GHC Compiler Pipeline

cyf, whz

June 1, 2017

GHC is structured into two parts:

- The ghc package (in subdirectory compiler), which implements almost all GHC's functionality. It is an ordinary Haskell library, and can be imported into a Haskell program by import GHC.
- The ghc binary (in subdirectory ghc) which imports the ghc package, and implements the I/O for the ghci interactive loop.

GHC is the root module for the GHC API, with very little code and just simple wrappers.

GhcMake implements –make and deals with compiling multiple modules.

DriverPipeline, after GhcMake, deals with compiling a single module through all its stages, including cpp, unlit, compile, assemble, link, etc. These stages/phases call other program and generate a series of intermediate files. The driver pipeline is not the same thing as compilation pipeline and the latter is part of the former.

When compile Foo.hs or Foo.lhs (Literate Haskell), the following phases are called (actually depend on file extensions or flags):

- The **unlit pre-processor** unlit, which locates at utils/unlit as C program, removes the literate markup and generates Foo.lpp.
- The **C preprocessor** cpp (when -cpp is specified), generates Foo.hspp
- The **compiler**, which does not start a separate process, generates files according to the flag given by user.

```
bash$ ghc -c Foo.hs -O -dshow-passes
*** Parser:
*** Renamer/typechecker:
*** Desugar:
Result size of Desugar (after optimization)
  = {terms: 7, types: 4, coercions: 0}
```

```
*** Simplifier:
Result size of Simplifier iteration=1
  = {terms: 6, types: 3, coercions: 0}
Result size of Simplifier = {terms: 6, types: 3, coercions: 0}
*** Specialise:
Result size of Specialise = {terms: 6, types: 3, coercions: 0}
*** Float out(FOS {Lam = Just 0, Consts = True, PAPs = False}):
Result size of Float out(FOS {Lam = Just 0,Consts = True,PAPs = False})
  = {terms: 8, types: 4, coercions: 0}
*** Float inwards:
Result size of Float inwards = {terms: 8, types: 4, coercions: 0}
*** Simplifier:
Result size of Simplifier iteration=1
  = {terms: 12, types: 6, coercions: 0}
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Demand analysis:
Result size of Demand analysis = {terms: 9, types: 5, coercions: 0}
*** Worker Wrapper binds:
Result size of Worker Wrapper binds
  = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Float out(FOS {Lam = Just 0, Consts = True, PAPs = True}):
Result size of Float out(FOS {Lam = Just 0,Consts = True, PAPs = True})
  = {terms: 9, types: 5, coercions: 0}
*** Common sub-expression:
Result size of Common sub-expression
  = {terms: 9, types: 5, coercions: 0}
*** Float inwards:
Result size of Float inwards = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Tidy Core:
Result size of Tidy Core = {terms: 9, types: 5, coercions: 0}
*** CorePrep:
Result size of CorePrep = {terms: 12, types: 6, coercions: 0}
*** Stg2Stg:
*** CodeOutput:
*** New CodeGen:
```

HscMain is used in compile phase of driver pipeline. It compiles a single module/expression/statement to bytecode/M.hc/M.s file. It is also called by GHCi.

GHC supports three backend code generators currently:

- native code generator
- C code generator
- llvm code generator

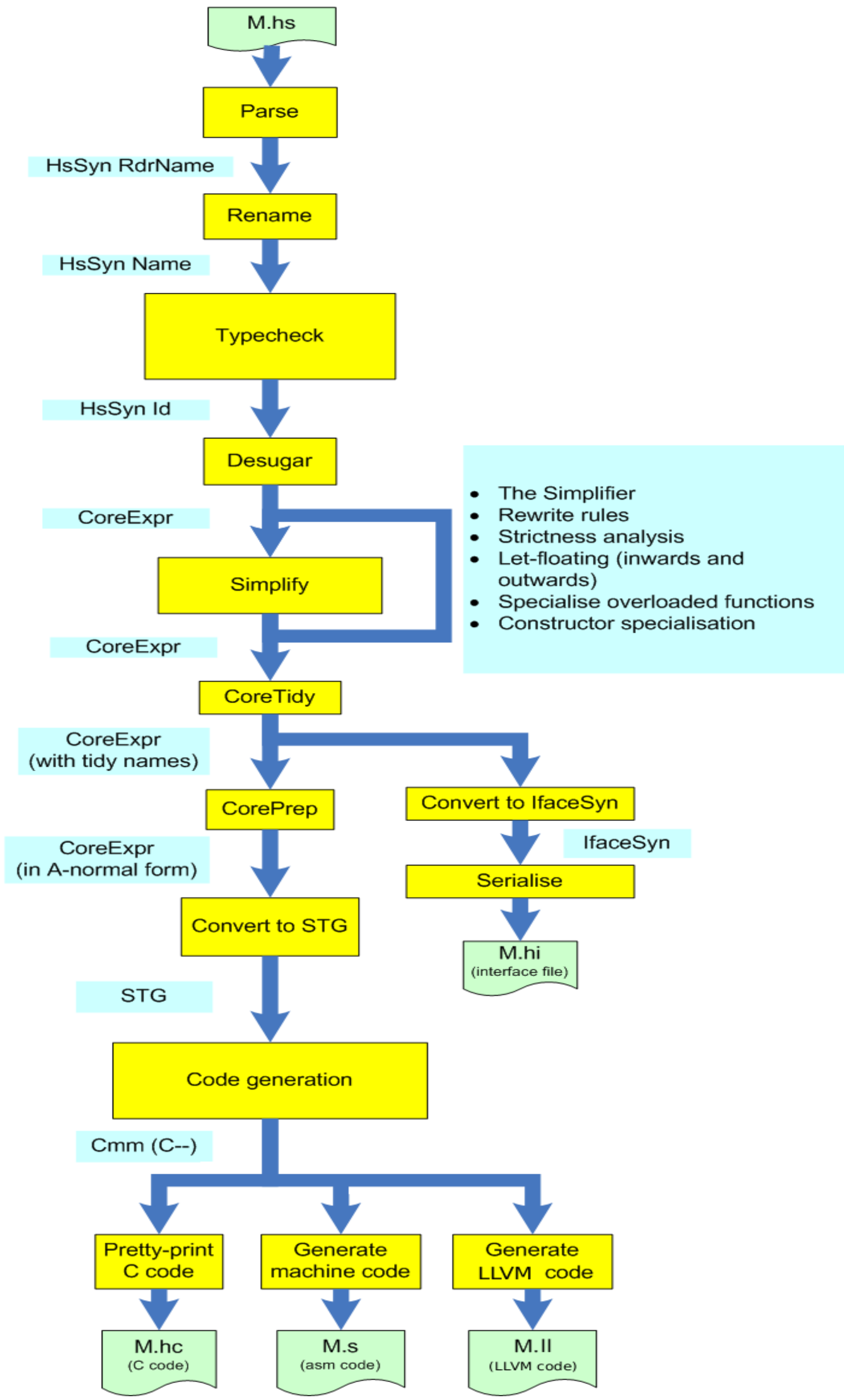
And the possible range of outputs depends on the backend used, all three support assembly output.

- **Object code** Foo.o: no flags required
- **Assembly code** Foo.s: -S required
- **C code**: -C required

The first two ways are supported by three backend and the last one is only supported by C backend.

Pipeline of compiler/main/HscMain.hs is following. They can be dumped by flags like -ddump-*

- The **Front End** processes the program in the big HsSyn type, which is parameterised over the types of the term variables it contains. These three passes detects all programmer errors, and sort them and report them to the user.
  - The **Parser** produces HsSyn parameterised by RdrName, which is approximately a string. (-ddump-parsed)
  - The **Renamer** transforms HsSyn such that it is parameterised by Name, which is approximately a string plus a Unique (number) that uniquely identifies it. (ddump-rn)
  - The **Typechecker** performs type reconstruction and further transforms HsSyn such that it is parameterised by Id, which is approximately a Name plus a type.
- The **Desugarer** (compiler/deSugar/Desugar.hs) converts the massive HsSyn to GHC's intermediate language CoreSyn, which is concise and amazing expressive power. It is much more common to desugar the program before typechecking or renaming, because that presents the renamer and typechecker with a much smaller language to deal with. However, GHC's organisation intend to display precisely error massages for users and avoid preserving type-inference properties.


M.hs
Parse
HsSyn RdrName
Rename
HsSyn Name
Typecheck
HsSyn Id
Desugar
CoreExpr
Simplify
CoreExpr
The Simplifier
Rewrite rules
Strictness analysis
Let-floating (inwards and outwards)
Specialise overloaded functions
Constructor specialisation
CoreTidy
CoreExpr (with tidy names)
CorePrep
Convert to IfaceSyn
IfaceSyn
Serialise
M.hi (interface file)
CoreExpr (in A-normal form)
Convert to STG
STG
Code generation
Cmm (C--)
Pretty-print C code
Generate machine code
Generate LLVM code
M.hc (C code)
M.s (asm code)
M.ll (LLVM code)

- The SimplCore pass (compiler/simplCore/SimplCore.hs) is a bunch of Core-to-Core passes that optimise the program.
  - The Simplifier
    * Inlining
    * Rewrite rules
    * Beta reduction
    * Case of case
    * Case of known constructor
    * etc etc etc...
  - Specialise overloading
  - Float out
  - Float in
  - Demand, cardinality, and CPR analysis
  - Arity analysis
  - Call-pattern specialisation (SpecConstr)

  The simplifier implements and applies lots of small, local optimisations to the program, making them cascade nicely. The float-out and float-in transformations move let-bindings outwards and inwards to optimise the program.

- The **CoreTidy** pass gets the code into a form in which it can be imported into subsequent modules.

- The **CorePrep** pass is the first step of feeding the tidied Core program to the Back End. Here a Core-to-Core pass puts the program into A-normal form (ANF).

- The **CoreToStg** pass is the second step to produce StgSyn data type.

- The **Code generator** converts the STG program to a c– program.