Adding an Optimisation Pass to the Glasgow Haskell Compiler

Olaf Chitil

Lehrstuhl für Informatik II, Aachen University of Technology, Germany
`chitil@informatik.rwth-aachen.de`
`http://www-i2.informatik.RWTH-Aachen.de/~chitil`

November 4, 1997

Abstract

The Glasgow Haskell compiler (GHC) with its over 40.000 lines of code is quite daunting for a newcomer. Here we give a short practical introduction based on our experiences in how to add an optimisation pass to GHC. Thus we hope to encourage other developers of optimisations to implement them in GHC.

These notes are meant to be extended and updated from time to time. Hence observe the date shown above.

Contents

1 The Glasgow Haskell Compiler

These notes give a short introduction into adding a program transformation to the Glasgow Haskell compiler (GHC) and demonstrates that — despite its size — extending GHC is not that difficult. Adding a transformation requires only very few modifications of the source code. For writing new transformations just a limited knowledge of GHC's internal structure is necessary.

Currently there exist two major versions of GHC. Version 2.08 implements Haskell 1.4, and version 0.29 implements Haskell 1.2 For future compatibility we extend version 2.08, but as recommended still use version 0.29 for compilation. The compiler consists of a front end which translates Haskell into a simpler intermediate language named Core, the transformations which optimise Core programs, and a back end which produces C-code.

The compiler is written in Haskell and contains already much documentation which — in contrast to most separate documentation — is up-to-date. This documentation should always be studied, because most papers about GHC are no longer correct in all aspects. The compiler proper is build from 216 Haskell modules and several yacc and lex files which are distributed over 22 directories. Fortunately, most of these are not relevant to program transformations, since they are either concerned with the front end or the back end. Several directories contain program transformations which may serve as examples.

2 Literate Style

Most of the source code is written in the literate style of the Glasgow literate programming system. Literate documents are LaTeX like files, with the compilable/executable code marked off by a `\begin{code}` ... `end{code}` pair. These documents can be both compiled/executed and turned into various formats for printing and viewing. The extensions of files written in literate style usually start with an l, for example, `.lhs` for Haskell programs and `.lprl` for Perl scripts.

Unfortunately, the literate programming tools currently do not work properly, especially indexes and thus (Html) links are not created correctly. Nonetheless, literate programming supports integration of documentation and program code and thus up-to-date documentation.

The existing source code contains much documentation that should always be studied, because most papers about GHC are no longer correct in all aspects. Similarly it makes sense to write your own extensions in literate style.

For pretty printing a literate file you can transform it into LaTeX:

```
lit2latex -S file.l*
latex file.tex
```

For browsing you may also transform literate files into Html:

```
lit2html file.l*
```

For more information about the literate programming system see [GLit92].

3 Structure of GHC

3.1 The Main Call Structure

GHC is invoked by calling `ghc`, which is a Perl script. This script processes the command line options and environment variables and then calls the components of the compiler:

`unlit`: which transforms literate Haskell programs into ”illiterate” Haskell programs
`hscpp`: a modified C-preprocessor; used e.g. for conditional compilation and hiding the incompatibilities of Haskell 1.2 and Haskell 1.3.
`hsc`: the actual Haskell compiler
`gcc`: the C-compiler
`as`: the assembler (called via `gcc`)
`ld`: the linker (called via `gcc`)

3.2 Directory Structure of Source Files

The compiler is in the directory `fptools/ghc` which has the following subdirectories:

`compiler`: the source code for the actual compiler `hsc`
`docs`: The user guide and notes on monadic-style programming
`driver`: Perl scripts for driving the whole compiler (`ghc`)
`includes`: C header files, mostly for the run-time system
`lib`: source code for the standard library `hslib` which contains the prelude
`misc`
`mk`: configuration files for the gnu make system to make `ghc`
`runtime`: C code for the run-time system
`utils`: several utility programs like `mkdepndHS` and `hscpp` (mostly written in Perl)

3.3 Structure of the Source for `hsc`

The source in `fptools/ghc/compiler` is organised into one level of directories.

- General parts
  - the module `Main` that calls all phases and some top level modules: `main`
  - wired-in knowledge about the Prelude: `prelude`
- Compiler phases
  - Reading of the source of a module

    - `reader`
    - `parser` (the most messy part, uses lex and yacc)
  - Renaming and dependency analysis passes: `rename`
  - Type checker: `typecheck`
  - Desugarer: `deSugar`
  - Core-to-core optimisations:
    - `simplCore`: drives all core-to-core transformations
    - `deforest`: normally not part of `hsc`
    - `specialise`
    - `stranal`
  - Translation of Core into Stg: `stgSyn/CoreToStg`
  - Stg-to-Stg optimisations: `simplStg`
  - Code generation:
    - `codeGen/CodeGen`: translation of Stg into abstract C
    - `nativeGen`: native code generators

- Abstract data types
  - Haskell abstract syntax definition: `hsSyn`
  - abstract syntax of types, used in both Haskell and Core: `types`
  - abstract syntax of the intermediate language Core: `coreSyn`
  - cost centre for profiling: `profiling`
  - abstract syntax of the STG language: `stgSyn`
  - abstract C: `absCSyn`
  - declarations of abstract data types that are used in many parts of the compiler: `basicTypes`
  - abstract data types that are of general utility, not just for GHC: `utils`
- an include file for the pre-processor to make the Haskell source compatible to either Haskell 1.2 or Haskell 1.4: `HsVersions.h`

4 Adding an Optimisation Pass

Here we identify those parts of the compiler that need to be modified for adding a new optimisation pass. Starting point of the modifications described here is an existing build tree of GHC (see [GInstall]) in a directory named `mybuild`.

4.1 Modification of Files and Rebuilding of the modified GHC

To simplify file management do not modify any files in `mybuild` but put all modified and new files into separate directories and just change the respective link in `mybuild`.

You have to modify `main/CmdLineOpts.lhs`, `simplCore/SimplCore.lhs`, and `mybuild/ghc/compiler/Makefile` as is described in the following subsections in detail. Probably you write your own `mybuild/mk/build.mk` as well.

Then just follow the standard making routine in the directory `mybuild`:

1. `./configure` (not necessary if you already executed it before the modifications)
2. `make boot` (important because of the new dependencies)
3. `make all`

Note that even if the build tree had already been built before, many object files are now rebuilt, because they directly or indirectly import the changed source files `main/CmdLineOpts.lhs` or `simplCore/SimplCore.lhs`.

While testing your optimisation pass do not forget to execute `make boot` after each modification of imported modules. Finally is a good idea to test this setup by extending GHC first with the empty transformation, that is the identity transformation.

4.2 Modification of `main/CmdLineOpts`

The command line options of the actual Haskell compiler `hsc` are processed by the module `main/CmdLineOpts`. Here a new data constructor has to be added to the data type for the optimisation options, and the function which parses the command line has to be extended:

```
data CoreToDo = ... | CoreDoNewTrans

...

sep ...
  = case (...) of
      ...
      "-fmy-new-trans" -> CORE_TD(CoreDoNewTrans)
```

4.3 Modification of `simplCore/SimplCore`

The `main` function of `hsc` invokes the various phases of the compiler. It calls the function `core2core` of module `simplCore/SimplCore` with the list of all optimisations (of type `CoreToDo`) to perform. This function again calls for every optimisation the module's function `do_core_pass`.

For adding a new transformation function `doNewTrans` we have to import it in the module `simplCore/SimplCore`. The function `do_core_pass` has to be extended to call `doNewTrans` when it is called with the option `CoreDoNewTrans`:

```
...
import MyNewTrans     ( myNewTrans )
...

    do_core_pass info@(binds, us, spec_data, simpl_stats) to_do =
      ...
          CoreDoNewTrans
            -> _scc_ "My new transformation"
               begin_pass "New transformation" >>
               case (myNewTrans binds) of  (binds2, statsText) ->
               hPutStr stderr statsText >>
               end_pass us2 binds2 spec_data simpl_stats
                 "My new transformation"
```

The new program transformation takes as input a value `binds` of type `[CoreBinding]`, which represents the Core program. It may also take an argument `us` of type `UniqSupply`, which provides an arbitrary number of different labels and serves for creating new names for the Core program. The transformation returns a new program of type `[CoreBinding]`. Our transformations also return a string with statistical data on the transformation for evaluation purposes. Since `do_core_pass` returns an IO monad, these statistics can be directly out-putted.

4.4 Modifying the Makefile

Finally the makefile in `mybuild/ghc/compiler` has to know about the new modules. We assume here that these modules all have the extension `*.lhs` or `*.hs` and reside in a directory *full-path*`/Transformation`. Change the makefile as follows:

```
# ---------------------------------------------------------------------
#               Set SRCS, LOOPS, HCS, OBJS
#
# First figure out DIRS, the source sub-directories
# Then derive SRCS by looking in them
#

NEW_DIR = full-path/Transformation


DIRS = \
  utils basicTypes types hsSyn prelude rename typecheck deSugar coreSyn \
  specialise simplCore stranal stgSyn simplStg codeGen absCSyn main \
  reader profiling parser $(NEW_DIR)
```

4.5 Further Hints

4.5.1 Sharing Object Files

Adding an optimising transformation to GHC leaves the major part of the compiler untouched. Hence when several persons write independent extensions it may make sense to share object files to save disc space. Already when building GHC from source you normally make a copy of the source tree, such that all each file is a symbolic link to the source file, and build the system in this build tree ([GInstall]). Thus the source tree can be shared by several different installations.

Similarly, you can make a copy of the existing build tree `shared-build`:

```
mkdir mybuild
cd mybuild
lndir full-path/shared-build
```

However, as mentioned before, many object files have to be rebuild, because they directly or indirectly import the changed source files `main/CmdLineOpts.lhs` or `simplCore/SimplCore.lhs`. Hence these two source files have to be shared as well by all other persons sharing the object code.

Beware that only files that are newly constructed in your build tree are yours. Changing a file that is shared implies changing the shared file, not replacing the link by a new file! To avoid such a modification to happen accidentially the shared build tree should be write protected. All files that are not meant to be shared have to be deleted in your build tree. Among these are

- `mybuild/ghc/compiler/.depend`, because it includes the dependencies of the individual extensions
- `mybuild/ghc/driver/ghc`, because it contains an absolute path to the build tree

You do not rerun `configure` on `mybuild`, because that changes several important files and thus implies loosing much sharing. However, therefore your individual `build.mk` has to contain the line

```
FPTOOLS_TOP_ABS = full-path/mybuild
```

Finally, for showing the difference between the original build tree and your build tree it is useful to know that the command

```
dircomp -s dir1 dir2
```

compares two directories.

4.5.2 Argument List Too Long

On some machines (e.g. Solaris 2.5) there is the problem that the size of the argument list for linking `hsc` is at the very limit and any extension leads to it being beyond the limit. We can resolve this problem by adding our own rule for linking `hsc` which does not use the `-i` option (it is for finding interface files and thus not necessary for linking).

Two changes are necessary in the makefile. First, at the beginning replace

```
HS_PROG=hsc
```

by

```
MY_HS_PROG=hsc
```

Second, append to the end of the makefile:

```
# Addition:
# linking without the -i options
# they are just removed to make the argument list shorter
MY_HC_OPTS = $(filter-out -i%, $(HC_OPTS))
all :: $(MY_HS_PROG)

$(MY_HS_PROG) :: $(HS_OBJS)
	$(HC) -o $@ $(MY_HC_OPTS) $(LD_OPTS) $(HS_OBJS) $(LIBS)
```

Do not forget that the indentation in front of `$(HC)` has to be made by the tabulator.

4.6 Calling the New Optimisation Pass

The the driver script `mybuild/ghc/driver/ghc` translates optimisation options like `-O` or `-O2` into a series of command line options for `hsc` which name the individual transformations to perform. Since arbitrary transformations can be invoked by using the `-Ofile` *file* option, *file* listing the desired transformations, `ghc` needs only to be modified when the new transformation shall finally become part of the standard optimisation options. See Figure 1 for an example of an option file. The list of individual optimisation passes of the standard optimisations (without any, `-O` and `-O2`) can most easily be obtained by compiling a program with option `-v`. Note that every option in *file* and even every single bracket (for the sub-options of the simplifier) has to be in a separate line and the last option has to be terminated by a newline. Note furthermore, that after the execution of all optimisations given in the options file the simplifier is run a last time for cleaning up the code. Hence a `-fprint-core` at the end of the file and `-ddump-simpl` show different Core code.

The native code generators cannot cope with code produced by some optmisations. Therefore, when optimisations are used, Haskell source has to be compiled via the C compiler. Do not forget that core lint is your dear friend! Thus testing your new optimisation should look as follows:

```
correct-path/ghc -fvia-C -dcore-lint -Ofileyour-options-file file.hs
```

Note that there is no space between `-Ofile` and *your-options-file*. It is convenient to make a link in your working directory pointing to the driver script `mybuild/ghc/driver/ghc`.

5 The Intermediate Language Core

The intermediate language of GHC, Core, is essentially the second-order λ-calculus augmented with `let(rec)`, `case`, data constructors, constants and primitive operations. A detailed description of Core and the objectives of its design is given in [PeySan97].

```
# list of core2core optimisations

# first simplifier to clean code of desugarer
# (especially to do dependency analysis)

-fignore-interface-pragmas
-fomit-interface-pragmas
-fsimplify
[
-ffloat-lets-exposing-whnf
-ffloat-primops-ok
-fcase-of-case
-freuse-con
-fpedantic-bottoms
-fsimpl-uf-use-threshold3
-fmax-simplifier-iterations4
]
-fprint-core
-fmy-new-trans
-fprint-core
```

Figure 1: Example file of optimisations for the `-ofile` option

5.1 The Data Types for the Intermediate Language Core

The main data types that represent the language inside GHC are given in Figure 2. All language constructs are parameterised with respect to the different kinds of occurring variables: binding and bound occurrences of value variables, type variables and use variables. Compared to the λ-calculus the syntax is restricted in that arguments of applications have to be literals or variables. The `Coerce` constructor implements a type coercion to support the `newtype` construct. The `SCC` constructor annotates an expression with a `CostCentre`. These cost centres are added when a program is compiled for subsequent profiling. They are essentially names under which all evaluation costs of the respective annotated expressions are recorded. Figure 3 shows an example of a pretty printed Core program.

5.2 Reading Core

Several options permit outputting Core programs between phases. The options `dppr-{user, debug, all}` influence the detail of the printed information (doesn't work correctly in 2.08; `-dppr-all` shows the least).

See the explanations in [GHCUser] and [Par94]. Information about identifiers is given as follows (see module `basicTypes/IdInfo`):

- `_U_` : update info exists
- strictness

```
type CoreBinding = GenCoreBinding  Id Id TyVar UVar
type CoreExpr    = GenCoreExpr     Id Id TyVar UVar
type CoreBinder  = GenCoreBinder   Id    TyVar UVar
type CoreArg     = GenCoreArg         Id TyVar UVar
type CoreCaseAlts    = GenCoreCaseAlts    Id Id TyVar UVar
type CoreCaseDefault = GenCoreCaseDefault Id Id TyVar UVar

data GenCoreBinding val_bdr val_occ tyvar uvar
  = NonRec   val_bdr (GenCoreExpr val_bdr val_occ tyvar uvar)
                                                        non-recursive variable binding
  | Rec    [(val_bdr, GenCoreExpr val_bdr val_occ tyvar uvar)]
                                                          mutually recursive bindings

data GenCoreExpr val_bdr val_occ tyvar uvar
  = Var val_occ                                                              variable
  | Lit Literal                                                         unboxed object
  | Con Id [GenCoreArg val_occ ...]                     saturated constructor application
  | Prim PrimOp [GenCoreArg val_occ ...]          saturated primitive operation application
  | Lam (GenCoreBinder ...) (GenCoreExpr ...)                              λ-abstraction
  | App (GenCoreExpr ...) (GenCoreArg ...)                                   application
  | Case (GenCoreExpr ...) (GenCoreCaseAlts ...)                         case expression
  | Let (GenCoreBinding ...) (GenCoreExpr ...)                               let binding
  | SCC CostCentre (GenCoreExpr ...)                              cost centre expression
  | Coerce Coercion (GenType tyvar uvar) (GenCoreExpr ...)                 type coercion

data GenCoreBinder val_bdr tyvar uvar
  = ValBinder   val_bdr                                           binding value variable
  | TyBinder    tyvar                                              binding type variable
  | UsageBinder uvar                                                binding use variable

data GenCoreArg val_occ tyvar uvar
  = LitArg      Literal
  | VarArg      val_occ
  | TyArg       (GenType tyvar uvar)
  | UsageArg    (GenUsage uvar)

data GenCoreCaseAlts val_bdr val_occ tyvar uvar
  = AlgAlts     [(Id,                                    alternatives: data constructor,
                  [val_bdr],                                   constructor's parameters,
                  GenCoreExpr val_bdr val_occ tyvar uvar)]                          rhs.
                (GenCoreCaseDefault val_bdr val_occ tyvar uvar)

  | PrimAlts    [(Literal,                                 alternatives: unboxed literal,
                  GenCoreExpr val_bdr val_occ tyvar uvar)]                          rhs.
                (GenCoreCaseDefault val_bdr val_occ tyvar uvar)

data GenCoreCaseDefault val_bdr val_occ tyvar uvar
  = NoDefault
  | BindDefault val_bdr                                           the form: var - > expr.
                (GenCoreExpr val_bdr val_occ tyvar uvar)
```

Figure 2: Data types of the Core language

`_bot_` : bottom guaranteed

`_S_` : strictness info

- demand info (only internal to strictness analysis)

  `{-# L #-}`: no info

  `{-#` ... `#-}`: the info

- arity info

  `_A_` *n*: arity is *n*

  `_A>_` *n*: arity is *n* or greater

6 Some Thoughts on Designing a New Transformation

First, a new optimisation has to be defined precisely. The principal problem is often to specify, when the transformation is applied exactly. The trigger for an optimisation may be an undecidable semantic property, for example strictness. Then a computable approximation of this property has to be found. This approximation may not just miss some admissible applications of the transformation but sometimes even permit a transformation when the semantic property would not. In the second case the transformation must obviously still be correct and at least not worsen the quality of the program.

The key feature of a transformation is naturally correctness, that is that it preserves the semantics of the transformed program.

Because the whole purpose of the transformation is optimisation, its effectiveness, that is its effect on the evaluation cost of the program, has to be determined. For lazy functional languages the costs of major interest are run-time of the program, total heap allocation, heap residency, that is the maximal space required by life objects on the heap at one time, and size of the program code.

Also the interaction between various transformations has to be studied. Transformations may both improve and worsen the effectiveness of later transformations. Finally, although a transformation may not be applied in every compilation, the cost of the transformation has to be determined and to be related to its effectiveness.

For examples of transformations implemented in GHC see [Chi97, PeySan97, San95].

7 Measuring the Effects of a Transformation

GHC has a large number of options for observing various aspects of a running program.

```
... (Definitions of constants and functions extracted from dictionaries)
f_aHT  ::  PrelBase.Int{-3g,p-}
{-# L #-}
f_aHT =
  let { ds_dP6  ::  PrelBase.Int{-3g,p-}
     {-# L #-}
     ds_dP6 =
   let {
    ds_dPi  ::  [PrelBase.Int{-3g,p-}]
    {-# L #-}
    ds_dPi =
      enumFromTo_aM5
        lit_aM8 lit_aM6
   } in
    sum_aMa
      ds_dPi
  } in
  let { ds_dPa  ::  PrelBase.Int{-3g,p-}
     {-# L #-}
     ds_dPa =
   let { ds_dPA  ::  [PrelBase.Int{-3g,p-}]
      {-# L #-}
      ds_dPA =
    let {
     ds_dPI  ::  PrelBase.Int{-3g,p-}
     {-# L #-}
     ds_dPI =
       negate_aM4
         lit_aMe
    } in
     enumFromTo_aMc
       ds_dPI lit_aMd
   } in
   sum_aMf
     ds_dPA
  } in
  +_aMb
    ds_dP6 ds_dPa
f{-rP,x-}  ::  PrelBase.Int{-3g,p-}
{-# L #-}
f{-rP,x-} =
  f_aHT
```

Figure 3: Desugared: `f = sum [1..100] + sum [-100..1]`

7.1 Without Special Compilation

For measuring the general effects of a transformation on a whole program no special compilation is necessary.

Just running a program with the option `+RTS -s -RTS` (`+RTS -S -RTS`) generates a file *`program`*`.stat` with a summary (detailed information) about garbage collection and run-time.

The program `mybuild/glafp-utils/runstdtest/runstdtest` has many options for testing and measuring programs systematically. Let *input*, *output* and *error* be files that contain the required input and expected output on stdin and stderr respectively. Those that are empty may be omitted. Running a program with

```
runstdtest -ghc-timing -iinput -o1output -o2error program
```

gives the following kind of output (`-i`, `-o1`, `-o2` may be omitted if the input/output is empty):

```
<<ghc:  31819776 bytes, 126 GCs, 5645/15436 avg/max bytes residency (63
samples), 0.01 INIT (0.00 elapsed), 10.14 MUT (10.25 elapsed), 0.09 GC
(0.11 elapsed) :ghc>>
```

The numbers are

- the sum of all bytes allocated on the heap
- the number of garbage collections (minor and major for the standard generational garbage collector)
- the average and maximal heap residency, that is that part of the heap that is life; these are determined at every major garbage collection, hence the number of samples is the number of major garbage collections.
- the time for initialising the run-time system in seconds
- the time for the actual computation
- the time spent on garbage collection

For all times both user time and real time (that includes time spent on other processes) are given.

When times are very short just use a slower machine for measurements. The residency information is normally very inaccurate, because only very few samples are taken. The number of major garbage collections can easily be increased by using `+RTS -jnumber -RTS`. This option forces a major garbage collection after every allocation of *number* bytes. Be aware that using this option increases run-time and even total heap allocation.

7.2 Profiling and Ticky-Ticky Profiling

For profiling individual parts of a program for space and time see [GHCUser].

8 Recommended Reading

What you should read when adding a new optimisation pass:

[GHCUser]	Glasgow Haskell compiler user's guide
[GInstall]	Glasgow Haskell compiler installation guide; especially to learn about the makefile architecture
[PeySan97]	a comprehensive overview of the compiler; to learn about Core and some optimisations
[San95]	if you want more detailed knowledge about some transformations
[Par94]	although dated and partially obsolete still useful information about how to add an optimisation
[Pey92]	if you want to know about the STG-machine to have a better understanding of the operational semantics of Core

The GHC web page [GHC] is a good starting point for finding more documentation about various aspects of the compiler.

Acknowledgement

The author thanks all members of the program transformation group at Aachen for numerous discussions and Simon Peyton Jones for answering many questions about GHC.

References

[Chi97] Olaf Chitil: *Common Subexpression Elimination in a Lazy Functional Language*; Implementation of Functional Languages, 9th International Workshop, 1997.

[GHC] *The Glasgow Haskell compiler*; `http://www.dcs.gla.ac.uk/fp/software/ghc/`.

[GHCUser] *Glasgow Haskell Compiler User's Guide*; part of the Glasgow Haskell compiler distribution.

[GInstall] *Building and installing the Glasgow Functional Programming Tools Suite*; part of the Glasgow Haskell compiler distribution.

[GLit92] The GRASP team: *Glasgow literate programming user's guide*; part of the Glasgow Functional Programming Tools distribution.

[MTW95] Christian Mossin, David N. Turner, and Philip Wadler: *Once upon a type*; Technical Report TR-1995-8, University of Glasgow, 1995. Extended version of *Once upon a type* in 7'th International Conference on Functional Programming Languages and Computer Architecture, June 1995.

[Par93] Will Partain: *The nofib benchmark suite of Haskell programs*; part of the GHC distribution.

[Par94] Will Partain: *How to add an optimisation pass to the Glasgow Haskell compiler (two months before version 0.23)*; part of the GHC 0.29 distribution, October 1994.

[Pey87] Simon L Peyton Jones: *The Implementation of Functional Programming Languages* Prentice-Hall, 1987.

[PeyLau91] Simon L Peyton Jones and John Launchbury: *Unboxed values as first class citizens in a non-strict functional language*; Conf. on Functional Programming Languages and Computer Architecture, 1991, pp 636–666.

[Pey92] Simon L Peyton Jones: *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*; J. Functional Programming, **2** (2):127–202, 1992.

[PeySan97] Simon L Peyton Jones and André L M Santos: *A transformation-based optimiser for Haskell*; submitted to Science of Computer Programming, 1997.

[SanPey95] Patrick M Sansom and Simon L Peyton Jones: *Time and space profiling for non-strict, higher-order functional languages*; 22nd ACM Symposium on Principles of Programming Languages, January 1995.

[San95] André L M Santos: *Compilation by transformation in non-strict functional languages*; PhD Thesis, University of Glasgow, July 1995.