeignObj#	ForeignObj# -> Int# -> Addr#
indexFloatOffForeignObj#	ForeignObj# -> Int# -> Float#
indexDoubleOffForeignObj#	ForeignObj# -> Int# -> Double#
indexInt8OffForeignObj#	ForeignObj# -> Int# -> Int#
indexInt16OffForeignObj#	ForeignObj# -> Int# -> Int#
indexInt32OffForeignObj#	ForeignObj# -> Int# -> Int#
indexInt64OffForeignObj#	ForeignObj# -> Int# -> Int64#
indexWord8OffForeignObj#	ForeignObj# -> Int# -> Word#
indexWord16OffForeignObj#	ForeignObj# -> Int# -> Word#
indexWord32OffForeignObj#	ForeignObj# -> Int# -> Word#
indexWord64OffForeignObj#	ForeignObj# -> Int# -> Word64#

The following are all reads from mutable locations relative to machine addresses.

Operation	Type
readCharOffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Char#
readWideCharOffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Char#
readIntOffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Int#
readWordOffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Word#
readAddrOffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Addr#
readFloatOffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Float#
readDoubleOffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Double#
readInt8OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Int#
readInt16OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Int#
readInt32OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Int#
readInt64OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Int64#
readWord8OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Word#
readWord16OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Word#
readWord32OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Word#
readWord64OffAddr#	%forall s . Addr# -> Int# -> State# s -> Z1U (State# s) Word64#

The following are all writes to mutable locations relative to machine addresses.

Operation	Type
writeCharOffAddr#	%forall s . Addr# -> Int# -> Char# -> State# s -> State# s
writeWideCharOffAddr#	%forall s . Addr# -> Int# -> Char# -> State# s -> State# s
writeIntOffAddr#	%forall s . Addr# -> Int# -> Int# -> State# s -> State# s
writeWordOffAddr#	%forall s . Addr# -> Int# -> Word# -> State# s -> State# s
writeAddrOffAddr#	%forall s . Addr# -> Int# -> Addr# -> State# s -> State# s
writeForeignObjOffAddr#	%forall s . Addr# -> Int# -> ForeignObj# -> State# s -> State# s
writeFloatOffAddr#	%forall s . Addr# -> Int# -> Float# -> State# s -> State# s
writeDoubleOffAddr#	%forall s . Addr# -> Int# -> Double# -> State# s -> State# s
writeInt8OffAddr#	%forall s . Addr# -> Int# -> Int# -> State# s -> State# s
writeInt16OffAddr#	%forall s . Addr# -> Int# -> Int# -> State# s -> State# s
writeInt32OffAddr#	%forall s . Addr# -> Int# -> Int# -> State# s -> State# s
writeInt64OffAddr#	%forall s . Addr# -> Int# -> Int64# -> State# s -> State# s
writeWord8OffAddr#	%forall s . Addr# -> Int# -> Word# -> State# s -> State# s
writeWord16OffAddr#	%forall s . Addr# -> Int# -> Word# -> State# s -> State# s
writeWord32OffAddr#	%forall s . Addr# -> Int# -> Word# -> State# s -> State# s
writeWord64OffAddr#	%forall s . Addr# -> Int# -> Word64# -> State# s -> State# s

4.4.13 MutVars

Mutable variables are just like single-element arrays.

Operation	Type	Meaning
`newMutVar#`	`%forall a s . a -> State# s -> Z1U (State# s) (MutVar# s a)`	create new var
`readMutVar#`	`%forall s a . MutVar# s a -> State# s -> Z1U (State# s) a`	read
`writeMutVar#`	`%forall s a . MutVar# s a -> a -> State# s -> State# s`	write
`sameMutVar#`	`%forall s a . MutVar# s a -> MutVar# s a -> PrelBase.Bool`	same var?

4.4.14 Exceptions

Working note: *These need proper explanation.*

Operation	Type	Meaning

`catch#`	`%forall a b . (State# (RealWorld) -> Z1U (State# (RealWorld)) a)`
	`-> (b -> State# (RealWorld) -> Z1U (State# (RealWorld)) a)`
	`-> State# (RealWorld) -> Z1U (State# (RealWorld)) a`
		handle

`raise#`	`%forall a b . a -> b`	raise
`blockAsyncExceptions#`	`%forall a . (State# (RealWorld) -> Z1U (State# (RealWorld)) a)` `-> State# (RealWorld) -> Z1U (State# (RealWorld)) a`	
`unblockAsyncExceptions#`	`%forall a . (State# (RealWorld) -> Z1U (State# (RealWorld)) a)` `-> State# (RealWorld) -> Z1U (State# (RealWorld)) a`	

4.4.15 MVars

In a non-concurrent setting, `MVar a` can be represented by `MutVar (Maybe a)`. Where a concurrent implementation of `takeMVar` (resp. `putMVar`) would block because the MVar is empty (resp. full), the sequential implementation can just die. The `Int#` values returned by the testing primitives represent booleans, where 0 means False and non-0 means True.

Operation	Type	Meaning
`newMVar#`	`%forall s a . State# s -> Z1U (State# s) (MVar# s a)`	create
`takeMVar#`	`%forall s a . MVar# s a -> State# s -> Z1U (State# s) a`	get contents; die if empty
`tryTakeMVar#`	`%forall s a . MVar# s a -> State# s -> Z2U (State# s) (Int#) a`	get contents if possible
`putMVar#`	`%forall s a . MVar# s a -> a -> State# s -> State# s`	put contents; die if full
`tryPutMVar#`	`%forall s a . MVar# s a -> a -> State# s -> Z1U (State# s) (Int#)`	put contents if possible
`sameMVar#`	`%forall s a . MVar# s a -> MVar# s a -> Bool`	same?
`isEmptyMVar#`	`%forall s a . MVar# s a -> State# s -> Z1U (State# s) (Int#)`	empty?

4.4.16 Concurrency

In a non-concurrent setting, these can be given trivial implementations as indicated.

Working note: *Presumably in a non-concurrent setting* `fork#` *should fail because we don't know how, and* `killThread#` *should fail because the only thread we can kill is us? What kind of failure is meant?*

Operation	Type	Meaning
`delay#`	`%forall s . Int# -> State# s -> State# s`	sleep specified number of microseconds
`waitRead#`	`%forall s . Int# -> State# s -> State# s`	block until input available on specified fd
`waitWrite#`	`%forall s . Int# -> State# s -> State# s`	block until output possible on specified fd
`fork#`	`%forall a . a -> State# (RealWorld)` `-> Z1U (State# (RealWorld)) (ThreadId#)`	fail??
`killThread#`	`%forall a . ThreadId# -> a -> State# (RealWorld)` `-> State# (RealWorld)`	fail??
`yield#`	`State# (RealWorld) -> State# (RealWorld)`	no-op
`myThreadId#`	`State# (RealWorld) -> Z1U (State# (RealWorld)) (ThreadId#)`	return single value of this type

4.4.17 Foreign Objects

Working note: *Why do we need these? Are they really any different from* `Addr#`*'s?*

Operation	Type	Meaning
`mkForeignObj#`	`Addr# -> State# (RealWorld) -> Z1U (State# (RealWorld)) (ForeignObj#)`	
`writeForeignObj#`	`%forall s . ForeignObj# -> Addr# -> State# s -> State# s`	
`foreignObjToAddr#`	`ForeignObj# -> Addr#`	
`touch#`	`%forall (o::?) . o -> State# (RealWorld) -> State# (RealWorld)`	

4.4.18 Weak objects

> ***Working note:*** *These need proper explanation. We don't really need them except to wrap around* `ForeignObj#`*'s in order to make user-level* `ForeignPtr`*'s.*

Operation	Type	Meaning
`mkWeak#`	`%forall (o::*) b (c::*) . o -> b -> c -> State# (RealWorld)` `-> Z1U (State# (RealWorld)) (Weak# b)`	
`deRefWeak#`	`%forall a . Weak# a -> State# (RealWorld)` `-> Z2U (State# (RealWorld)) (Int#) a`	

`finalizeWeak#`	`%forall a . Weak# a -> State# (RealWorld)`
	`-> Z2U (State# (RealWorld)) (Int#) (State# (RealWorld)`
	`-> Z1U (State# (RealWorld)) (Unit))`	

5 Static Semantics

> ***Working note:*** *To be done. A nearly-correct working type-checker exists.*

6 Dynamic Semantics

> ***Working note:*** *To be done. An interpreter describing the basic semantics exists, but leaves out some things of interest.*

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 (FTP:{\tt ftp.dcs.glasgow.ac.uk:\linebreak[1]pub/\linebreak[1]glasgow-fp/\linebreak[1]tech-reports/\linebreak[1]FP-94-05:state.ps.Z}).

[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.