A Guide Through

David Sabel

September 9, 2003

Contents

1 About HasFuse **2**

2 Building HasFuse **2**

3 Differences between HasFuse and GHC **3**

3.1 Direct-call I/O 3

3.2 GHC extensions you should not use 3

3.3 Optimisation levels in HasFuse 4

3.4 Compiler options, you should not use 4

3.5 New compiler options 7

4 Examples **7**

4.1 Different behaviours caused by a single transformation 7

4.1.1 Example for the “case eta expansion” 7

4.1.2 Example for the “full laziness” transformation 8

4.2 Basic applications of direct-call I/O 9

4.2.1 Encoding of directPutChar and directGetChar 9

4.2.2 Encoding of directPutStr and directGetLine 9

5 Further documentation **10**

1 About HasFuse

HasFuse (Haskell with FUNDIO-based side effects) is a modification of the Glasgow Haskell Compiler (GHC) ([The03b]). HasFuse allows you to use non-strict direct-call I/O within Haskell by using `unsafePerformIO` in any context. The semantics for this I/O are given by the FUNDIO calculus (see [Sch03]).

This document contains some information about building and using HasFuse. But for both you should use the GHC documentation ([The03b, The03a]) too, because the aim of this paper isn't to give a complete compiler reference, we present only the differences between HasFuse and GHC.

2 Building HasFuse

For building HasFuse you need

- The HasFuse source[1]
- The source of GHC version 5.04.3.
- Everything what's required to build GHC (another GHC, Happy). See [The03a].

Then unpack both source archives (GHC and HasFuse). Let's assume:

- GHC is in a tree with top-level directory `fptools` and
- HasFuse in a tree with top-level directory `hasfuse`.

Merge the source trees. This can easily be done, by copying the HasFuse files over the GHC files. For example you can do this step with:

```
cp -r hasfuse/* fptools/
```

After that, you can create a build tree or use the source tree with top-level directory `fptools` for building.

Building HasFuse is now identical to building GHC, so you should use the GHC documentation especially the GHC building guide ([The03a]) for this step.

Probably this works: Go to the `fptools` directory and type

[1] It's available at `http://www.ki.informatik.uni-frankfurt.de/~sabel`

```
autoconf
(cd ghc && autoconf)
```

Then create the mk/build.mk (probably an empty file is enough). After that, type

```
./configure
gmake
```

After building you should have a `ghc-inplace` at `fptools/ghc/compiler/ghc-inplace` (or in your build tree path) which can be used to compile programs with HasFuse.

3 Differences between HasFuse and GHC

3.1 Direct-call I/O

GHC offers the Haskell extension `unsafePerformIO`, but the use is limited to a few special cases (see [The03b, Chapter 13] for more information). The aim of this limitation is to preserve the pureness of Haskell.

HasFuse hasn't this limitation. You can use `unsafePerformIO` in any context you want. The underlying language is Haskell with `unsafePerformIO`, so this language is no longer pure, but the direct-call I/O is no longer unsafe, because the FUNDIO calculus gives the semantics for this I/O, which is modelled by nondeterminism.

3.2 GHC extensions you should not use

This section contains information about GHC-specific extensions of Haskell which you shouldn't use for programs, you want to compile with HasFuse.

INLINE pragmas: The reason for that is that Inlining is not safe in the meaning of the FUNDIO semantics. Notice: We didn't turn off the handling of INLINE-Pragmas ([The03b, section 7.6.1]) , so if you use them, there's no guarantee for correct behaviour of your compiled programs.

RULES pragmas: HasFuse ignores RULES ([The03b, section 7.6.6]), because it has been to much work, to prove the built-in rules of GHC.

3.3 Optimisation levels in HasFuse

GHC supports three different levels for different optimisation, which are available with the flags `-O0` (same as no optimisation flag), `-O1` (same as `-O`) and `-O2`. HasFuse supports the same flags, but the performed optimisations are different from those, which are performed in the GHC. The most important thing is, that the level 0 and 1 are proven for safeness. Level 2 isn't proven, so you should use the flag `-O2` **only** for testing purposes. Now we give a short summary about the different levels.

`-O0`: This level only performs local transformations, the compilation process is fast.

`-O1`: All optimisations that have been proven for safeness are performed. Most of the global transformations aren't performed, because we yet haven't analyzed them.

`-O2`: In difference to `-O1` all global transformations, which aren't obviously unsafe are performed. We have no counterexamples, which show different behaviours between this level and level `-O1`, but nevertheless this level is **not** proven as safe.

The tables 1 and 2 give a summary about the performed transformations depending of the optimisation level and some information about switching on or off a single transformation.

3.4 Compiler options, you should not use

This section contains information about some GHC options (flags), you shouldn't use for HasFuse. Most of these options are available in HasFuse, but they have no effect. This design decision has been chosen, so that the Makefiles for GHC can be used for HasFuse.

Common subexpression elimination: This transformation is unsafe in our meaning, so the transformation is never performed. The flag `-fno-cse` for turning off the transformation is available in HasFuse, but it has no effect, so you shouldn't use it.

Deforestation: This transformation hasn't been proved for correctness. So you shouldn't use the `-ffoldr-build-on` flag. Because this transformation is based on RULES pragmas, you can't turn on the transformation with this flag.

Interface Pragmas: HasFuse doesn't use pragmas from interface files. You can use the flag `-fno-ignore-interface-pragmas` to change this behaviour, but nevertheless HasFuse ignores RULES, so you shouldn't use this flag.

Tranformation	`-O` **level** 0	1	2	**correct**[a]	**comments**
eta expansion	✓	✓	✓	✓	can be turned off with the flag -fno-do-lambda-eta-expansion
let-to-case	×	✓	✓	✓[b]	
case merging	×	✓	✓	✓	can be turned on with -fcase-merge, can be turned off with -fno-case-merge
eta reduction	×	✓	✓	✓	can be turned on with -fdo-eta-reduction, can be turned off with f-no-do-eta-reduction
RULES pragmas	×	×	×	?[c]	
Interface pragmas	×	×	×	?[d]	can partly be turned on with -fno-ignore-interface-pragmas, but you shouldn't use this flag.

[a] in the meaning of the FUNDIO semantics
[b] depends on a conjecture, which hasn't been proven till now
[c] depends on the rule
[d] depends on the pragma

Table 1: Performed local transformations depending on the optimisation level

tranformation	`-O` **level** 0	1	2	**correct**[a]	**comments**
full laziness	×	×	×	×	
common subexpression elimination	×	×	×	×	
let floating in	×	✓	✓	✓	
strictness analysis	×	×	✓	?	can be turned on at optimisation level 1 with the flag -fstrictness
cpr analyse	×	×	✓	?	is only performed if strictness analysis is performed, if so it can be turned on by -fno-cpr-off or be turned off by -fcpr-off
worker/wrapper	×	×	✓	?	
specialising	×	×	✓	?	
specialising over constructors	×	×	✓	?	
deforestation	×	×	✓	?	can partly be turned on by -ffoldr-build-on (but you shouldn't use it), can be turned off by -fno-foldr-build-on
UsageSP-Analyse	×	×	×	?	can be turned on at level 2 with the flag -fusagesp

[a] in the meaning of the FUNDIO semantics

Table 2: Performed global transformations depending on the optimisation level

3.5 New compiler options

This section contains information about new options, which you can use for HasFuse. These options are not available in the GHC.

HasFuse related information The flag `--hasfuse` prints some help information about HasFuse.

Switching on strictness analysis Strictness analysis is off by default (except at the unsafe optimisation level 2, which comes with `-O2`), but you can turn on the strictness analysis at optimisation level 1 by using the flag `-fstrictness`. The results off the analysis are then used for local transformations[2]. The Worker/Wrapper transformation is a global transformation and so it's not performed at optimisation level 1, also if `-fstrictness` is used.

4 Examples

4.1 Different behaviours caused by a single transformation

4.1.1 Example for the "case eta expansion"

The following program behaves wrongly regarding the FUNDIO semantics, if it's compiled with the GHC. The reason here is the implementation of the so called "case eta expansion".

```
*****************************************************************
modul: etacase.lhs
*****************************************************************

> module Main(main) where

> import System.IO.Unsafe(unsafePerformIO)

> z = unsafePerformIO (putStr "Print this text!\n")
> f = \x -> z `seq` (\y -> y)

> main = (f True) `seq` return ()
```

The correct behaviour in the meaning of FUNDIO is to print the text before quitting the program.

With the GHC you can only get the correct behaviour, if you turn off the eta expansion:

[2]These are the so called let-to-case and case-elimination transformations

```
ghc -O0 -o etacase etacase.lhs
./etacase

ghc -fno-do-lambda-eta-expansion -O0 -o etacase etacase.lhs
./etacase
Print this text!
```

HasFuse compiles the example correct at all optimisation levels.

4.1.2 Example for the "full laziness" transformation

The following program isn't correct (in the FUNDIO meaning) compiled by GHC at optimisation level 1 or 2. The reason for the wrong behaviour is the "full laziness" transformation. This transformation isn't performed in HasFuse.

```
****************************************************************
modul: fulllazy.lhs
****************************************************************

> module Main(main) where

> import System.IO.Unsafe(unsafePerformIO)

> main = let f = \xs -> let z = unsafePerformIO getChar
>                       in z
>        in
>         do
>           putStr ( (f 1):" is the result of (f 1).\n" )
>           putStr ( (f 2):" is the result of (f 2).\n" )
```

The correct behaviour in FUNDIO is to perform two different calls to `getChar`, but the GHC compiled program behaves different:

```
ghc -O1 -o fulllazy fulllazy.lhs
./fulllazy
AB
A is the result of (f 1).
A is the result of (f 2).
```

Compilation with HasFuse shows the correct behaviour:

```
ghc-inplace -O1 -o fulllazy fulllazy.lhs
 _   _           _____
| | | | __ _ ___|  ___|   _ ___  ___  A modified version of GHC, version 5.04.3
| |_| |/ _` / __| |_ | | | / __|/ _ \ Type --hasfuse for details
|  _  | (_| \__ \  _|| |_| \__ \  __/ This software comes with
|_| |_|\__,_|___/_|   \__,_|___/\___| ABSOLUTELY NO WARRANTY!

./fulllazy
AB
A is the result of (f 1).
B is the result of (f 2).
```

4.2 Basic applications of direct-call I/O

In this section we present the encoding of direct-call variants of the monadic I/O functions `putChar` and `getChar`. Based on these definitions we give encodings of a function that prints a full string and a function that reads a line from the standard input. Both functions are encoded without using monadic I/O.

These definitions aren't very useful, but they should give you a feeling what you can do with direct-call I/O in lazy functional language.

4.2.1 Encoding of directPutChar and directGetChar

The encoding of these operators is easy: We apply the `unsafePerformIO`-Operator to the monadic functions. But we must be careful with `directGetChar`. This operator needs a dummy argument, so that the definition is an abstraction. If we didn't this, the right hand side of the definition would be updated after the first call with the result of this call.

```
> directPutChar :: Char -> ()
> directPutChar c = unsafePerformIO (putChar c)

> directGetChar :: a -> Char
> directGetChar _ = unsafePerformIO getChar
```

4.2.2 Encoding of directPutStr and directGetLine

For these functions we need a sequentialisation of the I/O call. So we use the `seq` operator.

```
> directPutStr :: String -> ()
> directPutStr []     = ()
> directPutStr (x:xs) = seq (directPutChar x) (directPutStr xs)

> directGetLine _ = let x = directGetChar ()
>                   in if x == '\n' then
>                        []
>                      else
>                        let
>                          res = (directGetLine ())
>                        in
>                          seq res (x:res)
```

If we used the following definition of `directGetLine`

```
> directGetLine _ = let x = directGetChar ()
>                   in if x == '\n' then
>                        []
>                      else
>                        x:(directGetLine ())
```

then the I/O would be lazy: A character is read at the time, when it's position in the resulting string is evaluated.

Now we can define a function `echo`, which firstly reads a string and then puts it on the standard output:

```
> echo = let
>          xs = directGetLine ()
>        in
>          seq xs (directPutStr xs)
```

5 Further documentation

HasFuse was developed as a part of [Sab03], where most of the local transformations of the GHC have been proven of correctness regarding the FUNDIO ([Sch03]) semantics.

For contact information or news about HasFuse take a look at `http://www.ki.informatik.uni-frankfurt.de/~sabel`.

References

[Sab03] David Sabel. Realisierung der Ein-/Ausgabe in einem Compiler für Haskell bei Verwendung einer nichtdeterministischen Semantik. Diplomarbeit (to appear), Institut für Informatik, J.W.Goethe-Universität, Frankfurt, 2003.

[Sch03] Manfred Schmidt-Schauß. FUNDIO: A Lambda-Calculus with a letrec, case, Constructors, and an IO-Interface: Approaching a theory of unsafePerformIO. Draft from 22.06.03, 2003.

[The03a] The GHC Team. Building the Glasgow Functional Programming Tools Suite. http://haskell.org/ghc/docs/5.04.3/, 2003.

[The03b] The GHC Team. The Glasgow Haskell Compiler User's Guide, Version 5.04. http://haskell.org/ghc/docs/5.04.3/, 2003.