Finding the Needle

Stack Traces for GHC

Tristan O.R. Allwood
Imperial College
tora@doc.ic.ac.uk

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Susan Eisenbach
Imperial College
susan.eisenbach@imperial.ac.uk

Abstract

Even Haskell programs can occasionally go wrong. Programs calling *head* on an empty list, and incomplete patterns in function definitions can cause program crashes, reporting little more than the precise location where *error* was ultimately called. Being told that one application of the *head* function in your program went wrong, without knowing which use of *head* went wrong can be infuriating.

We present our work on adding the ability to get stack traces out of GHC, for example that our crashing *head* was used during the evaluation of *foo*, which was called during the evaluation of *bar*, during the evaluation of *main*. We provide a transformation that converts GHC Core programs into ones that pass a stack around, and a stack library that ensures bounded heap usage despite the highly recursive nature of Haskell. We call our extension to GHC *StackTrace*.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Haskell

General Terms Algorithms, Languages

Keywords Stack Trace, Debugging

1. Motivation

Well-typed Haskell programs cannot seg-fault, but they can still fail, by calling *error*. For example, *head* is defined thus:

```
head :: [a] -> a
head (x:xs) = x
head []     = error "Prelude.head: empty list"
```

If the programmer calls *head*, and (presumably unexpectedly) the argument is [], the program will fail in the following cryptic fashion:

```
> ghc -o main Main.hs
> ./main.exe
main.exe: Prelude.head: empty list
```

At this point, a programmer new to Haskell will ask "Which of the zillions of calls to *head* in my program passed the empty list?". The message passed to the *error* function in *head* tells the programmer the *local* reason for the failure, but usually provides insufficient *context* to pinpoint the error.

Haskell'09, September 3, 2009, Edinburgh, Scotland, UK.

If the programmer is familiar with debugging an imperative language, they would expect to look at a stack trace; a listing that shows *main* called *foo*, *foo* called *bar* and *bar* called *head*, which then called *error* and ended the program. This information is readily available, since the run-time stack, which the debugger can unravel, gives the context of the offending call. However, in a lazy language, the function that *evaluates* (*head* []) is often not the function that *built* that thunk, so the run-time stack is of little or no use in debugging.

In the work described here, we describe a modest modification to GHC that provides a similar functionality to that offered by call-by-value languages:

- We describe a simple transformation that makes the program carry around an explicit extra parameter, the *debug stack* representing (an approximation of) the stack of a call-by-value evaluator. Section 3.

  The debug stack is reified as a value, and made available to functions like *error*, to aid the user in debugging their program.
- Crucially, the transformation inter-operates smoothly with precompiled libraries; indeed, only the parts of the program under scrutiny need be recompiled. Section 3.2.
- We give an efficient implementation of the stack data structure, that ensures that the debugging stacks take only space linear in the program size, regardless of the recursion depth, run-time, or heap residency of the program. Section 4.

  For Haskell programs to pass around stacks, we had to design an appropriate stack data structure with associated library functions, these are outlined in Section 4.2.
- We built a prototype implementation, called *StackTrace*, in the context of a full-scale implementation of Haskell, the Glasgow Haskell Compiler. We sketch the implementation and measure the performance overhead of our transformation in Section 6.
- Our prototype implementation raised some interesting issues, which we discuss in Section 5.

Although it is very simple in both design and implementation, debug stack traces have an extremely good power-to-weight ratio. Since "`Prelude.head: empty list`" has so little information, even a modest amount of supporting context multiplies the programmer's knowledge by a huge factor! Sometimes, though, that still may not be enough, and we conclude by comparing our technique with the current state of the art (Section 7).

2. The programmers-eye view

We begin with a simple example of our implemented system. The following program should print out the second Fibonacci number.

```
1  module Main where
2
3  import Error
4
5  main :: IO ()
6  main = print $ fib 2
7
8  fib :: Int → Int
9  fib 1 = 1
10 fib n
11     | n > 1 = fib (n − 1) + fib (n − 2)
12 fib n = error′ $ "Fib with negative number: "
13                  ++ show n
```

However our programmer has made a small mistake:

```
> ghc --make -o Fib Fib.hs
> ./Fib
Fib: Fib with negative number: 0
```

Of course, 0 is not a negative number, and our programmer has just missed out a base case. But the first thing programmer wants to know when faced with such an error is: what was the call site of the offending call to *fib*? Our new tool makes this easy to answer, by simply adding the `-fexplicit-call-stack-all` flag:

```
> ghc --make -fexplicit-call-stack-all -o Fib Fib
> ./Fib
Fib: Fib with negative number: 0
in error', Error.hs:7,14
in fib, Fib.hs:12,9
in fib, Fib.hs:11,27
in main, Fib.hs:6,16
in main, Fib.hs:6,1
```

This shows that the call to $error'$ was made in function *fib*, on line 12 and column 9; that is what "`in fib, Fib.hs:12,9`" means, where the line numbers are given in the code sample above. In turn, the offending call to `fib` was made in *fib* on line 11, column 27; the *fib* $(n-2)$ call. In effect, we are provided with a stack trace of the offending call.

2.1 Stack elision

Once the program has been recompiled with call stack information applied, we can use GHCi to experiment with other calls to *fib*:

```
Prelude Main> fib 20
*** Exception: Fib with negative number: 0
in error', Error.hs:7,14
in fib, Fib.hs:12,9
in fib, Fib.hs:11,27
in fib, Fib.hs:11,13
...
```

Here, the "..."s mean some of the stack has been elided, because we have recursively called the same function from the same call site. In this case the interactive request for *fib* 20 will have forced the call to *fib* $(n-1)$ on line 11, column 13, which will then call itself another 19 times before then calculating *fib* 1 + *fib* 0. The *fib* 0 (from line 11 column 27) then fails as before.

If we were instead to keep the full stack trace, a program that looped would consume ever-increasing memory for the ever-growing stack.

Here is another example of this behaviour (at the bottom of our `Fib.hs` file):

```
15 firstLetters = loopOver ["hi", "world", "", "!"]
16
17 loopOver [] = []
18 loopOver (x : xs) = head′ x : (loopOver xs)
```

Here we have a small recursive loop that turns a list of lists into a list by taking the *head* element of each of the sublists. Running this through GHCi we can see that some recursion happened before the program took the *head* element of an empty list.

```
*Main> firstLetters
"hw*** Exception: head: empty list
in error', exs/Error.hs:7,14
in head', exs/Error.hs:14,12
in loopOver, Fib.hs:18,19
in loopOver, Fib.hs:18,30
...
in firstLetters, Fib.hs:15,16
```

Of course, the more idiomatic way of writing this would be to use a *map* combinator.

```
21 firstLetters2 = map′ head′ ["hi", "world", "", "!"]
```

```
*Main> firstLetters2
"hw*** Exception: head: empty list
in error', exs/Error.hs:7,14
in head', exs/Error.hs:14,12
in firstLetters2, Fib.hs:21,22
```

Now the stack trace may appear at first to be surprising, as there is no mention of the map'[1] function in it. This is due to map' taking $head'$ as a higher-order argument, and at present we do not propagate stacks into higher-order arguments (a point we will return to in Section 5.1). However the stack trace obtained does accurately convey that it is some application of the $head'$ function referenced in the source of *firstLetters2* that caused the error.

2.2 Selective debugging

A critical design goal is that *a program can be debugged without recompiling the entire program.* Although it is theoretically unimportant, this goal is absolutely vital in practice for several reasons:

- Libraries may be available only in binary form.
- The program may simply be tiresomely voluminous, so that whole-program recompilation is painful (e.g. libraries, again).
- The overheads of generating and passing around a stack trace for the entire program may be substantial and unnecessary for all but a small critical path.

These have proved serious obstacles for tools based on whole-program transformation, including the cost-centres of GHC's own profiler (Section 7).

We therefore provide support for *selective* debugging on a function-by-function basis. A typical mode of use is this:

- Function *buggy* in module `Bug` crashes (by calling *error*).
- The programmer asks GHC to generate call-site information for *buggy* by adding a pragma (a bit like a INLINE pragma) thus:

[1] Several of the example functions used have primes (') suffixed on. Because of a currently unresolved bootstrapping issue, it is challenging to recompile all the standard libraries with our transform turned on, so we have just rewritten a few standard prelude functions and rebuilt them (with the exception of $error'$, which is discussed later).

```
{-# ANN buggy Debug #-}
```

- The system is recompiled passing `-fexplict-call-stack` to GHC. Modules that call *buggy* need to be recompiled (to pass their call site information), but that is all. (Except that if optimisation is on (the `-O` flag), more recompilation may happen because of cross-module inlining.)
- The programmer re-runs the program.
- Now *buggy* still crashes, but the trace tells that it crashed in module `Help`, function *bugCall*.
- That might already be enough information; but if not, the programmer asks GHC to debug *bugCall* in module `Help`, and recompiles. Again, depending on the level of optimisation, only a modest amount of recompilation takes place.
- The process repeats until the bug is nailed.

There is a shorthand for adding a *Debug* pragma to *every* function in a module, namely passing the `-fexplicit-call-stack-all` flag while compiling the module (which can reside in an `OPTIONS_GHC` pragma on a module by module basis).

2.3 Reifying the stack trace

We have seen that $error'$ prints out the stack trace. But in GHC, $error'$ is just a library function, not a primitive, so one might ask how $error'$ gets hold of the stack trace to print. StackTrace adds a new primitive *throwStack* thus:

$$throwStack :: \forall\ e\ a.Exception\ e \Rightarrow (Stack \rightarrow e) \rightarrow a$$

The implementation of *throwStack* gets hold of the current stack trace, reifies it as a *Stack* value, and passes it to *throwStack*'s argument, which transforms it into an exception. Finally, *throwStack* throws this exception. The *Stack* type is provided by our tool's support library, and is an instance of *Show*.

Given *throwStack*, we can define $error'$ as follows:

$$error' :: [Char] \rightarrow a$$
$$error'\ m = throwStack\ (\lambda s \rightarrow ErrorCall\ (m +\!\!+ show\ s))$$

It is also possible to reify the stack trace elsewhere, as we discuss in the case study that follows.

2.4 Debugging for real

GHC is itself a very large Haskell program. As luck would have it, in implementing the later stages of StackTrace we encountered a bug in GHC, which looked like this at runtime:

```
ghc.exe: panic! (the 'impossible' happened)
  (GHC 6.11 for i386-unknown-mingw32): idInfo
```

Fortunately the project was far enough advanced that we could apply it to GHC itself. The error was being thrown from this function:

$$varIdInfo :: Var \rightarrow IdInfo$$
$$varIdInfo\ (GlobalId\{\,idInfo\_ = info\,\}) = info$$
$$varIdInfo\ (LocalId\{\,idInfo\_ = info\,\}) = info$$
$$varIdInfo\ other\_var = pprPanic\ \texttt{"idInfo"}\ (ppr\ other\_var)$$

Rewriting it slightly to use our *throwStack* primitive, and recompiling with the transform allowed us to gain some extra context:

```
{-# ANN varIdInfo Debug #-}
```
$$varIdInfo :: Var \rightarrow IdInfo$$
$$varIdInfo\ (GlobalId\{\,idInfo\_ = info\,\}) = info$$
$$varIdInfo\ (LocalId\{\,idInfo\_ = info\,\}) = info$$
$$varIdInfo\ other\_var = throwStack\ (\lambda s \rightarrow pprPanic\ (\texttt{"idInfo\textbackslash n"} +\!\!+ show\ s)\ (ppr\ other\_var) :: SomeException)$$

```
ghc.exe: panic! (the 'impossible' happened)
  (GHC 6.11 for i386-unknown-mingw32): idInfo
in varIdInfo, basicTypes/Var.lhs:238,30
in idInfo, basicTypes/Id.lhs:168,10
```

We then chased through the functions sprinkling on further *Debug* annotations until we gained a full stack trace that we used to nail the bug.

```
ghc.exe: panic! (the 'impossible' happened)
  (GHC 6.11 for i386-unknown-mingw32): idInfo
in varIdInfo, basicTypes/Var.lhs:238,30
in idInfo, basicTypes/Id.lhs:168,10
in idInlinePragma, basicTypes/Id.lhs:633,37
in preInlineUnconditionally,
    simplCore/SimplUtils.lhs:619,12
in simplNonRecE, simplCore/Simplify.lhs:964,5
in simplLam, simplCore/Simplify.lhs:925,13
in simplExprF', simplCore/Simplify.lhs:754,5
in simplExprF, simplCore/Simplify.lhs:741,5
in completeCall, simplCore/Simplify.lhs:1120,24
in simplVar, simplCore/Simplify.lhs:1032,29
in simplExprF', simplCore/Simplify.lhs:746,39
...
in simplExprF', simplCore/Simplify.lhs:750,39
...
in simplLazyBind, simplCore/Simplify.lhs:339,33
in simplRecOrTopPair,
    simplCore/Simplify.lhs:295,5
in simplTopBinds, simplCore/Simplify.lhs:237,35
in simplifyPgmIO, simplCore/SimplCore.lhs:629,5
in simplifyPgm, simplCore/SimplCore.lhs:562,22
in doCorePass, simplCore/SimplCore.lhs:156,40
```

This story seems almost too good to be true, but we assure the reader that it happened exactly as described: the original failure was neither contrived nor anticipated, and the authors had no idea where the bug was until the trace revealed it. Simple tools can work very well even on very large programs.

3. Overview of the implementation

StackTrace is a simple *Core-to-Core compiler pass* that transforms the program in GHC's intermediate language (Core, [9]) to pass an additional argument describing the call site of the current function. This extra argument is called the *call stack*. StackTrace comes with a *supporting library* to be described shortly.

The basic transformation is extremely simple. Suppose we have a user-defined function `recip`, with a *Debug* pragma (Section 3.1), and a call to it elsewhere in the same module:

```
{-# ANN recip Debug #-}
```
$$recip :: Int \rightarrow Int$$
$$recip\ x = \textbf{if}\ x \equiv 0\ \textbf{then}\ error\ \texttt{"Urk foo"}\ \textbf{else}\ 1\ /\ x$$
$$bargle\ x = ...(recip\ x)\ ....$$

The transformation (elaborated in Section 3.2) produces the following code:

$$recip :: Int \rightarrow Int$$
$$recip\ x = recip\_deb\ emptyStack$$

```
{-# ANN recip (Debugged 'recip_deb) #-}
```
$$recip\_deb :: Stack \rightarrow Int \rightarrow Int$$

```
recip_deb stk x = if x ≡ 0 then error stk' "Urk foo"
                           else 1 / x
                  where
                    stk' = push "in recip:14,23" stk
bargle x = ....(recip_deb stk x) ....
  where
    stk = push "in bargle:19:22" emptyStack
```

Notice several things here:

- The transformed program still has a function *recip* with its original type, so that the source-language type-checking is not disturbed. Also any dependent modules can be compiled without enabling the transform and still work normally.
- In the transformed program, *recip* simply calls the debugging version *recip_deb*, passing an empty stack trace. The name "*recip_deb*" is arbitrary; in our real implementation it is more like *recip_ $ _351*, to ensure it cannot clash with programmer-defined functions.
- The transformation adds a new annotation *Debugged*, which associates the original function *recip* with its (arbitrarily-named) debugging version *recip_deb*. We discuss this annotation further in Section 3.2.
- The debugging version, *recip_deb*, contains all the original code of *recip*, but takes an extra stack-trace parameter, and passes on an augmented stack trace to the call to *error*.
- *recip_deb* does *not* pass a stack trace to $(\equiv)$ or $(/)$. Why not? Because it cannot "see" a debugging version of these functions; we describe how it identifies such functions in Section 3.1.
- Even though *bargle* is not not marked for debugging, the call to *recip* in *bargle* is transformed to call *recip_deb* with a singleton stack. In this way, a single *Debug* annotation may cause many call sites to be adjusted. That is the whole point!

3.1 Debug pragmas

As discussed earlier (Section 2.2), our tool supports *selective* tracing, using pragmas to specify which functions should be traced.

For these pragmas we use a recent, separate, GHC feature, called *annotations* [10]. The annotations feature allows a user to associate a top level function or module name with a Haskell value, using an *ANN* pragma, thus:

```
f x = ...
  {-# ANN f True #-}
data Target = GPU | CPU deriving (Data, Typeable)
  {-# ANN f GPU #-}
```

The first pragma adds the association $(f, True)$, while the second adds (f, GPU). The associated value is any Haskell value that implements both *Data* and *Typeable*. (In fact, the "value" is implicitly a Template Haskell splice, which is run at compile time to give the value.) These annotations are persisted into GHC interface files, and can be read off later by users of the GHC API, the GHC Core pipeline itself, and eventually GHC plugins.

StackTrace provides a datatype *Debug* (exported by the tool's support library *GHC.ExplicitCallStack.Annotation*) for annotating user functions with:

```
data Debug = Debug deriving (Data, Typeable)
```

This is then used with the *ANN* (annotate) pragma to mark functions for debugging:

```
import GHC.ExplicitCallStack.Annotation (Debug (..))
...
{-# ANN foo Debug #-}
foo = ...
```

Note the import of *GHC.ExplicitCallStack.Annotation*: the data constructor *Debug* must be in scope before it can be mentioned, even in an annotation.

$$
\begin{array}{rcl}
[\![f = e]\!] & = & \left\{ \begin{array}{l} \texttt{\{-\#}\ ANN\ f\ (Debugged\ 'f\_deb)\ \texttt{\#-\}} \\ f = f\_deb\ emptyStack \\ f\_deb\ s = [\![e]\!]_s \end{array} \right. \\
 & & \text{if } f \text{ has a } Debug \text{ pragma} \\
 & = & f = [\![e]\!]_{emptyStack} \\
 & & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
[\![throwStack]\!]_s & = & \lambda f \rightarrow throw\ (f\ s) \\
[\![x_l]\!]_s & = & x\_deb\ (push\ l\ s) \\
 & & \text{if } x \text{ has (Debugged 'x\_deb) ann} \\
 & = & x \quad \text{otherwise} \\
[\![e1\ e2]\!]_s & = & [\![e1]\!]_s\ [\![e2]\!]_s \\
[\![\lambda x \rightarrow e]\!]_s & = & \lambda x \rightarrow [\![e]\!]_s \\
[\![\textbf{case}\ e1\ \textbf{of}\ p \rightarrow e2]\!]_s & = & \textbf{case}\ [\![e1]\!]_s\ \textbf{of}\ p \rightarrow [\![e2]\!]_s \\
[\![\textbf{let}\ x = e1\ \textbf{in}\ e2]\!]_s & = & \textbf{let}\ x = [\![e1]\!]_s\ \textbf{in}\ [\![e2]\!]_s
\end{array}
$$

Figure 1. The stack-trace transformation

3.2 The transformation

When the user compiles their code with a command-line flag, `-fexplicit-call-stack`, we run an extra compiler pass that transforms the program as sketched above. This section gives the details of the transformation.

The GHC compiler pipeline parses Haskell into a large, data structure that is then typechecked. This typechecked source is then de-sugared into the simpler, typed intermediate language *Core*. The Core program is then optimised before being passed to the back-end compiler for turning into an executable or byte-code.

Although we have presented the StackTrace transform above in terms of the surface Haskell syntax, we implement it as a Core-to-Core transformation, because Core is a much, much smaller language than Haskell. However, the transformation is run early, just after the Haskell program has been desugared into Core, but before it has been optimised. At this stage the Core program still bears a close resemblance to the original Haskell, with some exceptions as noted later in Section 5.4. For example, top level Haskell functions become top-level bindings, pattern matching is expanded out to case statements, etc. Some information does get lost; for example it is difficult to know whether a Core `let` bound variable has come from a Haskell `let` or `where` statement or compiler created variable (for e.g. working with type class dictionaries). This can cause difficulties when trying to accurately talk about Haskell level function scopes and source locations from within Core.

The transformation itself is presented in Figure 1. The transformation is applied to each top-level definition $f = e$. If it has a *Debug* annotation then the transformation generates:

- A new function f_deb with argument s (of type *Stack*), whose right hand side is $[\![e]\!]_s$.
- An impedance-matching definition for the original f, which calls f_deb passing the empty stack, *emptyStack* (defined by the support library).
- A new annotation is generated for f, that associates it with the value $(Debugged\ 'f\_deb)$, where *Debugged* is a data constructor declared in the support library as follows:

$$\textbf{data}\ Debugged = Debugged\ TH.Name$$

Its argument is a Template Haskell name, in this case the name of f's debugging variant. (Such quoted names are written in Template Haskell with a preceding single quote.)

If f does *not* have a $Debug$ annotation (Section 3.1), then much less happens: the right hand side e is simply transformed with $[\![e]\!]_{emptyStack}$, where $emptyStack$ is the empty stack trace, reflecting the fact that a non-debugged function has no stack-trace context.

The term transformer $[\![e]\!]_s$, also defined in Figure 1, simply walks over the term e, seeking occurrences of functions that have debug variants. How are such functions identified? With the exception of the special primitive $throwStack$, discussed shortly, they are the ones that have a $Debugged$ annotation, which gives the name of the debugging variant to be substituted. Remember that imported functions, as well as functions defined in this module, may have a $Debugged$ annotation. The new $Debugged$ annotation attached to f by the transformation is automatically preserved in the module's interface file, and will thereby be seen by f's callers in other modules.

The stack passed to x_deb is $(push\ l\ s)$. Here, l is the source location (source file, line and column number etc.) of this occurrence of x, written informally as a subscript in Figure 1. The other parameter s is the stack trace of the context. The function $push$ is exported by the support library, and pushes a location onto the current stack trace. The implementation of stack traces is described in Section 4.

There is a small phase-ordering question here. Since the top-level functions of a module may be mutually recursive, we must add all their $Debugged$ annotations before processing their right-hand sides, so that their mutual calls are transformed correctly.

The transform has been designed to preserve the existing API of a module. The original function name f in the binding $f = e$ is still available at the original type. As the definition of f now uses the debugged version with an empty initial stack, libraries compiled without the transform can still depend on it with no changes, and gain limited stack-trace benefits for free.

The transform is fully compatible with non-transformed libraries: a call to a library function is left unchanged by the transformation unless the library exposes a $Debugged$ annotation for that function.

3.3 Implementing throwStack

The primitive $throwStack$ is implemented in our library very simply, as follows:

$$\begin{aligned} &throwStack :: \forall\ e\ a.Exception\ e \Rightarrow (Stack \to e) \to a \\ &throwStack\ f = throw\ (f\ emptyStack) \end{aligned}$$

This provides a safe default for when it is used without StackTrace being enabled. The transformation then treats references to $throwStack$ as a special case, although you can imagine a debugged version of $throwStack$ would take the following shape:

$$\begin{aligned} &\{\text{-\# ANN throwStack (Debugged 'throwStack\_deb) \#-}\} \\ &throwStack\_deb :: \forall\ e\ a.Exception\ e \\ &\qquad\qquad \Rightarrow Stack \to (Stack \to e) \to a \\ &throwStack\_deb\ s\ f = throw\ (f\ s) \end{aligned}$$

Any call elsewhere to $throwStack$ will be transformed to a call to $(throwStack\_deb\ s)$ where s is the stack trace at that call site. Then $throwStack\_deb$ simply passes the stack to f, and throws the result. Simple.

The reader may wonder why we did not give $throwStack$ the simpler and more general type $(Stack \to a) \to a$. Since $throwStack$ is a normal Haskell function, if it had the more general signature, it could lead to a subtle break of referential transparency. Consider the following program (assuming the more liberal $throwStack$):

$$\begin{aligned} &\ldots \\ &\{\text{-\# ANN main Debug \#-}\} \\ &main = print\ (bar \equiv bar) \\ &\{\text{-\# ANN bar Debug \#-}\} \\ &bar :: String \\ &bar = throwStack\ show \end{aligned}$$

When run normally, the program would print out `True` as expected. However, if `-fexplicit-call-stack` is enabled during compilation, it would instead print out `False`.

The two different contexts of the bar call in $main$ are now visible. Since a debugging library should not affect the control flow in pure Haskell code, we decided to require that $throwStack$ diverges. An expert Haskell programmer can of course resort to the $unsafe*$ black arts should they really desire the more liberal function.

$$\begin{aligned} &\textbf{module}\ Stack\ \textbf{where} \\ &\quad emptyStack :: Stack \\ &\quad push :: Stack \to StackElement \to Stack \\ &\quad throwStack :: \forall\ e\ a.Exception\ e \Rightarrow (Stack \to e) \to a \end{aligned}$$

Figure 2. The signature of the Stack library in StackTrace.

4. Call Stacks

A key component of StackTrace is the data structure that actually represents stack traces. It is implemented by our support library, and has the signature given in Figure 2. This section discusses our implementation of stack traces. A key design goal was this:

- The maximum size of the stack is statically bounded, so that the debugging infrastructure adds only a constant space overhead to the program.

To maintain a precise stack trace would take unbounded space, of course, because of recursion, so instead we abbreviate the stack with "..." elisions, in order to bound its size. Section 2 showed some examples of this elision. But just what should be elided? We established the following constraints:

- The top of the stack accurately reflects the last calls made up to an identifiable point. This is important for debugging, so the user can know exactly what they do and don't know about what happened.
- Any function that would be involved in a full stack trace is represented at least once in this stack trace.

4.1 Eliding locations in the Stack

Our stack design has the following behaviour when pushing a source location l (file name, line and column numbers) onto a stack:

- Place l at the top of the stack.
- Filter the rest of the stack to replace the previous occurrence of l (if it exists) with a sentinel value "...".
- If "..." were inserted and are directly above/below another "..."s, they are collapsed into a single "...".

Some examples of this behaviour are in Figure 3, which depicts a stack trace as a list of labels and elisions, such as `a,...,b,-`. The young end of the stack is at the left of such a list, with "`-`" representing the base of the stack. In examples (1) and (2) and (5)

	Push	onto stack	gives result
(1)	a	-	a,-
(2)	b	a,-	b,a,-
(3)	a	b,a,-	a,b,...,-
(4)	b	a,b,...,-	b,a,...,-
(5)	c	b,a,...,-	c,b,a,...,-
(6)	c	c,b,a,...,-	c,...,b,a,...,-
(7)	b	c,...,b,a,...,-	b,c,...,a,...,-
(8)	a	b,c,...,a,...,-	a,b,c,...,-

Figure 3. Pushing elements onto our Stack (young end to the left)

the element being pushed is not already in the stack and is placed on top as would be expected. In example (3) the element (a) is already present and is therefore its original reference is replaced with "..."s, while it is placed on top. In (4) the same happens with element b, although the new "..."s would be adjacent to the ones placed in (3), so they collapse together. In (8) we see an extreme example where three "..."s would end up adjacent and are all collapsed together.

An alternative way of imagining the results of this algorithm is this: given a real stack trace, you can convert it to our stack trace by sweeping down the stack from the top. Whenever you see a source location you have seen before, replace it with a sentinel value "...". If multiple sentinel values appear consecutively, collapse them together. To see this in practice, imagine reading the push column in Figure 3 from bottom to top (which represents the real stack trace), replacing any duplicate elements with "...". Doing this on any line will yield that line's result.

Given that all stacks must start out as empty, and the only mutation operator is to *push* a source location (i.e. you can never push an "..."), we get several nice properties:

- Any source location referring to a usage of a top-level function occurs at most once in the call stack.
- A "..." is never adjacent to another "..."
- The number of elements in the call stack is bounded at twice the number of possible source locations that refer to usages of top level functions (follows from the previous two). It is of course likely to be much, much less than this since not all program locations can call into each other.
- A "..." represents an unknown number of entries/calls in the stack trace. However the "..." can only elide functions that are mentioned above the "...".
- The top of the stack accurately reflects what happened, down to the first "...".

4.2 Stack Implementation

The run-time stack trace is implemented as an ordinary Haskell library. The data structure representing the stack takes advantage of the sentinel value ('...') only ever occurring between two stack elements, and maintains this invariant implicitly.

$$
\begin{aligned}
\textbf{data}\ \mathit{Stack} = \mathit{Empty} &\{\, \mathit{stackDetails} :: !\mathit{StackDetails} \,\} \\
\mid \mathit{Then} &\{\, \mathit{stackDetails} :: !\mathit{StackDetails} \\
&\,, \mathit{stackElement} :: !\mathit{StackElement} \\
&\,, \mathit{restOfStack} :: !\mathit{Stack} \,\} \\
\mid \mathit{RecursionThen} &\{\, \mathit{stackDetails} :: !\mathit{StackDetails} \\
&\,, \mathit{stackElement} :: !\mathit{StackElement} \\
&\,, \mathit{restOfStack} :: !\mathit{Stack} \,\}
\end{aligned}
$$

The *Empty* constructor represents the empty stack, and *Then* is the way of placing a `StackElement` upon an existing stack. The *RecursionThen* constructor is used to represent a sentinel value between its `StackElement` and the stack below it. The

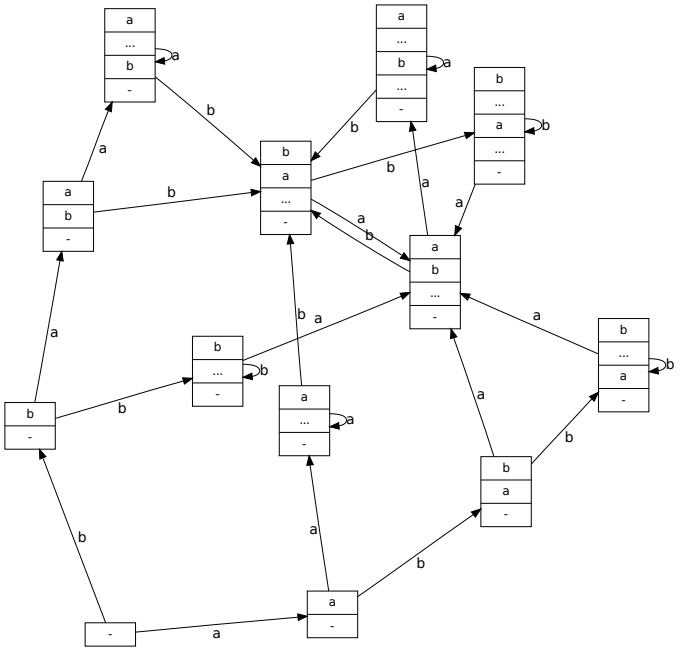


Figure 4. Complete transition diagram for our stack abstraction with two source locations. The empty stack is denoted by '-'. Edges represent pushing the named source location onto the stack.

`StackElement`s represent source locations. The `StackDetails` contain some bookkeeping information for each stack. When discussing stacks in constructor form, we will elide the `StackDetails`, meaning we can talk about stacks like a'Then'b'$\mathit{RecursionThen}$'Empty (which is a,b,...,-).

In Figure 4 we consider building a stack where we only have two possible items to put into it called a and b (these are actually locations in our source code, but the fact is unimportant for this example). The figure shows how the *push* function relates stacks via the source locations pushed onto them. The empty stack, '-', at the bottom left of the picture is the root of all the possible stack configurations.

For example, if a is the first source location reached, then the stack becomes a,- (following the a arrow from - to the right). From this position, if we reach source location b (following the b arrow to the right), then b is pushed onto the top of the stack as would be expected (giving b,a,-). If that source location b is recursively re-entered (following b again), then the first time the stack would transition to b,...,a,-, however any further pushes of the source location b would cause the stack to remain the same.

As the diagram shows, there are many possible configurations, and at runtime many of the shorter stacks appear in different contexts (for example main,- will be a suffix of all stacks).

4.3 Stack sharing and memoization

There are resource-related questions for stack traces:

- Every call $(\mathit{push}\ l\ s)$ must search s for occurrences of l. We would like to not do so repeatedly, giving *push* an amortised constant-time cost. We achieve this by memoising calls to *push*.
- Although elision means that each *individual* stack trace has bounded depth, there may be an unbounded number of them.

We would like to share their storage, so that the size of all stack traces *together* is bounded, independent of program runtime or data size. We can achieve this by hash-consing: that is, ensuring that for any particular stack trace there is at most one stack in the heap that represents it. Since the tail of a stack is also a stack, this implicitly means we share all suffixes of stacks.

We can memoise $push$ by attaching a memo table to each stack trace. The memo table for a stack trace s maps source locations l to the result of $(push\ l\ s)$. As a partial analogy, you could imagine that the arrows in Figure 4 represent the associations in the memo tables for each stack. The `StackDetails` data structure is where this memo table lives, which takes the following shape:

$$\begin{array}{l}\textbf{data}\ StackDetails\\ \quad = StackDetails\{\\ \qquad stackUnique :: !Unique,\\ \qquad stackTable\ \ :: !(MVar\ (HashTable\ StackElement\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Stack))\\ \quad\ \ \}\end{array}$$

The $stackTable$ is used to memoize the push calls. When $(push\ l\ s)$ is called, the stack s checks its $stackTable$ to see if the new stack has already been calculated (looking it up and returning if necessary); otherwise the new appropriate stack is built and the $stackTable$ is updated. Since we are using hashtables, the $Stack$s also need to be comparable for equality, and we use $stackUnique$ to provide a quick equality check.

4.4 The implementation of $push$

The use of memo tables alone, however, does not guarantee that all stacks in the heap are unique. The problem is that it could be possible to reach the same stack in multiple different ways. For example, the stack `a,b...,-` could be reached by: $push$ `a` $\circ push$ `b` $\circ push$ `a` $\$emptyStack$ or $push$ `a` $\circ push$ `b` $\circ push$ `b` $\$emptyStack$. In order to ensure each stack is only created once, we make our $push$ function generate new stacks using a canonical set of pushes upon a known memoized stack. The idea is to build all stacks incrementally using two "smart constructors" that can only alter the top of the stack, only ever operate on stacks that have already been memoized correctly and do not feature the program location about to be pushed. If these preconditions are met, they guarantee that all stacks are only ever created once and share all tails correctly.

- $pushl$ is the smart constructor for $Then$. It takes program location l and a known memoized stack s (not containing l), and checks s's memo table for l. If the check succeeds, it returns the memoized result. Otherwise it uses $Then$ to build a new stack trace, adds it to s's memo table, and returns it.
- $pushr$ is the smart constructor for $RecursionThen$. To guarantee all stacks are correctly shared, this constructor ensures that (for example) the generation of the stack `a,...,rest` given a known memoized stack `rest`: `a,rest` is memoized and the memo table for `a,rest` knows that when `a` is pushed upon it the result it `a,...,rest`.

  It achieves this by (using this example of $pushr$ `a rest`):

  - First using $pushl$ to build or lookup the stack `a,rest`
  - It then does a memo table check in `a,rest` for pushing `a`. If the check succeeds, it just returns the result. If it fails it picks apart the top of the stack and swaps the $Then$ for a $RecursionThen$, and then adds the mapping for pushing `a` onto `a,rest` to `a,...,rest`, before returning `a,...,rest`.

With these smart constructors in hand, the implementation of $(push\ l\ s)$ is easy:

$push$ `b (a,b,c,-)`

Action	Queue	Stack
(1) split stack at `b`	`b,a,...,`	`c,-`
(2) $pushr$ `a`		
(3) $pushl$ `a`	`b,`	`a,c,-`
(4) replace `a,` with `a,...,`	`b,`	`a,...,c,-`
(5) $pushl$ `b`		`b,a,...,c,-`

Figure 5. Example use of the smart constructors

1. Look up l in s's memo table. If the check succeeds, return the pre-computed result.
2. Search s for an occurrence of l. If none is found, just tail-call $(pushl\ l\ s)$ to push l onto s.
3. Starting from the suffix just below the occurrence of l (which cannot contain l), rebuild the stack using $pushl$ and $pushr$, omitting l. Finally use $pushl$ to push l onto the re-built stack.

We illustrate the third step with an example in Figure 5. In this example we are pushing `b` onto the stack `a,b,c,-`. In (1), $push$ splits the stack into a queue of things to be pushed, and the known memoized stack being built up. Notice that `b` has been placed at the front of the queue, and its original location replaced with a "...".

In (2) we take the last item of the queue (`a,...` which is really a value representing $a`RecursionThen`$), and since we need to create a $RecursionThen$, use $pushr$ to place that on the top of the new stack. $pushr$ first uses $pushl$ to put `a` on the top of the stack in (3), and then replaces the $Then$ constructor on the top of the new stack with $RecursionThen$ in (4). In (5) we take the next item off the queue, and since that needs to be separated using $Then$, we use $pushl$ to place it on the top of the stack.

Once the queue is empty, $push$ then updates the memo table of the original (pre-queue) stack to point to the final stack when (in this example) `b` is pushed.

4.5 Run-time example

We now demonstrate how our algorithm for pushing elements onto the stack, using memo tables, results in a bounded heap footprint. Using the following program:

$$\begin{array}{l}a = b\\ b = a\\ main = a\end{array}$$

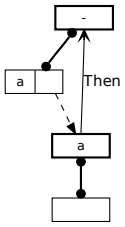
Imagine that $main$ is not being debugged, so our stack traces will only refer to the source locations b and a.

Initially there is a global constant empty stack available, with an empty memo table attached (Figure 6 - 1). Pushing program location `a` onto this stack first checks the memo table, but as it is empty we need to compute the new stack, and update the memo table. As the stack does not contain `a` already, $pushl$ can simply create a new $Then$ stack element (with its own empty memo table) and update $Empty$'s memo table to point to it (2).

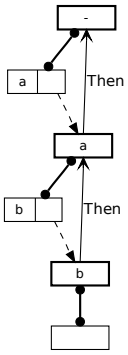
Pushing `b` onto this new stack follows similarly, giving a heap as in (3). Now we come to pushing `a` on top of `b,a,-`. Again the memo table is empty, so we need to compute a new stack. However the existing stack already contains an `a`, so $push$ splits the stack at `a`, giving a known memoized stack `-`, and a queue of `a,b,...`.

So in this example, the first item off the queue is `b,...`, which means $push$ will delegate to $pushr$. This then delegates to $pushl$ to first push b on to $Empty$, giving the heap layout in (4). Then, since we want a $RecursionThen$ between $Emtpy$ and b, $pushr$ will replace the top $Then$ with a $RecursionThen$, giving the situation in (5). Notice in this step we have initialized the new memo table

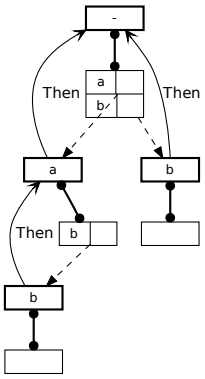
1. Empty Stack (-) with empty memo table.



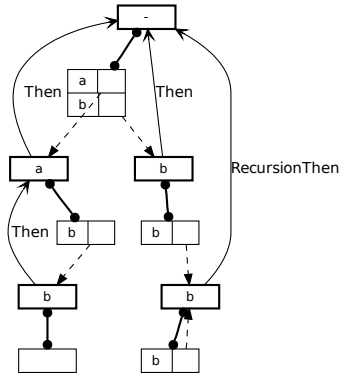
2. Pushing `a` onto the stack



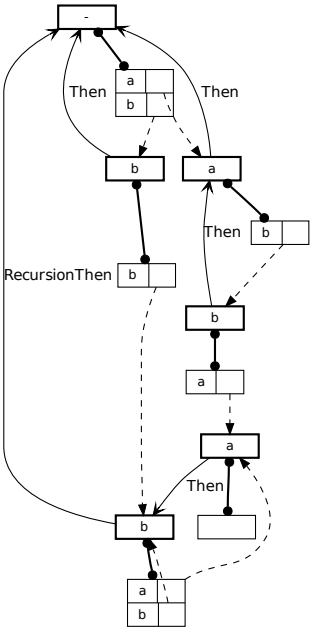
3. Pushing `b` onto the stack



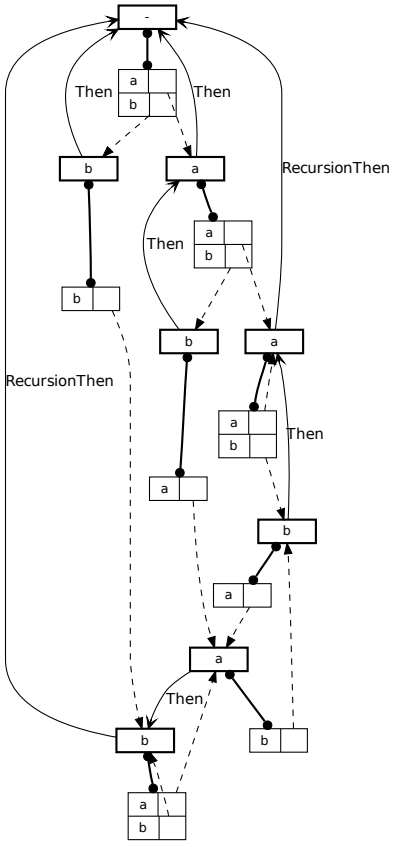
4. Pushing `b` onto `Emtpy`



5. Pushing `b` again to create a `RecursionThen`



6. Stack structure after re-entering `b` for the first time.



7. The final Stack structure.

Figure 6. Stack Pushing Example

with a self-reference loop because any further pushes of b will return to the same stack.

The only item left in the queue is the `a,`, which is pushed using $pushl$. Finally $push$ updates the `b,a,-` memo table to point to the resulting `a,b,...,-` stack (6).

The next iteration of the loop then pushes another `b`, transitioning the stack from `a,b,...,-` to `b,a,...,-` with associated updates to form the heap in (7). (7) also includes the final arc that the subsequent pushing of `a` creates.

5. Future Work

For the most part, StackTrace as described so far works well; well enough, for example, for it to be helpful in debugging GHC itself (Section 2.4). However there are some thorny open issues that need to be investigated to make it complete. How to deal with type classes is one problem, as these have non-trivial, cross-module interactions that a rewriting transform must take into account. Our stack trace transform also has potential negative effects on constant functions and the translation of mutually recursive functions with polymorphic / type-class arguments.

5.1 Stack traces for Higher Order Functions

There are times when it could be useful to have a more flexible call stack to the one currently implemented. Higher order functions are a good motivator of this. For example, consider the map function:

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f\ [\,] = [\,] \\
&map\ f\ (x : xs) = (f\ x) : map\ f\ xs
\end{aligned}
$$

and a use site:

$$1\ foo = map\ (error'\ \texttt{"..."})\ [1,2,3]$$

The call stack will be:

```
error ``...''
in foo, Blah.hs:1,12
in <foo's calling context>
```

even if we add an annotation to explicitly say we want to debug map, there will be no reference to map in the call stack. The reason for this is that map's argument f is never told (and has no way to know) that it is being applied inside map.

A natural solution to this problem would be to let the user somehow indicate that that the first argument to map should also accept a stack, giving a debugged version and new stack trace like so:

$$
\begin{aligned}
&map\_deb :: Stack \to (Stack \to a \to b) \to [a] \to [b] \\
&map\_deb\ s\ f\ [\,] = [\,] \\
&map\_deb\ s\ f\ (x : xs) \\
&\quad = f\ (push\ loc1\ s)\ x : map\ (push\ loc2\ s) \\
&\qquad\qquad (\lambda s' \to f\ (push\ loc3\ s') \\
&\qquad\qquad\quad xs) \\
&foo = \lambda stack \to map\ (push\ loc4\ stack) \\
&\qquad (\lambda stk \to error'\ (push\ loc5\ stk) \\
&\qquad\quad \texttt{"..."})\ [1,2,3]
\end{aligned}
$$

```
error "..."
in foo at loc5
in map at loc1
in foo at loc4
in <foo's calling method>
```

Now f also takes a stack indicating where it is used, and in the recursive case of $mapDebugged$, the fact that it is called inside map at *loc1* is presented to it.

The complications with implementing this scheme would be establishing which function arguments (or in fact any locally declared variable) could be useful to debug, and then keeping track of these so that we know to propagate the stack. The difficulty comes from realising that f is a local variable, whereas previously all debugged variants of things were top-level declarations that could easily be referred to in GHC.

5.2 Constant Applicative Form Expressions

Another problem area is the treatment of expressions in Constant Applicative Form (CAF's). Due to GHC's evaluation strategy, these will be evaluated once and their end result stored, as opposed to recomputing their value each time they are demanded. For example:

$$
\begin{aligned}
&e = expensive\ `seq`\ f \\
&main = print\ e \gg print\ e
\end{aligned}
$$

Here $expensive$ will only be computed once, the second reference to e in main will just get the result of whatever f evaluated to.

However, by adding the stack argument, and threading it through into $expensive$, we can dramatically change the runtime of the program:

$$
\begin{aligned}
&e\_deb\ stack \\
&\quad = expensive\ `seq`\ (f\_deb\ (push\ loc\ stack)) \\
&main = print\ (e\_deb\ (push\ loc1\ emptyStack)) \gg \\
&\qquad print\ (e\_deb\ (push\ loc2\ emptyStack))
\end{aligned}
$$

Now, since e_deb accepts an argument (which is different in both cases), and GHC is unaware of our invariant that stacks do not change user-visible control flow, then both invocations of e_deb will require the recomputation of $expensive$, each with the different stack variable passed in.

This is a very hard problem to solve in general, although we mitigate this by allowing the user to explicitly state which parts of the program should be rewritten - which allows stack traces to remain performant even in the presence of expensive CAF expressions.

5.3 Type Class Design Space

We want the StackTrace pass to degrade gracefully if future modules compiled without StackTrace are compiled against StackTrace altered modules. This means any changes StackTrace makes to a module have to preserve the existing interface of the module. For simple functions, record selector functions and even mutually recursive functions, no definition can cross a module boundary and so making a change in an API compatible way is straightforward. However type classes can be instantiated in different modules to where they are declared, and used in a different set of modules again. It could be possible, for instance, for a use-site of a type-class instance to see declared instances that have come from modules both compiled with and without StackTrace enabled.

Consider the following two modules:

$$
\begin{aligned}
&\textbf{module}\ MClassC\ \textbf{where} \\
&\quad \textbf{class}\ C\ a\ \textbf{where} \\
&\qquad c :: a \to Bool \\
&\textbf{module}\ MUseC\ \textbf{where} \\
&\quad \textbf{import}\ MClassC \\
&\quad useC :: C\ a \Rightarrow a \to Bool \\
&\quad useC = \neg \circ c
\end{aligned}
$$

Here we have a module declaring a type class C with a simple function c. And a module that just uses class C in a generic way.

If we `Debug` annotate $useC$, and propagate the stack into the c in its definition, the debugged version of $useC$ would be:

$$useC\_deb\ stack = \neg \circ (c\_deb\ (push\ loc\ stack))$$

The question is now, where does the c_deb name come from? Is it is generated by rewriting the type-class `C` as follows?

```
module MClassC where
  class C a where
    c :: a → Bool
    c_deb :: Stack → a → Bool
    c_deb _ = c
```

Now the original class declaration is expanded with a new function, and we give it a default implementation to ensure later clients compiled without StackTrace have a sensible implementation of it.

Instance declarations for class C that are compiled with the transform turned on could then generate a c_deb function to give a stack propagating version of their c instance, others would get the API safe, but stackless, default implementation.

However there are downsides to this approach. Firstly, GHC the internal representation of a type-class is currently fixed very early on in the compiler pipeline, and altering that fixed definition would invalidate some invariants in later stages of the compiler.

The second problem is that it requires the class declaration itself to be available to be annotated by the user. If the class declaration is buried deep in a library without a debugged annotation attached, then any user code that has control flow through a user instance declaration would have its stack essentially reset.

An alternative approach would be to create a new typeclass that contains the debugged definitions of functions and to change the rewritten functions to require the presence of the new typeclass (if it exists) instead of the original. So for our example, we would generate instead:

```
class (C a) ⇒ C_Deb a where
  c_deb :: Stack → a → Bool
useC_deb :: (C_Deb a) ⇒ Stack → a → Bool
useC_deb stack = ¬ ∘ (c_deb stack)
```

However, we currently have some open questions for this design. If we allow the user to declare that the c function should have a debugged version available, but not need to annotate the class declaration in its declaring module, then we have to ensure that any potential users of the debugged version can see the declaration of the debugged version. For this example, it may require an extra import in $MUseC$ to pull in the new declaration. It also requires that any instance declarations can see the debugged version of the typeclass so they can make instances of it.

There are some other, more serious, issues however. For example imagine a class with two functions; and imagine that separately we create two debugged versions of the class, each debugging a different function. Now we can have a function that can witness both of these debugged versions - do we create debugged versions of it for all possibilities of debug information available?

```
module Urg where

class Urg a where
  u1 :: a → Bool
  u2 :: a → Bool

module Urg1 where
  import Urg
 {-# ANN u1 Debug #-}     -- Which generates:
class (Urg a) ⇒ Urg_Deb_1 a where
  u1_deb :: Stack → a → Bool

module Urg2 where
  import Urg
 {-# ANN u2 Debug #-}     -- Which generates:
class (Urg a) ⇒ Urg_Deb_2 a where
  u2_deb :: Stack → a → Bool

module UseUrgs where
  import Urg1, Urg2, Urg
 {-# ANN d Debug #-}
d :: Urg a ⇒ a → Bool
d x = u1 x ∧ u2 x
```

Our Urg module exports a typeclass with two member functions. Then in separate modules, we request that the member functions be debugged. Finally in module $UseUrgs$ we ask to debug the function d. The question is now, do we expand out all the possibilities for the debugged version of d, such as:

```
d_Deb_1 :: Urg_Deb_1 a ⇒ Stack → a → Bool
d_Deb_1 stack x
  = u1_deb (push loc stack) x ∧ u2 x
d_deb_2 :: Urg_Deb_2 a ⇒ Stack → a → Bool
d_deb_2 stack x
  = u1 x ∧ u2_deb (push loc stack) x
d_deb_1_2 :: (Urg_Deb_1 a,
              Urg_Deb_2 a) ⇒ Stack → a → Bool
d_deb_1_2 stack x
  = u1_deb (push loc stack) x ∧
    u2_deb (push loc stack) x
```

5.4 Mutually recursive functions with type parameters / type class dictionaries

One of the few cases in which GHC Core does not intuitively resemble the original Haskell source is in the treatment of mutually recursive functions with type parameters / type class dictionaries.

By default, the following set of bindings:

```
f 0 = error' "Argh!"
f x = g (x − 1)
g x = f x
```

Normally desugars into (roughly) the following Core language:

```
fg_tuple = Λa.λd_num : Num a →
  let {
    d_eq = getEqDict d_num
    f_lcl = λx : a → case (((≡) a d_eq) 0 x) of
      True → error' "Argh"
      False → g_lcl (((−) a d_num) x 1)
    g_lcl = λx : a → f_lcl x
  } in
    (f_lcl, g_lcl)
f = Λa.λd_num : Num a →
    case (fg_tuple a d_num) of
      (f_lcl, g_lcl) → f_lcl
g = Λa.λd_num : Num a →
    case (fg_tuple a d_num) of
      (f_lcl, g_lcl) → g_lcl
```

The actual definitions of f and g end up living in f_lcl and g_lcl inside the **let** in fg_tuple. Hoisting them into this **let** means that the functions do not need to apply their counterparts to the type variable a and dictionary d_num (the arguments to fg_tuple) on the recursive call, as they are just in scope. This has obvious benefits in terms of keeping the code size down (it could blow up exponentially otherwise), but also (because the calculation of the `Eq` dictionary d_eq, needed for finding the definition of $(\equiv)$, becomes cached) maintains the full laziness property that GHC supports. A fuller explanation for this can be found in [4].

However, when we add the stack transform, this occurs:

```
fg_tuple = λstack.Λa.λd_num : Num a →
  let {
    d_eq = getEqDict d_num
    f_lcl = λx : a → case (((≡) a d_eq) 0 x) of
      True → error′ (push pos stack) "Argh"
      False → g_lcl (((−) a d_num) x 1)
    g_lcl = λx : a → f_lcl x
  } in
    (f_lcl, g_lcl)
f = λstack.Λa.λd_num : Num a →
      case (fg_tuple (push pos stack) a d_num) of
        (f_lcl, g_lcl) → f_lcl
g = λstack.Λa.λd_num : Num a →
      case (fg_tuple (push pos stack) a d_num) of
        (f_lcl, g_lcl) → g_lcl
```

The stack is modified in f and g when entering fg_tuple, and again in f_lcl before calling $error'$ (the latter causing the non-Haskell-source variable fg_tuple to appear in the stack trace). However the stack does not get modified when the recursion between f_lcl and g_lcl occurs. This means invocations of say $f\ 100$ and $f\ 0$ will produce the same output stacks, despite the fact that a lot of recursion will have happened in the former case.

In theory it could be easy to detect the code structure above and special-case-modify it to pass the call stack as desired. Unfortunately by the time we get to the desugared Core, the link between the tuple fg_tuple and the top-level selectors being used to encode mutually recursive functions is gone. There is no way to know that the let-bound f_lcl, g_lcl are really the implementations of top-level functions.

To get around this, we have added an optional flag to the desugarer to do a more naive translation. However this can result in large code-blowup and duplication, and removes the full laziness property. We present some preliminary results from using this transform in the following section.

An alternative approach would be to add an annotation to the Core as it is being generated to describe the mutually recursive bind. However how this annotation would be persisted in the presence of core-rewriting optimisations is an open question.

6. Evaluation

Although this work is prototypical and experimental in nature, we have used the *nofib* [7] benchmark suite to gain an insight into the possible compile and runtime costs of StackTrace on non-erroneous programs. The full logs of the *nofib* results are available from [1].

We ran the test-suite three times. Once using a clean GHC head snapshot, and twice using our patched version of the GHC head, once using only our simple desugaring rule for mutually recursive functions (`-fds-simple`, see Section 5.4) and once rewriting all sources to pass stacks through (`-fexplicit-call-stack-all`).

As none of the *nofib* programs crash, and do not use our *throwStack* function anywhere, we are not going to see call stacks at runtime, however it is useful to see the performance impact of this work when enabled on full programs.

Our prototype implementation was able to compile and run all programs with `-fds-simple` enabled, and 75 of the 91 programs could be tested under `-fexplicit-call-stack-all`.

Comparing the original GHC to our modified version with `-fds-simple` turned on, we see that there is an average of over 11% cost in terms of runtime and memory allocations for just using the simple desugaring strategy (though the maximum increase in time was over thirteen in the multiplier program). Compile times (excluding those programs that failed to compile) were on average 2.5% slower, although one standard deviation ranged from -18.5% to 28.5%.

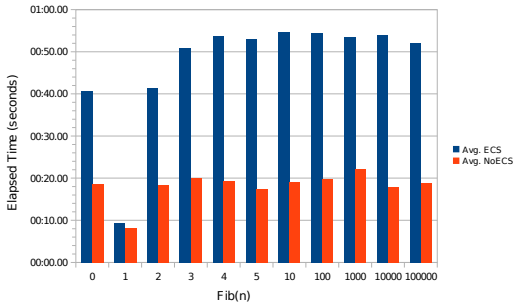


Figure 7. Graph of average runtimes for the erroneous Fibonacci function with and without StackTrace enabled

Comparing the original GHC to our modified version with `-fexplicit-call-stack-all` turned on, we see that there is an average of over five times the cost in terms of runtime and memory allocations. Compile times were on average 71% slower, with one standard deviation ranging from 14.0% to 157.4%.

The experiments with the *nofib* benchmark suite indicate that some work is still necessary in ironing out the bugs in the prototype. There are many different parts in the entirety of the GHC pipeline, and some of the *nofib* programs have teased out otherwise undiscovered interactions between the pipeline and the changes necessary to enable the stack transform. However, for the vast majority of programs, it is possible to apply our stack passing transform to the entire program, and still run it with a modest, but perfectly acceptable, performance hit.

As a smaller benchmark, we have taken the example erroneous *fib* program from the Example in Section 2, and compared its runtime with and without the explicit call stack transform enabled. Our benchmark calls *fib* with the indicated n, forcing the resulting exception (if there is one). This is done 10,000,000 times in a loop. For each n, we performed this experiment 5 times. The average results are presented graphically in Figure 7.

Calling *fib* where n is 1 doesn't call $error'$, and indicates there is less than a 20% cost in just adding the call stack information to the program. When n is 10 or greater, the resulting stack from the error is always the same, and calculating it increases the runtime by approximately 180%.

What the results also show is that the overhead is mostly in printing the stack (which most normal use-cases would do only once), as opposed to any calculation that occurs with each push onto the stack, as there is no consistent increase in runtime as the size of the *fib* argument increases from 10 to 100 to 1000 etc.

There is an increase in performance when n is 0 or 2 compared to when n is 10 or greater with the transform enabled. When n is 0 or 2, the resulting stack is smaller and simpler (it features no recursion) than in the other cases - again this is indicative that the formatting of the stack is much more expensive than the actual creation of the stack.

7. Related Work

There are already several ways of debugging existing Haskell programs. GHC currently ships with an interactive mode that features several debugging features, [5], [3]. Along with the standard options for setting breakpoints, and inspecting current local variables when execution is paused, it also features a `:trace` mode, which allows the build up of a dynamic execution stack. Currently this is

limited to the last 50 execution steps. It is also only available for code that is run in interpreted mode.

The difference in approach in keeping an accurate but bounded stack, versus our abstracted stack has advantages and disadvantages. For cases where the program control flow does not exceed 50 execution steps deep then certainly the accurate stack is more helpful. However a tight loop of greater than 50 iterations would remove any of the preceding context, and would not provide any more information beyond the loop running for over 50 iterations. Our abstracted stack on the other hand would indicate that the loop was re-entered at least once, and would keep the (abstracted) context above the loop. It is possible that some form of hybrid approach that keeps the full stack up to some limit and then starts abstracting away recursion could provide the best of both worlds, which we leave open to future work.

Another existing tool is the Haskell Tracer, HAT [11]. This provides the ability to trace Haskell 98 (plus most common extensions) programs and extract a Redex Trail (a full record of all the reductions that happened in the program). From this Redex Trail, they provide several different views with the trace that can aid in debugging a program. One of these is a call stack (provided through the tool *hat-stack*). As the authors note, this call stack (and ours) is not the real lazy evaluation stack, but

> "gives the virtual stack showing how an eager evaluation model would have arrived at the same result."

Although building a full Redex Trail could be quite expensive for a large application, HAT is designed to stream this out to disk and thus not cripple performance on large programs. Also of note is the difference in when the tracing code is applied; HAT works by first pre-processing the program, whereas we have integrated directly with GHC. While this in theory gives us the advantage of being able to reasonably easily track new GHC extensions to Haskell (because we are buffered from them by using Core unlike HAT which has to then upgrade its parser, internal model and other features), we do not yet have a good story for tracing (for example) type-classes, which HAT can currently do perfectly.

It is also possible to re-use the GHC profiling tools in order to get stack traces out of GHC. When profiling, GHC associates the runtime costs (memory / cpu use) to cost centers [8], and it builds up an abstracted stack of these at runtime as different functions are evaluated. The abstraction scheme used is to prune the stack back to the previous entry for a cost center when one is recursively re-entered. When a program crashes, it is possible to acquire the current cost-center stack, and thus get an indication of what the root causes of the crash could be. Although the abstraction scheme employed is somewhat lossy, in practice this is probably not an issue; the success or failure of using the cost center stacks for stack traces depends on the accuracy and resolution of the cost centers themselves. By default GHC creates a single cost center for an entire function definition, and so tracing through individual cases can be tricky. However the user is free to declare a new cost center anywhere by annotating an expression with an `SCC` pragma.

Another related tool that has an integrated component into GHC is HPC [2] (Haskell Program Coverage). This transforms a Haskell program into one that uses *tick boxes* to record when expressions are evaluated at runtime, and then allows visualisation of this data in terms of marked-up source code to see which expressions where or where not executed. Unlike our approach of rewriting GHC Core, they perform their transform earlier in the pipeline, just before the Haskell AST is desugared into Core. This means they have a data structure that much more closely resembles the original source program to work with. As a possible alternative target in the pipeline for a fuller implementation, HPC demonstrates that before Core is a reasonable target.

JHC [6] features an annotation, `SRCLOC_ANNOTATE`, that instructs the compiler to make any use sites of a function call an alternate version that receives the call-site location. Although this requires more work from the user (they also have to implement the version of the function that is passed call-site information), it is a simple and flexible tool.

8. Conclusions

We have presented StackTrace, our prototype for adding the ability to get stack traces out of crashing GHC-Haskell programs. We have given an intuitive overview of how Haskell programs are rewritten to pass an explicit stack around, and then given details on the actual transformation used on the GHC Core language. Accompanying the stack passing transform is a stack data structure and associated API that models the current call stack, while ensuring bounded heap usage by abstracting away recursively entered functions. We have discussed some current limitations and areas for future work, and presented some initial results from using our work on the *nofib* benchmark suite.

Acknowledgments

This work was undertaken while Tristan Allwood was on an internship at Microsoft Research Cambridge. We like to thank Thomas Schilling, Max Bolingbroke and Simon Marlow for long and interesting discussions and guidance during this work. We also wish to thank the anonymous reviewers for their detailed comments. Tristan is supported by EPSRC doctoral funding.

References

[1] T. Allwood, S. P. Jones, and S. Eisenbach. Explicit call stack paper resources. `http://code.haskell.org/explicitCallStackPaper/`.

[2] A. Gill and C. Runciman. Haskell program coverage. In G. Keller, editor, *Haskell*, pages 1–12. ACM, 2007.

[3] G. U. Guide. The ghci debugger. `http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci-debugger.html`.

[4] S. P. Jones and P. Wadler. A static semantics for haskell. Draft paper, Glasgow, 91.

[5] S. Marlow, J. Iborra, B. Pope, and A. Gill. A lightweight interactive debugger for haskell. In G. Keller, editor, *Haskell*, pages 13–24. ACM, 2007.

[6] J. Meacham. Jhc. `http://repetae.net/computer/jhc/jhc.shtml`.

[7] W. Partain. The nofib benchmark suite of haskell programs. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1992.

[8] P. Sansom and S. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Langauges and Systems*, 19(1), 1997.

[9] M. Sulzmann, M. M. T. Chakravarty, S. L. P. Jones, and K. Donnelly. System F with type equality coercions. In F. Pottier and G. C. Necula, editors, *TLDI*, pages 53–66. ACM, 2007.

[10] G. Trac. Annotations. `http://hackage.haskell.org/trac/ghc/wiki/Annotations`.

[11] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In R. Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, Sept. 2001. Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).