# An External Representation for the GHC Core Language (For GHC 6.10)

Andrew Tolmach, Tim Chevalier ({apt,tjc}@cs.pdx.edu)
and The GHC Team

September 4, 2008

**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.[1] 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.

## 1 Introduction

The Glasgow Haskell Compiler (GHC) uses an intermediate language, called "Core," as its internal program representation within the compiler's simplification phase. Core resembles a subset of Haskell, but with explicit type annotations in the style of the polymorphic lambda calculus ($F_\omega$).

GHC's front-end translates full Haskell 98 (plus some extensions) into Core. The GHC optimizer then repeatedly transforms Core programs while preserving their meaning. A "Core Lint" pass in GHC typechecks Core in between transformation passes (at least when the user enables linting by setting a compiler flag), verifying that transformations preserve type-correctness. Finally, GHC's back-end translates Core into STG-machine code [Peyton Jones, 1992] and then into C or native code.

Two existing papers discuss the original rationale for the design and use of Core [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. In particular, with the advent of GHC support for generalized algebraic datatypes (GADTs) [Peyton Jones et al., 2006] Core was extended beyond its previous $F_\omega$-style incarnation to support type equality constraints and safe coercions, and is now based on a system known as $F_C$ [Sulzmann et al., 2007].

Researchers interested in writing just *part* of a Haskell compiler, such as a new back-end or a new optimizer pass, might like to use 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. As another example, they might like to use Core as the target language for a front-end compiler of their own design, feeding externally synthesized Core into GHC in order to take advantage of GHC's optimizer, code generator, and run-time system. Without external Core, there are two ways for compiler writers to do this: they can link their code into the GHC executable, which is an arduous

[1] 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.

Support for generating external Core (post-optimization) was originally introduced in GHC 5.02. The definition of external Core in this document reflects the version of external Core generated by the HEAD (unstable) branch of GHC as of May 3, 2008 (version 6.9), , using the compiler flag `-fext-core`. We expect that GHC 6.10 will be consistent with this definition.

process, or they can use the GHC API [Haskell Wiki, 2007] to do the same task more cleanly. Both ways require new code to be written in Haskell.

We present a precisely specified external format for Core files. The external format is text-based and human-readable, to promote interoperability and ease of use. We hope this format will make it easier for external developers to use GHC in a modular way.

It has long been true that GHC prints an ad-hoc textual representation of Core if you set certain compiler flags. But this representation is intended to be read by people who are debugging the compiler, not by other programs. Making Core into a machine-readable, bi-directional communication format requires:

1. precisely specifying the external format of Core;
2. modifying GHC to generate external Core files (post-simplification; as always, users can control the exact transformations GHC does with command-line flags);
3. modifying GHC to accept external Core files in place of Haskell source files (users will also be able to control what GHC does to those files with command-line flags).

The first two facilities will let developers couple GHC's front-end (parser, type-checker, desugarer), and optionally its optimizer, with new back-end tools. The last facility will let developers write new Core-to-Core transformations as 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) idiosyncracies 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. Indeed, for the time being, we aim to support only the first two facilities and not the third: we define and implement Core as an external format that GHC can use to communicate with external back-end tools, and defer the larger task of extending GHC to support reading this external format back in.

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).

GHC supports many type system extensions; the External Core printer built into GHC only supports some of them. However, External Core should be capable of representing any Haskell 98 program, and may be able to represent programs that require certain type system extensions as well. If a program uses unsupported features, GHC may fail to compile it to Core when the `-fext-core` flag is set, or GHC may successfully compile it to Core, but the external tools will not be able to typecheck or interpret it.

Formal static and dynamic semantics in the form of an executable typechecker and interpreter are available separately in the GHC source tree[2] under `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. Thus, we had to make some compromises. Specifically:

- In order to avoid explosion of parentheses, we support standard precedences and shortcuts for expressions, types, and kinds. Thus we had to introduce multiple non-terminals for each of these syntactic categories, and as a result, the concrete grammar is longer and more complex than the underlying abstract syntax.
- On the other hand, we have kept the grammar simpler by avoiding special syntax for tuple types and terms. Tuples (both boxed and unboxed) are treated as ordinary constructors.

[2] http://darcs.haskell.org/ghc

- All type abstractions and applications are given in full, even though some of them (e.g., for tuples) could be reconstructed; this means a parser for Core does not have to reconstruct types.[3]
- The syntax of identifiers is heavily restricted (to just alphanumerics and underscores); this again makes Core easier to parse but harder to read.

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 |

[3] 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.

| | | | | |
|---|---|---|---|---|
| Module | *module* | → | %module *mident* { *tdef* ; } { *vdefg* ; } | |
| Type defn. | *tdef* | → | %data *qtycon* { *tbind* } = { [ *cdef* { ; *cdef* } ] } | algebraic type |
| | | \| | %newtype *qtycon 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 (*aty*) *exp* %of *vbind* { *alt* { ; *alt* } } | case expression |
| | | \| | %cast *exp aty* | type coercion |
| | | \| | %note " { *char* } " *exp* | expression note |
| | | \| | %external ccall " { *char* } " *aty* | external reference |
| | | \| | %dynexternal ccall *aty* | external reference (dynamic) |
| | | \| | %label " { *char* } " | external label |
| 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* | $\rightarrow$ | *tyvar* | type variable |
| | | $\mid$ | *qtycon* | type constructor |
| | | $\mid$ | `(` *ty* `)` | nested type |
| Basic type | *bty* | $\rightarrow$ | *aty* | atomic type |
| | | $\mid$ | *bty aty* | type application |
| | | $\mid$ | `%trans` *aty aty* | transitive coercion |
| | | $\mid$ | `%sym` *aty* | symmetric coercion |
| | | $\mid$ | `%unsafe` *aty aty* | unsafe coercion |
| | | $\mid$ | `%left` *aty* | left coercion |
| | | $\mid$ | `%right` *aty* | right coercion |
| | | $\mid$ | `%inst` *aty aty* | instantiation coercion |
| Type | *ty* | $\rightarrow$ | *bty* | basic type |
| | | $\mid$ | `%forall` { *tbind* }$^+$ `.` *ty* | type abstraction |
| | | $\mid$ | *bty* `->` *ty* | arrow type construction |
| Atomic kind | *akind* | $\rightarrow$ | `*` | lifted kind |
| | | $\mid$ | `#` | unlifted kind |
| | | $\mid$ | `?` | open kind |
| | | $\mid$ | *bty* `:=:` *bty* | equality kind |
| | | $\mid$ | `(` *kind* `)` | nested kind |
| Kind | *kind* | $\rightarrow$ | *akind* | atomic kind |
| | | $\mid$ | *akind* `->` *kind* | arrow kind |
| Identifier | *mident* | $\rightarrow$ | *pname* `:` *uname* | module |
| | *tycon* | $\rightarrow$ | *uname* | type constr. |
| | *qtycon* | $\rightarrow$ | *mident* `.` *tycon* | qualified type constr. |
| | *tyvar* | $\rightarrow$ | *lname* | type variable |
| | *dcon* | $\rightarrow$ | *uname* | data constr. |
| | *qdcon* | $\rightarrow$ | *mident* `.` *dcon* | qualified data constr. |
| | *var* | $\rightarrow$ | *lname* | variable |
| | *qvar* | $\rightarrow$ | *[ mident* `.` *] var* | optionally qualified variable |
| Name | *lname* | $\rightarrow$ | *lower* { *namechar* } | |
| | *uname* | $\rightarrow$ | *upper* { *namechar* } | |
| | *pname* | $\rightarrow$ | { *namechar* }$^+$ | |
| | *namechar* | $\rightarrow$ | *lower* $\mid$ *upper* $\mid$ *digit* | |
| | *lower* | $\rightarrow$ | `a` $\mid$ `b` $\mid$ ... $\mid$ `z` $\mid$ `_` | |
| | *upper* | $\rightarrow$ | `A` $\mid$ `B` $\mid$ ... $\mid$ `Z` | |
| | *digit* | $\rightarrow$ | `0` $\mid$ `1` $\mid$ ... $\mid$ `9` | |

# 3 Informal Semantics

At the term level, Core resembles a explicitly-typed polymorphic lambda calculus ($F_\omega$), with the addition of local `let` bindings, algebraic type definitions, constructors, and `case` expressions, and primitive types, literals and operators. Its type system is richer than that of System F, supporting explicit type equality coercions and type functions. [Sulzmann et al., 2007]

In this section we concentrate on the less obvious points about Core.

## 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*. A module identifier consists of a *package name* followed by a module name, which may be hierarchical: for example, `base:GHC.Base` is the module identifier for GHC's Base module. Its name is `Base`, and it lives in the `GHC` hierarchy within the `base` package. Section 5.8 of the GHC users' guide explains package names [The GHC Team, 2008a]. In particular, note that a Core program may contain multiple modules with the same (possibly hierarchical) module name that differ in their package names. In some of the code examples that follow, we will omit package names and possibly full hierarchical module names from identifiers for brevity, but be aware that they are always required.[4]

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 a coercion name; 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 potentially 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., `base:GHC.Base.Bool`, `base:GHC.Base.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., `main:MyModule.foo = ...`); otherwise, it is internal (e.g., `bar = ...`). Note that Core's notion of an external identifier does not necessarily coincide with that of "exported" identifier in a Haskell source module. An identifier can be an external identifier in Core, but not be exported by the original Haskell source module.[5] However, if an identifier was exported by the Haskell source module, it will appear as an external name in Core.

Core modules have no explicit import or export lists. Modules may be mutually recursive. Note that because of the latter fact, GHC currently prints out the top-level bindings for every module as a single recursive group, in order to avoid keeping track of dependencies between top-level values within a module. An external Core tool could reconstruct dependencies later, of course.

There is also an implicitly-defined module `ghc-prim:GHC.Prim`, 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:Main` having an exported value called `main:ZCMain.main` of type `base:GHC.IOBase.IO a` (for some type `a`). (Note that the strangely named wrapper for `main` is the one exception to the rule that qualified names defined within a module `m` must have module name `m`.)

Many Core programs will contain library modules, such as `base:GHC.Base`, which implement parts of the Haskell standard library. In principle, these modules are ordinary Haskell modules, with no special status. In practice, the requirement on the type of `main:Main.main` implies that every program will contain a large subset of the standard library modules.

## 3.2 Namespaces

There are five distinct namespaces:

1. module identifiers (`mident`),
2. type constructors (`tycon`),
3. type variables (`tyvar`),

[4] A possible improvement to the Core syntax would be to add explicit import lists to Core modules, which could be used to specify abbrevations for long qualified names. This would make the code more human-readable.

[5] Two examples of such identifiers are: data constructors, and values that potentially appear in an unfolding. For an example of the latter, consider `Main.foo = ... Main.bar ...`, where `Main.foo` is inlineable. Since `bar` appears in `foo`'s unfolding, it is defined and referenced with an external name, even if `bar` was not exported by the original source module.

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 data and type constructors begin with an upper-case character, and that term and type variables begin with a lower-case character.

Primitive types and operators are not syntactically distinguished.

Primitive *coercion* operators, of which there are six, *are* syntactically distinguished in the grammar. This is because these coercions must be fully applied, and because distinguishing their applications in the syntax makes typechecking easier.

A given variable (type or term) may have multiple definitions within a module. However, definitions of term variables never "shadow" one another: the scope of the definition of a given variable never contains a redefinition of the same variable. Type variables may be shadowed. Thus, if a term variable has multiple definitions within a module, all those definitions must be local (let-bound). The only exception to this rule 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.

Finally, note that hierarchical module names are z-encoded in Core: for example, `base:GHC.Base.foo` is rendered as `base:GHCziBase.foo`. A parser may reconstruct the module hierarchy, or regard `GHCziBase` as a flat name.

## 3.3 Types and Kinds

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

### 3.3.1 Types

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 $\geq 0$. 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 `ghc-prim:GHC.Prim.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 `GHC.Prim` module, but have no special syntax. `%data` and `%newtype` declarations introduce additional type constructors, as described below. Type constructors are distinguished solely by name.

### 3.3.2 Coercions

A type may also be built using one of the primitive coercion operators, as described in Section 3.2. For details on the meanings of these operators, see the System FC paper [Sulzmann et al., 2007]. Also see Section 3.5 for examples of how GHC uses coercions in Core code.

### 3.3.3 Kinds

As described in the Haskell definition, it is necessary to distinguish well-formed type-expressions by classifying them into different *kinds* [Peyton Jones, 2003, p. 41]. In particular, Core explicitly records the kind of every bound type variable.

In addition, Core's kind system includes equality kinds, as in System FC [Sulzmann et al., 2007]. An application of a built-in coercion, or of a user-defined coercion as introduced by a newtype declaration, has an equality kind.

### 3.3.4 Lifted and Unlifted Types

Semantically, a type is *lifted* if and only if it has bottom as an element. We need to distinguish them because operationally, terms with lifted types may be represented by closures; terms with unlifted types must not be represented by closures, which implies that any unboxed value is necessarily unlifted. We distinguish between lifted and unlifted types by ascribing them different kinds.

Currently, all the primitive types are unlifted (including a few boxed primitive types such as `ByteArrayzh`). Peyton Jones and Launchbury [1991] described the ideas behind unboxed and unlifted types.

### 3.3.5 Type Constructors; Base Kinds and Higher Kinds

Every type constructor has a kind, depending on its arity and whether it or its arguments are lifted.

Term variables can only be assigned types that have base kinds: the base kinds are `*`,`#`, and `?`. The three base kinds distinguish the liftedness of the types they classify: `*` represents lifted types; `#` represents unlifted types; and `?` is the "open" kind, representing a type that may be either lifted or unlifted. Of these, only `*` ever appears in Core type declarations generated from user code; the other two are needed to describe certain types in primitive (or otherwise specially-generated) code (which, after optimization, could potentially appear anywhere).

In particular, no top-level identifier (except in `ghc-prim:GHC.Prim`) has a type of kind `#` or `?`.

Nullary type constructors have base kinds: for example, the type `Int` has kind `*`, and `Int#` has kind `#`.

Non-nullary type constructors have higher kinds: kinds that have 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, type variables may have higher kinds; however, much more commonly they have kind `*`, so that is the default if a type binder omits a kind.

### 3.3.6 Type Synonyms and Type Equivalence

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 $a$ `->` $b \equiv$ `ghc-prim:GHC.Prim.ZLzmzgZR` $a$ $b$

## 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 an associated representation type, corresponding to a source Haskell `newtype` declaration. However, unlike in source Haskell, a `%newtype` declaration does not introduce any data constructors.

Each `%newtype` declaration also introduces a new coercion (syntactically, just another type constructor) that implies an axiom equating the type constructor, applied to any type variables bound by the `%newtype`, to the 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 ZCCoU = Bool;
u :: U = %cast (True)
           ((%sym ZCCoU));
v :: Bool = not (%cast (u) ZCCoU);
```

The newtype declaration implies that the types `U` and `Bool` have equivalent representations, and the coercion axiom `ZCCoU` provides evidence that `U` is equivalent to `Bool`. Notice that in the body of `u`, the boolean value `True` is cast to type `U` using the primitive symmetry rule applied to `ZCCoU`: that is, using a coercion of kind `Bool :=: U`. And in the body of `v`, `u` is cast back to type `Bool` using the axiom `ZCCoU`.

Notice that the `case` in the Haskell source code above translates to a `cast` in the corresponding Core code. That is because operationally, a `case` on a value whose type is declared by a `newtype` declaration is a no-op. Unlike a `case` on any other value, such a `case` does no evaluation: its only function is to coerce its scrutinee's type.

Also notice that unlike in a previous draft version of External Core, there is no need to handle recursive newtypes specially.

## 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 the operational semantics for Core interprets strings as C-style null-terminated strings, strings should not contain embedded nulls.

In Core, value applications, type applications, value abstractions, and type abstractions are all explicit. 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 $\Lambda$ 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.

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). The scrutinee is preceded by the type of the entire `%case` expression: that is, the result type that all of the `%case` alternatives have (this is intended to make type reconstruction easier in the presence of type equality coercions).

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 ((Bintree a)) 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 (`%_`), whose right-hand side will be evaluated if none of the other alternatives match. The default is optional except for in 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, then the list of alternatives must contain a default alternative and nothing else. 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.

`%cast` is used to manipulate newtypes, as described in Section 3.5. The `%cast` expression takes an expression and a coercion: syntactically, the coercion is an arbitrary type, but it must have an equality kind. In an expression `(cast e co)`, if `e :: T` and `co` has kind `T :=: U`, then the overall expression has type `U` [GHC Wiki, 2006]. Here, `co` must be a coercion whose left-hand side is `T`.

Note that unlike the `%coerce` expression that existed in previous versions of Core, this means that `%cast` is (almost) type-safe: the coercion argument provides evidence that can be verified by a typechecker. There are still unsafe `%cast`s, corresponding to the unsafe `%coerce` construct that existed in old versions of Core, because there is a primitive unsafe coercion type that can be used to cast arbitrary types to each other. GHC uses this for such purposes as coercing the return type of a function (such as error) which is guaranteed to never return:

```
case (error "") of
  True -> 1
  False ->  2
```

becomes:

```
  %cast (error @ Bool (ZMZN @ Char))
  (%unsafe Bool Integer);
```

`%cast` has no operational meaning and is only used in typechecking.

A `%note` expression carries arbitrary internal information that GHC finds interesting. The information is 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). External Core supports two kinds of external calls: `%external` and `%dynexternal`. Only the former is supported by the current set of stand-alone Core tools. In addition, there is a `%label` construct which GHC may generate but which the Core tools do not support.

The present syntax for externals 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: for example, we ignore all type abstractions and applications. The denotational semantics of 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 [Launchbury, 1993].

This section describes the model informally; a more formal semantics is separately available as 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 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. Evaluating a program means evaluating the expression `main:ZCMain.main`.

To make sure that expression evaluation is shared, we make use of a *heap*, which contains *heap entries*. A heap entry can be:

- A *thunk*, representing an unevaluated expression, also known as a *suspension*.
- A *WHNF*, representing an evaluated expression. The result of evaluating a thunk is a WHNF. A WHNF is always a closure (corresponding to a lambda abstraction in the source program) or a data constructor application: computations over primitive types are never suspended.

*Heap pointers* point to heap entries: at different times, the same heap pointer can point to either a thunk or a WHNF, because the run-time system overwrites thunks with WHNFs as computation proceeds.

The suspended computation that a thunk represents might represent evaluating one of three different kinds of expression. The run-time system allocates a different kind of thunk depending on what kind of expression it is:

- A thunk for a value definition has a group of suspended defining expressions, along with a list of bindings between defined names and heap pointers to those suspensions. (A value definition may be a recursive group of definitions or a single non-recursive definition, and it may be top-level (global) or `let`-bound (local)).
- A thunk for a function application (where the function is user-defined) has a suspended actual argument expression, and a binding between the formal argument and a heap pointer to that suspension.
- A thunk for a constructor application has a suspended actual argument expression; the entire constructed value has a heap pointer to that suspension embedded in it.

As computation proceeds, copies of the heap pointer for a given thunk propagate through the executing program. When another computation demands the result of that thunk, the thunk is *forced*: the run-time system computes the thunk's result, yielding a WHNF, and overwrites the heap entry for the thunk with the WHNF. Now, all copies of the heap pointer point to the new heap entry: a 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

When no pointers to a heap entry (whether it is a thunk or WHNF) remain, the garbage collector can reclaim the space it uses. We assume this happens implicitly.

With the exception of functions, arrays, and mutable variables, we intend that values of all primitive types should be held *unboxed*: they should not be heap-allocated. This does not violate call-by-need semantics: all primitive types are *unlifted*, which means that values of those types must be evaluated strictly. 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 with data dependencies on the pseudo-values representing the threads.

An implementation must specially support the **`raisezh`** and **`handlezh`** primitives: for example, by using a handler stack. Again, real-world threading guarantees that they will execute in the correct order.

# 4 Primitive Module

The semantics of External Core rely on the contents and informal semantics of the primitive module `ghc-prim:GHC.Prim`. 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).

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.

For a full listing of the names and types of the primitive operators, see the GHC library documentation [The GHC Team, 2008b].

## 4.1 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 |

# References


GHC Wiki. System FC: equality constraints and coercions. `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/FC`, 2006.

Haskell Wiki. Using GHC as a library. `http://haskell.org/haskellwiki/GHC/As_a_library`, 2007.

J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993. URL `citeseer.ist.psu.edu/launchbury93natural.html`.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.

S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *Workshop on Implementing Declarative Languages*, 1999. URL `http://research.microsoft.com/Users/simonpj/Papers/inlining/inline.pdf`.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the 2006 ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM. URL `http://research.microsoft.com/Users/simonpj/papers/gadt/index.htm`.

S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992. URL `http://citeseer.ist.psu.edu/peytonjones92implementing.html`.

S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 636–666, Cambridge, Massachussets, USA, 26–28 August 1991. Springer-Verlag LNCS523. URL `http://citeseer.ist.psu.edu/jones91unboxed.html`.

S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998. URL `http://citeseer.ist.psu.edu/peytonjones98transformationbased.html`.

M. Sulzmann, M. M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 53–66, New York, NY, USA, 2007. ACM. URL `http://portal.acm.org/citation.cfm?id=1190324`.

The GHC Team. The Glorious Glasgow Haskell Compilation System user's guide, version 6.8.2. `http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html`, 2008a.

The GHC Team. Library documentation: GHC.Prim. `http://www.haskell.org/ghc/docs/latest/html/libraries/base/GHC-Prim.html`, 2008b.