CHALMERS

Stack Traces in Haskell

Master of Science Thesis

ARASH ROUHANI

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, March 2014

Stack Traces for Haskell

A. ROUHANI

©A. ROUHANI, March 2014.

Examiner: J. SVENNINGSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering

Department of Computer Science and Engineering
Göteborg, Sweden March 2014

Abstract

This thesis presents ideas for how to implement Stack Traces for the Glasgow Haskell Compiler. The goal is to come up with an implementation with such small overhead that organizations do not hesitate to use it for their binaries running in production. Since the implementation is aiming for efficiency it will be heavily tied to only GHC. This work has been made possible thanks to a very recent contribution [1] that implements debug data for binaries compiled with GHC. Thanks to that contribution, this thesis can almost entirely focus on managing the GHC stack. Three different designs of stack values is presented, they allow creation in constant time and we implement one of these designs. The overhead of these designs can be kept small by utilizing laziness and the special linked list structure of the GHC stack. The other contribution is the work on the Haskell API that is exposed to programmers. We have implemented an API where the Haskell programmer can create the stack value at will and examine its content. Different ways of incorporating stack traces into the catching and throwing mechanism have been analyzed and we have found a rethrowing semantics for Haskell that is backwards compatible, convenient to use and easy to implement in GHC. The design in this paper allows stack values to be first class values.

Acknowledgements

I would like to thank my supervisor Josef and my opponent Dima at Chalmers for always being available and helpful. This work would have never been possible without Peter, who helped me every time I got stuck. Simon have been guiding and reviewing my work and was kind and arranged for me to meet Peter in person at Facebook's office in London. I am also very grateful for Pepe's feedback and review of my thesis. Last but definitely not least, I would like to thank my two parents and my two brothers, who have been there for me throughout my whole life.

Arash Rouhani, Göteborg, Sweden, March 2014

Contents

1 Introduction 1

2 Background 2
2.1 Stack traces . . . 2
2.2 Haskell . . . 3
2.2.1 Error handling in Haskell . . . 4
2.2.2 Functional Programming Concepts . . . 5
2.3 Glasgow Haskell Compiler . . . 9
2.3.1 The stack in GHC . . . 9
2.3.2 The runtime system . . . 10
2.4 From source to machine code . . . 10
2.4.1 The intermediate representations in GHC . . . 10
2.4.2 Generating debug data . . . 13
2.5 DWARF . . . 15

3 Related work 17
3.1 Debugging Haskell . . . 17
3.1.1 GHCi Debugger . . . 18
3.1.2 ghc-vis . . . 18
3.2 Avoiding Crashing . . . 18
3.2.1 Catch . . . 18
3.3 Inefficient stack traces . . . 19
3.3.1 Explicit call stack . . . 19
3.3.2 Stack traces with profiling . . . 20
3.4 Recent work . . . 20

4 The Execution Stack 23
4.1 Number of stacks . . . 23
4.2 What's on the Stack? . . . 24
4.2.1 Fields and arguments . . . 26

4.2.2 The members of the stack . . . 27
4.3 Structure . . . 29
4.3.1 Current stack pointer . . . 29
4.3.2 Buffering . . . 30
4.3.3 Stack squeezing . . . 30

5 Reifying the Stack **33**
5.1 Frames of interest . . . 33
5.1.1 Update frames . . . 35
5.1.2 The other frames . . . 38
5.1.3 Artificial frames . . . 39
5.2 Efficient reification . . . 40
5.2.1 Reifying a constant number of frames . . . 42
5.2.2 Static analysis . . . 42
5.2.3 Stack freezing . . . 43
5.2.4 Stack thawing . . . 44
5.2.5 Chunked reifying . . . 45

6 A Haskell Interface **47**
6.1 User-invoked reification . . . 47
6.2 Exception system . . . 51
6.3 Adding the trace . . . 55
6.3.1 Catching the stack . . . 55
6.3.2 Rethrowing the stack . . . 58

7 Conclusions **61**
7.1 Did we answer the question formulation? . . . 61
7.2 What should be done next? . . . 62

Bibliography **70**

1

Introduction

PART OF SOFTWARE DEVELOPMENT IS DEBUGGING. Debugging is the activity of diagnosing flawed software, in practice this means finding programming mistakes in the program source code. Debugging is usually initiated when the running software behaves unexpectedly, like crashing. When a program crashes, it'll be required for a programmer to diagnose it in order to find the root cause and correct it. As software systems become increasingly complex, they will grow in code size and the debugging phase becomes a more involved process.

From an economical perspective, a lack of funding or commercial success can stagnate the growth of a software project. But from a technological perspective, a software project stagnates due to lack of technology that scales. There are software tools that make big software systems manageable, even for hundreds of developers and millions of lines of code. These tools include development environments, version control and programming language features like interfaces. All of which are part of a programmers day to day work. Another set of tools that become essential as software systems grows are those that ease debugging.

This work implements and analyzes an implementation of *stack traces* for the programming language *Haskell*. Stack traces is a language feature that prints out extra context on a program crash, making the task of debugging the software easier [2]. Haskell is both a research language [3, 4] and used in industry [5, 6].

2

Background

The reader of this thesis is likely to know Haskell very well and read this thesis to understand the GHC stack (chapter 4) or to understand the stack traces proposal of this thesis (chapter 5 and 6). However, this thesis must be approachable to my fellow class mates according to the thesis regulations at Chalmers. That includes students with almost no prior Haskell knowledge. Haskell is a big language with many aspects and it takes many years to master. Therefore, the Haskell background which is in section 2.2 focuses on what is going to be essential to understand *this* thesis.

2.1 Stack traces

When a computer program crashes, the runtime of some programming languages gives some context to where in the code the program crashed. Typically, a *stack trace* is printed. A stack trace is the listing of the functions that have called each other and have not exited yet, so they have all been part of the crash. The first function in the stack trace is always the program entry point, the last function is where the crash actually occurred.

The Rosetta Code wiki contains a code sample in Ruby illustrating a stack trace [7], reproduced here in figure 2.1. As the figure shows, some languages can print stack traces at any time, not only after crashes. Stack traces can not be implemented as a regular user-level library, stack traces will need to look at the internal state of the run time system or interpreter. In the case of Ruby, they have the magical primitive `caller` which retrieves the call chain. It would not be possible for a user to implement `caller` in pure Ruby. To implement stack traces in Haskell is no exception, the implementation we present in this work will need to look at the internal state of the runtime system and has to target a specific Haskell compiler. The compiler we target is GHC.

```
def outer(a,b,c)
  middle a+b, b+c
end

def middle(d,e)
  inner d+e
end

def inner(f)
  puts caller(0)
  puts "my arg is #{f}"
end

outer 2,3,5
```

(a) Ruby code printing a stack trace.

```
$ ruby stacktrace.rb
stacktrace.rb:10:in ‘inner’
stacktrace.rb:6:in ‘middle’
stacktrace.rb:2:in ‘outer’
stacktrace.rb:14
my arg is 13
```

(b) Output of running the Ruby program.

Figure 2.1: Illustration of a simple stack trace.

```
main = print (fibonacci 10)

fibonacci :: Int -> Int
fibonacci 1 = 0
fibonacci 2 = 1
fibonacci x = fibonacci (x - 1) + fibonacci (x - 2)
```

Figure 2.2: A simple Haskell program.

2.2 Haskell

Haskell is a lazy, functional, general-purpose programming language [8]. Haskell first appeared in 1990 [9] and has since released the major standards Haskell 98 and Haskell 2010 [8].

Figure 2.2 shows a simple program in Haskell. Two functions are defined in this program, `main` and `fibonacci`. The explicit type signature for the function `fibonacci` means that it takes an int and returns an int. If a type signature is omitted, like for `main`, Haskell will infer it automatically.

```java
int integerDivision (int nom, int den) throws ArithmeticException {
  if (den == 0) {
    throw ArithmeticException("Division by zero");
  }
  else {
    return nom / den;
  }
}
```

(a) A total function in Java.

```haskell
integerDivision :: Int -> Int -> Maybe Int
integerDivision n 0 = Nothing
integerdivision n d = Just (n ‘div‘ d)
```

(b) A total function in Haskell.

Figure 2.3: Two total functions.

2.2.1 Error handling in Haskell

In order for stack traces to be relevant for a programming language, programs must have the notion of *crashing*. Intuitively, crashing causes sudden stops in execution, either by the operating system or by the language's own exception handling. Program crashes can be disastrous, since they will also terminate processes that are supposed to be long-running. Hence there are language constructs to eliminate some causes of program crashes. For instance, evaluating a well-typed expression in Haskell can not segfault [10].

In theoretical computer science, there's a notion of a function being *total*. Meaning that a function will terminate and not return any error. Therefore, such a function can not crash. Unfortunately, as of the famous halting problem it's not possible to decide if a function will terminate or not. That implies that you can't in general verify that a function is total [11, p.380]. The good news, though, is that whenever we explicitly *choose* to crash, we can systematically avoid it. This is not only true in the language Haskell, Java implements this through the `throws` keyword [12]. To annotate the function signature with `throws ArithmeticException` would mean that the function may throw an `ArithmeticException`. Figure 2.3a shows a Java integer division function that is total.

In Haskell, the most common way to implement "safe" division is to return the special value `Nothing` instead of dividing by zero. To do this in Haskell, the *Maybe* wrapper must be put in the function signature. See figure 2.3b.

The two functions in figure 2.3 will not crash when dividing by zero, rather, they

```java
int integerDivisionUnsafe (int nom, int den) {
  return nom / den;
}
```

(a) A partial function in Java.

```haskell
integerDivisionUnsafe :: Int -> Int -> Int
integerDivisionUnsafe n 0 = error "Division by zero"
integerDivisionUnsafe n d = n `div` d
```

(b) A partial function in Haskell.

Figure 2.4: Two partial functions.

gracefully return a value of either the division or a value representing failure. But there's a drawback, both these functions are cumbersome to use. In Java the programmer needs to explicitly catch the Exception combining the `try` and `catch` constructs [13, 14]. In Haskell, an additional layer of pattern matching is required. Due to this inconvenience, both languages allow for carrying out integer division without forcing the caller to do any error handling. Figure 2.4 shows two partial functions. Partial functions are controversial in Java [15] and discouraged when unnecessary in Haskell [16].

For the first time we now see the `error` function in Haskell (figure 2.4b). `error` is a special built-in function that terminates execution and outputs the provided message. While it's not entirely accurate, we could think of `error` being the only gateway to crashing a Haskell program. That means that all the typical dangerous operations like integer division by zero or indexing outside an array would just invoke the `error` function. We define "crashing" to be whenever `error` is called.

The conclusion is that Haskell has two major types of error values [17] [18], errors in total functions (figure 2.3b) and errors in partial functions (figure 2.4b). In this thesis we only care about the latter.

2.2.2 Functional Programming Concepts

The language that we want to add stack traces to is a lazy functional programming language. It is important to be aware of this when implementing stack traces for Haskell. In this subsection we will look at how Haskell expressions are evaluated. The concepts presented here is fundamental knowledge in Haskell programming.

Equational Reasoning

We will not go into details of how the Haskell language specification defines evaluating an expression, but there is no concept of a entering a function then let it return. Instead

```haskell
-- We can define a few functions (or equations)
f x = g x + h x
g x = 5 + h x
h x = x + 2

-- If Haskell is evaluating a value like (f 10), you can apply the
-- equations from above and reach the same result as your Haskell
-- program would.
f 10                      ==> -- equation for f
g 10 + h 10               ==> -- equation for g
(5 + h 10) + h 10         ==> -- equation for h
(5 + (10 + 2)) + h 10 ==> -- (+) (we treat it as a primitive)
(5 + 12) + h 10           ==> -- (+)
17 + h 10                 ==> -- equation for h
17 + (10 + 2)             ==> -- (+)
17 + 12                   ==> -- (+)
29                            -- Done!
```

Figure 2.5: Haskell can be understood by equational reasoning.

you think of it as successively applying equations until you reach a completely evaluated value. Applying an equation having the form $lhs = rhs$ on an expression means to substitute lhs in the expression to rhs, figure 2.5 shows this by example. To understand equational reasoning is helpful when learning Haskell.

Recursion and Tail Calling

Haskell has no statements, only expressions[1]. So obviously there can be no loop-statements like the `for`-statement or `while`-statement. As a general purpose language, Haskell naturally has a replacement for loops, namely *recursion*. Figure 2.6 shows two implementations of the mathematical function $fun(limit, acc) = acc + \sum_{x=1}^{limit} x$, one in Haskell and one in the imperative language C.

When recursion is used as a replacement for loops in C programming, it is often a poor choice. Recursion uses stack space, it needs to save both local variables and a return address on the stack for each function call. But this is not always true, in some cases it is possible for the compiler to optimize the recursive code into code using loops! This optimization is called *tail call optimization*. For functional programming languages, this optimization is of utmost importance, as recursion is the standard way of doing control flow. A tail call is a like a regular call, only that the caller is replacing its current stack

[1]Haskell has no statements, but the code can be in an imperative looking style when using the `do`-syntax

```haskell
fun acc 0      = acc
fun acc limit =
  fun (acc + limit)
      (limit - 1)
```

(a) Haskell version. Implemented with recursion.

```c
// 'acc' is like a start-value
int fun(int acc, int limit) {
  while (limit != 0) {
    acc   = acc + limit;
    limit = limit - 1;
  }
  return acc;
}
```

(b) C version. Implemented with loops.

Figure 2.6: Two functions.

frame instead of creating a new stack frame. The insight is that if a call will not return to the calling function (say if the call is the last statement), it is safe to overwrite the current stack frame.

In Haskell, the intuition is the same, but the technical explanations don't carry over. There are no statements, so there is no notion of a last statement. In chapter 4 we will see how function calls and jumps in Haskell are implemented for the Glasgow Haskell Compiler.

Purity

Haskell is a *pure* language where functions do not have side effects and this is a good consequence of equational reasoning. So any function `fun` can be run twice with the same arguments and will always return the same value. There will also be no side effects of running it twice. From figure 2.7 we see that purity is a necessity for equational reasoning. Note how the first step is turning *one* occurrence of `g` into *two* occurrences by applying the equation for `f`. If running `g` would have had side effects, this would not have been possible.

Laziness

The critical reader would question why equational reasnoning evaluates expressions from the outside like we do in figure 2.7. When expanding the equation for `f` we get two occurrences of `(g 10)`. Why did we not evaluate `(g 10)` first? It would have reduced the number of evaluation steps since we evaluate `(g 10)` to `13` and then continue by evaluating `(f 13)`. It would have worked in this case, but we can not in general reduce `(f (g 10))` to `(f 13)`. That would violate the *laziness* property of the language. Another phrasing for using this evaluation strategy is that Haskell does *outermost reductions*.

Figure 2.8 shows an expression that must be evaluated with lazy evaluation. A strict evaluation strategy would evaluate the arguments before applying the equation for `myIf`, leading to a crash instead of `5`.

```
-- Equations:
f x = x + x
g y = y + 3

-- Let's evaluate (f (g 10))
f (g 10)            ==> -- equation for f
(g 10) + (g 10)     ==> -- equation for g
(10 + 3) + (g 10) ==> -- (+)
13 + (g 10)         ==> -- equation for g
13 + (10 + 3)       ==> -- (+)
13 + 13             ==> -- (+)
26                  ==> -- Done!
```

Figure 2.7: A pure language is a requirement for equational reasoning.

```
-- Equations:
myIf True  onTrue onFalse = onTrue
myIf False onTrue onFalse = onFalse
crash = error "scary side effect!"

-- Let's evaluate (myIf False crash (2 + 3))
myIf False crash (2 + 3) ==> -- equation for myIf
2 + 3                    ==> -- (+)
5                        ==> -- Done!
```

Figure 2.8: Laziness is not an implementation detail, it is a mathematical necessity.

Thunks

But how do we then ensure that we only evaluate `(g 10)` once? Any sensible Haskell implementation will not evaluate `(g 10)` twice like in figure 2.7. The solution is to substitute `(f (g 10))` with `(f t) where t = (g 10)` where `t` is a *thunk*. Figure 2.9 shows the same value being evaluated as in figure 2.7 but with thunks. Note that it requires fewer steps to do evaluation with thunks. Haskell (in most compilers, in most cases) uses this improved evaluation strategy with thunks, this evaluation strategy is called *outermost reductions with graph reduction* [19].

```
-- Let's evaluate (f (g 10))
f t
  where t = (g 10) ==> -- equation for f
t + t
  where t = (g 10) ==> -- equation for g
t + t
  where t = 10 + 3 ==> -- (+)
t + t
  where t = 13     ==> -- (+)
26                 ==> -- Done!
```

Figure 2.9: Equational reasoning with thunks.

2.3 Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is a Haskell2010 compatible compiler [20]. With it you can compile Haskell source code to an executable binary. Here's an invocation of the compiler on the program sample from figure 2.2.

```
$ ghc --make Fibonacci.hs
...
$ ./a.out
34
```

GHC as of today supports many features in addition to the Haskell2010 standard, like parallelism, many optimizations, a LLVM backend, profiling and more [20].

2.3.1 The stack in GHC

Most programmers have a decent picture of how programming languages implement functions. Whenever a function is called, its arguments are pushed on the stack by the caller and the caller jumps to the function's code. When the function finally exits, it returns to where it was called from and pops the stack arguments[2]. This was a short reminder of how the *regular stack* works. Most programming languages use this to implement functions.

Due to the nature of Haskell, it is not clear if the stack that worked so naturally for languages like C can be used to implement Haskell. How does it work with partial applications? How does it work for laziness? Instead one might look at creating a completely new execution machine. The *Spineless Tagless G-Machine* (STG) from [21] is implemented in GHC [22] in the sense that Haskell is compiled down to the STG

[2]Whether if the caller or the callee should pop the arguments will depend on the *call convention*, but we do not need to worry about it here.

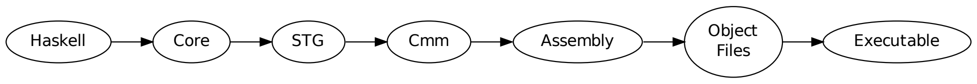


Figure 2.10: The GHC phases of compilation. The LLVM and C backend are not shown.

language at some point during compilation. The STG machine has a stack called the *execution stack*. The details of the execution stack changes as new versions of GHC are released. We have documented the execution stack in chapter 4 by examining the source code of the 7.6.2 version of GHC.

2.3.2 The runtime system

Many implementations of programming languages have a run time system (RTS) and GHC has a run time system too. The RTS in GHC is written mostly in C. The feature list is long, but the rule of thumb is that whatever can not be implemented in pure Haskell has to be implemented in the run time system. Some examples include garbage collection, an implementation of arrays, synchronization primitives, Software Transactional Memory, green threads and actual parallelization [23].

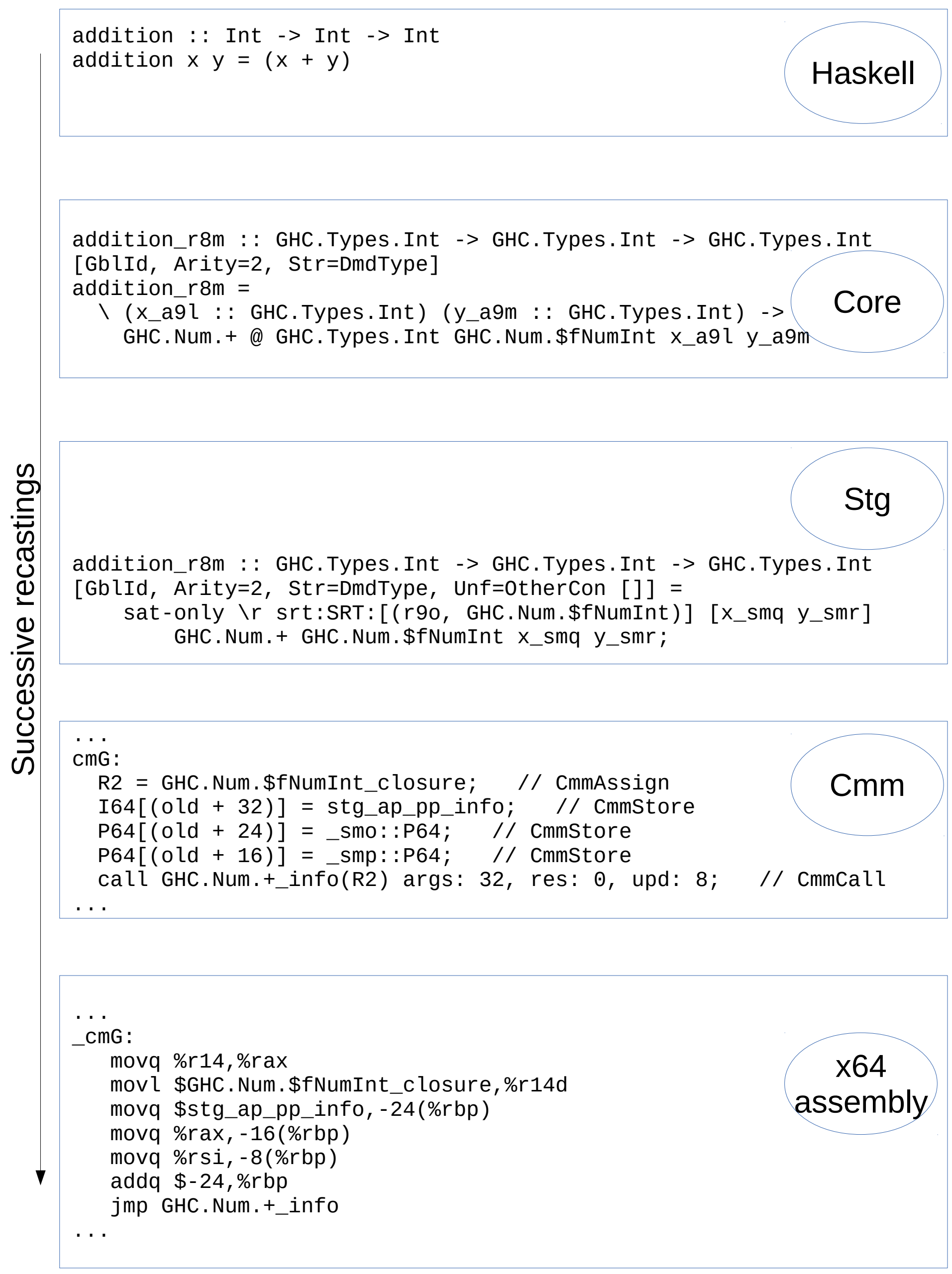
In this thesis, the C-code usually pertains to code from the run time system. When we say that control is passed from Haskell-land to C-land, we mean that some Haskell code have called a primitive function that is implemented in the RTS.

2.4 From source to machine code

Typically, compilers take source code and convert it to machine code. It would be overwhelming to go directly to machine code, instead most compilers have some intermediate representations (IR) in the pipeline [24, p.358]. For GHC, the pipeline is illustrated in figure 2.10 [25], the figure only includes the native code (assembly) backend. Throughout this thesis we will ignore the C and LLVM backend entirely.

2.4.1 The intermediate representations in GHC

Figure 2.10 only showed the names of the phases. To give a rough idea of how each intermediate representation might look like, we compiled a very small Haskell program with `ghc` passing the flags `-ddump-parsed -ddump-simpl -ddump-stg -ddump-cmm -ddump-asm`. The output is too verbose to reproduce in full, instead figure 2.11 shows some interesting excerpts. Naturally, the IRs closer to the hardware (Cmm and assembly) will contain more code and have therefore been truncated much more in figure 2.11.



```
addition :: Int -> Int -> Int
addition x y = (x + y)
```

```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType]
addition_r8m =
  \ (x_a9l :: GHC.Types.Int) (y_a9m :: GHC.Types.Int) ->
    GHC.Num.+ @ GHC.Types.Int GHC.Num.$fNumInt x_a9l y_a9m
```

```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType, Unf=OtherCon []] =
    sat-only \r srt:SRT:[(r9o, GHC.Num.$fNumInt)] [x_smq y_smr]
        GHC.Num.+ GHC.Num.$fNumInt x_smq y_smr;
```

```
...
cmG:
  R2 = GHC.Num.$fNumInt_closure;   // CmmAssign
  I64[(old + 32)] = stg_ap_pp_info;   // CmmStore
  P64[(old + 24)] = _smo::P64;   // CmmStore
  P64[(old + 16)] = _smp::P64;   // CmmStore
  call GHC.Num.+_info(R2) args: 32, res: 0, upd: 8;   // CmmCall
...
```

```
...
_cmG:
   movq %r14,%rax
   movl $GHC.Num.$fNumInt_closure,%r14d
   movq $stg_ap_pp_info,-24(%rbp)
   movq %rax,-16(%rbp)
   movq %rsi,-8(%rbp)
   addq $-24,%rbp
   jmp GHC.Num.+_info
...
```

Figure 2.11: The IRs in GHC for a simple function. The assembly is also included even though it is not an intermediate representation (it is the final representation).

```haskell
doubleAddition :: Int -> Int -> Int
doubleAddition x y = tot + tot
  where tot = x + y

globalThunk :: Int
globalThunk = doubleAddition 2 3
```

(a) Haskell code.

```
doubleAddition = FUN(x y ->
    let { tot = THUNK(plusInt x y);
        }
    in plusInt tot tot);

globalThunk = THUNK(doubleAddition two three);

-- And some imported functions

two = CON(I 2);
three = CON(I 3);

plusInt = FUN(x y -> ... ); -- Definition omitted
```

(b) STG code with the Ministg syntax.

Figure 2.12: Haskell code and STG code.

The STG IR

There is nothing that says that Haskell must be implemented using some sort of stack. In section 2.3.1 we saw that GHC, one popular Haskell compiler, chooses to implement the language using a stack called the execution stack. The STG intermediate representation in GHC is interesting to us since it is a representation of code where it is specified how it will interact with the execution stack [21, 22]. The STG code in 2.11 does not have a clean syntax, instead we use the syntax from the Ministg project [26]. Figure 2.12 shows Haskell code and the corresponding Ministg code. In the Haskell world, `doubleAddition` and `globalThunk` are both just values, in theory we do not really know if multiple uses of `globalThunk` will be memoized. Such implementation details are not relevant in the Haskell world. Looking at the STG however, we can be sure whether the value `globalThunk` will be memoized or not. By examining the STG code in figure 2.12b we can make the following observations.

- The implementation of `doubleAddition` is a function that takes two arguments, as expected[3].
- The value of `tot` is put into one thunk. Another possible implementation would be to inline it to `((x + y) + x) + y`. But inlining would make it worse, because the compiler would have to create *two* thunks instead. One for `(x + y)` and one for `((x + y) + x)`.
- The value `globalThunk` will be a thunk. When a thunk is defined at the top level, it can become a *global thunk*. A global thunk will only be evaluated at most once during the execution of a Haskell program. Some Haskell programmers are more familiar with the term CAF or constant applicative form. In this thesis we call them global thunks.
- We are also reminded that literals like `2` and `3` are wrapped in constructors.

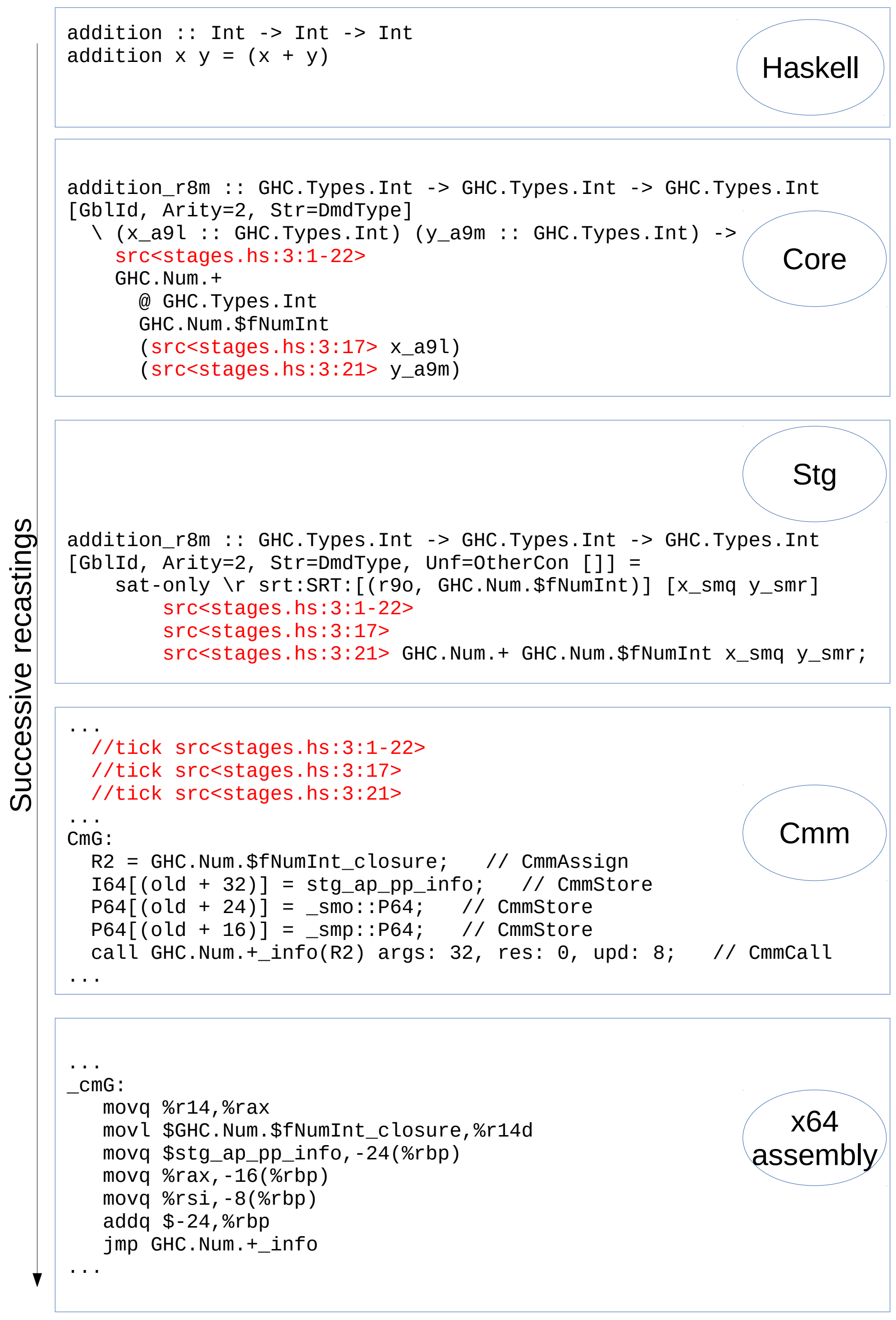
The experienced Haskell programmer will know all of this by heart by looking at the Haskell code (in this case, the programmer was so confident that he named the value `globalThunk` with such a suggestive name!). However, the only way to be certain is to look at the intermediate code that GHC emits.

2.4.2 Generating debug data

There is one common complex problem that must be solved to enable debugging tools: The programmer thinks of the program as its source code and the semantics of the language. However, the processor only runs machine code. Unfortunately, there is no way to associate the machine code to the source code that it originated from. This is a problem for all applications of debugging, not limited to stack traces [27]. As a consequence, any compiler that wants to support debugging has to do the truly overwhelming task of threading along information about the original source code that got compiled into each intermediate step, this information must also be retained and transformed accordingly during all the optimization steps. Figure 2.13 shows debug information that has been retained through all the transformations in the GHC pipeline when compiling a simple function.

One important observation is that any implementation can only be a best effort implementation. Consider figure 2.14 which contains two Haskell functions with *exactly* the same implementation. A clever compiler will be able to realize that the two function bodies are identical, it would then be safe for the compiler to remove one of the functions and just change the call sites of the removed function to use the other function. But this has a drawback because a stack trace involving the removed function can not exist. Figure 2.15 highlights this problem, Haskell can't report which function caused the crash, since that function is optimized away. In conclusion, an implementation of stack traces that have no effect on performance can only be a best effort attempt.

[3]It could in theory be a function taking one argument returning another function of one argument.



```
addition :: Int -> Int -> Int
addition x y = (x + y)
```

```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType]
  \ (x_a9l :: GHC.Types.Int) (y_a9m :: GHC.Types.Int) ->
    src<stages.hs:3:1-22>
    GHC.Num.+
      @ GHC.Types.Int
      GHC.Num.$fNumInt
      (src<stages.hs:3:17> x_a9l)
      (src<stages.hs:3:21> y_a9m)
```

```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType, Unf=OtherCon []] =
    sat-only \r srt:SRT:[(r9o, GHC.Num.$fNumInt)] [x_smq y_smr]
        src<stages.hs:3:1-22>
        src<stages.hs:3:17>
        src<stages.hs:3:21> GHC.Num.+ GHC.Num.$fNumInt x_smq y_smr;
```

```
...
  //tick src<stages.hs:3:1-22>
  //tick src<stages.hs:3:17>
  //tick src<stages.hs:3:21>
...
CmG:
  R2 = GHC.Num.$fNumInt_closure;   // CmmAssign
  I64[(old + 32)] = stg_ap_pp_info;   // CmmStore
  P64[(old + 24)] = _smo::P64;   // CmmStore
  P64[(old + 16)] = _smp::P64;   // CmmStore
  call GHC.Num.+_info(R2) args: 32, res: 0, upd: 8;   // CmmCall
...
```

```
...
_cmG:
   movq %r14,%rax
   movl $GHC.Num.$fNumInt_closure,%r14d
   movq $stg_ap_pp_info,-24(%rbp)
   movq %rax,-16(%rbp)
   movq %rsi,-8(%rbp)
   addq $-24,%rbp
   jmp GHC.Num.+_info
...
```

Figure 2.13: The same figure as 2.11, only that here each IR has debug data attached to it. The debug data tells us what Haskell source code the IR code is compiled from. For clarity the debug data annotations have a red font.

```
kilometerToMeter = (*1000)
kilogramToGram = (*1000)
```

Figure 2.14: Two functions with identical implementations.

```
reciprocal_1 :: Int -> Int
reciprocal_1 x = 1 `div` x

reciprocal_2 :: Int -> Int
reciprocal_2 = (1 `div`)

main = do
  print (reciprocal_1 5)
  print (reciprocal_2 0) -- crash!
```

(a) Original program.

```
reciprocal_1 :: Int -> Int
reciprocal_1 x = 1 `div` x

main = do
  print (reciprocal_1 5)
  print (reciprocal_1 0) -- crash!
```

(b) After optimizations.

Figure 2.15: An example showing why optimizations can give inaccurate stack traces.

Finally, the information about the source-level functions that the compiler has held tight throughout the IRs must get packaged into the binary. This concern arises naturally in the final IR stage (Cmm in the case of GHC). How does the compiler emit the debug information? How is it stored in a way so it doesn't get in the way of the actual code? A debugging format answers these questions. One such debugging format is DWARF.

2.5 DWARF

In 1988, DWARF was created hoping to solve a quite general problem. DWARF is a language agnostic debugging format that is still producing updated revisions. DWARF 5 is planned to be released in 2014 [27]. The DWARF data that is stored in the binary can be understood by a debugger like `gdb`. For example, it could help `gdb` explain how some data should be displayed, for instance if a particular byte is a 8-bit number or a character.

Looking at figure 2.13 again, we see that neither the first phase (the original Haskell source code) nor the last phase (the output assembly) has any debug information attached to it. This does make sense because programmers should not need to annotate their source code to get stack traces, neither should the performance of the program degrade by changing the output assembly. But then where is the final debug data emitted? It must be included with the binary of course. The binary is divided into *sections* [28], some of these sections are DWARF sections. The command line tool `dwarfdump`

```
< 1><0x0000008d>    DW_TAG_subprogram
                      DW_AT_name                  "addition"
                      DW_AT_MIPS_linkage_name     "r8m_info"
                      DW_AT_external              no
                      DW_AT_low_pc                0x00000020
                      DW_AT_high_pc               0x00000054
                      DW_AT_frame_base            DW_OP_call_frame_cfa
< 2><0x000000b3>      DW_TAG_lexical_block
                        DW_AT_name                  "cmG_entry"
                        DW_AT_low_pc                0x00000029
                        DW_AT_high_pc               0x0000004b
< 2><0x000000cf>      DW_TAG_lexical_block
                        DW_AT_name                  "cmF_entry"
                        DW_AT_low_pc                0x0000004b
                        DW_AT_high_pc               0x00000054
```

Figure 2.16: The DWARF data generated from the debug annotations in figure 2.13.

can inspect the DWARF data in object files. Figure 2.16 shows some of the DWARF data that GHC included in the object file it created. The figure shows only some of the relevant debug information, for example, the line numbers are not stored anywhere in the contents of the figure.

As will be revealed in section 3.4, this thesis work was made possible thanks to that DWARF got integrated in GHC.

3

Related work

To programmers outside of the Haskell community, it could sound surprising that a mature language like Haskell doesn't support stack traces. This might raise the following questions:

- Since stack traces are difficult, what other means of debugging are there?
- Are stack traces in Haskell really necessary?
- Is there at least any inefficient way to get stack traces?
- How close is the Haskell community in solving stack traces?

The overall structure of this chapter is that we answer the questions by looking at related work. Most of the related work is recent, usually less than 5 years from the time of publishing of this work. The first question is answered in section 3.1. Section 3.2 shows that Haskell is a language producing robust programs, which alleviates the need for stack traces. The third question is answered in section 3.3 which shows many working implementations of stack traces, all of which have significant overhead. The last question is answered in section 3.4.

3.1 Debugging Haskell

Examining stack traces falls in the category of debugging. Programmers examine stack traces printed from a handled exception or a crash. The amount of time the program runs before it crashes can be anywhere between a few nanoseconds to many years. Stack traces are most valuable for programs that crash unexpectedly after a long time of stable execution, because it might be hard to reproduce the error in order to diagnose it. Ideally, the stack trace aids the programmer in writing a minimal reproducible test case that exercises the original bug. Once programmers have an easy to reproduce bug, they look for tools that help them better understand why the bug is happening. In this subsection we'll look at existing tools for GHC that let programmers step through the program's execution and even print variables. Neither of which the stack trace implementation in this paper can provide.

3.1.1 GHCi Debugger

GHC comes with its own interactive *read evaluate print loop* (REPL) which has a built-in debugger. It's rich in features, supporting break points, single-stepping, breaking on crashes, a "tracing mode" and even variable inspection. The implementation works only with interpreted code [29, 30]. So there will be significant overhead both from the fact that the code is interpreted and that the debugger is running.

3.1.2 ghc-vis

Debuggers are a view into the otherwise opaque executing program. The GHCi debugger interface is text based, the programmer enters a command to the debugger and it responds in text. ghc-vis on the other hand is a graphical debugger. It allows users to visualize variables and interact with them with the mouse pointer. For example, when clicking on an yet unevaluated expression (remember, Haskell is a lazy language) it will evaluate the expression. Making it great for stepping through your program without losing the big picture. ghc-vis is hooking itself into the program by running a thread inside of GHCi. So it will only work with GHCi [31].

3.2 Avoiding Crashing

If a program never crashes, it will not matter if our language prints stack traces or not. Never-crashing programs is a research area, sometimes called formal verification. There are many approaches to formal verification. One can statically analyze C-programs [32], use finite automata or formal grammars [33, 34], use type system tricks [35] or use total functional programming [36]. In the end though, none of these methods are perfect, otherwise we would not need stack traces.

There are also statical analysis tools for Haskell. HALO is a tool where the programmers write contracts about their own programs and then let HALO prove them [37]. HALO seems to be inspired from [38] which in turn is inspired by [39]. Another tool is HipSpec which does automatic proof finding instead of having the programmer spell out the properties to validate [40]. Yet another tool that's mentioned on Haskellwiki is Catch where its described to "detect common sources of runtime errors" [41].

3.2.1 Catch

Catch is a static analyzer for Haskell. It can detect if a pattern-matching is sufficiently covering, even if the cases aren't collectively exhaustive. Figure 3.1 shows a function where the pattern match isn't exhaustive but sufficiently so. Catch can prove that such a pattern matching is safe by doing flow analysis and ruling out impossible patterns for

```haskell
safeFunction = nonExhaustivePatterns False
  where
    nonExhaustivePatterns False = 42
      -- NOTE: No pattern for True
```

Figure 3.1: A safe function even though the non-exhaustive matching. A totality checker like Catch can ensure that it's safe.

the scrutiny (the expression that we `case` on). This eliminates the need for the human programmer to manually check what can be automatically proven [42].

3.3 Inefficient stack traces

There are already many successful stack trace implementations in Haskell. Unfortunately, they all have a significant overhead. In this section we will look at previous work about stack traces for Haskell. There are two common sources of overhead in existing implementations:

- By building an explicit call stack (Subsection 3.3.1)
- By depending on expensive runtime settings (Subsection 3.3.2)

3.3.1 Explicit call stack

Stack traces can be achieved by doing some methodological source level transformations. Figure 3.2 shows a program transformed into one producing stack traces on calls to `error`. This transformation is essentially:

- Changing all top level functions to take one additional string argument. Except for the program entry-point `main`.
- Transform all equations to define the new call stack `stack'` and pass it as the first argument to all calls of top level functions.
- Transform all calls to `error` to also print out the call stack.

This transformation is similar to [43] and a complete source-to-source implementation called *hat* exists already [44]. But explicit call stack implementations don't need to work on a source level.

StackTrace

Allwood et al implemented a Intermediate Representation (IR) transformation pass called *StackTrace*. It's operating on the GHC Core IR. Since Core is like a small subset of Haskell, its implementation will do something similar to what figure 3.2 illustrates.

```
main = print (f 100)

f :: Int -> Int
f x = g (5*x)

g :: Int -> Int
g 7 = error "Bang"
g x = 100 * x
```

(a) Original program.

```
main = print (f stack' 100)
  where
    stack' = "main \n"

f :: String -> Int -> Int
f stack x = g stack' (5*x)
  where
    stack' = "f (case 1)\n" ++ stack

g :: String -> Int -> Int
g stack 7 = error ("Bang" ++ stack')
  where
    stack' = "g (case 1)\n" ++ stack
g stack x = 100 * x
  where
    stack' = "g (case 2)\n" ++ stack
```

(b) Transformed program.

Figure 3.2: An example of how a Haskell program can be transformed to one that will print stack traces on errors. The syntax '`str1 ++ str2`' is string concatenation.

Among its complications are the handling of higher order functions, linking with code that doesn't have stack traces and an efficient non-naive implementation of the passed along stack [10]. Functional programming in particular relies on efficient tail call optimizations, which requires the passed around call stack to efficiently handle this.

3.3.2 Stack traces with profiling

A mature and stable implementation of stack traces for Haskell is present in GHC since GHC 7.4.1 which was released in February 2012. No paper has been produced from this effort. But a talk were given at Haskell Implementors Workshop in September 2012 [45]. The implementation is only working in conjunction with the profiling mode of GHC. In Profiling mode the execution of programs can expect to be twice as slow as their plain counterparts. The cost centre stack traces have its own set of problems and is only an approximation of what Haskell really is executing.

3.4 Recent work

Around the time when this thesis started, Peter Wortmann, a PhD candidate at University of Leeds showed a proof of concept stack trace in Haskell that was based on the execution stack [46]. Peter had been working on non intrusive profiling for GHC. To

```
$wfibonacci =
  \ (ww_sp0 :: Int#) ->
    case ww_sp0 of ds_Xph {
      __DEFAULT -> ...
```

(a) Without ticks, the default option.

```
$wfibonacci =
  \ (ww_sp0 :: Int#) ->
    src<0:Fibonnaci.hs:(4,1)-(6,51)>
    case ww_sp0 of ds_Xph {
      __DEFAULT -> ...
```

(b) With ticks by passing the flags `-g -dppr-ticks`.

Figure 3.3: The same excerpt of the Core generated from the program in figure 2.2. Only the version on the right have debug data included.

accomplish this, he had developed a theory of causality of computations in Haskell and his work extended even to optimized code [1]. To do profiling he needed to map instruction pointers to the corresponding source code. He added source code annotations that propagated through the pipeline of IRs and optimizations and finally emitted DWARF debugging data. Figure 3.3 shows how a code annotation has been placed in the Core IR. With his patches, GHC now emits DWARF, making a stack trace implementation to be low hanging fruit, enabling the quite sizeable problem of stack traces to be worked on during the limited scope of a master's thesis.

But the original stack traces produced from Peter's simple demo are not satisfactory. The stack trace from running the program in figure 3.4 looks like 3.5a. The outputted stack trace doesn't contain a single function from the program in figure 3.4. The bottom of the stack is `stg_catch_frame_ret`, which is the default catch-handler that is installed at the root of Haskell programs, before `main` is run. The `writeBlocks` and `showSignedInt` give vague hints that we're printing an `Int`-type, possibly to stdout (they come from the usage of `print`). Parts of chapter 5 will look how to improve the stack. Ideally the stack should look like in figure 3.5b. Yet, we have not looked deeply into the internals of GHC and its execution stack, we will do this right now throughout the whole of chapter 4.

```haskell
main :: IO ()
main = do print 1
          a
          print 2

a, b, c :: IO ()
a = do print 10
       b
       print 20

b = do print 100
       c
       print 200

c = do print 1000
       print (crashSelf 2)
       print 2000

crashSelf :: Int -> Int
crashSelf 0 = 1 `div` 0
crashSelf x = crashSelf (x - 1)
```

Figure 3.4: A sample Haskell program that will crash when run.

```
 0: stg_bh_upd_frame_ret
 1: stg_bh_upd_frame_ret
 2: stg_bh_upd_frame_ret
 3: showSignedInt
 4: stg_upd_frame_ret
 5: writeBlocks
 6: stg_ap_v_ret
 7: bindIO
 8: bindIO
 9: bindIO
10: bindIO
11: stg_catch_frame_ret
```

(a) A stack trace, clearly based on the execution stack. Since the execution stack is bound to GHC's specific implementation of Haskell, it'll be difficult for programmers to interpret.

```
0: crashSelf
1: crashSelf
2: print
3: c
4: b
5: a
6: main
```

(b) An ideal, fictive, stack trace, without implementation details of the execution stack. It's rather a semantic stack.

Figure 3.5: Two stack traces.

4

The Execution Stack

In this chapter we describe the execution stack. In section 4.1 we'll look at the quantity and ownership of execution stacks. Section 4.2 will examine the entries of the stack, called stack frames. We'll see what data layout a stack frame has to adhere to and look at some common stack frames. Section 4.3 explains the structure of the stack and how the convenient abstraction of the `Sp`-register works.

The content in this chapter will stick to objective facts, mostly based on the GHC source code. Difficulties of implementing stack traces will be withheld for later chapters.

4.1 Number of stacks

Haskell's `base` library exports the following primitive [47]:

```
forkIO :: IO() -> IO()
```

Intuitively `forkIO` just creates another "green" thread running its argument. Since there will be two concurrent threads running after this, clearly another execution stack have been created somehow. In this section we look at how many execution stacks there will be by looking at where they are referenced from.

The implementation of `forkIO` is that it will create another *Thread State Object* (TSO). As can be seen from figure 4.1, a TSO points at a Stack object fully reproduced in figure 4.2. Thread State Objects themselves are usually referenced from *Capabilities*. A Capability contains all essential values for executing Haskell code. It can be thought as a virtual CPU, it contains the virtual register values of the STG abstract machine. A capability also contains a singly-linked deque of all TSOs that are scheduled to run, meaning there is a one to many relationship between a capability and an execution stack. Capabilities themselves also come in multitude in the run time system (with the default settings). The number of capabilities is configurable through a runtime option, as a rule of thumb it should be set to as many as the number of cores on the computer [48].

```
typedef struct StgTSO_ {
    StgHeader               header;
    // deleted lines ...
    struct StgStack_       *stackobj;
    // ...
    struct Capability_*     cap;
    // ...
    StgWord32  tot_stack_size;
} *StgTSOPtr;
```

Figure 4.1: The definition of `StgTSO` from the GHC run-time system code.

```
typedef struct StgStack_ {
    StgHeader  header;
    StgWord32  stack_size;      // stack size in *words*
    StgWord32  dirty;           // non-zero => dirty
    StgPtr     sp;              // current stack pointer
    StgWord    stack[FLEXIBLE_ARRAY];
} StgStack;
```

Figure 4.2: The definition of `StgStack` from the GHC run-time system code.

Figure 4.3 shows an example of all the capabilities and TSOs at a particular moment of a running Haskell program. The number of threads is dynamic of course, since new threads can get created with `forkIO` and existing threads can finish executing. Since GHC 7.6.1, the number of capabilities is dynamic too [49].

In conclusion, there is one execution stack for each green thread (TSO).

4.2 What's on the Stack?

In STG-land, all values have the layout shown in figure 4.4 [50], these values are called *heap objects*. In figure 4.4, we see that the first value of a heap object is a pointer to executable code. It should also be noted that the code and info table is in static memory while heap objects mostly are in allocated memory (global thunks being one exception). Typically there would be multiple live heap objects pointing to the same info table (or its code). When heap objects are no longer used they are removed during the next garbage collection. Some info tables do not have any code, but mostly this is not the case, any heap object pointing to an info table with runnable code is called a *closure*.

A *stack frame* is a closure whose info table's type is any of the types listed in figure 4.5.

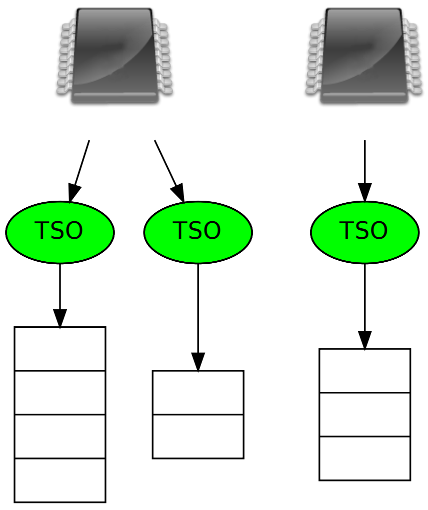


Figure 4.3: Two virtual CPUs (Capabilities) running a total of three green threads. Each green thread has its own execution stack.

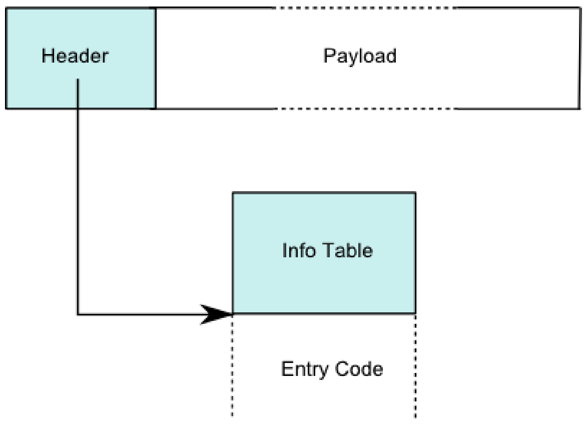


Figure 4.4: Structure of heap objects in Haskell. Image is taken from [50].

```
#define RET_BCO                 31
#define RET_SMALL               32
#define RET_BIG                 33
#define RET_FUN                 34
#define UPDATE_FRAME            35
#define CATCH_FRAME             36
#define UNDERFLOW_FRAME         37
#define STOP_FRAME              38
// ... some omitted non-stack closure types ...
#define ATOMICALLY_FRAME        57
#define CATCH_RETRY_FRAME       58
#define CATCH_STM_FRAME         59
```

Figure 4.5: The subset of closure types that are present on the stack. The excerpt is from [52].

```
f :: Bool -> Bool -> Bool
f = \x -> if x then not else id
```

(a) Haskell function.

```
f = FUN(x -> case x of {
               True  -> not;
               False -> id });
```

(b) STG function, note that the number of arguments is explicitly just one.

Figure 4.6: In Haskell-land, the function can be fed up to two arguments. But we say that the arity is one, because the STG function that will be compiled takes one argument and returns a `FUN`. Note that `not` and `id` are themselves `FUN`s.

Which types that exist on the stack is based on [51].

In this section we'll dive into some details about the stack frames. Other details are omitted, like how garbage collection is treating the closures.

4.2.1 Fields and arguments

We are used to think of the *arity* of a function as the number of *arguments* that it takes. That is valid in the context of the STG-machine [53]. But the arity of the STG-function does not always equal with the number of times a Haskell function can be applied to. Figure 4.6 shows one Haskell function which can be applied to two arguments, but is compiled to an STG-function with an arity of one.

The return convention in the Cmm IR is to jump to the code of the topmost stack frame [54]. If the code takes arguments, they are pushed on the stack before jumping to the code. In addition to arguments that are just pushed at invocation time, there are

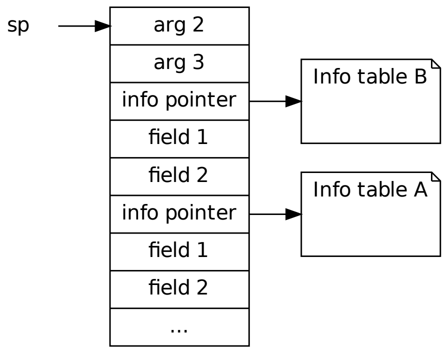


Figure 4.7: The stack when the topmost function on the stack is invoked. In this example we pass the first argument by register, so `arg 1` is stored in the virtual register `R1`.

already *fields* fields residing in the stack frame. Fields are similar to arguments but are already placed on the stack and are part of the stack frame. Fields are pushed just before we push the info pointer and together they constitute the stack frame. Arguments on the other hand are passed when code returns. When a function is entered, the arguments are above the info pointer and the fields are below [55], like shown in figure 4.7. As an optimization some of the arguments will be passed by the available registers of the virtual CPU, the exact number of arguments that are passed by register will depend on the backend that is targeted [56].

Figure 4.8 shows examples of Haskell code that compile to code that has both fields and arguments. As soon will be revealed in the next section, `case` expressions will be splitting points in the code generation. The pre-`case` code will push the post-`case` code on the stack. The post-`case` code is also called a *case continuation.* In order to not loose the variables when evaluating the scrutiny, the live variables from the pre-`case` code will be pushed along as the fields of the post-`case` code. Jointly they form the *case continuation frame.* When a variable is not needed by the case continuation, it will not be a field, like in figure 4.8b.

4.2.2 The members of the stack

Previously we defined what a stack frame is and enumerated its types. However stack frames can also be categorized by their purpose rather than their type. In this section we look at stack frames, categorized by what they do.

```
... = \x y -> ... case x of
                    4 -> x + y
                    _ -> x - y
```

(a) The case continuation will have y as a field and x as an argument.

```
... = \x y -> ... case x of
                    4 -> x + 5
                    _ -> x + 10
```

(b) The case continuation doesn't depend on y, so it will have no fields and x as its only argument.

Figure 4.8: Two examples of code that will be turned into case continuations.

Case continuation frames

Case continuations is what comes naturally from having lazy evaluation in conjunction with pattern matching. When evaluating a case-expression, the code first pushes the case continuation frame *(*`case` • `of` *...)* and then jumps to the entry code of the scrutiny. When the scrutiny code returns, the code of the case continuation will have that value as argument. Since that value is evaluated by now, it is possible to have a C language style `switch-case` statement corresponding to the original Haskell `case` statement, or at least corresponding to the STG `case` statement.

Update frames

Consider the following code:

```
let x = 2 + 3
in x + x
```

Here, `x` will only be evaluated once. This is implemented through update frames. Update frames *(Upd • t)* are pushed when a `THUNK` *x* is evaluated, the frame's only field will be the thunk itself [57]. After the frame is pushed, the entry code for `x` is entered. When the code returns, the result is passed to the update frame as an argument, which will overwrite the thunk with an indirection to its argument (the result of `x`) [58].

Call continuation frames

When evaluating

```
f True False
```

where `f` is from figure 4.6, we have the following scenario:

- The function `f` has arity 1.
- We have an application of 2 arguments.

- The code has already type checked. We assume there is no programming error and that `f True` will return another function.

We can't just jump to the code of `f` and forget about the last argument `False`. Instead, we first put a call continuation frame *(• False)* and then jump to the code of `f` [22]. When evaluation of `f True` is complete it returns to the entry code of the call continuation, the call continuation would first put its fields (`False`) on the stack (or write to register `R1`) and then jump to the entry code of the argument it got passed (the result of `f True`). All call continuations are of closure type `RET_SMALL` [59].

Underflow frames

Underflow frames allow the stack itself to dynamically grow or shrink. Their significance is discussed in section 4.3.

Other frames

Figure 4.5 showed that there are other closure types that play a role on the Haskell execution stack, including retry frames for Software Transactional Memory. We will not examine these other frames further.

4.3 Structure

Since version 7.2.1, GHC switched its underlying structure of the execution stack to use a chunked singly linked list rather than a dynamically growing array [60, 61]. When the stack chunk that `tso->stackobj` is referencing gets full, it *overflows*, then a new stack chunk is created with a reference to the first stack and `tso->stackobj` is set to the new stack. Conversely, when a stack chunk gets depleted, it *underflows*, then we just set `tso->stackobj` to point to what the underflow frame is referencing, the garbage collector will later remove the abandoned chunk. Figure 4.9 shows one execution stack owned by a TSO whose stack has two chunks, hence it must have exactly one underflow frame.

4.3.1 Current stack pointer

Each stack chunk is represented by the `StgStack`-struct which has a member `sp` which is a stack pointer. Since the execution stack is the collection of these chunks the execution stack has multiple stack pointers. What the stack pointers more exactly are pointing at is illustrated in figure 4.10. But the only relevant stack pointer is what we call the *current stack pointer*, which is the one pointed by `CurrentTSO->stackobj->sp`. `CurrentTSO` is a STG virtual register. In C-land, the `CurrentTSO` virtual register is stored in `cap->r->rCurrentTSO`, where `cap` is the Capability that contains the `CurrentTSO` register. So there is one current stack pointer for each Capability.

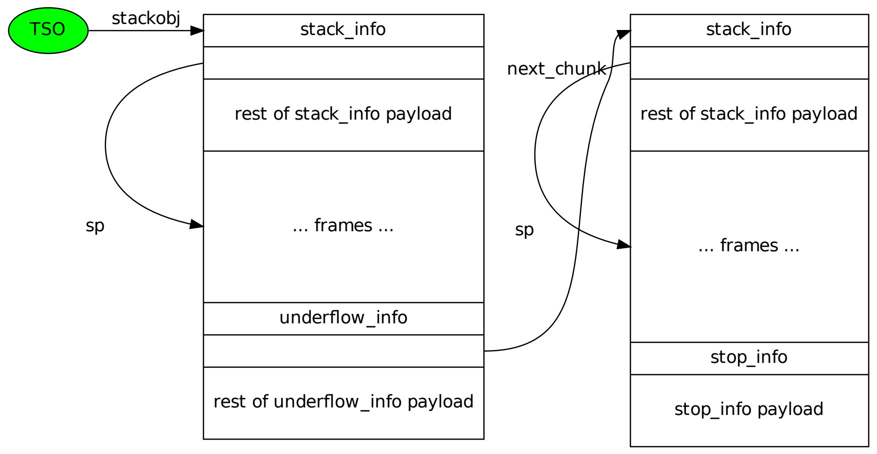


Figure 4.9: Structure of the stack. The TSO is pointing at the current chunk and the older chunks are referenced by underflow frames.

Syncing

The virtual CPU has the register `Sp` which also is a stack pointer. `Sp` should be safe to assume to be in sync with the current stack pointer. To make this so, whenever one jumps from Cmm-land to C-land (from virtual CPU to real CPU), or vice verse, it syncs. The exception is when it is not necessary, then there is no syncing due to its overhead. Syncing does happen however on stack underflows [62]. Note that while there is a syncing between the virtual register `Sp` and the memory location `CurrentTSO->stackobj->sp` [63], it also happens for all the other virtual registers.

4.3.2 Buffering

Each time an overflow happens, there is some overhead. In the worst case, a long alternating sequence of pushing and popping will cause perpetual over and underflows. *Buffering* is implemented to combat this, buffering will copy a few frames from the old chunk to the new chunk [64], requiring that more than one frame have to be popped before an underflow will happen.

4.3.3 Stack squeezing

When control is passed to an update frame, it will update its thunk and then pass control to the next frame on the stack. An interesting scenario is when there are consecutive update frames on the stack. In this case all the consecutive update frames will be passed the exact same value and will modify their respective thunks to an indirection to the

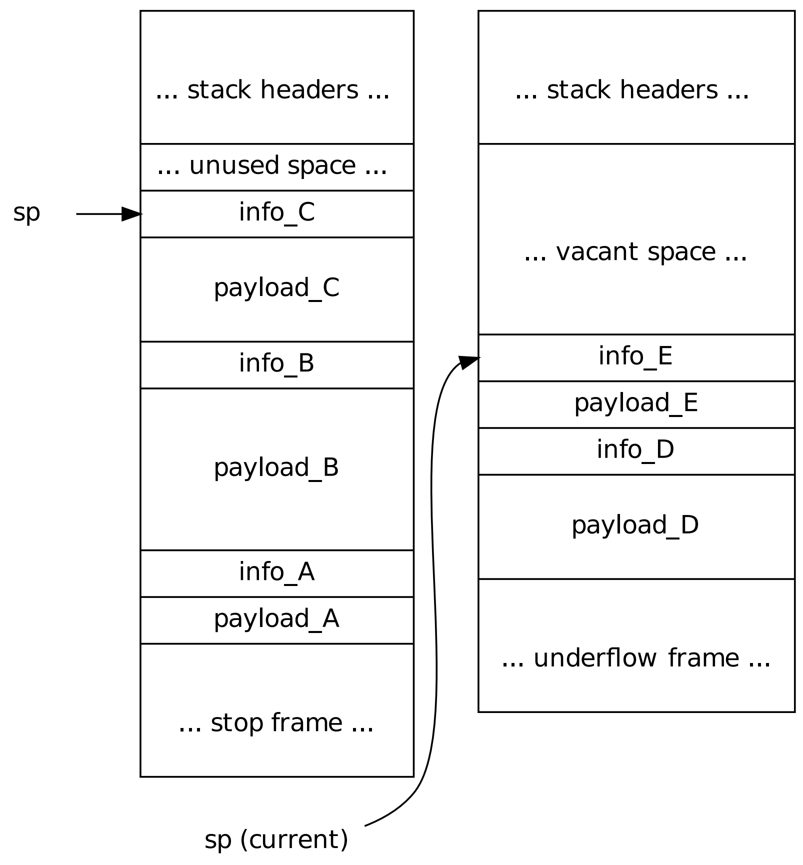
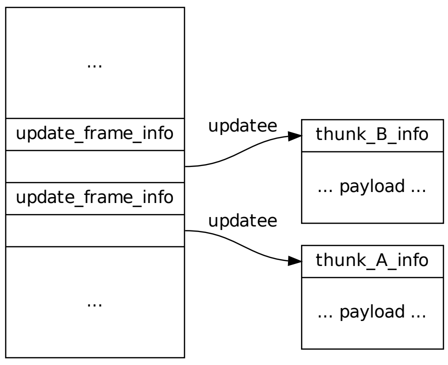
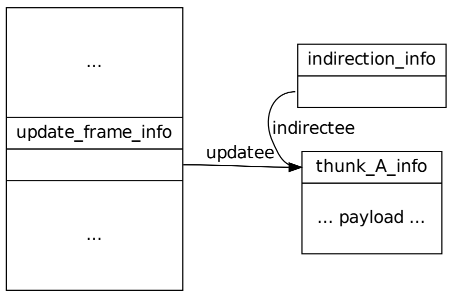


Figure 4.10: Clarification of where the stack pointers are pointing. Note that these two chunks (can) belong to the same execution stack.



(a) Before squeezing.



(b) After squeezing.

Figure 4.11: An illustration of stack squeezing.

same result. *Stack squeezing* is to detect this scenario and remove all but one remaining update frame and instead turn the affected thunks into indirections pointing to the thunk that the remaining update frame points to. One example is shown in figure 4.11 [65].

Stack squeezing happens on overflows and is only run on the current stack chunk. If the stack squeeze is successful the overflow gets cancelled [66].

5

Reifying the Stack

In section 3.4 we saw Peter Wortmann's prototype for execution stack based stack traces. It was demonstrated in August 2013 [46]. This is a rough sketch of how it is implemented:

1. The program starts by installing a catch-all handler. This handler will print the stack trace on a crash.
2. Program runs and crashes.
3. The run time system handles the crash by walking through the whole execution stack and saving it in a separate array. It then invokes the installed handler and passes the array.

To store the essential contents of the stack in a new value, possibly for later trace-printing, is called *reifying* the stack. The reification from Peter Wortmann's demonstration is quite simple, figure 5.1 illustrates this method of reifying the stack.

This chapter will go into stack reification in detail. By critically looking at the prototype, we find room both for improvement and discussion. In section 5.1 we see that the stack traces can become more readable by using the extra information in the payloads of each stack frame. Section 5.2 deals with the issue of wasting resources when reifying the stack without ever using it.

5.1 Frames of interest

Figure 3.5 showed that the stack trace from Peter Wortmann's demonstration is very far from the ideal stack trace. Worse, the only information we have to work with is the execution stack, remember, we maintain no explicit call stack for performance reasons. Still, the stack can become clearer by using the payload of the stack frames. For the readers convenience, we reproduce the stack trace here again as figure 5.2.

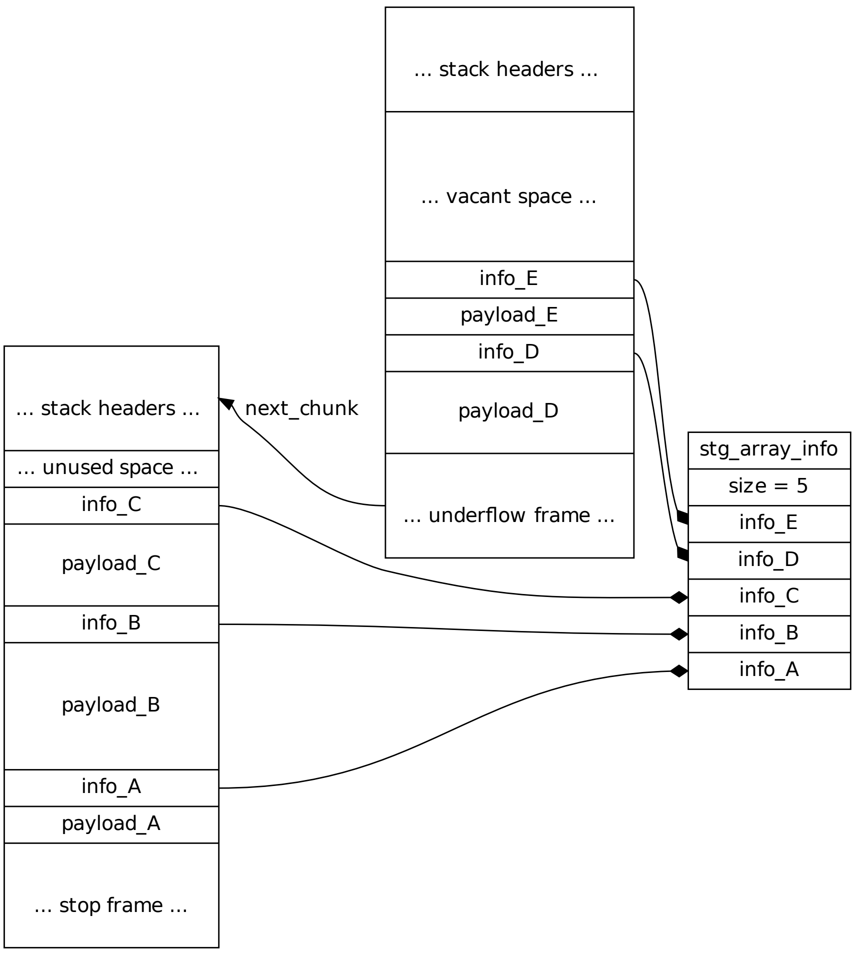


Figure 5.1: An illustration of stack reification, the rightmost box is the array allocated to store the result of the reification, the diamond arrows show where the content is copied from.

```
 0: stg_bh_upd_frame_ret
 1: stg_bh_upd_frame_ret
 2: stg_bh_upd_frame_ret
 3: showSignedInt
 4: stg_upd_frame_ret
 5: writeBlocks
 6: stg_ap_v_ret
 7: bindIO
 8: bindIO
 9: bindIO
10: bindIO
11: stg_catch_frame_ret
```

Figure 5.2: Stack trace (same as in figure 3.5a).

In section 5.1.1 and 5.1.2 we will revisit the stack frames we looked at in section 4.2.2. This time we will look at how their frame-specific payload can be utilized to make the stack traces more useful. In addition to utilizing the existing frames, we take a stab at creating our own artificial stack frame in section 5.1.3 that could improve stack traces.

5.1.1 Update frames

Frame 0,1,2 and 4 from figure 3.5a are all update frames. Recall that the code for a thunk will push one update frame. Thunks are common in Haskell and therefore update frames are common as well. If we were able to print something better than the cryptic `stg_bh_upd_frame` and `stg_upd_frame`, many frames in the stack trace would improve.

The actual code of the thunk that pushed the update frame could be anything. To only see the info pointer of the update frames is hardly helpful. But we know that the field of the update frame is a reference to the heap object the update frame is going to update, the *updatee*. The updatee's info table contains the code that pushed the update frame. This is really good, because we can copy that info pointer to our array instead, as illustrated in figure 5.3. Unfortunately this trick only works in some cases and it will depend on the kind of update frame we have on the stack. There are three different kinds of update frames [67].

- `stg_bh_upd_frame` is the update frame used for global thunks. Global thunks are better known as CAFs.
- `stg_upd_frame` is the update frame used for local thunks.
- `stg_marked_upd_frame` is created by overwriting one of the other two kinds of frames, it happens during garbage collection and this phase is named *blackhol-*

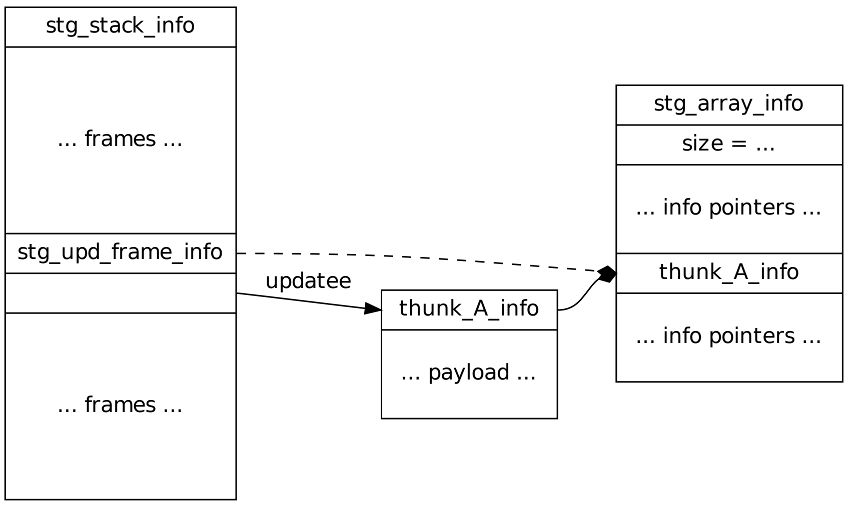


Figure 5.3: An improved way of reifying update frames. The old method is showed in the dashed line.

ing. Blackholing converts any update frame to a marked update frame [68] and overwrites the updatee's info pointer to point to a ”black hole” [69].

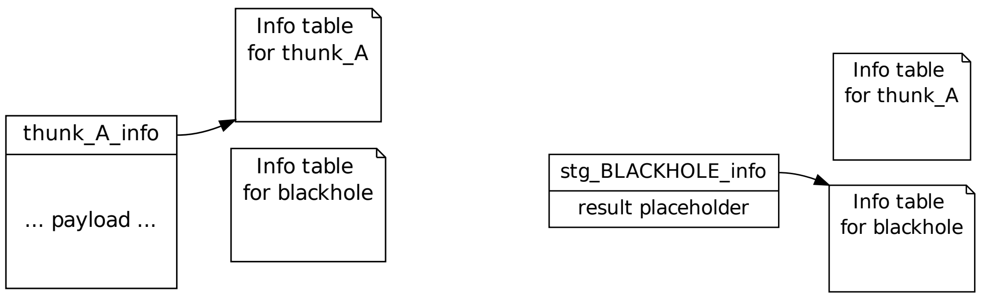
Unfortunately, only the updatee of local thunks point at interesting code and not a black hole. The marked update frame always gets its updatee's info pointer overwritten to a black hole [69] and the update frame for global thunks' updatee point at a black hole to begin with [70].[1]

Blackholing have become a quite complicated part of the run-time system, having multiple purposes [71] and is implemented with low-level tricks like pointer tagging. But the traditional black hole as described in [72] is having clear purposes, blackholing fixes a class of space leaks and it detects some cases of nontermination. In concurrent Haskell, blackholing have synchronization purposes that increases sharing (avoids re-computation). Blackholing is illustrated in figure 5.4.

Retaining code reference on blackholing

The bad effect of blackholing from a stack trace perspective is that thunks lose the reference to the code that pushed the update frame (recall figure 5.3 and figure 5.4). So

[1]The names `stg_bh_upd_frame` and `stg_marked_upd_frame` are very confusing. In both cases they point at black holes. A `stg_marked_upd_frame` points at a `BLACKHOLE` and a `stg_bh_upd_frame` points at a `CAF_BLACKHOLE`. The names are kept true to their name in the GHC source code to ease verifiability.



(a) A thunk. **(b)** A blackhole.

Figure 5.4: The same thunk, before and after blackholing.

instead of being able to print something useful, like `... 4: print ...`, we have to print `... 4: stg_upd_frame_ret ...`.

Unfortunately, blackholing is not optional [73]. It is however possible to retain the reference by just adding a field for all thunks and copying the reference there. The extended field is shown in figure 5.5.

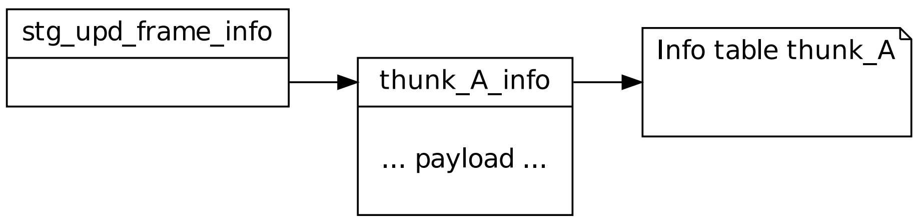
The extra field has a runtime cost of course. The number of thunks ever created during the lifetime of a Haskell program has no upper bound. However, there is only a constant number of global thunks, which means that the performance cost would be insignificant if we only applied the idea from figure 5.5 to global thunks. When only changing the global thunks, we are able to identify the `stg_bh_upd_frame` frames but not the `stg_marked_upd_frame` frames. For example, the segment

```
0: stg_bh_upd_frame_ret
1: stg_bh_upd_frame_ret
2: stg_bh_upd_frame_ret
...
```

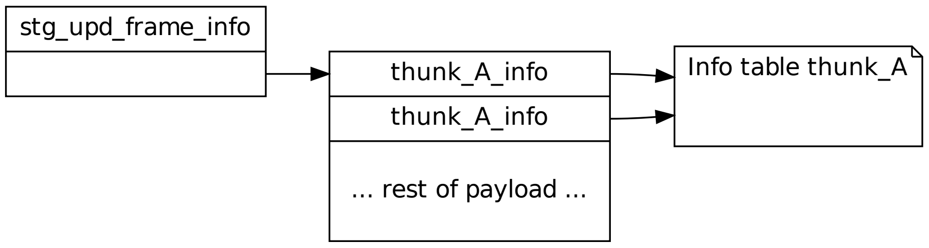
could instead be

```
0: divZeroError
1: crashSelf
2: c
...
```

by utilizing the new payload. The second stack trace is of course *much* more useful than the first stack trace. Unfortunately, to only add the field for global thunks will only be useful for a short time of the life of the thunk, since blackholing is happening



(a) The current layout.



(b) A layout that has a copy of the original info pointer, since the first info pointer will be overwritten during blackholing.

Figure 5.5: Two different layout for thunks.

intermittently. If instead all thunks had the extra field, the blackhole thunks would also retain the useful reference. Luckily though, if a global thunk is containing code that crashes, it is probable that its update frame will be left intact if it crashes early. In fact, we should consider ourselves extra lucky that top-level values like `divZeroError` memoize. If there were no memoization, there would be no thunk or no update frames, instead there would be regular tail calls and the first three frames would not even be on the stack.

5.1.2 The other frames

Some of the stack frames don't need any analysis because their info pointer is what we want to print, other frames seem hopeless and are not investigated in depth. In this subsection we will give a brief mention of the frames that we have not analyzed in depth.

For *case continuations*, the info pointer is pointing to code generated from Haskell, so we consider case continuations as a trivial frame. For *Call continuations*, we've already jumped to the interesting code and just left a trace of leftover arguments. There is nothing to do for call continuations since the leftover arguments say nothing about the over-applied function itself. *Catch frames* will hold a reference to the handler, but it is not what is currently evaluating. The good news is that just seeing any catch frame is helpful, with some context the programmer might be able to determine which `catch` in the source code pushed the catch frame.

5.1.3 Artificial frames

So far we have looked at how to utilize the payload of *already existing* stack frames. In this chapter we look at how we can add new *artificial* stack frames. Artificial frames are stack frames whose only purpose is to improve the stack trace. The intended use case is for programmers that have got a reproducible bug and a stack trace without enough information. This should be a common situation, because stack traces derived from the execution stack often lack frames due to tail calls. A versatile programmer might switch to one of the many other stack trace implementations from section 3.3. But in practice that is usually an overwhelming task due to technical difficulties like compiler flags and missing libraries. Instead we present the programmers with an inbuilt primitive to force a stack frame. When a stack is later reified, these artificial frames would work as checkpoints and be printed.

There are two fundamentally different approaches here. Either you create an actual function like `pushStackFrame :: a -> a` that takes one argument, pushes its arguments info pointer on the execution stack (encapsulated in a special stack frame) and then evaluates the argument. The other approach is to create a "macro"-like function that will expand `(forceCaseContinuation x)` to `(case x of _ -> x)` in such a way that the case expression can not be optimized away. This approach will create a unique case continuation for every use site of `forceCaseContinuation`. The fundamental difference is that the first approach will save the *jump target* in the artificial frame's field. Meanwhile the second approach pushes a case continuation that will uniquely identify the *jump source*. It is not obvious which method will be most useful and do what the programmer expects. One nice property of the jump source implementation is that it will include function f in the stack trace if `forceCaseContinuation` is put inside function f and its argument is currently evaluating. Further, the jump target approach has a pitfall: When a nontrivial expression is passed to it, the nontrivial expression will be put into a thunk that will have a source location associated to it from the jump source. So the jump target will often be the same as the jump source. The programmers using `pushStackFrame` should therefore understand when GHC decides to create thunks. On the other hand, `pushStackFrame` can be the right tool for the job and will push an artificial stack frame containing an interesting jump target.

5.2 Efficient reification

When executing the program from figure 3.4, the reification need to happen at the crash site and not in the handler. When control has been passed to the handler the execution stack has been unrolled already, so the stack is inaccessible and maybe even overwritten. At the same time, there is a cost associated with reifying the stack and the cost is growing linearly with the size of the execution stack. So when we *know* that we're going to print the stack, it is acceptable to have the additional linear cost of reifying the stack, but we shouldn't tolerate the cost when we are not using the reified stack value. One consequence of always reifying the stack is that functions that use `throw` for control flow will become slower depending on how big the stack is when they're called. To have the time complexity of a total Haskell function depending on the state of the underlying runtime system is not acceptable for a mature compiler like GHC. Note that control flow can be an anticipated use case for exceptions [74] (however, a more recent paper does not anticipate it [75]). `lazyReadBuffered` uses exception handling for control flow [76]. In this section we will look for alternative solutions to unconditional and strict stack reification.

Before looking at theoretical solutions, we will test our hypothesis and convince ourselves by experiment that stack reification really does take linear time. Figure 5.6 shows a Haskell program that runs the exact same pure computation twice, the first time it runs the computation with a small execution stack and the second time a lot of frames have been pushed on the execution stack. We also run the program twice, the first time with stack traces enabled and the second time with stack traces disabled:

```
$ ./Benchmark +RTS --stack-trace -RTS
Takes 332 ms with small stack
Takes 18344 ms with large stack

$ ./Benchmark
Takes 175 ms with small stack
Takes 178 ms with large stack
```

The results clearly match the hypothesis. If we do not reify the stack at all like in the second program run. The two functions are equally fast. But if we do one stack reification per exception (as happens in `stupidFunction`), the program gets slower as seen from the results in the first program run. We can also see that the program gets *much* slower if the execution stack is already big. This is considered to be a serious problem because `pureFunction` is not supposed to depend on its environment. The tests were run on a Ubuntu 13.10 laptop, Intel(R) Core(TM) i7-4800MQ CPU at 2.70GHz with GHC revision [77], we compile with `-rtsopts -g -make` as flags to `ghc`.

As we just saw, reifying the stack is costly and we will dedicate this whole section for finding more efficient alternatives. The two natural solutions are to either reify the stack *conditionally* or to reify it *lazily*. When reifying the stack conditionally, we only reify the stack when we know for sure that we are going to print the stack. Subsection 5.2.2

```haskell
import Control.Exception ( catch
                         , SomeException (SomeException)
                         , evaluate)
import System.IO.Unsafe (unsafePerformIO)
import Data.List (foldl')
import System.CPUTime (getCPUTime)

main :: IO()
main = do
    evaluate (stackBuilder 1) -- For fairness with CAFs
    putStrLn $ "Takes " ++ show fast ++ " ms with small stack"
    putStrLn $ "Takes " ++ show slow ++ " ms with large stack"
  where
    fast = stackBuilder 0
    slow = stackBuilder 1000

getMilliSeconds :: IO Integer
getMilliSeconds = fmap (`div` 1000000000) getCPUTime

timeIt :: Integer -> IO Integer
timeIt x = do t0 <- getMilliSeconds
              evaluate x
              t <- getMilliSeconds
              return (t - t0)

-- Returns the time it takes to evaluate pureFunction
stackBuilder :: Integer -> Integer
stackBuilder x | x < 1 = unsafePerformIO (timeIt (pureFunction x))
stackBuilder x         = x - x + stackBuilder (x-1) -- Push frames

-- It has to take an argument to not become a thunk
pureFunction :: Integer -> Integer
pureFunction zero =
    foldl' (+) 0 (map stupidFunction [1..(zero + 1000000)])

stupidFunction :: Integer -> Integer
stupidFunction x = unsafePerformIO (action `catch` handler)
  where
    action = evaluate (5 `div` 0)
    handler (SomeException _) = return x
```

Figure 5.6: A Haskell program we use for benchmarking purposes.

looks at solutions in this category. The second approach would be to let the stack value be lazily evaluated. If creating the thunk for the execution stack value could be done in constant time, we have a satisfactory solution. Subsection 5.2.3 and 5.2.5 are implementation ideas for lazy stack values. But first we take a look at a simple and unsophisticated solution in subsection 5.2.1.

5.2.1 Reifying a constant number of frames

A very simple solution would be to just reify a constant number of frames. The time complexity for any reification would then be constant. The exact number of frames could be specified through a run-time parameter or from Haskell-land. There is also a few other benefits with this approach, for one there would be no user-interface issues with too long stack traces being printed to the screen, second, as we saw from subsection 5.1.1, the top of the stack is less likely to have been blackholed. The drawback is that the stack trace could be too truncated to be useful.

To verify our speculations, we run the benchmarking program from figure 5.6 with the frame size limits 10 and 500:

```
$ ./Benchmark +RTS --stack-trace --reify-x-frames 10 -RTS
Takes 232 ms with small stack
Takes 236 ms with large stack

$ ./Benchmark +RTS --stack-trace --reify-x-frames 500 -RTS
Takes 315 ms with small stack
Takes 4120 ms with large stack
```

This makes sense. In the case with a cap on reifying 10 frames, we see that both computations take about the same time. It also takes more time than when not reifying a stack at all (since $10 > 0$), but it takes less time than when the stack is unbounded, since 10 is less than the number of frames on the stack. For the case when reifying 500 frames, we see that the computation on a small stack is taking as much time as it did then we reified the whole stack, this is because the whole stack is less than 500 frames. The computation done with the larger stack takes about one fourth of what it was when we reified the whole stack, this is because we call `stackBuilder` 1000 times, and each time it pushes two frames (one addition, one subtraction).

5.2.2 Static analysis

By analyzing Haskell source code at compile time we can get information that could help decide whether we should reify the stack or not. For instance, when the compiler generate code for the `throw` primitive operation, the *compiler* could choose if a stack should be reified or not at this particular usage of `throw`. Another idea is to do static analysis on the uses of the `catch` primitive, one could mark the catch frames that are using the stack value and choose at run-time to reify the stack if the catch frame needs it.

To do static analysis on the uses of `catch` has no runtime cost, but there many problems. When the execution stack contains a series of catch statements and if the topmost catch frame indicates that it doesn't need the stack, it will not be possible to rethrow the stack trace to the second catch frame which might need it. Section 6.3.2 in the next chapter will discuss semantics of rethrowing in detail.

Another serious obstacle with static analysis is that the code generation phase would require a lot more context to decide what kind of catch or throw site it is. Catch and throw sites have to generate code defensively (like with all compiler optimizations). While code from remote libraries must be compiled defensively as one would expect, the biggest issue is thunks! Since thunks passed as arguments can contain arbitrary code, even code that throws, the compiler have to assume the worst.

We have not found any sensible way to do static analysis that prove that a throw at a throw site don't need to reify the stack. We reach the same conclusion when doing analysis on catch sites. Even if a handler can be proven to not use stack traces, we might still need reify the stack in order to pass it down to the next handler.

5.2.3 Stack freezing

One issue with lazy reification is that the stack is a mutable data structure, so it is not enough to have a reference to the topmost chunk. The issue of mutability goes away if we make the stack structure immutable when reifying[2], we call that *stack freezing*. To freeze the whole stack would mean to freeze each individual stack chunk. A frozen chunk's content should be regarded as read-only and the reference to the top of the stack should not change either. It turns out that thanks to already existing machinery, much is already implemented.

Freezing a particular chunk (a `StgStack` value) would be trivial by just setting the chunk's `stack_size` to be zero and saving the `sp`-value to another field. By setting the `stack_size` to zero, the overflow check described in section 4.3 would automatically kick in as a copy-on-write mechanism. Since we saved the `sp` value at the time of the reification, the `sp` value can change, which means that the stack chunk can still safely shrink. The stack value itself would be a value with a reference to the stack chunk that was current at the time of the exception, as seen in figure 5.7.

So far this solution is linear, since freezing the whole stack would mean to traverse through each chunk and freezing them. The number of chunks is linear in the size of the stack. Luckily, we can get away with freezing the whole stack by only freezing the chunk where the handler lies. Because when we get an underflow, control is passed to the RTS [62]. From the RTS code we could freeze the next chunk of the underflow frame, which can be thought of as freezing lazily.

There are multiple drawbacks of freezing the stack. When doing a garbage collection, a stack value has a reference to the old stack and it would be wasteful to retain the

[2]So far, all stack reifications have been for the purpose of creating a stack value. The name reification has been used since all methods discussed until now have been about copying the essentials of the stack and store it in changed format. To avoid too much terminology in this paper, we call *all* methods of creating a stack value for "reification".

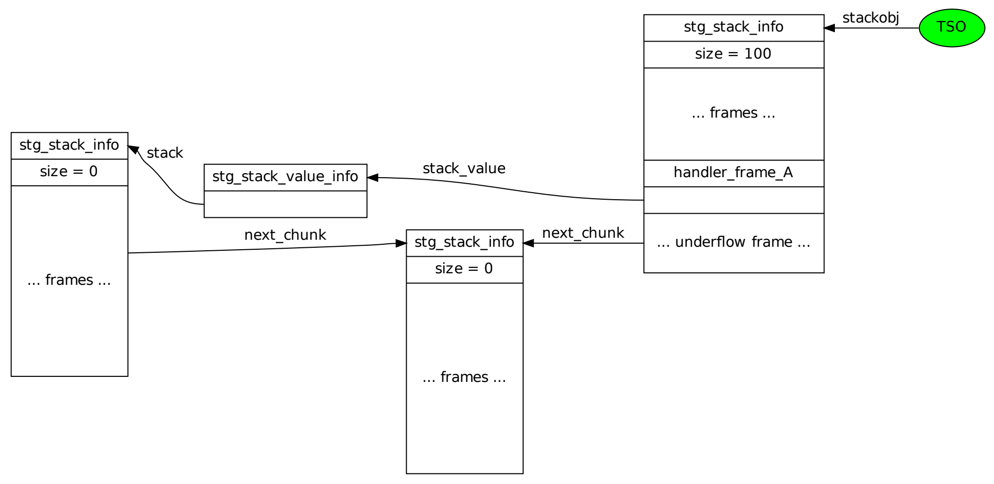


Figure 5.7: Illustration of stack freezing. The frames with `size = 0` are the frames that have been frozen by the run time system when a crash was detected.

payloads and all their references if the eventual printed stack trace is only based on the info pointers. Another drawback is that it is not clear when (if ever) to *thaw* the stack. Thawing would be the process of undoing the freezing of the stack. Freezing the stack and the chunk is very cheap, but a frozen chunk will cause overflows on every push. Thawing can be done in constant time by just restoring the `stack_size` of the current stack chunk. Thawing would ideally happen when the last stack value become unreachable from Haskell code. Subsection 5.2.4 will look at how the RTS can know when to initiate the thawing.

The overall advantage of stack freezing is that we would never actually need to create another array, so then all payloads are still accessible. If the payloads are available from the stack value in the handler, the optimizations from section 5.1 could be configurable from code in Haskell-land. Even better, one day the stack traces could utilize the stack frames even more by for example printing the values of the free variables in a case continuation.

5.2.4 Stack thawing

When we defined freezing and thawing in subsection 5.2.3 we said that both freezing and thawing were quite simple operations. However, we still need some sort a destructor mechanism for heap objects since it is only safe to thaw the `StgStack` chunk when it is not used anymore. The RTS in GHC does support *weak pointers*. Unlike regular pointers, weak pointers do not maintain liveliness for the object it is pointing at. During garbage collection, The weak pointers are visited after all other live pointers have been visited, in that way it is possible to see if a heap object have died since the last garbage collection. When a dead weak pointer is visited, the pointer is removed and a *finalizer* is run. It is tempting to let thawing be the finalizer of stack values, but there could be values that are created from the stack value that relies on the stack being left intact. For example, we could have that the stack is exposed to the programmer as a Haskell list. If any of the list values are alive the stack must not thaw as the data source for the list (the stack) can then be overwritten. One way to ensure that the finalizer is not run prematurely is for every heap object that depends on the stack being alive to have a pointer to the stack value. This way the stack value will be alive as long as its dependents are alive.

Another design question is if the finalizer should thaw the stack. The stack is owned by a particular capability, but the finalizer can be run by any capability doing the garbage collection. We propose that thawing only should happen on stack underflows, because then we are certain that the running thread is thawing its own stack and we do not need to worry about the overhead of locks and there being race conditions. But then what would the finalizer do? How would a underflowing stack chunk know if it should thaw the succeeding chunk? We propose a scheme like in figure 5.8: Every `StgStack` chunk has a linked list of subscribers that require the stack to be kept frozen (in the figure we just have one pointer and do not show the linked list structure). The only weak pointer is pointing at the stack value and its finalizer is to mark the subscriber as dead. When a chunk underflows it will traverse its linked list and remove the entries pointing to dead subscribers. The next chunk will then be be frozen if and only if the subscriber list is nonempty. The list will then be inherited by the succeeding chunk.

One alternative for a linked list to subscribers could be to have a counter containing the number of subscribers. But that would require synchronization primitives.

5.2.5 Chunked reifying

Chunked reifying of the stack is another idea about doing the stack reification lazily. In chunked reification we do not freeze the stack, instead, we observe that the stack is *almost* immutable already! The stack can only modify itself frame by frame unlike an array which can access any element. This is a powerful property since this means that the stack is almost immutable. Like for stack freezing we want to have some sort of copy-on-write mechanism and we will save frames once they are popped from the stack. It is too expensive to copy over one frame each time the stack pointer goes below its yet lowest point since the reification. Instead we can copy over stack frames in chunks. Luckily, the execution stack implemented in GHC is already chunked and we could decide to reify the next chunk on each underflow. One very useful trick is to save the reified stack chunk in the chunk that underflowed and not in its successor. It might feel more natural at first to store the reified frames for chunk A in chunk A, but if chunk A overflows again and does another stack reification, then A has to hold two *different* reified chunks and the stack values would not know which one belongs to them. We avoid this problem by storing the chunk in the overflowing chunk.

Like with the approach of stack freezing, it would improve performance to do some-

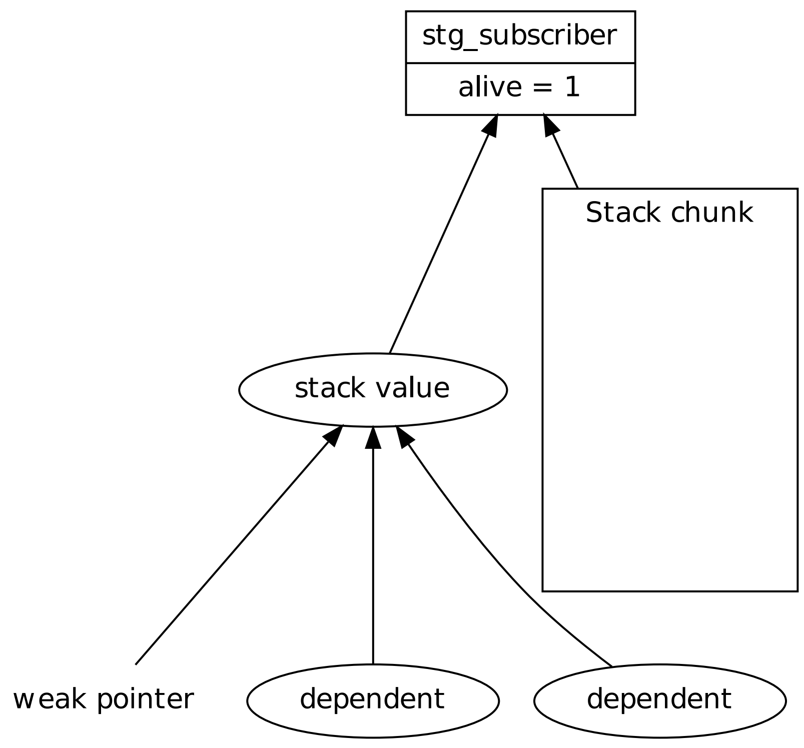


Figure 5.8: A scheme for stack thawing without locks.

thing similar to thawing the stack when stack values become inaccessible. Even if chunked reification does not freeze the stack it must be able to tell stack parents that they do not need to reify on underflows when no stack value is alive anymore. The finalizer scheme could be the same as the one described in subsection 5.2.4.

6

A Haskell Interface

Our goal is for Haskell to ”have stack traces”. This could just mean that the language prints a stack trace when the program crashes, however it would be more powerful if the programmer has some control of this. With a Haskell-API, the programmer can print the stack trace at will and not only on crashes. Further, the programmer could examine what is on the stack, for example one could check if the function `foo` is on the stack or not. For this to be possible, some interface has to exist in Haskell-land. One contribution of this thesis is a Haskell API to the execution stack, section 6.1 explains the design choices when creating that interface. Aside from the API, it would also be great if stack traces got integrated into the exception system, since once an exception has occurred, it is too late to get the original stack trace, somehow the stack trace has to be passed to the exception handler. This will be discussed in section 6.3. Section 6.2 covers the preliminaries needed to understand section 6.3.

6.1 User-invoked reification

The stack traces shown in section 5.1 look something like this:

```
3: showSignedInt
4: print
5: writeBlocks
```

We basically have assumed that the DWARF debug data gives us a simple mapping from instruction pointer to function names. We could model this in Haskell:

```
-- | Location in source code.
data LocationInfo = LocationInfo
    { startLine    :: !Word16
    , startCol     :: !Word16
    , endLine      :: !Word16
    , endCol       :: !Word16
    , fileName     :: !String
    , functionName :: !String
    }
    deriving(Eq)
```

Figure 6.1: The information necessary to display a *row* from the stack trace.

```
import Foreign.Ptr (Ptr)

data Instruction -- Empty data declaration (for type safety)
type FunctionName = String

lookupWithDwarf :: Ptr Instruction -> IO FunctionName
```

But we actually get much more information than just the function name from the DWARF info. We also know in which file it was defined and more exactly where in the function. A stack trace including all this information could look like this:

```
3: showSignedInt (at libraries/base/GHC/Show.lhs:432:1-434:56)
4: print (at libraries/base/System/IO.hs:281:29-281:37)
5: writeBlocks (at libraries/base/GHC/IO/Handle/Text.hs:584:4-609:31)
```

We have created a Haskell data type to encapsulate one frame like shown above. We call the data type a `LocationInfo` and its definition is in figure 6.1.

Yet, there is even more information stored in the DWARF data. For a given instruction pointer, you would usually associate it to only one place in the source code, but the situation is not always clear cut due to code transformations like inlining. Instead we have a mapping from one instruction pointer to *many* source functions.

```
3: showSignedInt (at libraries/base/GHC/Show.lhs:432:1-434:56)
4: print (at libraries/base/System/IO.hs:281:29-281:37)
   putStrLn (at libraries/base/System/IO.hs:267:37-267:38)
   print (at libraries/base/System/IO.hs:281:35-281:36)
5: writeBlocks (at libraries/base/GHC/IO/Handle/Text.hs:584:4-609:31)
```

With the fact that each instruction pointer maps to multiple source locations, it would make sense for `lookupWithDwarf` to return a list of `LocationInfo`s. Perhaps the signature should be like the following?

```haskell
lookupWithDwarf :: Ptr Instruction -> IO StackFrame
```

Figure 6.2: Given an instruction pointer, you get a `StackFrame` value.

```haskell
data StackFrame = StackFrame
    { unitName      :: !String
    , procedureName :: !String
    , locationInfos :: ![LocationInfo] -- Empty without -g flag to ghc
    }
```

Figure 6.3: The information necessary to display a *frame* from the stack trace.

```haskell
lookupWithDwarf :: Ptr Instruction -> IO [LocationInfo]
```

Unfortunately, DWARF information is not always available in the executable, for example if a Haskell module have been compiled without the `-g` flag. Luckily, symbol table information might still be packaged inside the binary, which would be able to get out some useful information.

Note that only *some* modules might have been compiled with or without `-g`, so a stack trace could have some stack frames actually resolved while others just contain information extracted from the symbol table. Here is an excerpt of a stack trace where the Haskell module `GHC.Show` was compiled without the `-g` flag.

```
3: c1lo_entry (using libraries/base/GHC/Show.lhs)
4: print (at libraries/base/System/IO.hs:281:29-281:37)
   putStrLn (at libraries/base/System/IO.hs:267:37-267:38)
   print (at libraries/base/System/IO.hs:281:35-281:36)
5: writeBlocks (at libraries/base/GHC/IO/Handle/Text.hs:584:4-609:31)
```

While `c1lo_entry` is not too useful (but it hints that it could be a case continuation), the second piece of information is useful. The file `Show.lhs` would probably have something to do with converting a value to a string. Therefore, it would help if the Haskell value includes this information when it is present. With this in mind, we decide on the API shown in figure 6.2 and 6.3. The structure fields `unitName` and `procedureName` from figure 6.3 could example be `libraries/base/GHC/Show.lhs` and `c1lo_entry` respectively, as in the last stack trace sample. The exact procedure of calculating the `unitName` and `procedureName` is hard wired in the run time system. In essence, the values will be based on DWARF values, like those shown in figure 2.16 [78]. If there is no DWARF data stored in the binary, the symbol table of the binary can be used as a last resort [79].

These decisions are debatable. What happens if there is no symbol table? What happens if the frame on the execution stack was an update frame, but we reified the code pointer of its payload (recall section 5.1.1)? Both the signature

```
lookupWithDwarf :: Ptr Instruction -> IO (Maybe StackFrame)
```

and

```
lookupWithDwarf :: Ptr Instruction -> IO [StackFrame]
```

makes sense. But the API should preferably be simple and natural. Most programmers might be used to a one to one mapping between instruction pointer and source code. Therefore we choose the signature in figure 6.2. Then, when programmers looks at what a `StackFrame` is, *then* they will be gently introduced to the fact that a stack frame may contain multiple `LocationInfo`s. Contrast this to giving an API based on these observations:

- One execution stack frame maps to multiple stack frames (like for update frames).
- We "meld" `StackFrame` and `LocationInfo`s, basically substituting `StackFrame` with `[LocationInfo]`.
- We have a function `getStackFrames` that returns a list of the frames on the execution stack.

Then we would end up with a function like:

```
getStackFrames :: IO [[[Locationinfo]]]
```

Or if we commented the signature:

```
getStackFrames :: IO
  [ -- The execution stack has multiple frames
   [ -- Each execution stack frame maps to multiple frames
    [ -- An instruction pointer maps to multiple source code locations
    Locationinfo
    ]
   ]
  ]
```

No, this is not a user-friendly API. Still, we need to discuss two issues:

1. All DWARF data and the symbol table could have been stripped away.
2. Update frames can be thought as being multiple frames.

We solve the first issue by simply setting the `String`-fields of a `StackFrame` to a descriptive value like `"<No data>"` or just `""` if we do not find anything in the symbol table.

In the case of having an execution stack frame that corresponds to multiple frames (like for update frames), we pick the most helpful stack frame (the updatee over the updater). After all, most reification methods from the previous chapter is only capable of storing one info pointer per stack frame. But if we were making all frames accessible from the Haskell API, one idea would be to squeeze in another field `next :: Maybe StackFrame` in the definition of `StackFrame` from figure 6.3. Then that particular field's documentation could explain how it works.

Lastly, the API will include some non-controversial functions like pretty-printing of the execution stack. Since there is only one natural implementation for these functions, we do not discuss them further.

6.2 Exception system

The *exception system* of a language is how throwing and catching is done in that language. The exception system for GHC got an overhaul when version `6.10.1` was released in November 2008 [80]. The design of the new exception system was introduced in a paper from 2006 [75]. In this section we will look at how this relatively new and current exception system works. The most recent Haskell report does not mention this exception system [8, ch. 42], but we only care about GHC here and still consider "Haskell" to mean the language implemented in GHC.

The exposed exception system in Haskell is surprisingly similar to Java's, but their implementation differs vastly. There is a root exception type in Java called `Exception` and the Haskell equivalent is called `SomeException`. One subclass of exceptions are arithmetic exceptions, Java has `ArithmeticException` and Haskell has `ArithException`. So Java and Haskell are similar in the sense that they both have an *extensible and hierarchical* exception system. Haskell, unlike Java, do not have anything like the `throws` annotation[1] in the type system, but it can be implemented as a library [81, 82].

One way to learn the Haskell exception system is to study the documentation of the `Control.Exception` module from the `base` library [83]. That module is quite sizeable, so we will only study the `catch` and `throw` functions since that is enough background to be able to discuss how stack traces can be added to the exception system. The documentation contains many functions with various best practices the programmer should know, `catch` and texttttthrow is not enough to write good code. Let's continue by looking at some example uses of `catch` and `throw` in GHCi.

[1]Introduced in section 2.2.1.

```
-- We import what we need
> :set -XScopedTypeVariables
> import Control.Exception

-- It is equivalent to cause the exception and to throw it manually
> (1 ‘div‘ 0)
*** Exception: divide by zero
> throw DivideByZero
*** Exception: divide by zero

-- You can catch the exception
> catch (throw DivideByZero) (\ (e :: SomeException) -> putStrLn "Yaay")
Yaay

-- But if the type of the error you are catching is not a superclass
-- of the exception being thrown, the catch will not be caught
> catch (throw DivideByZero) (\ (e :: IOException ) -> putStrLn "Yaay")
*** Exception: divide by zero
```

The type signatures of `catch` and `throw` can be found in figure 6.4. The signatures are reproduced from the documentation [83]. The type signature for both functions are in the form `:: Exception e => ...`, this means that `e` is a constrained type. The type variable `e` can not be of any type, under common circumstances it can not be an `Int` or a `String`[2], it has to be some sort of an `Exception`. `Exception` is not a type (unlike `SomeException` and `ArithException`), `Exception` is *type class*[3]! For a particular type to qualify as being an `Exception` type, it must adhere to the following interface:

- Any value of type `e` can be converted into a value of type `SomeException`. The programmer has to provide the function:

  ```
  toException :: e -> SomeException
  ```

- All `SomeException` values must have a more concrete type underneath[4], the concrete type could for example be an `ArithException`. In general we would ask ourselves if a particular `SomeException` value is somewhere down the hierarchy built using a node of type `e`. If so, we say that we can *downcast* a `SomeException` value to a value of type `e`, essentially we are looking for a downcasting function in Haskell. This will probably be achieved using dynamic casting in Haskell. To fulfill the `Exception` interface the programmer has to provide the following downcasting-like function:

[2] But it can if an `Exception` instance is provided for one of these types

[3] The name “class” is unfortunate since it is not at all like classes in object oriented programming, it is more like interfaces.

[4] At least all total and finite `SomeException` values

```
-- | Throw an exception.  Exceptions may be thrown from purely
-- functional code, but may only be caught within the 'IO' monad.
throw :: Exception e => e -> a

-- | This is the simplest of the exception-catching functions. It
-- takes a single argument, runs it, and if an exception is raised
-- the "handler" is executed, with the value of the exception passed
-- as an argument. Otherwise, the result is returned as normal.
catch   :: Exception e
        => IO a         -- The computation to run
        -> (e -> IO a)  -- Handler to invoke if an exception is raised
        -> IO a
```

Figure 6.4: The function signatures for `catch` and `throw` and some excerpts of the documentation.

```
    fromException :: SomeException -> Maybe e
```

So as indicated from the type signatures of `throw` and `catch`, they do use the `toException` and `fromException` functions (otherwise they would not need the type class constraint). The definition of `throw` uses `toException` meanwhile `catch` uses `fromException` [84].

The implementation of `toException` and `fromException` differs for every type. But the convention is that `toException` wraps a bare exception value into wrappers, one wrapper per level of hierarchy. Similarly `fromException` removes the layers. For the hierarchy shown in figure 6.5, the `toException` and `fromException` functions would work like this:

$$\begin{tikzcd}
 & \texttt{SomeException} \arrow[d] & \\
 & \texttt{SomeCompilerException} \arrow[d] & \\
 & \texttt{SomeFrontendException} \arrow[dl] \arrow[dr] & \\
\texttt{MismatchedParentheses} & & \texttt{TypeCheckerFailure}
\end{tikzcd}$$

Figure 6.5: One example exception hierarchy.

```
> toException MismatchedParentheses
SomeException
  (SomeCompilerException
    (SomeFrontendException MismatchedParentheses))

> toException TypeCheckerFailure
SomeException
  (SomeCompilerException
    (SomeFrontendException TypeCheckerFailure))

> (fromException :: SomeException -> MismatchedParentheses)
    (toException MismatchedParentheses)
(Just MismatchedParentheses)

> (fromException :: SomeException -> TypeCheckerFailure)
    (toException MismatchedParentheses)
Nothing
```

The multiple layer of constructors is added by `toException` and `fromException` checks if its argument was wrapped by the corresponding `toException` function. When seeing this it should be clear that we are not really doing downcasting in the OOP sense. When using `fromException` we are actually losing information, we can not upcast the value back to its *original* value of type `SomeException`. Contrast this to the real downcasting that is done in Java, when an `Exception` in Java is successfully downcasted to a `ArithmeticException` value it is still the same value, only with a new type. Upcasting is always possible in Java by doing a normal type cast.

In section 6.3 we will look at how we can catch exceptions with stack traces. So far we have only looked at how the exception system is implemented. The implementation details are necessary background so we can discuss the proposal that modifies the existing exception system.

6.3 Adding the trace

In this section we will look at two problems. First, how can we give the handler a stack value? Second, which stack should be reified when we throw inside the handler? When answering these questions, we say that we have a type called `ExecutionStack`, values of this type will be named `executionStack` by convention. The underlying data representation of the stack value that was discussed in section 5.2 should not matter.

6.3.1 Catching the stack

To catch the stack should have an intuitive interface. Possibly something like

```
catchWithStack :: Exception e => IO a -> Handler e -> IO a
```

Where `Handler e` is either `(e -> ExecutionStack -> IO a)` or `(e -> Maybe ExecutionStack -> IO a)`. Another approach would be if there was a function like

```
getStack :: Exception e => e -> ExecutionStack
```

This would be *very* convenient and would solve both catching and throwing at the same time. It is now time to look at possible implementations that would give a decent Haskell-interface to the programmer.

Implementations in pure Haskell

We can change the type of `SomeException` from

```
data SomeException =
  forall e . Exception e => SomeException e
```

to

```
data SomeException =
  forall e . Exception e => SomeException e ExecutionStack
```

But this breaks compatibility because you change the type of `SomeException`. Furthermore it is not as convenient as it seems. If the programmer catches an exception downcasted to say an `ArithException`, it would be intuitive that the stack trace would be reused if you throw the exception variable because the stack trace is stored in the exception value. But that is not true once you downcast the `SomeException` value.

Another design is to add an *optional* layer containing the stack. If `(toException DivideByZero)` evaluates to `(SomeException DivideByZero)`, we could imagine a function `toExceptionWithStack` where `(toExceptionWithStack DivideByZero)` evaluates to `(SomeException (WithStack DivideByZero executionStack))`. While this does not break backwards compatibility, users has to actively choose `toExceptionWithStack` over `toException`.

Both these solutions would be accompanied with small changes to library functions like `throw` and small changes to the RTS. Ideally all raised exceptions will have the stack trace in it without needing to change any external library code. However, we have not found any work around for the fundamental issue of no proper downcasting. That rethrowing the same exception only propagates the stack when the exception is a `SomeException` is confusing, particularly for programmers used to languages with a hierarchical exception system based on subtyping like Java and Python.

RTS-based solutions

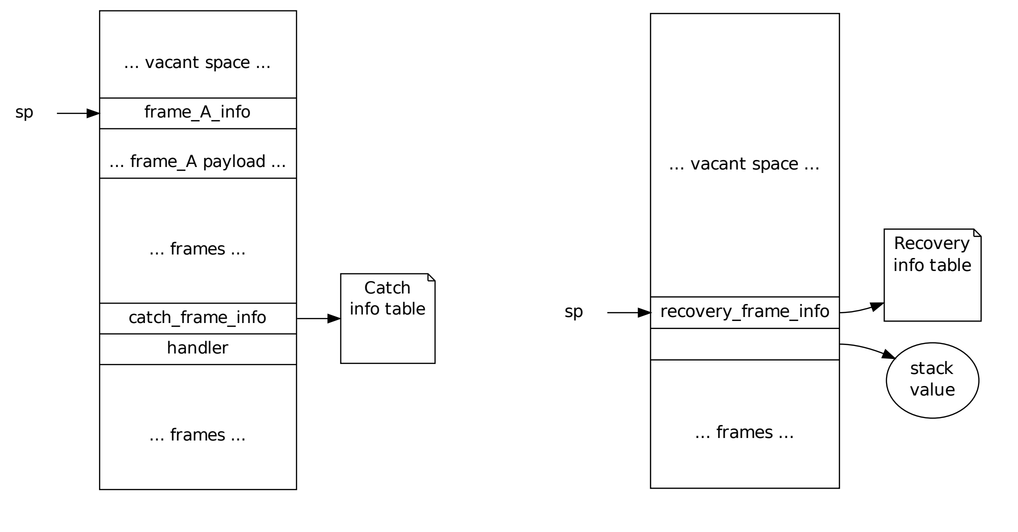
It would be desirable for our solution to be implemented as much as possible in a Haskell library instead of the run time system, it would then work for any Haskell compiler that has the same exception system as GHC. Unfortunately we did not find a satisfactory solution when limiting ourselves to solutions in Haskell-land (obviously there could exist clever solutions we have not discovered). We will now broaden the solution space to include RTS changes.

Currently, control is passed to the run time system when an exception is thrown. The RTS gets passed control when functions like `throw` [85] and `error` are invoked, because both will eventually call `raise#` which is is a RTS primitive [86]. When `raise#` is called, it will walk the stack towards the first frame type that (possibly) can handle exceptions (but we only consider `CATCH_FRAME`) [87, 88]. When a `CATCH_FRAME` frame is found, `raise#` will run the frame's handler with the exception passed as an argument. The handler on the stack should therefore be of arity 1, more precisely it should conform to `(Exception e => e -> IO a)`. The way to get this catch frame on the stack in the first place is to use the primitive `catch#`, for example it could be invoked with `(catch# ioThatCanCrash myHandler)`. The type signature for `catch#` is[5]:

```
catch# :: IO a -> (exception -> IO a) -> IO a
```

There are many interesting possibilities here, for example in Peter Wortmann's demonstration from section 3.4 he changed the `catch#` primitive to have the following signature:

[5]The type system for RTS primitives is weaker, we can not constrain the type variable `exception` to belong to the type class `Exception`.



(a) The stack when `raise#` is invoked. **(b)** The stack when the handler is invoked.

Figure 6.6: The stack before and after an exception.

```
catch# :: IO a -> (exception -> ByteArray# -> IO a) -> IO a
```

Where `ByteArray#` contains the reified stack. With this change it would be trivial to implement `catchWithStack` and to reimplement `catch` in terms of `catchWithStack`. But also the `raise#` primitive must change its implementation. After all `raise#` is the function that invokes the handler (`catch#` just pushes it on the stack). The stack reification can happen in `raise#` which will then pass it as an argument to the handler it finds on the stack.

Another approach would be to not change the signature of `catch#`. But how would `catchWithStack` be implemented if the undermost handler does not take the execution stack as an argument? Of course it could be put in the exception value, but we propose to instead push a *recovery frame*, the recovery frame would contain the stack value. Figure 6.6 illustrates when and where a recovery frame would be pushed. Once the handler gets control, we know that stack pointer `Sp` points at the recovery frame. It would then be a constant time operation to retrieve the stack value. We can imagine a function with this signature:

```
-- Traverse the stack and look for a recovery frame
recoverExecutionStack :: IO (Maybe ExecutionStack)
```

This approach will not break backwards compatibility since it does not change any function in Haskell land. This time `catchWithStack` is defined in terms of `catch` and

	Inside the handler	Outside the handler
Throwing the caught exception	Rethrow	Rethrow
Throwing a new exception	Throw anew	Throw anew

Table 6.1: The throw semantics in Java.

not the other way around. But one of the best benefits with this approach is that it introduces a nice way to implement rethrowing of stack traces.

6.3.2 Rethrowing the stack

When throwing an exception for the first time we reify the stack at the crash site. After reification the exception will be passed to a handler. We can categorize the resulting effects of running the handler, either:

- The handler can resolve the exception. For example it can print an error message.
- The handler can declare itself incapable of handling the exception, the exception must then be passed to the next handler on the stack. We say that the handler *rethrows* the exception. When rethrowing we always want the next handler to get the original stack trace.
- The handler itself may crash and generate a completely new exception on its own, we say that the handler is *throwing anew*. When throwing anew we always want the next handler to get a new stack trace.

These scenarios do not directly map to an obvious Haskell interface. In Java, the stack traces are stored *inside* the exception. To do a rethrow you must throw the exception value that you caught. If you create a new exception (even if it is of the same type) a new stack trace will be reified from inside the handler. Whether if you throw from inside the code of the handler[6] or not does not matter in Java. The way a language defines rethrowing can be summarized in a table, for Java it would look like be like table 6.1. The stated behavior of Java is based on the experiments in [89].

When designing any programming language, we will obviously be steered by what is technically possible. When we tried to put the stack trace value inside the `SomeException` type, we realized that the hierarchical exception system from [75] is inferior to actual class hierarchies and "real" downcasting that exists in the object oriented programming language Java. Due to the technical reasons we can not conveniently store the stack trace in the `SomeException` value, so the semantics of throwing in Haskell can not be identical to Java. Instead we propose that Haskell have the semantics shown in table 6.2, so that we count a throw as a rethrow if it is within the handler.

[6]in Java we say that the `catch`-block is the handler.

	Inside the handler	Outside the handler
Throwing the caught exception	Rethrow	Throw anew
Throwing a new exception	Rethrow	Throw anew

Table 6.2: One suggested throwing semantics for Haskell.

```haskell
try :: Exception e => IO a -> IO (Either e a)
try a = catch (a >>= \ v -> return (Right v)) (\e -> return (Left e))

-- | A variant of 'try' that takes an exception predicate to select
-- which exceptions are caught. If the exception does not match the
-- predicate, it is re-thrown.
tryJust :: Exception e => (e -> Maybe b) -> IO a -> IO (Either b a)
tryJust p a = do
  r <- try a
  case r of
        Right v -> return (Right v)
        Left  e -> case p e of
                        Nothing -> throwIO e
                        Just b  -> return (Left b)
```

Figure 6.7: The definition of `try` and `tryJust`.

Snatching the trace in the recovery frame

Now we can contemplate over how to implement rethrowing to follow the specification in table 6.2. Recall the recovery frame, it can be used for rethrowing too: Say that we are inside a handler, then a recovery frame must be somewhere on the execution stack. If the handler now does a rethrow, we do not need to pass the stack trace to the throw function, because the handler-search in `raise#` will walk the stack downwards anyway, when doing that it can look for the recovery frame and snatch the stack trace stored within it.

One serious issue is that sometimes a function might intend to rethrow but do the throwing outside the handler. Consider the implementation of `tryJust` from [90] reproduced in figure 6.7. The implementation of `tryJust` would not do a rethrow if Haskell had the rethrow semantics shown in table 6.2 and it is obvious that `tryJust` should be rethrowing.

There is another problem with using the semantics that throwing in handlers mean rethrowing. Imagine a long-running handler, the handler could be calling a separate function that throws on its own, both rethrowing and throwing anew are likely intents here. What can we do about this? It would be disastrous if the user would get an incorrect stack trace! Of course, we will provide throw functions where a programmer can explicitly say if we throw anew or if we should look for a recovery frame. But there will still be code using `throw` and for those cases the stack traces might be confusing. One comfort will be that long running handlers that do install their own catch handler will not be problematic. If a handler install its own catch frame, the handler will not get the old stack trace since it lies below the inner catch frame on the stack. Whether if long-running handlers are common have not been investigated in this thesis.

Long running handlers also have another issue, the recovery frame will point to a stack trace value, which means that it will not get garbage collected until the recovery frame is popped. For some of the implementations suggested in section 5.2, the stack trace value can keep ridiculous amounts of memory from being garbage collected. The stack trace value implementation of stack freezing from subsection 5.2.3 keeps the whole execution stack at the crash site alive.

These two issues with long-running handlers can be fixed by having a way to remove the pushed recovery frame. It must be called immediately in the handler (just as with `recoverExecutionStack`):

```haskell
-- This function must be called first thing in the handler.
-- It will remove the recovery frame. Returns true iff successful.
removeRecoveryFrame :: IO Bool
```

Asynchronous throwing and other complications

The ideas presented in this section have not been implemented (not even partially). There are many features in GHC that have not been discussed that must be considered before the ideas in this thesis can be considered implementable. For instance Haskell supports *asynchronous exceptions* [91]. While we do not see any immediate problem with incorporating asynchronous exceptions and stack traces there could be subtleties around the corner, like in [92]. Another issue not brought up is concurrency. So the final word of this section is a warning that these ideas most probably will not work without some modifications that only will be realized down the road.

7

Conclusions

This thesis is titled “Stack traces in Haskell” which is a problem that have been attempted and solved many times before. The title was chosen to be understandable rather than specific. The actual question formulation is “Can efficient stack traces be implemented in GHC?”. We will now try to conclude how we managed to answer the question formulation and to guess how the future could be for GHC.

7.1 Did we answer the question formulation?

A prototype had already been made when this thesis work started. The prototype had set up a very promising start: The debug data did not interfere with optimizations and the debug data was emitted in the DWARF format. DWARF sections are stored separately in an executable and should therefore not affect performance. So the question was solved in the ”code generation” sense already, but the run time details of reifying the stack were problematic. Chapter 5 identified the efficiency issues with the stack reification method in the prototype. Some solutions were proposed that are able to reify the stack in constant time. Chapter 6 dealt with the problem that the original prototype was always reifying a new stack trace regardless of the kind of throw, but you do not want a new reification if you are propagating an exception. It turned out that the rethrow semantics must be a language decision and the semantics that is easy to implement in Java will not work in Haskell. The important result is that a consistent semantic of throwing can be implemented.

In this thesis we have come closer to incorporating efficient stack traces since we both have an idea of how to represent the stack trace value internally (Chapter 5) and how to expose it to users in Haskell-land (Chapter 6). Also, chapter 4 translated the code of the stack related parts in the run time system into high level text and illustrations. That chapter should shorten the time it takes to understand the execution stack for new GHC contributors.

7.2 What should be done next?

The contributions in lines of code from this thesis is quite small. Most ideas from chapter 5 and chapter 6 was never implemented. Which means that we do not know if they actually work and we have not been able to benchmark their performance. The ideas are however usually quite small changes and the theoretical analysis are indicating that the overhead should be low.

I expect to slowly try to implement some of the ideas and successively get patches merged. At the time of writing, all of my stack trace patches must wait for Peter Wortmann's debug data patches and DWARF patches to get merged first.

As of date, Haskell is a *backwards* language in regards to stack traces, but the ideas in this thesis could push Haskell towards becoming a forwards language. If one would implement the lazy reification idea of stack freezing from subsection 5.2.3 in combination with the idea of recovery frames from section 6.3, then Haskell have the potential to have stack trace values with the following properties:

- *Efficient* – The reification is done in constant time with respect to the stack size. This must obviously happen *lazily*, any strict implementation would need to be at least linear in the size of the stack. Efficient reification is a must if programmers are using exceptions for control flow.

- *Available post mortem* – The stack can not only be reified at will, but also at the time when an exception is thrown.

- *With variables* – To print the values of the variables is out of reach today, but the stack value will keep the payloads alive as well. So only thing hindering variables to be printed is that the instruction pointer to source code mapping does not describe the fields[1].

- *First class value* – The stack value would act like any other Haskell value. You can put it in an `IORef`, pass it around between functions and let other threads use it. As long as the stack value is live it is accessible. Contrast this to the Python programming language, which relies on the magical and "thread and stack frame local" function `sys.exc_info()` [93].

- *Lock free* – Synchronization primitives are slow. By only attempting thawing when underflowing, we can be sure that only one virtual CPU is mutating the thread state objects and its stack chunks at a time.

Yet, there is still a lot of work on stack traces not even considered throughout this thesis. In particular how stack traces will work in conjunction with foreign C calls and in calls (C code calling into Haskell). Haskell code in general is quite robust and free from errors thanks to its type system and its users appreciation for totality. So even if

[1]Well, Haskell is a lazy and non-total language, so evaluating a variable in the payload could both take time and have side effects.

just 1% of Haskell code is dealing with foreign calls and in calls, we should expect for it to be responsible for a much higher percentage of the crashes. In particular, Haskell users from industry are interested in stack traces for foreign calls, the typical scenario would be that a company has a large code base written in C and just wants a small Haskell program using the larger C code base. Fortunately, the problem of stack traces for segfaulting code have been studied as of very recently [94]. Indeed, stack tracing *is* a very sought-after problem to solve for Haskell programmers.

Bibliography

[1] Peter M. Wortmann and David J. Duke. “Causality of optimized Haskell: what is burning our cycles?” In: *Haskell*. Ed. by Chung chieh Shan. ACM, 2013, pp. 141–152. ISBN: 978-1-4503-2383-3.

[2] Adrian Schroter, Nicolas Bettenburg, and Rahul Premraj. “Do stack traces help developers fix bugs?” In: *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE. 2010, pp. 118–121.

[3] Haskellwiki. *Research papers*. Nov. 2013. URL: `http://www.haskell.org/haskellwiki/Research_papers`.

[4] Jason Dagit. *Getting Started With GHC Hacking*. Nov. 2013. URL: `http://blog.codersbase.com/posts/2013-08-03-getting-started-with-ghc-hacking.html`.

[5] Haskellwiki. *Haskell In Industry*. Oct. 2013. URL: `http://www.haskell.org/haskellwiki/Haskell_in_industry`.

[6] FP Complete. *Case Studies*. Oct. 2013. URL: `https://www.fpcomplete.com/business/resources/case-studies/`.

[7] Rosetta Stone Wiki. *Stack traces*. Dec. 2013. URL: `http://rosettacode.org/mw/index.php?title=Stack_traces&oldid=171024#Ruby`.

[8] Simon Marlow. *Haskell 2010 Language Report*.

[9] Paul Hudak et al. “A history of Haskell: Being lazy with class”. In: *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III)*. ACM Press, 2007, pp. 1–55.

[10] Tristan O. R. Allwood, Simon L. Peyton Jones, and Susan Eisenbach. “Finding the needle: stack traces for GHC.” In: *Haskell*. Ed. by Stephanie Weirich. ACM, 2009, pp. 129–140. ISBN: 978-1-60558-508-6. URL: `http://dblp.uni-trier.de/db/conf/haskell/haskell2009.html#AllwoodJE09`.

[11] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201441241.

[12] Oracle. *Method Throws.* Jan. 2014. URL: http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.6.

[13] Oracle. *Compile-Time Checking of Exceptions.* Jan. 2014. URL: http://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html#jls-11.2.

[14] Oracle. *The catch Blocks.* Nov. 2013. URL: http://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html.

[15] Oracle. *Unchecked Exceptions - The Controversy.* Jan. 2014. URL: http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html.

[16] Haskellwiki. *Avoiding partial functions.* Jan. 2014. URL: http://www.haskell.org/haskellwiki/index.php?title=Avoiding_partial_functions&oldid=47858.

[17] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real world haskell.* O'Reilly Media, Inc., 2008.

[18] Edward Z. Yang. *8 ways to report errors in Haskell revisited.* Mar. 2014. URL: http://blog.ezyang.com/2011/08/8-ways-to-report-errors-in-haskell-revisited/.

[19] Wikibooks. *Haskell/Graph Reduction.* Mar. 2014. URL: http://en.wikibooks.org/w/index.php?title=Haskell/Graph_reduction&oldid=2562184.

[20] GHC Website. *What is GHC?* Oct. 2013. URL: http://www.haskell.org/ghc/.

[21] Simon. "Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine". In: *Journal of Functional Programming* 2.2 (1992), pp. 127–202. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.3729.

[22] Simon Marlow and Simon Peyton Jones. "Making a fast curry: push/enter vs. eval/apply for higher-order languages". In: *Journal of Functional Programming* 16.4–5 (July 2006), pp. 415–449. URL: http://community.haskell.org/~simonmar/papers/evalapplyjfp06.pdf.

[23] GHC Commentary. Mar. 2014. URL: https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts?version=29.

[24] Alfred V Aho et al. "Compilers: Principles, Techniques, & Tools with Gradiance". In: (2007).

[25] David Anthony Terei and Manuel MT Chakravarty. "Low level virtual machine for Glasgow Haskell Compiler". PhD thesis. The University of New South Wales, Sydney, Australia, 2009.

[26] Haskellwiki. *Ministg.* Mar. 2014. URL: http://www.haskell.org/haskellwiki/index.php?title=Ministg&oldid=45213.

[27] Michael J Eager and Eager Consulting. *Introduction to the DWARF debugging format.* 2012.

[28] Oracle. *Object File Format.* Mar. 2014. URL: `http://docs.oracle.com/cd/E19683-01/817-3677/chapter6-46512/index.html`.

[29] Simon Marlow et al. “A lightweight interactive debugger for haskell”. In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop.* ACM. 2007, pp. 13–24.

[30] GHC. *The GHCi Debugger.* Nov. 2013. URL: `http://www.haskell.org/ghc/docs/7.6.3/html/users_guide/ghci-debugger.html`.

[31] Dennis Felsing. “Visualization of Lazy Evaluation and Sharing”. Bachelor’s Thesis. Germany: Karlsruhe Institute of Technology, Sept. 2012.

[32] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004).* Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[33] Neil T. Dantam and Mike Stilman. “The Motion Grammar: Analysis of a Linguistic Method for Robot Control”. In: *IEEE/RAS Transactions on Robotics* 29.3 (2013), pp. 704–718.

[34] Arash Rouhani, Neil T. Dantam, and Mike Stilman. “Software-Synthesis via LL(*) for Context-Free Robot Programs”. In: *4th Workshop on Formal Methods for Robotics and Automation, RSS.* June 2013.

[35] James Cheney and Ralf Hinze. *First-class phantom types.* Tech. rep. Cornell University, 2003.

[36] D. A. Turner. “Total Functional Programming”. In: *Journal of Universal Computer Science* 10.7 (July 28, 2004), pp. 751–768.

[37] Dimitrios Vytiniotis et al. “HALO: Haskell to logic through denotational semantics”. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. 2013, pp. 431–442.

[38] Dana N Xu, Simon Peyton Jones, and Koen Claessen. “Static contract checking for Haskell”. In: *ACM Sigplan Notices.* Vol. 44. 1. ACM. 2009, pp. 41–52.

[39] Dana N Xu. “Extended static checking for Haskell”. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell.* ACM. 2006, pp. 48–59.

[40] Koen Claessen et al. “Automating inductive proofs using theory exploration”. In: *Automated Deduction–CADE-24.* Springer, 2013, pp. 392–406.

[41] Haskellwiki. *Development Libraries and Tools - Static Analysis Tools.* Jan. 2014. URL: `http://www.haskell.org/haskellwiki/index.php?title=Development_Libraries_and_Tools&oldid=55967#Static_Analysis_Tools`.

[42] Neil Mitchell and Colin Runciman. "Not All Patterns, But Enough - an automatic verifier for partial but sufficient pattern matching". In: *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*. Victoria, BC, Canada: ACM, Sept. 2008, pp. 49–60. ISBN: 978-1-60558-064-7. DOI: `http://doi.acm.org/10.1145/1411286.1411293`. URL: `http://community.haskell.org/~ndm/downloads/paper-not_all_patterns_but_enough-25_sep_2008.pdf`.

[43] GHC Wiki. *Maintaining an explicit call stack*. Oct. 2013. URL: `http://ghc.haskell.org/trac/ghc/wiki/ExplicitCallStack#Transformationoption1`.

[44] Hat Team. *Hat - generating and viewing traces*. Nov. 2013. URL: `http://projects.haskell.org/hat/`.

[45] Haskellwiki. *HaskellImplementorsWorkshop/2012*. Dec. 2013. URL: `http://www.haskell.org/haskellwiki/HaskellImplementorsWorkshop/2012#Programme`.

[46] GHC Trac. *Show stack traces*. Dec. 2013. URL: `https://ghc.haskell.org/trac/ghc/ticket/3693#comment:44`.

[47] Hackage. *forkIO*. Feb. 2014. URL: `http://hackage.haskell.org/package/base-4.6.0.1/docs/Control-Concurrent.html#v:forkIO`.

[48] GHC Commentary. Feb. 2014. URL: `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Scheduler?version=23#Capabilities`.

[49] GitHub. Feb. 2014. URL: `http://www.haskell.org/ghc/docs/7.6.1/html/users_guide/release-7-6-1.html`.

[50] GHC Commentary. Feb. 2014. URL: `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects?version=33#HeapObjects`.

[51] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/3c9aa40f1cb3f228a86b359466ac8f058583e157/rts/sm/Scav.c#L1640-L1769`.

[52] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/4f603db253434ba0758142c42109d02c95a0ceda/includes/rts/storage/ClosureTypes.h`.

[53] GHC Commentary. Feb. 2014. URL: `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/HaskellExecution/FunctionCalls?version=3#FunctionCalls`.

[54] GHC Commentary. Feb. 2014. URL: `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/HaskellExecution/CallingConvention?version=3#ReturnConvention`.

[55] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/58e5843a4118ca19fd1c93f52f2365d90bb1b9b6/compiler/cmm/CmmParse.y#L167-L179`.

[56] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/2f69aaea7066b8d11034925d9376fadd67361eca/includes/stg/MachRegs.h`.

[57] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/ea584ab634b17b499138bc44dbec777de7357c19/compiler/codeGen/StgCmmBind.hs#L554-L581`.

[58] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/1c160e588706f4ff6b4e391602e38f0a2044ec13/rts/Updates.cmm#L37`.

[59] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/2f69aaea7066b8d11034925d9376fadd67361eca/utils/genapply/GenApply.hs#L525`.

[60] GHC Blog. Feb. 2014. URL: `https://ghc.haskell.org/trac/ghc/blog/stack-chunks`.

[61] GHC Changeset. Feb. 2014. URL: `https://ghc.haskell.org/trac/ghc/changeset/f30d527344db528618f64a25250a3be557d9f287/ghc`.

[62] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/1bffa2b2e7b7d1a829dff44256a5d1da3f2aef88/rts/StgMiscClosures.cmm#L25-L42`.

[63] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/5f98d44d8617756971cf47c040f2556de4e98f63/compiler/codeGen/StgCmmForeign.hs#L261-L275`.

[64] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/d3b24e10d419f48e0839b08eb740d7138e56b390/rts/Threads.c#L602-L661`.

[65] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/9562f18769b18cd44290d14628dd8d9a45e7d898/rts/ThreadPaused.c`.

[66] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/d3b24e10d419f48e0839b08eb740d7138e56b390/rts/Threads.c#L561`.

[67] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/1c160e588706f4ff6b4e391602e38f0a2044ec13/rts/Updates.cmm`.

[68] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/9562f18769b18cd44290d14628dd8d9a45e7d898/rts/ThreadPaused.c#L237`.

[69] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/9562f18769b18cd44290d14628dd8d9a45e7d898/rts/ThreadPaused.c#L326-L328`.

[70] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/95854ca5276e3f4063ade7fe3a934bed46648270/rts/sm/Storage.c#L376`.

[71] GHC Changeset. Feb. 2014. URL: `https://ghc.haskell.org/trac/ghc/changeset/5d52d9b64c21dcf77849866584744722f8121389/ghc`.

[72] Richard Jones. “Tail recursion without space leaks”. In: *J. Funct. Program.* 2.1 (1992), pp. 73–79.

[73] GitHub. Feb. 2014. URL: `https://github.com/ghc/ghc/blob/3c9aa40f1cb3f228a86b359466ac8f058583e157/rts/sm/Scav.c#L1690`.

[74] Simon Peyton Jones et al. “A semantics for imprecise exceptions”. In: *ACM SIGPLAN Notices*. Vol. 34. 5. ACM. 1999, pp. 25–36.

[75] Simon Marlow. “An extensible dynamically-typed hierarchy of exceptions”. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. ACM. 2006, pp. 96–106.

[76] GitHub. Mar. 2014. URL: https://github.com/ghc/packages-base/blob/master/GHC/IO/Handle/Text.hs#L417-L424.

[77] GitHub. Mar. 2014. URL: https://github.com/Tarrasch/ghc/tree/d60c99748d837f4cc5f448bf36e61fe10b849a69.

[78] GitHub. Mar. 2014. URL: https://github.com/Tarrasch/ghc/blob/ffc0c799822d19565666c7587108230d02f169b2/rts/Dwarf.c#L569-L584.

[79] GitHub. Mar. 2014. URL: https://github.com/Tarrasch/ghc/blob/ffc0c799822d19565666c7587108230d02f169b2/rts/Dwarf.c#L467-L471.

[80] GitHub. Mar. 2014. URL: http://www.haskell.org/ghc/docs/6.10.1/html/users_guide/release-6-10-1.html.

[81] GitHub. Mar. 2014. URL: https://github.com/pepeiborra/control-monad-exception/tree/d32767e95756e3970909e65fc7b020af262ca04f.

[82] Hackage. *The control-monad-exception package*. Mar. 2014. URL: http://hackage.haskell.org/package/control-monad-exception.

[83] Hackage. *Control.Exception*. Mar. 2014. URL: http://hackage.haskell.org/package/base-4.6.0.1/docs/Control-Exception.html.

[84] GitHub. Mar. 2014. URL: https://github.com/ghc/packages-base/blob/8c249173042f978a1ce8503a76682547e61c8039/GHC/IO.hs#L280.

[85] Hackage. *GHC/Exception.lhs*. Mar. 2014. URL: http://hackage.haskell.org/package/base-4.6.0.1/docs/src/GHC-Exception.html#throw.

[86] GitHub. Mar. 2014. URL: https://github.com/ghc/ghc/blob/3fb19a913f7bf79bd7895c85c750b98308ddb1cf/rts/Exception.cmm#L431-L617.

[87] GitHub. Mar. 2014. URL: https://github.com/ghc/ghc/blob/3fb19a913f7bf79bd7895c85c750b98308ddb1cf/rts/Exception.cmm#L463-L465.

[88] GitHub. Mar. 2014. URL: https://github.com/ghc/ghc/blob/8f3ea7d7b88b7dac26756e8af9f9defd4208e521/rts/Schedule.c#L2627-L2727.

[89] GitHub. Mar. 2014. URL: https://gist.github.com/Tarrasch/7431590/5f868a3d0dc1c9ec456b5e260f0c82773b2f4859#file-inout-java.

[90] Hackage. *Control.Exception.Base*. Mar. 2014. URL: http://hackage.haskell.org/package/base-4.6.0.1/docs/src/Control-Exception-Base.html#tryJust.

[91] Simon Marlow et al. "Asynchronous exceptions in Haskell". In: *ACM SIGPLAN Notices*. Vol. 36. 5. ACM. 2001, pp. 274–285.

[92] Edsko de Vries. *The darker corners of throwTo*. Mar. 2014. URL: http://www.edsko.net/2013/06/11/throwto/.

[93] Python Software Foundation. *System-specific parameters and functions*. Mar. 2014. URL: http://docs.python.org/3.3/library/sys.html#sys.exc_info.

[94] GitHub. Mar. 2014. URL: https://github.com/blitzcode/ghc-stack/tree/7e7e941a04d2c0b3faf79204a675ffa1a1df4d62.