# Transforming Haskell for Tracing

Olaf Chitil, Colin Runciman, and Malcolm Wallace

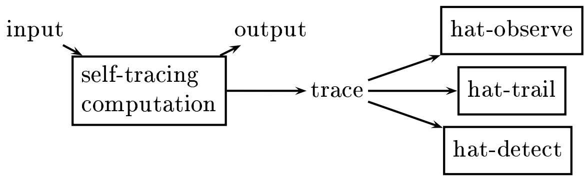
The University of York, UK

**Abstract.** Hat is a programmer's tool for generating a trace of a computation of a Haskell 98 program and viewing such a trace in various different ways. Applications include program comprehension and debugging. A new version of Hat uses a stand-alone program transformation to produce self-tracing Haskell programs. The transformation is small and works with any Haskell 98 compiler that implements the standard foreign function interface. We present general techniques for building compiler independent tools similar to Hat based on program transformation. We also point out which features of Haskell 98 caused us particular grief.

## 1 Introduction

Tools such as profilers and tracers are essential for wider adoption of a programming language [8]. For more than 20 years researchers have been proposing ways to build tracers for lazy higher-order functional languages (see the related work sections of [3, 4, 10]). However, most of this work has never been widely used, because it has been done for locally used implementations of local dialect languages. Haskell[1] 98 was designed with the explicit goal of solving the language diversity problem. Hence the remaining question is: Can we build a powerful tracer for Haskell that does not depend on any specific implementation of Haskell?

**Tracing with Hat** A tracer gives the programmer access to otherwise invisible information about a computation. It is a tool for understanding how a program works and for locating errors in a program. Tracing a computation with Hat consists of two phases, trace generation and trace viewing:

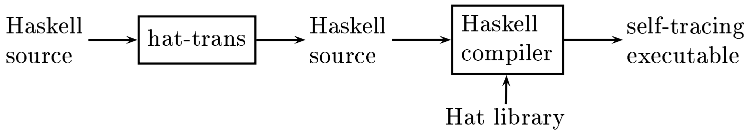



First, a transformed version of the program runs. In addition to its normal input/output behaviour it writes a trace into a file. Second, after the program has terminated, the programmer studies the trace with a collection of viewing

[1] In the following the name Haskell always refers to Haskell 98 as defined by the Haskell 98 report [5].

tools. The trace as concrete data structure liberates the views from the time arrow of the computation. The trace and the viewing tools are described in [9]. Here we focus on the trace generation part of Hat.

**Trace Generation through Transformation** Until recently the production of the self-tracing executable was integrated into the Haskell compiler nhc98[2]. We have now separated Hat from its host Haskell compiler. The new program hat-trans transforms the original Haskell program into a Haskell program that, when compiled and linked with a library provided with Hat, is self-tracing:




The separation between Hat and the compiler has the following advantages:

- Hat-trans and the Hat library together capture the essence of tracing.
- The small size of hat-trans and the library minimised the implementation effort and makes it easy to make experimental changes in the course of research.
- The future life of Hat is not tied to the future life, that is, continued support, of a specific compiler.
- Hat can be combined with Haskell compilers that have different characteristics, for example with respect to availability on certain computing platforms, compilation speed, or optimisation for speed or space.
- Hat is more easily accepted by programmers who wish to continue using a familiar compiler.

The obvious disadvantage of the new architecture compared to the old one is that hat-trans has to duplicate some work of a Haskell compiler, for example parsing. However, we will show that this duplicate work can be kept to a minimum and the implementation of nearly all duplicated phases can be shared between hat-trans and nhc98 without compromises.

In principle, the old nhc98-based implementation already consisted of a single transformation phase that was inserted into the front end of the compiler [7]. However, the implementation had several limitations and many small but crucial modifications had been made in the remainder of the compiler. For the separated hat-trans we had to develop various techniques to overcome a number of problems. Here we describe problems and solutions in the hope that the discussion will be useful for other people who build similar tools. In addition, we also point out features of Haskell that made our job particularly hard. Our observations may be taken into consideration in the future development of Haskell or similar languages.

The new Hat using the compiler-independent program transformation has been publicly released as Hat 2.0 (`http://www.cs.york.ac.uk/fp/hat`).

[2] `http://www.cs.york.ac.uk/fp/nhc98/`

## 2 How Tracing Works

It is not our aim in this paper to define the structure of Hat's trace, nor to describe its generation in all details. To get an understanding of what the trace looks like and how a transformed program generates it during its computation, let us consider an example.

**The Trace of a Reduction** Figure 1 shows several intermediate stages of the trace during the reduction of the term `f True`, using the definition `f x = g x`. Initially (a) there is the representation of the term as one application and two atom nodes. The first entry of each node points to a representation of the parent, the creator of the expression. Because our computation starts with `f True`, the parent is just a special node Root. In stage (b) the redex `f True` is "entered"; the result pointer of the application node changes from a null value to $\perp$. In stage (c) the representation of the reduct has been generated in the trace. The application node of the redex is the parent of all new nodes of the reduct. Finally (d) the result pointer of the redex is updated to point to its reduct.

A trace with its parent, subexpression and result pointers is a complex graph that is traversed by Hat's viewing tools. The "entered" mark $\perp$ is essential information when a computation aborts with an error. In general, several redexes may be "entered" at one time, because pattern matching forces the evaluation of arguments before a reduction can be completed.

**Augmented Expressions** The central idea for the tracing transformation is that every expression is augmented with a pointer to its description in the trace. Thus expression and its description "travel together" throughout the computation, so that when expressions are plumbed together by application, the corresponding descriptions can also be plumbed together to create the description of the application.

We transform an expression of type $T$ into an expression of type `R` $T$, where

```
data R a = R a RefExp
```

A value of `RefExp` is simply an integer offset of a node in the trace file.

**Transformed Program** Figure 2 shows the result of transforming our example, including additional definitions used. We assume `f` came with type signature `Bool -> Bool`. The program has been simplified for explanatory purposes.

In the first argument of `f`, respectively `g`, a pointer to its parent is passed. The original type constructor `->` is replaced by the new type constructor `Fun`. A self-tracing function needs to take an augmented argument and return an augmented result. The pointer to the parent of the right-hand side of the function definition, the redex, also needs to be passed as argument. Hence this definition of `Fun`.

The tracing combinator `ap` realises execution and tracing of an application. The primitive tracing combinators `mkAt`, `mkApp`, `entRedex` and `entResult` write to the trace file. They are side-effecting C-functions used via the standard FFI [1].

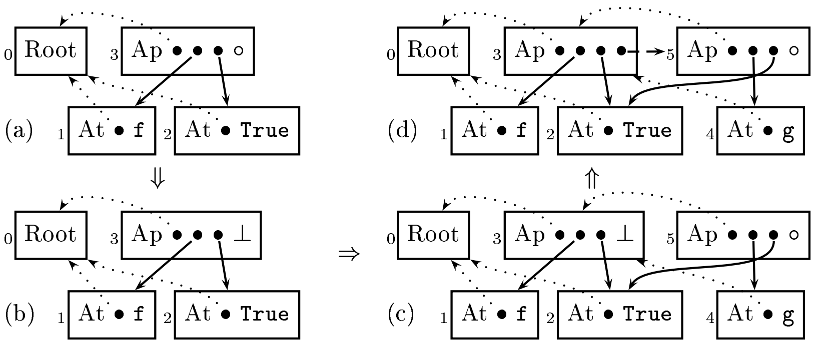



**Fig. 1.** Trace generation for a reduction step

```
f :: RefExp -> Fun Bool Bool
f p = R (Fun (\x r -> ap r (g r) x)) (mkAt p "f")
g p = R (...) (mkAt p "g")

newtype Fun a b = Fun (R a -> RefExp -> R b)
ap :: RefExp -> R (Fun a b) -> R a -> R b
ap p (R (Fun f) rf) a@(R _ ra) =
  let r = mkAp p rf ra
  in R (entRedex r `seq` case f a r of R y ry -> updResult r ry `seq` y) r
```

**Fig. 2.** Transformed program with additional definitions

```
    case (let p=mkRoot in ap p (f p) (R True (mkAt p "True"))) of R x _-> x
⇒*  case (ap • (R (...) (mkAt • "f")) (R True (mkAt • "True")) of R x _ -> x
                                                          mkRoot
⇒*  entRedex • `seq` case ((\x r->ap r (g r) x) (R True •) •) of
                 R y ry -> updResult • ry `seq` y
                                                       mkAp • • •
                   mkAt • "f"    mkAt • "True"                  mkRoot
(a)
⇒*  entRedex 3 `seq` case ((\x r->ap r (g r) x) (R True 2) 3) of
                 R y ry -> updResult 3 ry `seq` y
(b)
⇒*  case ((\x r->ap r (g r) x) (R True 2) 3) of
      R y ry -> updResult 3 ry `seq` y
⇒*  case (ap 3 (R (...) (mkAt 3 "g")) (R True 2)) of
      R y ry -> updResult 3 ry `seq` y
⇒*  updResult 3 • `seq` (entRedex • `seq` ...)
(c)                                 mkAp 3 • 2          mkAt 3 "g"
⇒*  updResult 3 5 `seq` (entRedex 5 `seq` ...)
(d)
⇒*  entRedex 5 `seq` ...
```

**Fig. 3.** Evaluation of self-tracing expression

**Tracing a Reduction** Figure 3 shows the reduction steps of the transformed program that correspond to the original reduction `f True` $\Rightarrow$ `g True`. The first line shows the result of transforming `f True`. The surrounding `case` and `let` are there, because it is the initial expression of the computation. The arrows indicate sharing of subexpressions, which is essential for tracing to work. Values of `RefExp` are the same integers as used in Figure 1. The reduction steps perform the original reduction and write the trace in parallel. In the sequence of reductions we can see at (a) how strictness of `entRedex` forces recording of the redex in the trace, at (b) the redex is "entered", at (c) strictness of `updResult` forces recording of the reduct, and at (d) the result pointer of the redex is updated.

**Properties of the Tracing Scheme** Only by bypassing the IO monad the transformed program can mostly preserve the structure of the original program. This preservation of structure ensures that the Haskell compiler determines the evaluation order, not Hat. Otherwise Hat would not be transformation-based but would need to implement a full Haskell interpreter. Only in few places the order of evaluation is enforced by `seq` and the fact that the primitive trace-writing combinators are strict in all arguments.

To simplify the transformation, `RefExp` is independent of the type of the wrapped expression. The correctness of the transformation ensures that the trace contains only representations of well-typed expressions.

The new function type constructor `Fun` is defined specially, different from all other types, because reduction of function applications is the essence of a computation and its trace. The transformation naturally supports arbitrary higher-order functions.

All meta-information that is needed for the creation of the trace, for example identifier names and source positions, is made available by the transformation as literal values. Thus Hat does not require any reflection features in the traced language.

## 3 The Hat Library

The Hat library comprises two categories of combinators: primitive combinators such as `entRedex` and `mkApp1` that write the trace file and high-level combinators that manipulate augmented expressions. The high-level combinators structure and simplify the transformation. The transformation only grows a program by a factor of 5-10. For the development of Hat it is useful that a transformed program is well readable and most changes to the tracing process only require changes of the combinator definitions, not the transformation.

Haskell demands numerous combinators to handle all kinds of values and language constructs, from floating point numbers to named field update. Figure 4 shows an excerpt of the real Hat library. The types `RefAtom`, `RefSrcPos` and `RefExp` indicate that there are different sorts of trace nodes. The trace contains references to positions in the original program source. The combinators `fun`$n$

```
fun1 :: RefAtom -> RefSrcPos -> RefExp  -> (R a -> RefExp -> R z)
     -> R (Fun a z)
fun1 var sr p f = R (Fun f) (mkValueUse p sr var)

ap1 :: RefSrcPos -> RefExp -> R (Fun a z) -> R a -> R z
ap1 sr p (R (Fun f) rf) a@(R _ ra) =
  let r = mkApp1 p sr rf ra in  wrapReduction (f a r) r

fun2 :: RefAtom -> RefSrcPos -> RefExp -> (R a -> R b -> RefExp -> R z)
     -> R (Fun a (Fun b z))
fun2 var sr p f = R (Fun (\a r -> R (Fun (f a)) r) (mkValueUse p sr var)

ap2 :: RefSrcPos -> RefExp -> R (Fun a (Fun b z)) -> R a -> R b -> R z
ap2 sr p (R (Fun f) rf) a@(R _ ra) b@(R _ rb) =
  let r = mkApp2 p sr rf ra rb
  in  wrapReduction (pap1 sr p r (f a r) b) r

pap1 :: RefSrcPos -> RefExp -> RefExp -> R (Fun a z) -> R a -> R z
pap1 sr p r wf@(R (Fun f) rf) a = if r == rf then f a r else ap1 sr p wf a

wrapReduction :: R a -> RefExp -> R a
wrapReduction x r =
  R (entRedex r `seq` case x of R y ry -> updResult r ry `seq` y) r
```

**Fig. 4.** Combinators from the Hat library

allow a concise formulation of function definitions of arity $n$. The combinators `wrapReduction` and `pap1` are just helper functions.

***N*-ary Applications** The combinator `ap2` for an application with two arguments could be defined in terms of `ap1`, but then two application nodes would be recorded in the trace. For efficiency we want to record $n$-ary application nodes as far as possible. Then we have to handle explicitly partial and oversaturated applications. The `pap1` combinator recognises when its first wrapped argument is a saturated application by comparing its parent with the parent passed to the function of the application. The `fun`$n$ are defined so that a partial application just returns the passed parent. If the function of `ap2` has arity one, then `pap1` uses `ap1` to record the application of the intermediate function to the last argument.

The fact that the function has arity one can only be recognised after recording the oversaturated application in the trace. Therefore the `ap2` combinator does not record the desired nested two applications with one argument each. Instead it constructs an application with two arguments whose reduct is an application with one argument. Because both applications have the same parent, the viewing tools can recognise applications of this sort in the trace and patch them for correct presentation to the user.

Often the function in an $n$-ary application is a variable `f` that is known to be of arity $n$. In that case the construction of `Fun` values and their subsequent destruction is unnecessary; the wrapped function can be used directly. A similar and even simpler optimisation is realised for data constructors; their arity is always known and they cannot be oversaturated.

**Further Optimisations** Preceding the transformation, list and string literals could be desugared into applications of `:` and `[]`. Such desugaring would however increase size and compile time of the transformed programs. Instead, special combinators perform the wrapping of these literals at runtime.

There is still considerable room left for further optimising combinators, which have not been the main focus in the development of Hat.

## 4 The Transformation Program Hat-trans

The tracing transformation hat-trans parses a Haskell module, transforms the abstract syntax tree, and pretty prints the abstract syntax in concrete syntax. Hat-trans is purely syntax-directed. In particular, hat-trans does *not* require the inclusion of a type inference phase which would contradict our aim of avoiding the duplication of any work that is performed by a Haskell compiler. Figure 5 shows the phases of hat-trans.

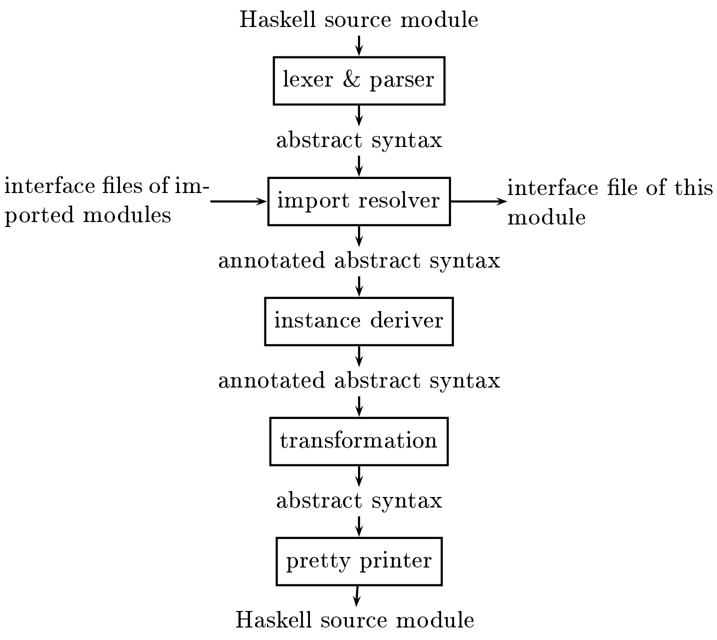



**Fig. 5.** Phases of hat-trans

To enable separate transformation of modules, an interface file is associated with every module, similar to the usual `.hi`-file. Haskell already requires for complete parsing of a module some sort of interface files that contain the user defined associativities and priorities of imported operators. Hat interface files also associate various other sorts of information with exported identifiers, such as arity of a function, the type constructor to which a data constructor belongs, etc. Hat-trans does not use the `.hi`-files of its collaborating compiler, because, first, this would always require the compilation of the original program before the tracing compilation and, second, every compiler uses a different format for its `.hi`-files. Hat-trans also does not generate its interface files in a format used by any compiler, because `.hi`-files always contain the type of every exported variable but Hat does not have these types.

The import resolver uses the import declarations of a module to determine for each identifier from where it is imported. This phase also finalises the parsing of operator chains and augments every occurrence of an identifier with the information which for imported identifiers is obtained from the interface files and otherwise is obtained syntactically by a traversal of the syntax tree. Whereas the import resolution phase of the nhc98 compiler qualifies each identifier with the identifier of the module in which it is defined, hat-trans leaves identifiers unchanged to ensure that pretty printing will later create a well-formed module.

The instance deriver replaces the `deriving` clause of a type definition by instances of the listed standard classes for the defined type. These derived instances need to be transformed (cf. Section 8) and obviously a Haskell compiler cannot derive instances of the transformed classes. To determine the context of a derived instance, Haskell requires full type inference of the instance definition. Because hat-trans does not perform type inference, it settles on generating a canonical context, that is, for an instance $C(T a_1 \ldots a_n)$ it generates the context $(C a_1, \ldots, C a_n)$. In principle, if this canonical context is incorrect, the Hat user has to write the correct instance by hand. But in practice we have not yet come across this problem.

The implementations of the lexer and parser and of the pretty printer are reused from nhc98. The import resolver and instance deriver have similarities to the corresponding phases of nhc98, but had to be implemented specially for hat-trans.

## 5 The Transformation

The transformation is implemented through a single traversal of the annotated abstract syntax tree.

**Namespace** The transformation leaves class, type, type variable and data constructor identifiers unchanged. Only special identifiers such as `(,)` and `:` have to be replaced by new identifiers such as `TPrelBase.Tuple2` and `TPrelBase.Cons`, qualified to avoid name clashes. Because many new variables are needed in the transformed program, every variable identifier is prefixed by a single letter. Different letters are used to associate several new identifiers with one original identifier, for example the definition of `rev` is transformed into definitions of `grev`, `hrev` and `arev`. The identifier and filename of a transformed module is prefixed by the letter “T”, for example “Module.hs” is transformed into “TModule.hs”. The Hat combinator library is imported qualified as “T” and qualified identifiers are used to avoid name clashes. As a result the development of Hat profits from readable transformed programs.

**Types** Because every expression has to be augmented with a description, in particular arguments of data constructors, type definitions need to be transformed. For example:

```
   data Point = P Integer Integer
⇝  data Point = P (T.R Integer) (T.R Integer)
```

Predefined types such as `Char`, `Integer` and `Bool` can be used unchanged, because the definition of an enumeration type does not change.

As type definitions are transformed, type signatures require only replacement of special syntactic forms and additional parent and source position arguments. For example:

```
   sort :: Ord a => [a] -> [a]
⇝  gsort :: Ord a => T.RefSrcPos -> T.RefExp -> T.R (Fun (List a) (List a))
```

The transformation has to accept any Haskell program and yield a well-formed Haskell program. Because partially applied type constructors can occur in Haskell programs, a transformation for the full language cannot just replace types of kind $*$, but has to replace type constructors. On the other hand, Haskell puts various restrictions on types that occur in certain contexts. For example, a type occurring in a qualifying context has to be a type variable or a type variable applied to types; a type in the head of an instance declaration has to be a type constructor, possibly applied to type variables. So it is important that the transformation does not change the form of types, in particular it maps type variables to type variables.

**Type Problems** In the last example the `Ord` in the transformed type refers to a different class than the `Ord` in the original type. The method definitions in the instances of `Ord` have to be transformed for tracing and hence also the class `Ord` needs to be transformed to reflect the change in types. Sadly the replacement of classes by new transformed classes means that the defaulting mechanism of Haskell cannot resolve ambiguities of numeric expressions in the transformed program. Defaulting applies only to ambiguous type variables that appear only as arguments of Prelude classes. Hence Hat requires the user to resolve such ambiguities. In practice, if an ambiguity error occurs when compiling the transformed program, a good tactic for the user is to add the declaration `default ()` to the original program and compile it to obtain a meaningful ambiguity error message. The ambiguities in the original program can then be resolved by type signatures or applications of `asTypeOf`.

The transformation of type definitions cannot preserve the strictness of data constructors. The transformation

```
    data RealFloat a => Complex a = !a :+ !a
⇝   data RealFloat a => Complex a = !(T.R a) :+ !(T.R a)
```

would not yield the desired strictness for `:+`. Ignoring this strictness issue only yields programs that are possibly less space efficient but it does not introduce semantic errors. Nonetheless, the transformation can achieve correct strictness by replacing all use occurrences of `:+` by a function that is defined to call `:+` but uses `seq` to obtain the desired strictness.

**Expression and Function Definitions** Figures 6 and 7 show the original and the transformed definition of a list reversal function `rev`. Most of the original definition is transformed into the new definition of `hrev`. The transformation wraps the patterns with the `R` data constructor to account for the change in types. The combinator `projection` records an indirection node (cf. [6]) and `con2` the application of a constructor to two arguments. The type of `hrev` still contains the standard function type constructor instead of the tracing function type constructor `Fun`. The function `grev` is the fully augmented tracing version of `rev`. The remaining new variables refer to meta-information about variables and expressions, for example `p3v13` refers to a position in line 3 column 13 of the original program.

```
rev :: [a] -> [a] -> [a]
rev [] ys = ys
rev (x:xs) ys = rev xs (x:ys)
```

**Fig. 6.** Original definition of list reversal

```
grev :: T.RefSrcPos -> T.RefExp -> T.R (Fun (List a) (Fun (List a) (List a)))
grev p j = T.fun2 arev p j hrev

hrev :: T.R (List a) -> T.R (List a) -> T.RefExp -> T.R (List a)
hrev (T.R Nil _) fys j = T.projection p3v13 j fys
hrev (T.R (Cons fx fxs) _) fys j =
  T.ap2 p4v17 j (grev p4v17 j) fxs (T.con2 p4v26 j Cons aCons fx fys)

arev = T.mkVariable tMain 3 1 3 2 "rev" TPrelBase.False

tMain = T.mkModule "Main" "Reverse.hs" TPrelBase.True

p3v13 = T.mkSrcPos tMain 3 13
p4v17 = T.mkSrcPos tMain 4 17
p4v26 = T.mkSrcPos tMain 4 26
```

**Fig. 7.** Transformed definition of list reversal

**Tricky Language Constructs** Most of Haskell can be handled by a simple, compositionally defined transformation, but some language constructs describing a complex control flow require a context-sensitive transformation.

A guard cannot be transformed into another guard. The problem is that the trace of the reduct must include the history of the computation of *all* guards that were evaluated for its selection, including all those guards that failed. Hence a sequence of guards is transformed into an expression that uses continuation passing to be able to pass along the trace of all evaluated guards.

The pattern language of Haskell is surprisingly rich and complex. Matching on numeric literals and $n + k$ patterns causes calls to functions such as `fromInteger`, `==` and `-`. The computation of these functions need to be recorded in the trace, in particular when matching fails. In general it is not even easy to move the test from a pattern into a guard, because Haskell specifies a left-to-right matching of function arguments.

An irrefutable pattern may never be matched within a computation but all the variables within the pattern may occur in the right hand side of the equation and need a sensible description in the trace. For variables that are proper subexpressions of an irrefutable pattern, that is those occurring within the scope of a `~` or the data constructor of a newtype, the standard transformation does not yield any description, because the `R` wrappers are not matched. We do not present the transformation of arbitrary patterns here, because it is the most complex part of the transformation.

**Preservation of Complexity** Currently a transformed program is about 70 times slower with nhc98 and 180 times slower with GHC[3] than the original program. This factor should be improved, but it is vital that it is only a constant factor. We have to pay attention to two main points to ensure that the transformation preserves the time and space complexity of the original program.

Although by definition Haskell is only a non-strict language, all compilers implement a lazy semantics and thus ensure that function arguments and constants (CAFs) are only evaluated once with their values being shared by all use contexts. To preserve complexity, constants have to remain constants in the transformed program. Hence the definition of a constant is transformed differently from the definition of a function. The definition of a constant *name* is transformed into the definition of a function `g`*name* and a constant `s`*name*. In Haskell not every variable defined without arguments is a constant; the variable may be overloaded. Fortunately the monomorphism restriction requires that an explicit type signature is given for such non-constant variables without arguments. Thus such cases can be detected without having to perform type inference. For correct sharing the new constant `s`*name* has to be defined in the same scope as in the original program. Hence every class obtains for every original method *name* not just the new method `g`*name*, but also a method `s`*name*. The latter is only used in an instance where the method is defined as constant.

[3] `http://www.haskell.org/ghc/`

Figures 6 and 7 demonstrate that a tail recursive definition is transformed into a non-tail recursive definition. Although the transformation does not preserve tail recursion, the stack usage of the tracing program is still proportional to the stack usage of the original program. This is, because the `ap2` combinator, which makes the transformed definition non-tail recursive, calls `wrapReduction`. That combinator immediately evaluates to an `R` wrapper whose first argument is returned after a single reduction step — not full evaluation.

## 6 Error Handling

Because debugging is the main application of Hat, programs that abort with an error or are interrupted by Control-C must still record a valid trace. An error message, a pointer to the trace of the redex that raised the error, and some buffers internal to Hat need to be written to the trace file before it can be closed.

**Catching Errors** Because Haskell lacks a general exception handling mechanism, Hat combines several techniques to catch errors:

- To catch failed pattern matching all definitions using pattern matching are extended by an equation (or case clause) that always matches and then calls a combinator which finalises the trace.
- The Prelude functions `error` and `undefined` are implemented specially, so that they finalise the trace.
- The C signalling mechanism catches interruption by Control-C and arithmetic errors.
- The transformed `main` function uses the Haskell exception mechanism to catch any IO exceptions.
- There exist variants of the Hat library for nhc98 and GHC that catch all remaining errors, in particular blackholes and out-of-memory errors. These variants take advantage of the extended exception handling mechanism of GHC (which does not catch all errors) and features of the runtime systems.

**The Trace Stack** The redex that raised an error is the last redex that was "entered" but whose result has not yet been updated. Most mechanisms for catching an error do not provide a pointer to the trace of this redex. In these cases the pointer is obtained from the top of an internal trace stack.

The trace stack contains pointers to the traces of all redexes that have been "entered" but not yet fully reduced. Instead of writing to the trace, `entRedex` $r$ puts $r$ on the trace stack. Later `updResult` $r$ $ry$ pops this entry from the stack and updates the result of $r$ in the trace (cf. Section 2). The trace stack shadows the Haskell runtime stack, that is, the two stacks grow and shrink synchronously. Besides a successful reduction, an IO exception also causes shrinking of the runtime stack. To detect the occurrence of a (caught) IO exception, `updResult` $r$ $ry$ compares its first argument with the top of the stack and keeps popping stack elements until the entry for the description $r$ is popped.

The stack not only enables the location of the redex that caused an error, it also saves the time of marking each "entered" application in the trace file. Only when an error occurs must all redexes on the stack be marked as "entered" in the trace file. Because sequential writing of a file is considerably more efficient than random access, `updResult` does not perform its update immediately either but stores it in a buffer. When the buffer is full all updates are performed at once. On our computers the use of stack and buffer nearly halves the runtime of the traced program.

## 7 Connecting to Untraced Code

For some functions a self-tracing version cannot be obtained through transformation, because no definition in Haskell is available. This is the case for primitive functions on types that are not defined in Haskell: for example, addition of `Int`s, conversion between `Int`s and `Char`s, IO operations and operations on `IOError`. We need to define self-tracing versions of such functions in terms of the original functions instead of by transformation. In other words, we need to lift the original function to the tracing types with its `R`-wrapped values.

**Calling Primitive Haskell Functions** Hat-trans (mis)uses the foreign function interface notation to mark primitive functions. For example:

```
    foreign import haskell "Char.isUpper" isUpper :: Char -> Bool

⇝   gisUpper :: T.RefSrcPos -> T.RefExp -> T.R (Fun Char Bool)
    gisUpper p j = T.ufun1 aisUpper p j hisUpper
    hisUpper :: T.R Char -> T.RefExp -> R Bool
    hisUpper z1 k = T.fromBool k (Char.isUpper (T.toChar k z1))
    aisUpper = T.mkVariable tPrelude 8 3 3 1 "isUpper" Prelude.False
```

The variant `ufun1` of the combinator `fun1` ensures that exactly the application of the primitive function and its result are recorded in the trace, no intermediate computation.

**Type Conversion Combinators** The definition of combinators such as

```
toChar :: T.RefExp -> T.R Char -> Prelude.Char
fromBool :: T.RefExp -> Prelude.Bool -> T.R Bool
```

that convert between wrapped and unwrapped types is mostly straightforward.

For a type constructor that takes types as arguments, such as the list type constructor, the conversion combinator takes additional arguments. The conversion combinators are designed so that they can easily be combined:

```
toList :: (T.RefExp -> T.R a -> b) -> T.RefExp -> T.R (List a) -> [b]
toString :: T.RefExp -> T.R String -> Prelude.String
toString = toList toChar
```

Some types have to be handled specially:

– No values can be recorded for abstract types such as `IO`, `IOError` or `Handle`. Instead of a value only its type is recorded and marked as abstract.
– For primitive higher-order functions such as `>>=` of the IO monad we also need combinators that convert functions. When a wrapped higher-order function calls a traced function, the latter has to be traced and connected to the trace of the whole computation.
  The function type is not only abstract but it is also contravariant in its first argument. The contravariance shows up in the types of the first arguments of the combinators. Only because `toFun` needs a `RefExp` argument, all unwrapping combinators take a `RefExp` argument.

```
toFun :: (T.RefExp -> c -> T.R a) -> (T.RefExp -> T.R b -> d)
      -> T.RefExp -> T.R (Fun a b) -> (c -> d)
toFun from to r (T.R (Fun f) _) = to r . f r . from r

fromFun :: (T.RefExp -> T.R a -> b) -> (T.RefExp -> c -> T.R d)
        -> T.RefExp -> (b -> c) -> T.R (Fun a d)
fromFun to from r f = T.R (Fun (\x _ -> (from r . f . to r) x))
                          (T.mkValueUse r T.mkNoSrcPos aFun)

aFun = T.mkAbstract "->"
```

**IO Actions** Although a value of type IO is not recorded in the trace, the output produced by the execution of an IO-action is. Primitive IO functions such as `putChar` are wrapped specially, so that their output is recorded and connected to the trace of the IO expression that produced it.

## 8 Trusting

Hat allows modules to be marked as trusted. The internal workings of functions defined in a trusted module are not traced. Thus Hat saves recording time, keeps the size of the trace smaller and avoids unnecessary details in the viewing tools. By default the Prelude and the standard libraries are trusted.

**No (Un)Wrapping for Trusting** The obvious idea is to access untransformed trusted modules with the wrapping mechanism described in the previous section. Thus the functions of trusted modules could compute at the original speed and their source would not even be needed, so that internally they could use extensions of Haskell that are not supported by Hat. However, this method cannot be used for the following reasons:

– It can increase the time complexity. Regard the list append function `++`: In evaluation `++` traverses its first argument but returns its second argument as part of the result without traversing it. However, the wrapped version of `++` has to traverse both arguments to unwrap them and finally traverse the whole result list to wrap it. Therefore the computation time for `xs ++`

(xs ++ ... (xs ++ xs)...) is linear in the size of the result for the original version but quadratic for the lifted version. Also the information that part of the result was not constructed but passed unchanged is lost.
– Overloaded functions cannot be lifted. For example, the function `elem` uses the standard `Eq` class, but its wrapped version `gelem` has to use the transformed `Eq` class. No combinator can change the class of a function, because it cannot access the implicitly passed instance (dictionary). Instances are not first class citizen in Haskell.

**Combinators for Trusting** So trusted modules have to be transformed as well. The same transformation is applied, only different combinators used. The computation of trusted code is not traced, but the combinators have to record in the trace for each traced application of a trusted function its call, the computations of any traced functions called by it, and its result.

In our first implementation of trusting, combinators did not record any reductions in trusted code, but all constructions of values. The disadvantage of this implementation is that not only the result value of a trusted function but also all intermediate data structures of the trusted computation are recorded.

Our current implementation takes advantage of lazy evaluation to only record those values constructed in trusted code that are *demanded* from traced code. Thus no superfluous values are recorded. However, sadly also values that are first demanded by trusted code and later demanded by traced code are not recorded either. It seems impossible to change this behaviour without losing the ability to record cyclic data structures, for example the result of the standard function `repeat`. The limitations of the current implementation of trusting are acceptable for tracing most programs, but not all.

The result values of trusted functions may contain functions. These functions are currently only recorded as abstract values, because otherwise they could show arbitrary large subexpressions of trusted code. The connection between trusting and abstraction barriers needs to be studied further.

# 9 Conclusions

We described the design and implementation of Hat's program transformation for tracing Haskell programs.

**Compiler Independence** We have used the new Hat together with both nhc98 and GHC (Hugs[4] and hbc[5] currently do not support the standard foreign function interface). Compiling a self-tracing program with both compilers and running the excecutables does not yield an identical trace file, because side effects of the trace recording combinators are performed at different times. However, manual comparison of small traces shows the *graph structure* of these traces to

[4] `http://www.haskell.org/hugs/`
[5] `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`

be the same. The size of large trace files differs by about $10^{-6}$ of their size, proving that sometimes different structures are recorded. We will have to build a tool for comparing trace structures to determine the cause of these differences. Semantic-preserving eager evaluation may cause structural differences, but otherwise the trace structure is fully defined by the program transformation, not the compiler.

**Language** The implementation of tracing through program transformation owes much to the expressibility of Haskell. Higher-order functions and lazy evaluation allowed the implementation of a powerful combinator library, describing the process of tracing succinctly.

Nonetheless we also faced a number of problems with Haskell. The source-to-source transformation exposed several irregularities and exceptions in the language design, for example the limited defaulting mechanism and the fact that class instances are not first class citizens. The limited exception handling mechanism forced us to have a tiny compiler-specific part in the Hat library. Finally, the sheer size of Haskell makes life hard for the builder of a tool such as Hat. Most language constructs can be translated into a core language, but because traces must refer to the original program, the program transformation has to handle every construct directly.

**Tracing through Program Transformation** Considering all the problems discussed here, is building a tracer based on a compiler-independent program transformation a good idea?

The Haskell tracing tool Hood [2] consists of a library only. Hence its implementation is much smaller and it can even trace programs that use various language extensions without having to be extended itself. Hood's architecture is actually surprisingly similar to that of Hat: the library corresponds to Hat's combinator library and Hood requires the programmer to annotate their program with Hood's combinators and add specific class instances, so that the program uses the library. Hat's trace contains far more information than Hood's and hence requires a more complex transformation with which the programmer cannot be burdened.

On the other end of the design space is the algorithmic debugger Freja [3], a compiler developed specially for the purpose of tracing. Its self-tracing programs are very fast. However, implementing and maintaining a full Haskell compiler is a major undertaking. Freja only supports a subset of Haskell and runs only under Solaris. Modifying an existing compiler is also near to impossible, because all existing Haskell compilers translate a program into a core language in early phases, but a trace must refer to all constructs of the original program. The implementation of a tracing Haskell interpreter would still require more work than the implementation of hat-trans, and achieving similar or better trace times would still be hard. Hat-trans reduces the duplication of implementation effort to a minimum.

Hat is an effective tool for locating errors in programs. We use it to locate errors in the nhc98 compiler and recently people outside York located subtle bugs in complex programs with Hat. Nonetheless, there is still considerable potential for extensions, and we hope that future developments will benefit from Hat's modular design and portable implementation.


## Acknowledgements

The work reported in this paper was supported by the Engineering and Physical Sciences Research Council of the United Kingdom under grant number GR/M81953.


## References


1. M. Chakravarty et al. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2002.
2. A. Gill. Debugging Haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. 2000 ACM SIGPLAN Haskell Workshop.
3. H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.
4. A. Penney. *Augmenting Trace-based Functional Debugging.* PhD thesis, University of Bristol, UK, September 1999.
5. S. L. Peyton Jones, J. Hughes, et al. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, Feb. 1999.
6. J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. In C. Clack, K. Hammond, and T. Davie, editors, *Selected papers from 9th Intl. Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS Vol. 1467, Sept. 1997.
7. J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS Vol. 1292, Sept. 1997.
8. P. Wadler. Functional programming: Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, Aug. 1998. Functional programming column.
9. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, UU-CS-2001-23. Universiteit Utrecht, 2001.
10. R. D. Watson. *Tracing Lazy Evaluation by Program Transformation.* PhD thesis, Southern Cross, Australia, Oct. 1996.