ART Memo 9 (Version 1)
Towards a Portable Tracer

Olaf Chitil

June 13, 2001

Contents

1 Introduction

The Hat viewers are separate programs, but the Hat transformation is still integrated into nhc. Here we collect some ideas about how a separate transformation can yield Haskell programs that can be compiled with other Haskell compilers. Advantages are:

- An optimising Haskell compiler such as ghc can produce better (especially faster) code.
- The obtained modularity may improve the understanding of tracing further. The trace generating program is a complete and standard Haskell program. The transformation relates it to the original Haskell program and its semantics relates it to the ART trace.

Note that the separate transformation will not lead to supporting Haskell extensions implemented by various Haskell compilers. It might facilitate such extensions, because only the transformation but not the back end of nhc need to be modified for any such extension.

The transformed Haskell modules will use a library of support functions, which include C-functions accessed via the FFI. The library should be implementation independent.

To minimise the development effort we intend to obtain the separate transformation by modifying the current integrated transformation.

2 The Current State

Here we enumerate how the Hat transformation and nhc are currently intertwined and the problems we have to solve for separating the two.

2.1 An Example

For the program

```
insert :: Ord a => a -> [a] -> [a]

insert x [] = [x]
insert x (y:ys)
  | x > y = x : insert x ys
  | otherwise = x : y : ys

sort xs = foldr insert [] xs

main = print (sort [3,2,1])
```

we can obtain with the options `-tracefns -noshowqualified` the following result:

```
insert ::
  Ord t1 =>
  SR -> Trace -> R (Trace -> (R t1) -> R (Trace -> (R [t1]) -> R [t1]))

insert v346 v345 =
  fun2 (mkNTId' 337) Main.insert_347 v346 v345
  where
```

```
  Main.insert_347 v348 v340 (R ([]) v351) =
    con2 (mkSR' 2) v348 (:) (mkNTConstr' 311) v340
      (con0 (mkSR' 2) v348 ([]) (mkNTConstr' 310))
  Main.insert_347 v348 v341 (R (v342 : v343) v352) =
    case ap2 (mkSR' 7) v348 ((mkSR' 7) > v348) v341 v342 of
      R v357 v358 ->
        let
          v353 =
            mkTAp2 v348 (mkTNm v348 mkNTGuard (mkSR' 10)) v358 v348 (mkSR' 10)
          in
          (myseq v353
            (if v357
              then con2 (mkSR' 9) v353 (:) (mkNTConstr' 311) v341
                (ap2 (mkSR' 8) v353 (insert (mkSR' 8) v353) v341 v343)
              else case otherwise (mkSR' 3) v353 of
                R v355 v356 ->
                  let
                    v354 =
                      mkTAp2 v353 (mkTNm v353 mkNTGuard (mkSR' 6)) v356 v353
                        (mkSR' 6) in
                    (myseq v354
                      (if v355
                        then con2 (mkSR' 5) v354 (:) (mkNTConstr' 311) v341
                          (con2 (mkSR' 4) v354 (:) (mkNTConstr' 311) v342 v343)
                        else fatal v354))))
  Main.insert_347 v348 _ _ =
    fatal v348


sort v360 v359 =
  fun1 (mkNTId' 338) Main.sort_361 v360 v359
  where

  Main.sort_361 v362 v344 =
    rap3 (mkSR' 11) v362 (foldr (mkSR' 11) v362) (insert (mkSR' 12) v362)
      (con0 (mkSR' 13) v362 ([]) (mkNTConstr' 310)) v344
  Main.sort_361 v362 _ =
    fatal v362


main _ _ =
  Main.main_365

Main.main_365 =
  lazySat
    (rap1 (mkSR' 14) nt_364 (print (mkSR' 14) nt_364)
      (ap1 (mkSR' 15) nt_364 (sort (mkSR' 15) nt_364)
```

```
    (con2 (mkSR' 19) nt_364 (:) (mkNTConstr' 311)
      (fromConInteger (mkSR' 16) nt_364 3L)
      (con2 (mkSR' 19) nt_364 (:) (mkNTConstr' 311)
        (fromConInteger (mkSR' 17) nt_364 2L)
        (con2 (mkSR' 19) nt_364 (:) (mkNTConstr' 311)
          (fromConInteger (mkSR' 18) nt_364 1L)
          (con0 (mkSR' 19) nt_364 ([]) (mkNTConstr' 310))))))) nt_364
where

nt_364 =
  mkTNm mkTRoot (mkNTId' 339) (mkSR' 20)
```

2.2 Pretty Printing the Syntax Tree as a Haskell Module

Besides the subsequently listed points, the pretty printer also makes a number of small mistakes (especially parenthesis) which have to be corrected to obtain a well-formed Haskell program.

2.2.1 The Module Header: Imports & Exports

The module name, the export list and various imports have to be added to the output. This information is not part of the declaration list but should be available. However, for reasons discussed elsewhere the information might need to be modified with respect to the original program.

2.2.2 Qualified Names

The pretty printer either always uses qualified names or always unqualified names. The latter leads to ambiguities, the former make the output unreadable for a human and the qualification refers to the definition module, not the module from which the named entity is imported. The name originally used in the program (qualified or unqualified) is lost during renaming.

2.2.3 Missing Declarations

Several kind of declarations containing types are missing from the output, for example definitions of types and type synonyms, types in expressions and type declarations introduced by the transformation (which may sometimes be necessary to enable type inference). This is because there exist two kinds of representations of types in nhc. `Context` and `Type` values are produced by the parser. The renamer transforms them into `NewType` and `NT` values. For the latter there is currently no suitable pretty printer.

2.2.4 Pragmas

Comments delimited by `{-#` and `#-}` contain pragmas for the compiler. Unfortunately the parser already dismisses these pragmas, they are not part of the syntax tree. Hence the question arises of how to get them into the tracing program.

For some pragmas simple copying of their content into the tracing program may not suffice, they have to be transformed as well. The most obvious example for such a pragma is the RULES pragma. It contains a rewriting rule that is to be used by the compiler, mostly for optimisations. It consists of terms and types and would need to be transformed similar to a variable definition. Because of the modified types an untransformed rule does not make any sense in the tracing program. Possibly we do not want rules at all, because we may not want to trace optimised programs. However, many optimisations of ghc are now based on rules and without them ghc will produce rather inefficient code, especially for the prelude and standard libraries (so rules are connected with the problem of how to handle the prelude).

2.3 The Source Reference and Identifier Tables

For each Haskell module exists a table with information about source references and a table with information about identifiers. The tables are used for sharing the information, entries are usually referred to several times in the transformed Haskell code. These tables are currently generated by the code generator.

Earlier generation at the Haskell level is possible. However, whereas source reference entries are only referred to in the same module, references to identifier table entries may be to tables of other modules.

2.4 The Prelude

Because the definitions of the prelude are not fixed, just the semantics, it is not sensible to trace the prelude. However, even a trusted prelude requires transformation.

Writing the whole tracing prelude by hand would probably be rather error-prone and hard work. Most of the tracing prelude needs to be created by the tracing transformation from nhc's normal prelude. However, for some primitive parts hand-written wrappers are still required.

The final aim will be that the traced trusted prelude is produced automatically from the identifiers and types of the original prelude and consists of wrappers that call the original prelude.

2.4.1 Unchanged Prelude

A Haskell system knows the semantics of all prelude functions and is free to use them internally in the compiler. For example, pattern matching of numeric constants and $n + k$ patterns is translated into the use of overloaded `==` and `-`. The tracing transformation removes such syntactic sugar itself, but in principal a Haskell system may make use of its knowledge of prelude functions for any kind of purposes, especially optimisation.

Hence a traced program has to import both the original, unchanged prelude (as qualified) and a trusted traced prelude (which needs a new name). All explicit imports of the prelude have to be changed to imports of the traced prelude.

2.4.2 Primitive Functions

Some prelude functions are defined as primitive (should be FFI). Some have magic implementations as byte code. For the latter wrappers have to be provided that call the magic implementations.

2.4.3 Primitive Types

The tracing counterparts of primitive types like numbers, `Char` and list have to be defined. Because they need to be normal Haskell data types, special syntax such as `[]` is not admissible for them. Hence the transformation also has to replace constructors appropriately.

2.4.4 Traced Trusted Prelude as Valid Haskell Modules

Nhc's prelude consists of numerous files, all with the same module header. The traced trusted prelude needs to consist of normal Haskell modules.

2.4.5 Interface Files for Bootstrapping

Currently the transformation needs the interface files of the transformed prelude. They are written by hand.

For the portable version these interface files may change. How can they be generated automatically to avoid bootstrapping problems?

2.4.6 IO

To write a computation's output to file and to connect it to the trace, some nhc internal IO functions are implemented specially.

For portability, the traced prelude IO functions must be defined in terms of the untraced prelude IO functions.

2.5 Other Issues

2.5.1 Primitive Functions Used by Tracing Combinators

The combinator library contains several C-functions which are used via nhc's primitive mechanism: `fatal`, `cPointerEquality` (still used?), `cEnter`, `cInitializeDebugger`, `_tprim_setOutputContext`, The purpose of each of these functions has to be determined. For several of them just using them via the FFI may suffice. Some are addressed in the following.

2.5.2 Abortion of a Traced Computation with Error Message

The transformation introduces calls of the C-function `fatal` with a parent as argument. It aborts the computation with the message "pattern match failure".

Somehow calls of `error` also abort the computation.

2.5.3 Interruption of a Traced Computation

An important feature of Hat is that any traced computation can be interrupted by pressing Control-C. The C-function `cEnter` is called every time a closure is entered. It tests if Control-C is pressed and if that is the case then terminates the computation. Some other C-function must set up a handler for Control-C. Both functions should be used via the FFI and their definitions should be portable. They should not conflict with the exception handling mechanism of GHC. (Maybe polling for Control-C can be avoided altogether; cleaning up at termination probably only requires access to C-variables which should be in a consistent state.)

2.5.4 Initialisation of Trace Generation

The C-function `cInitializeDebugger` seems to perform some initialisation for trace generation. It is currently called from some nhc internal module `_Driver` which calls `main`. For portability all initialisation should be performed by `main`.

2.5.5 `main`

The function `Main.main` needs to have type `IO ()`, but all transformed functions have a different more complex types. The transformed version of `Main.main` should have a different name, to enable calls to it (currently impossible).

2.5.6 Numeric Patterns

Numeric patterns, that is, numeric constants and $n+k$ patterns, are currently handled specially in various parts of the compiler, also in code generation.

They have to be handled by the transformation. A simple method would be to ignore the involved reduction steps in the trace and rely on numeric patterns of the compiler (which must use the untransformed prelude functions). Otherwise a transformation has to be defined, which probably needs untransformed prelude functions.

2.5.7 `fromInteger/fromRational`

Where are these implicit conversion functions for numeric literals introduced explicitly? Probably they are currently traced specially.

There is nothing to do about them, if nhc introduces them after the transformation and they refer to the original prelude.

3 Design Options

3.1 Remaining Integration with nhc

The separate transformation can be built as a separate program, or nhc obtains a new option to perform the transformation, generating a Haskell module, without generating code. The former has the following advantages:

- Tracing transformation can be more efficient, because it does not need to perform any superfluous work needed by nhc.
- A user only has to install the transformation, not full nhc.
- Easier to support Haskell extensions that nhc does not support.

The latter has these advantages:

- A natural result of stepwise development of the transformation.
- For compiling with nhc no second parsing of transformed Haskell program necessary.
- Nhc and transformation keep consistent during further development.
- The transformation can use any information available in nhc's front end (e.g. types); hence future changes may be easier to implement.

3.2 Position of the Transformation

Currently the transformation takes place rather late in the front end of nhc, just before type inference.

We might want to move the transformation just after the parser. Some rethinking might be required parse operators without knowing their fixity and to have cross-module references to identifier tables (Section 2.3), but the resulting implementation would be much simpler and smaller, without any worries about interface files.

On the other hand doing so would mean giving up the flexibility to use – in the future – additional information that is availabe in nhc (interface files; even types, if the transformation is moved).

3.3 Interface Files

Shall the transformation introduce its own interface file format? Shall it use nhc's interface files? In the latter case, does it produce its own interface file?

Information currently needed from other modules: fixity information for parsing; references to identifier table entries.

3.4 Module and Identifier Names

Shall the transformation change the names of modules? A transformed modules needs a name and it should certainly not overwrite the original module. For the tracing prelude a new name is needed anyway. Note that the main module must still have the name `Main`.

Shall the names of identifiers be changed? By preceding each identifier by a character we obtain a name space for identifiers introduced by the transformation. Also there may be several new identifiers associated with an old one (e.g. transformed identifier and identifier table entry). Identifiers for constructor table entries are needed.