KIT

Karlsruhe Institute of Technology

Visualization of Lazy Evaluation and Sharing

Bachelor Thesis of

Dennis Felsing

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer: Prof. Gregor Snelting
Advisor: Joachim Breitner

Duration: 2012-06-01 – 2012-09-27

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text, and have followed the rules of the KIT for upholding good scientific practise.

Karlsruhe, 2012-09-27

. . . .
          (Dennis Felsing)

Abstract

This thesis details the motivation, implementation and evaluation of ghc-vis, a tool for visualizing Haskell data structures inside of GHCi and GHC compiled programs at runtime. The layout of objects on the GHC heap is used as the basis for two types of visualizations, which respectively aim to functionally be supersets of GHCi's :print and vacuum-cairo. Examples for using ghc-vis to further the understanding of lazy evaluation and sharing in Haskell are given.
The visualization of sharing and lazy evaluation in Haskell data structures can be useful to enhance the comprehension of functional data structures and algorithms and their perfomance in the environment of education, development and debugging.
A new library for parsing and viewing graphs in Haskell, called xdot, was created in order to interactively view graphs created by Graphviz.

Acknowledgements

I am grateful to Joachim Breitner for offering me this topic and advising me throughout the time I worked on it. His patience, competent answers and remarks have been very helpful in the creation of this thesis.

My parents receive my deepest gratitude for supporting me throughout my life.

Finally I'm thankful to my friends, maybe one of them will read this.

Contents

5 Conclusion   47

Bibliography   49

Appendix   51

CHAPTER 1

Introduction

Functional programming languages like Haskell offer a way to write well-structured software, which is easy to understand and still performs well. In order to achieve this one has to be able to consciously use features that result from the paradigm of functional programming.

This thesis aims to provide a tool for visualizing two features of this kind, namely lazy evaluation and sharing, in a way that can be used in the (self-)teaching of Haskell and the development and debugging of Haskell software.

A common example for the power of lazy evaluation are infinite data structures. They are easy to define and convenient to use in Haskell. We can write

$$ones = [1, 1 ..]$$

to construct an infinite list of 1s, which we can now access by the suggestive name *ones*. How does this work? Clearly the entire list can't be stored in memory as we do not possess the infinite tape of a turing machine.

We can write a function to retrieve the nth member of a list:

$$at\ 0\ (x : xs) = x$$
$$at\ n\ (x : xs) = at\ (n - 1)\ xs$$

Evaluating $at\ 2\ ones$ extracts the second member of the list *ones*, which gives us the integer 1. In the same manner $at\ 3\ ones$ and $at\ 4\ ones$ give us 1. But what is happening to the infinite list of *ones* as we access its members?

Figure 1.1 provides a view of the actual representation of the list in memory. The solution is that the infinite list of ones gets constructed just as we access it. Thinking about this leads us to a problem: When we evaluate $at\ (10 \uparrow 7)\ ones$ a huge list gets constructed and a lot of memory is used to store it. There must be a way to avoid this and indeed there is. We define the infinite list of ones in a different way:

$$ones' = 1 : ones'$$

The resulting list is the same and we can access it in the same way. But when we evaluate $at\ (10 \uparrow 7)\ ones'$ memory usage does not seem to increase at all. What is the reason?

As Figure 1.2 reveals this time no big list is constructed in memory, instead a list is created that references itself as the rest of the list.
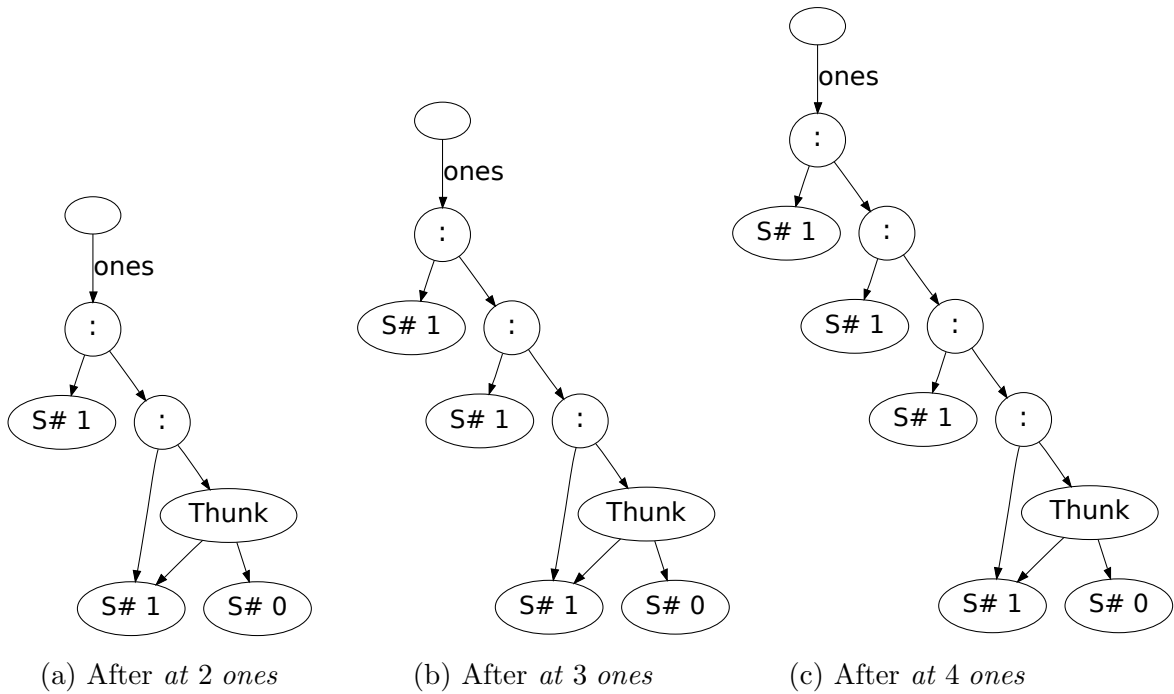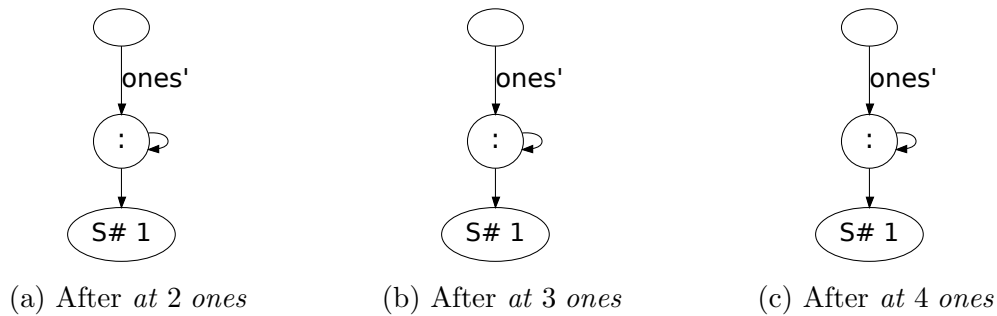
(a) After *at 2 ones* (b) After *at 3 ones* (c) After *at 4 ones*

Figure 1.1: Infinite list of *ones* gets lazily evaluated

(a) After *at 2 ones* (b) After *at 3 ones* (c) After *at 4 ones*

Figure 1.2: An infinite list of ones that uses constant space

The reader may want to experiment with these and the more complex examples provided later. All the Haskell code used in this thesis is available at `http://felsin9.de/nnis/ghc-vis/thesis`.

The tool developed as part of this thesis, *ghc-vis*, was made available as an open source package on Hackage: `http://hackage.haskell.org/package/ghc-vis`.

ghc-vis can be easily installed from the command line:

```
$ cabal install ghc-vis
$ echo ":script $HOME/.cabal/share/ghc-vis-0.4/ghci" >> ~/.ghci
```

Most testing was done on Linux systems. GTK, Cairo and Pango have to be installed and Cabal's package list should be updated if any problems occur during the installation:

```
$ cabal update
$ cabal install gtk2hs-buildtools
```

Now ghc-vis can be used inside of *GHCi*:

```
$ ghci
GHCi, version 7.4.2: http://www.haskell.org/ghc/  :? for help
ghci> :vis
ghci> let ones = [1,1..]
ghci> :view ones
```

An example for using ghc-vis will be presented in Subsection 3.1.1 and further documentation is available at `http://felsin9.de/nnis/ghc-vis`.

CHAPTER 2

Background

2.1 Haskell

In this section the aspects of Haskell which are relevant for this thesis will be outlined. For a more thorough introduction to Haskell consult for example Hutton's "Programming in Haskell"[1] or Davie's "introduction to functional programming systems using Haskell"[2].

Haskell is a *functional* programming language, which means the basic method of computation is the application of functions to arguments. In contrast to this the basic method of computation of imperative programming languages is changing stored values.[1]

To fully utilize functional programming functions are *first class* entities in Haskell, which means they

- can be named,

- can be the value of some expression,

- can be members of a list,

- can be elements of a tuple,

- can be passed to a function as a parameter and

- can be returned from a function as its result.[2]

Consider the following function definiton, taken from Haskell's base library, as a short example of Haskell code, highlighting some of the language features and syntax:

```
-- | 'takeWhile', applied to a predicate p and a list xs, returns the
-- longest prefix (possibly empty) of xs of elements that satisfy p:
--
-- > takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
-- > takeWhile (< 9) [1,2,3] == [1,2,3]
-- > takeWhile (< 0) [1,2,3] == []
--
```

These lines are comments, describing how the function works. Basic comments start with the characters   --   and reach until the end of the line.

$$takeWhile \quad :: (\alpha \rightarrow Bool) \rightarrow [\alpha] \rightarrow [\alpha]$$

This line contains the function's type signature, which tells us that the function we are defining takes another function of type $\alpha \rightarrow Bool$ as its first argument, a list of type $[\alpha]$ as it's second argument and returns a list of type $[\alpha]$. *takeWhile* is a *polymorphic* function, as α is a type variable that can be substituted with concrete types.

$$takeWhile \ \_ \ [\,] = [\,]$$

Here we define the function *takeWhile* for the case where the second argument, the list to be passed, is empty. This is done using *pattern matching*: The second argument to *takeWhile* has to fit the pattern $[\,]$, which denotes the empty list.

$$
\begin{aligned}
&takeWhile \ p \ (x : xs) \\
&\quad | \ p \ x \qquad = x : takeWhile \ p \ xs \\
&\quad | \ otherwise \ = [\,]
\end{aligned}
$$

This is the second case of the *takeWhile* function. Here the second argument is pattern matched to be of the type *(x:xs)*, which means that x is the first member of the list and xs is the rest of the list. The first argument, p, is a function of type $\alpha \rightarrow Bool$, which means a function that takes a value of type α and returns a Boolean. p is called with the first list member x as its argument. If the result is *true*, a new list $x : takeWhile \ p \ xs$ is returned. Otherwise an empty list is returned.

Defining a function that maps a function to the values of a list is very easy in Haskell:

$$
\begin{aligned}
&map \qquad\quad :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\
&map \ \_ \ [\,] \quad\;\; = [\,] \\
&map \ f \ (x : xs) = f \ x : map \ f \ xs
\end{aligned}
$$

An expression is said to have *side effects* when it interacts with the outside world, for example by reading from a file or modifying a global variable. Haskell is a *pure* programming language, which means side effects are prohibited.[1] The result of an expression depends solely on its inputs, which means the evaluation of a pure function returns the same result every time it is called with the same inputs, which leads us to the following property:

> "[. . .] [I]f we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value. Any other features of the sub-expression, such as its internal structure, the number and nature of its components, the order in which they are evaluated or the colour of the ink in which they are written, are irrelevant to the value of the main expression."[3, page 19]

This property is called *referential transparency*[2], and as a consequence of it expressions do not have to get evaluated in any specific order. Another favourable effect of referential transparency is that it allows clear equational reasoning.[5]

---

[1]There is of course a way to cause side effects in Haskell, the IO Monad.

[2]The term *referential transparency* has its origin in philosophy, where it means that replacing a part of a sentence with another term that refers to the same thing does not change the meaning of the sentence.[4]

2.2 Lazy Evaluation

Haskell uses referential transparency by deferring evaluations to the latest possible moment: When they are actually needed. This is called *lazy evaluation* and is the default evaluation strategy in Haskell.

Lazy evaluation automatically avoids unnecessary computations, because they will never be needed. Consider this example:

$$
\begin{aligned}
&squares && :: [Integer] \\
&squares && = map\ (\uparrow 2)\ [1 .. 10] \\
&smallSquares && :: [Integer] \\
&smallSquares && = takeWhile\ (\leqslant 20)\ squares
\end{aligned}
$$

The variable *squares* is defined to be the squares of the numbers 1 to 10, namely $[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$. *smallSquares* is defined to be the list of squares until the point where they grow greater than 20, namely $[1, 4, 9, 16]$.

Let us think about what has to get evaluated to calculate the value of smallSquares. As smallSquares stops when it encounters the first square number that is greater than 20 only the first five list members of squares get evaluated. The fifth member of squares gets evaluated to 25, which is checked to be greater than 20 and therefore no further values are evaluated.

Lazy evaluation allows carrying this to an extreme, by defining infinite data structures. Just as before they get only evaluated as far as they are needed. Consider the following example:

$$
\begin{aligned}
&squares && = map\ (\uparrow 2)\ [1 ..] \\
&smallSquares && = takeWhile\ (\leqslant 20)\ squares
\end{aligned}
$$

The only difference to the example above is that *squares* refers to the list of all squares now. Just as before only the first five list members of squares actually get evaluated.

Lazy evaluation is important for creating modular Haskell programs. A common approach is to split a function into two modules: A generator that constructs many possible solutions and a selector that checks whether those values actually satisfy the conditions for being a solution.[6]

2.3 Sharing

A consequence of Haskell being a pure programming language is that data is *immutable*. Values cannot be directly altered. Instead a copy containing the modified data gets created. Because data never gets altered there is no need to copy unmodified values. Instead they get *shared* between data structures. The above code illustrates this. As the *smallSquares* contain the first four values of *squares* they are not copied, but instead the old values are also used in smallSquares.

Sharing is a vital part of lazy evaluation. Because functions are first class entities in Haskell they can also be shared.

$$
\begin{aligned}
&doubleTotal = total + total \\
&\quad \textbf{where}\ total = sum\ [1 .. 100]
\end{aligned}
$$

Instead of evaluating the value for *total* every time it is needed it is only calculated once and when it is requested a second time the old value is returned directly. Thanks to referential transparency sharing does not influence the result of a compuation.

Sharing is useful when designing performant functional algorithms, but is rarely used consciously as sharing is unobservable in pure Haskell, which makes it hard to understand and preserve.[3][8]

2.4 Glasgow Haskell Compiler

The Glasgow Haskell Compiler (*GHC*)[9] is a compiler and interactive environment for Haskell, which is mainly developed by Simon Peyton Jones and Simon Marlow. GHC is available for several platforms and processor architectures, has extensive optimisations and is probably the most popular Haskell compiler. It is also the foundation of the *Haskell Platform*, which considers itself "the easiest way to get started with programming Haskell"[10]. This makes it the optimal target for ghc-vis, which is aimed at Haskell professionals and novices alike.

GHC's interactive environment, *GHCi* provides a command line interface, which gives immediate feedback for Haskell expressions entered. It can be used to interactively evaluate Haskell expressions and interpret programs.

2.5 GHC Heap

In GHC all dynamic objects reside in the heap in the form of a *closure*.[4] Closures are laid out as shown in Figure 2.1. Static objects, which already exist when the program is started, reside in the data section of compiled object code.

The header of a closure points to an info table which specifies

- the *layout* of the payload: How many words does it contain, how many of those are pointers,

- the *type* of this particular closure, for example a data constructor, a function or a thunk,

- the *SRT Bitmap*, a field used for garbage collection, which is of no further concern to us.

The relevant types of closures for this work are:[12]

Data Constructors provide basic data storage. This is for example used for Integers, Chars, the list constructor *cons* (:), the boolean values *True* and *False* or user defined constructors like *Leaf* and *Node* in the following example:

> **data** *Tree = Leaf*
>   *| Node Tree Int Tree*

---

[3]Actually the Haskell 2010 Language Report[7] does not mention sharing at all, but it is very practical. Because of referential transparency results will be the same whether sharing is utilized or not.

[4]A commonly used definition of a *closure* is a block of code together with its free variables. This is *not* the definition used in this thesis. Our definition, which is taken from [11], is more general and includes all dynamic objects.
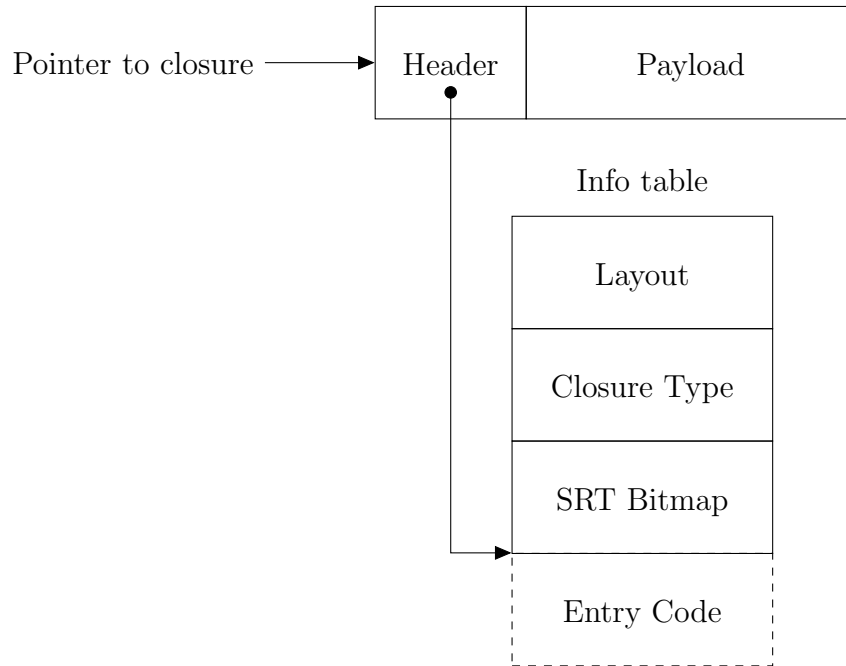
Figure 2.1: General layout of a closure

> $t :: Tree$
> $t = Node\ (Node\ Leaf\ 2\ Leaf)\ 1\ Leaf$

The layout of *t's* closure on the heap can be seen in Figure 2.2.

The closure of *t* contains a header, which tells us that there are 3 pointers and 0 nonpointers in the closure, that we are dealing with a closure of Type CONSTR and that the constructor is called *main::Interactive.Node*.

Function Closures represent Haskell functions, without any arguments, but containing pointers to the function's free variables.

Partial Applications represent functions with some arguments applied to them, but less than they take in total.

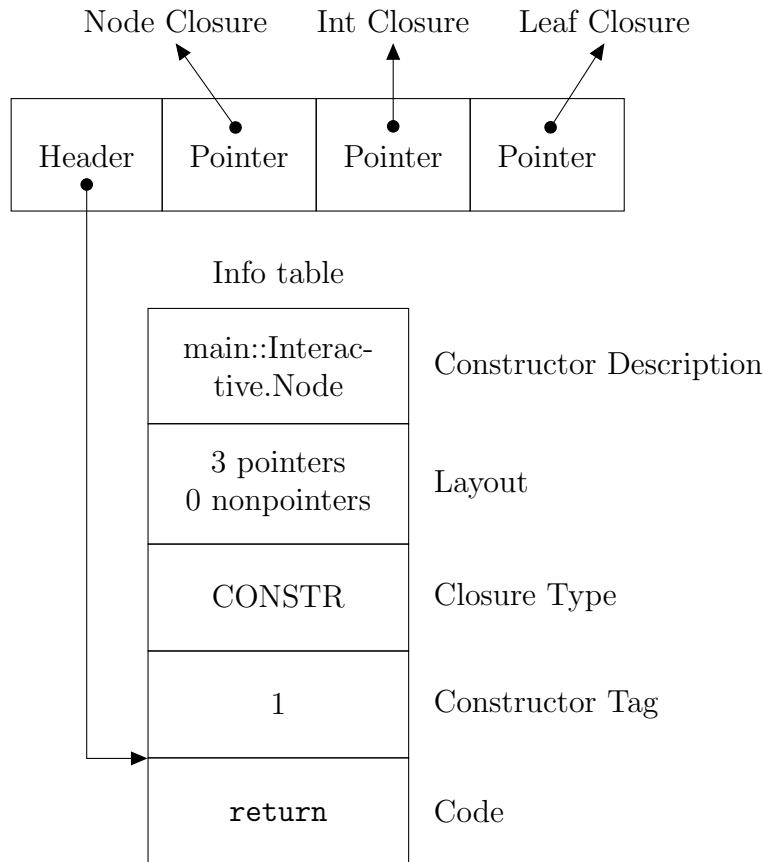**Thunks** represent suspended computations, which can be evaluated further.

When a thunk gets evaluated it is replaced with the result of its evaluation.

Generic Applications are like partial applications, but all arguments are given. Therefore they are *thunks* and can be evaluated. Generic Applications are mainly used in interpreted code.

Selector Thunks perform simple selections. They are necessary in order to prevent space leaks, that would happen when a part of a data structure is selected, but another part of it is not needed anymore.[13]

Indirections simply point to other closures. They are introduced when a thunk gets evaluated to point to its value and are necessary because it might not be possible to simply replace a thunk with its result when the resulting closure is too big.

Byte Code Objects (BCOs) contain byte code that can be executed by the interpreter.

Node Closure   Int Closure   Leaf Closure

Header	Pointer	Pointer	Pointer

Info table

main::Interac- tive.Node	Constructor Description
3 pointers 0 nonpointers	Layout
CONSTR	Closure Type
1	Constructor Tag
return	Code

Figure 2.2: Layout of Tree t's closure on the heap

Blackhole Closures appear when an evaluation is happening and later get replaced by the result. They are used to avoid space leaks that would happen when the closure which is getting evaluated points to closures that are not actually needed anymore. Blackhole closures prevent the garbage collector from discovering other pointers in the closure, which can then be made available for recycling.[14]

The library ghc-heap-view[15] by Joachim Breitner makes it possible to investigate closures on the GHC heap directly in Haskell.

Closures may be moved around or evaluated at any moment by the garbage collector, which only uses the information available on the heap. GHC uses a two-space stop-and-copy collector, that divides the heap memory into two spaces. To perform a garbage collection all currently used closures are copied from one space to the the other. To realize which closures to copy the garbage collector follows the pointers of closures.[11]

When a closure is evaluated it gets updated with its *weak-head normal form* (WHNF), which means the outermost part of the expression is evaluated. Note that this does not imply that the expression is in *normal form*, for which the expression would have to be fully evaluated. Data constructors, function closures and partial applications are always in WHNF, because when they are the top level closure they cannot be further evaluated. Thunks, selector thunks and generic applications are not necessarily in WHNF and therefore it is only those that can get evaluated.

There are two typical evaluation models that could be used in GHC:

Push/Enter : The caller pushes the function arguments on the stack and calls the function. The function itself figures out how many arguments it has been passed.

Eval/Apply : The caller examines the function closure and calls the function with the right number of arguments. Three cases may occur:

1. If there are not enough arguments the function gets turned into a partial application.

2. If the right number of arguments are given the function is simply called.

3. If there are too many arguments, the function is applied to the arguments it can take and then the resulting function is examined in the same way.[16]

GHC uses the Eval/Apply evaluation model. We will look at GHC's behavior concerning closures on the heap later, using the developed tool, ghc-vis.

2.6 Related Work

There have been some projects with regards to visualizing data structures in Haskell. Some are to be used in a similar way as ghc-vis, in a live environment, whereas others allow debugging code after it has been run.

2.6.1 GHCi Debugger

GHCi comes with its own debugger, which has been developed in 2006 under the Google Summer of Code intiative by José Iborra.[17] The debugger provides the *:print* command. It inspects data structures at runtime without forcing evaluation of them.

```
$ ghci
GHCi, version 7.4.2: http://www.haskell.org/ghc/  :? for help
ghci> let a = [1..3]
ghci> :print a
a = (_t1::[Integer])
```

We can see that *a* currently is completely unevaluated, it simply consists of a thunk.

```
ghci> head a
1
ghci> :print a
a = 1 : (_t2::[Integer])
```

Now that we have peeked inside of *a* it has been partially evaluated.

```
ghci> head $ tail a
2
ghci> :print a
a = 1 : 2 : (_t3::[Integer])
ghci> a
[1,2,3]
ghci> :print a
a = [1,2,3]
```

We see that *:print* enables us to see lazy evaluation in action. Unfortunately sharing cannot be seen as easily:

```
ghci> let b = a ++ a
ghci> head b
1
ghci> :print b
b = 1 : (_t3::[Integer])
ghci> b
[1,2,3,1,2,3]
ghci> :print b
b = [1,2,3,1,2,3]
```

Even though the numbers in *b* are shared with *a* and even within *b* itself *:print* does not tell us about this in any way.

2.6.2 Vacuum

Another tool that has been developed to inspect data structures in Haskell is vacuum.[18] It can tell us how values are shared within b:

```
ghci> System.Vacuum.Cairo.view b
```
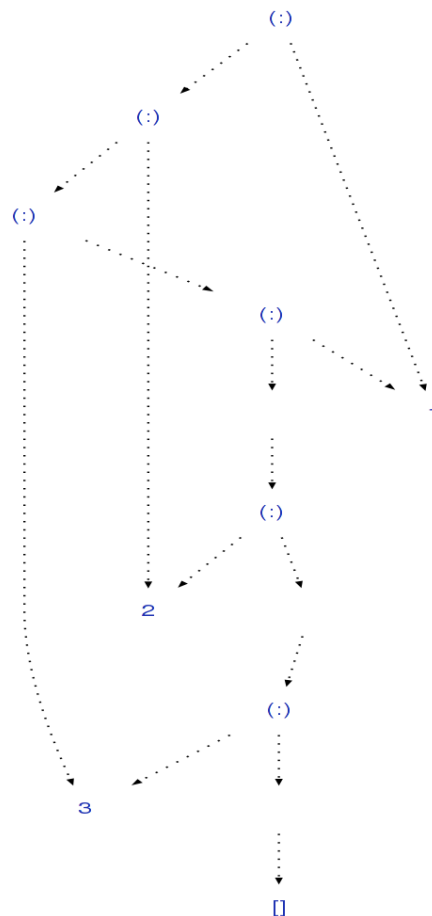
Figure 2.3: Visualization of a list using vacuum

This causes the output seen in Figure 2.3 to appear on the screen. From this graph it is clear how the members of the list are shared instead of being copied. Unfortunately vacuum-cairo evaluates data structures fully before showing them. To also inspect lazy evaluation using vacuum we can do the following:

```
ghci> let a = "foo"
ghci> let b = a ++ a
ghci> head b
'f'
ghci> GHC.Vacuum.GraphViz.graphToDotFile "vacuum2"
          Data.GraphViz.Commands.Png $ GHC.Vacuum.nameGraph
          (GHC.Vacuum.vacuumLazy (a,b))
```
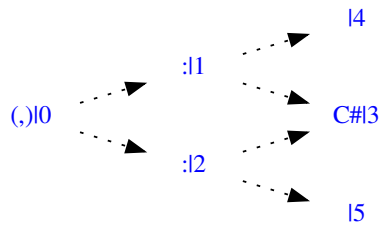

Figure 2.4: Lazy visualization using vacuum

Which leads to the output in Figure 2.4. We can see that the value *C#l3* is referenced both in *a* and *b*. That's the `'f'` which was evaluated by calling *head b*. What can not be seen is that the *l5* thunk references *a* and has another value shared with *a*.

Later, in Subsection 4.1, we will come back to these examples and compare ghc-vis with vacuum and :print to illustrate how ghc-vis solves the issues that were just shown.

2.6.3 Hood / GHood

Hood[19] observes Haskell data structures in GHC compiled programs. *GHood*[20] creates an animation of the evaluation of data structures which can be watched in a Java program after execution. It is possible to step through the evaluation and observe how the data structure changes, including the evaluation of internal thunks. See Figure 2.5 for an example.

Sharing can not be observed in GHood and as an animation first has to be created before one can visualize it, GHood can not be used in GHCi to visualize live data structures.

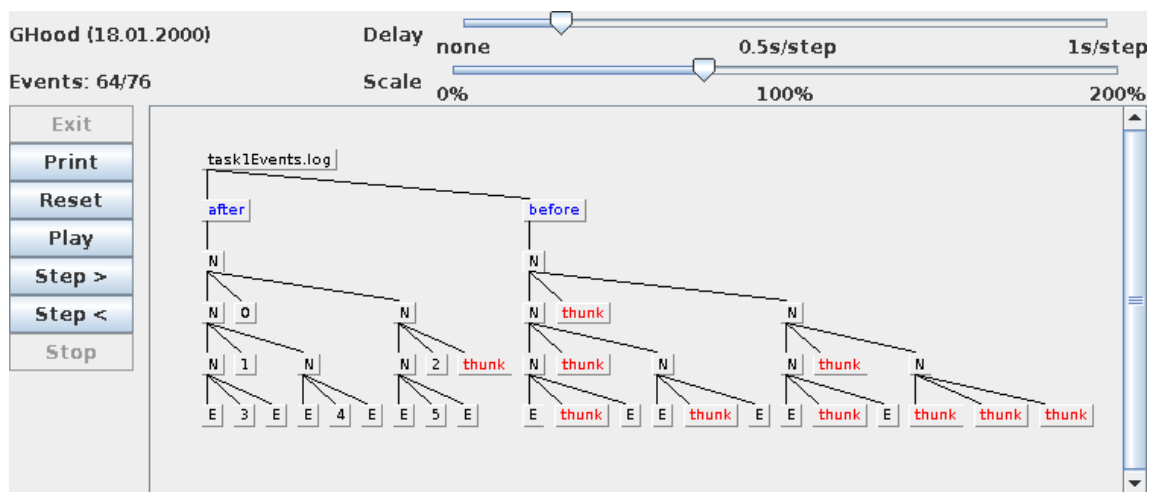Figure 2.5: Visualization of data structures using GHood

CHAPTER 3

ghc-vis

The main part of this thesis was the design and implementation of *ghc-vis*, a tool to visualize lazy evaluation and sharing in Haskell. In this chapter the design, implementation and challenges of *ghc-vis* will be discussed.

3.1 Design

In order to be useful ghc-vis has to provide two functions:

- Variables defined in GHCi can be supplied to ghc-vis for visualization.

- When a closure is clicked in ghc-vis' visualization window GHCi evaluates it.

There are multiple possibilities for the high level integration of ghc-vis into GHCi, which could achieve those functions:

1. Modify the source code of GHCi. This allows for easy bidirectional communication between ghc-vis and GHCi.

2. Supply commands to GHCi on stdin.

3. Run a thread inside of GHCi.

Because of its simplicity the third approach was chosen. ghc-vis is run directly inside of GHCi in a seperate thread. The graphical interface is created using the Haskell library bindings to GTK, Cairo and Pango.[21]

The interface to ghc-vis from GHCi is defined by GHCi commands that map to simple Haskell functions. The main commands are listed and explained in Table 3.1.

We will now take a quick look at the usage of ghc-vis to illustrate how it works, before we continue discussing it.

3.1.1 Demonstration

We begin by running ghci and starting ghc-vis inside of it. User input is prepended by `ghci>`. All other lines are output by GHCi:

Command	Description
`:vis`	Run the visualization window
`:view x`	Add the Haskell expression x to the visualization window
`:eval t`	Force evaluation to WHNF of the thunk named t
`:switch`	Switch between the visualization type (linear, graph)
`:update`	Update the visualization window, in case an expression has changed
`:clear`	Remove all visualized expressions
`:export file`	Export the current visualization to a file; *SVG*, *PDF*, *PS* and *PNG* supported

Table 3.1: Main commands of ghc-vis

```
$ ghci
GHCi, version 7.4.2: http://www.haskell.org/ghc/  :? for help
ghci> :vis
[Package loading output cut]
```

The visualization window, which currently does not show anything. We can add expressions to the visualization:

```
ghci> let a = [1..3]
ghci> :view a
ghci> let b = cycle a
ghci> :view b
ghci> :view "foo" ++ "bar"
```
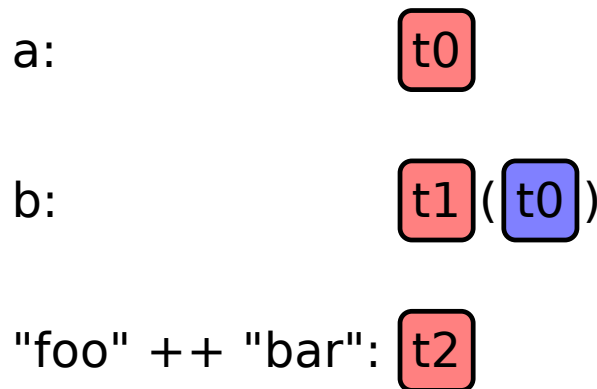

Figure 3.1: Demonstration of ghc-vis

The output in Figure 3.1 appears in the visualization window. What we're seeing now is the *linear view*. It shows values in a linear form, similar to *:print*, but with graphical highlighting and deeper inspections. See Table 3.2 for the mapping of linear view objects to actual heap closures.

In Figure 3.1 we can see that *a* is referenced by *b*, because the thunk *t0* is linked in *b*. Hovering an object highlights all other objects in the visualization window which link to this object or are linked by it. In order to force evaluation of an object in the visualization we can click on it or enter the following in GHCi:

```
ghci> :eval t1
```

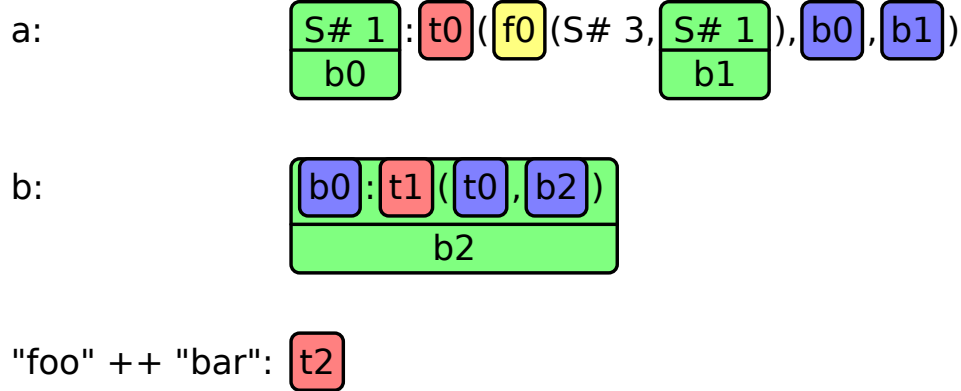| Name	Object	Meaning
Unnamed	S# 1	Represents Data Constructors
Thunk	t0	Represents Thunks, Generic Applications
Function	f0	Represents Functions, Partial Applications
Named	S# 1 / b0	A named object, so that it can be linked elsewhere
Link	b0	Links to an object which is already visualized

Table 3.2: Linear view objects

a: S# 1 / b0 : t0 (f0 (S# 3, S# 1 / b1), b0 , b1)

b: b0 : t1 (t0 , b2) / b2

"foo" ++ "bar": t2

Figure 3.2: Demonstration of ghc-vis

We see in Figure 3.2 that the first member of *b* has been evaluated. It's called *b0* and is a reference to the value that's also referenced as the head of *a*, as they share the same name. *S# 1* stands for Integer data constructor for small Integers, which is called *S#* in GHC and its argument *1*.

The linear view may not always be ideal, especially for bigger data structures. The heap can be thought of as a graph with closures as vertices and their pointers to each other as directed edges. This is represented in the *graph view* of ghc-vis, which we can switch to like this:

```
ghci> :switch
```

The output in Figure 3.3 appears on the screen.

When an object is updated by accessing it *:update* has to be called to refresh the visualization window. Alternatively any object can be clicked to perform an update.

```
ghci> a !! 2
3
ghci> :update
```

This causes the visualization window to be updated to the view in Figure 3.4.

3.1.2 Implementation

ghc-vis has been implemented to work with the Glasgow Haskell Compiler version 7.4.2. More recent versions of the GHC will likely require adjustments in ghc-vis because of the interaction with GHC internals which are likely to change in the future.
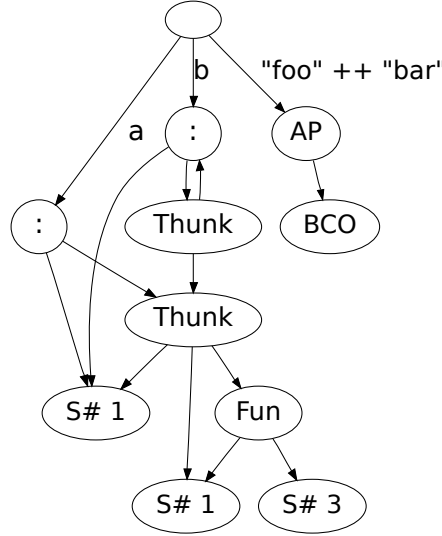
Figure 3.3: Demonstration of ghc-vis

In order to analyze the representation of data structures on the heap we represent
closures on the heap as a *HeapEntry* tuple and the part of the heap we are interested
in as a *HeapMap*:

$$\textbf{data } Box \qquad\quad = Box\ Any$$
$$\textbf{type } HeapEntry = (Maybe\ String, Closure)$$
$$\textbf{type } HeapMap \quad = [(Box, HeapEntry)]$$

The first element of a *HeapEntry* identifies the entry on the heap. It is used to denote
top-level closures and closures we have already visited in order to prevent infinite
recursions. The second element is the closure, containing the parsed information of
the raw heap closure we actually are interested in.

The *HeapMap* is a simple list, identifying raw Haskell expressions with their corre-
sponding closure information. We want to use the HeapMap to look up the corre-
sponding *HeapEntries* of *Box*es, which is only possible in $O(n)$ in a list of length
n:

$$lookup \qquad\quad :: Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$$
$$lookup\ \_\ \ \ [\,] \ = Nothing$$
$$lookup\ key\ ((x, y) : xys)$$
$$\quad | \ key \equiv x \quad = Just\ y$$
$$\quad | \ otherwise = lookup\ key\ xys$$

Let us investigate why we are using a *List* and equality (type class *Eq*) based com-
parisons instead of a totally ordered (type class *Ord*) *Map*, which performs *lookup*s
in $O(logn)$ on a balanced binary tree:

$$\textbf{data } Map\ k\ a = Tip$$
$$\qquad\qquad\qquad | \ \ Bin\ \{\text{-\# UNPACK \#-}\}\ \ !\,Size\,!\,k\ a\,!\,(Map\ k\ a)\,!\,(Map\ k\ a)$$
$$lookup \qquad :: Ord\ k \Rightarrow k \rightarrow Map\ k\ a \rightarrow Maybe\ a$$
$$lookup\ k\ t = \textbf{case } t\ \textbf{of}$$
$$\quad Tip \rightarrow Nothing$$
$$\quad Bin\ \_\ kx\ x\ l\ r$$
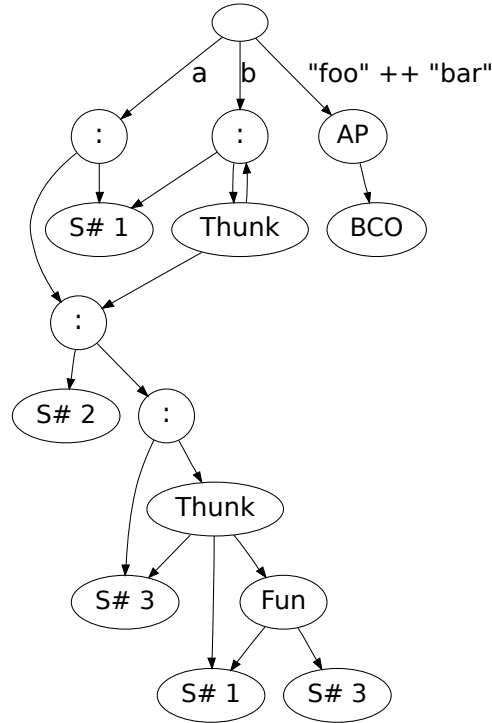
Figure 3.4: Demonstration of ghc-vis

$$\rightarrow \textbf{case } compare \; k \; kx \textbf{ of}$$
$$LT \;\rightarrow lookup \; k \; l$$
$$GT \rightarrow lookup \; k \; r$$
$$EQ \rightarrow Just \; x$$

As the type signature of *Map*'s lookup tells us, we need to provide an instance of *Ord* for the *Box* data type, thereby defining a total order of *Box*es.

```
ghci> let x = asBox 3 :: Box
ghci> let y = asBox 4 :: Box
ghci> x
0x00007f8ee7f796b8
ghci> y
0x00007f8ee7f5a6b8
ghci> System.Mem.performGC
ghci> x
0x00007f8ed0ffb090
ghci> y
0x00007f8ed105eaa0
```

First we defined two new *Box*es. Then we print their actual memory locations. After we perform a garbage collection (*performGC*) we print their memory locations again. The garbage collector run switched the values' order in the memory around. As we have seen in Section 2.5 the garbage collector is allowed to do that at any time. Therefore comparing the values' memory locations does not help us in establishing any consistent order between them. There is no other information about the stored value available except its memory location.

StableName[22] is a data type that is supposed to provide stable ordering of pointers

according to the paper in which they are introduced, but unfortunately the current implementation of them does not instantiate the *Ord* type class. If the naive list implementation turned out as a performance bottle neck a hash table with Stable-Names could be used instead of it.

Creating the *HeapMap* at first seems like the problem already solved by *reify-Graph*[23], a function to observe sharing in Haskell. It works by instantiating the *MuRef* type class for the data type one wants to observe, thereby defining how to traverse the data structure. The vertex and edge labels in the graph created are also user defined.

A closer look reveals that we have to deal with a number of problems that do not exist when using *reifyGraph*, which mostly stem from these differences:

- We want more than observing sharing, namely keep unevaluated data structures unevaluated to visualize lazy evaluation.

- We operate on a lower level, looking inside the heap.

- We do not want the user to supply any information about what he is going to visualize.

Some of the resulting problems are described in Section 3.3.

Our function for creating the *HeapMap* from the actual heap, called *walkHeap*, works as follows:

1. Add a dummy node to the HeapMap that links to all expressions to be visualized

2. Add the expressions to the HeapMap

3. Start a recursive heap walk along the pointers of the heap representations of the supplied expressions:

   a) If we have already added this closure to the HeapMap, return

   b) Add the closure to the HeapMap

   c) Visit all relevant pointers the closure has

3.2 Views

3.2.1 Linear View

The *linear view* is similar to *:print*. In order to create it a linear list of *VisObject*s is generated from the HeapMap.

```
data VisObject = Unnamed String
             | Named    String [VisObject]
             | Link     String
             | Thunk    String
             | Function String
    deriving Eq
```

VisObject		Visualization
Unnamed `"S# 1"`		S# 1
Named	`"b0"` [*Unnamed* `"S# 1"`]	S# 1 / b0
Link	`"b0"`	b0
Thunk	`"t0"`	t0
Function	`"f0"`	f0

Table 3.3: VisObjects

The VisObjects directly correspond to the objects drawn in the visualization, as can be seen in Table 3.3.

The list of VisObjects gets created in this way:

1. Create a HeapMap

2. Recursively walk the HeapMap for every expression to be visualized:

   a) If the current closure already has a name, return a link to the name

   b) Create a corresponding *VisObject* for the closure

   c) Append the result of recursively walking the pointers of the closure to the VisObject we already have

   d) If the VisObject has multiple pointers pointing to it and is not already named, create a Named VisObject to surround the entire list of VisObjects

   e) Return the resulting list of VisObjects

3.2.2 Graph View

The graph view is similar to *vacuum-cairo*. If ghc-vis used a non-interactive graph view the process of creating the graph would look like this:

1. Convert HeapMap to DotGraph

2. Use graphviz[24] to convert DotGraph to SVG

3. Draw resulting SVG

But in order to support interactivity we had to take another course:

1. Convert HeapMap to DotGraph

2. Use graphviz to convert DotGraph to XDotGraph

3. Parse drawing operations of XDotGraph

4. Execute drawing operations, allowing for manipulation by user input (hovering, clicking)

Steps 3 and 4 have been externalized into a seperate library called *xdot*. It uses the Haskell graphviz library bindings[25] to convert a regular graph to a DotGraph, supply it to the `dot` program and finally parse the resulting XDot file in again.

As the Haskell graphviz library does not support parsing XDot operations[26], this functionality was added to xdot. The relevant data type used for this is *Operation*, which translates all XDot operations into a Haskell data type:

```
data Operation =
  Ellipse  { xy         :: Point
           , w          :: Double
           , h          :: Double
           , filled     :: Bool
           } |
  Polygon { points      :: [Point]
           , filled     :: Bool
           } |
  Polyline { points     :: [Point]
           } |
  BSpline { points      :: [Point]
           , filled     :: Bool
           } |
  Text     { baseline   :: Point
           , alignment  :: Alignment
           , width      :: Double
           , text       :: String
           } |
  Color    { rgba       :: (Double, Double, Double, Double)
           , filled     :: Bool
           } |
  Font     { size       :: Double
           , name       :: String
           } |
  Style    { style      :: String
           } |
  Image    { xy         :: Point
           , w          :: Double
           , h          :: Double
           , name       :: String
           }
  deriving Show
```

All of the operations except for *Image*, which integrates an external image into the graph, but was not needed for ghc-vis, are supported for viewing inside a GTK window using Cairo and Pango.[21]

3.3 Problems and Solutions

3.3.1 Garbage Collector

When we have stored the layout of the heap into our representation of it, the
HeapMap, the garbage collector may cause the *HeapMap* to become corrupted. Let
us look at an example, for reasons of brevity some members of the heap map are
omitted:

```
ghci> let x = [1..2]
ghci> hm <- walkHeap [(asBox x,"x")]
ghci> hm
[(0x00007f732dac52f0,(Nothing,BCOClosure {info = StgInfotable
  {ptrs = 4, nptrs = 0, tipe = BCO, srtlen = 0},
  instrs = 0x00007f732dac55a0, literals = 0x00007f732dac55e8,
  bcoptrs = 0x00007f732dac5618, arity = 0, size = 6, bitmap = 0}))
,(0x00007f732dac5380,(Just "x",APClosure {info = StgInfotable
  {ptrs = 0, nptrs = 0, tipe = AP, srtlen = 0}, arity = 116,
  n_args = 0, fun = 0x00007f732dac52f0, payload = []}))
,...]
ghci> x
[1,2]
ghci> hm2 <- walkHeap [(asBox x, "x")]
ghci> hm2
[...
,(0x00007f732e8dbc40/2,(Nothing,ConsClosure {info = StgInfotable
  {ptrs = 2, nptrs = 0, tipe = CONSTR_2_0, srtlen = 1},
  ptrArgs = [0x00007f732e8dbce0,0x00007f732e8dbcc8/2],
  dataArgs = [], pkg = "ghc-prim", modl = "GHC.Types", name = ":"}))
,(0x00007f732dac5380,(Just "x",BlackholeClosure {info = StgInfotable
  {ptrs = 1, nptrs = 0, tipe = BLACKHOLE, srtlen = 0},
  indirectee = 0x00007f732e8dbc40/2}))
,...]
```

We see that the APClosure at memory address `0x00007f732dac5380` was replaced
with a BlackholeClosure which points to the ConsClosure that resulted from evalu-
ating the generic application. After waiting for a few seconds another look at *hm2*
reveals a surprising result:

```
ghci> hm2
[...
,(0x00007f732dcad4b0/2,(Nothing,ConsClosure {info = StgInfotable
  {ptrs = 2, nptrs = 0, tipe = CONSTR_2_0, srtlen = 1},
  ptrArgs = [0x00007f732dcad890,0x00007f732dcad878/2],
  dataArgs = [], pkg = "ghc-prim", modl = "GHC.Types", name = ":"}))
,(0x00007f732dcad4b0/2,(Just "x",BlackholeClosure {info = StgInfotable
  {ptrs = 1, nptrs = 0, tipe = BLACKHOLE, srtlen = 0},
  indirectee = 0x00007f732dcad4b0/2}))
,...]
```

Suddenly the ConsClosure and the BlackholeClosure have the same memory location:
`0x00007f732dcad4b0/2`. How did this occur?

The BlackholeClosure was only created while the evaluation was happening to prevent space leaks. When the garbage collector ran in the few seconds we waited it replaced the BlackholeClosure, which has become useless now that the evaluation has finished, with the resulting ConsClosure. That means the BlackholeClosure isn't actually on the heap anymore, but our HeapMap wasn't updated.

To deal with problems introduced by the garbage collector running in the time between creating a HeapMap and visualizing it, a garbage collection is performed right before creating a HeapMap.

3.3.2 Changing Pointers

When the garbage collector runs not after we have created a HeapMap, but *during* the creation, even bigger problems may appear. The same problems can occur when a user evaluates data structures while the HeapMap gets created.

An example for this problem works as follows:

1. We start parsing a closure from the heap into a Haskell data structure

2. Closure gets evaluated by the garbage collector or a user

3. The closure on the heap gets replaced by another closure

4. We continue parsing the closure, now reading wrong values and pointers because we assume the closure to be of the old type

In order to deal with problems of this kind failures in *walkHeap* are not fatal, but instead trigger a rerun.

3.3.3 Byte Code Objects

Byte Code Objects (BCOs) on the heap may link to rather complex data structures, as in Figure 3.5a, which we are not interested in:

- List of instructions to be executed

- List of pointers that are used in the instructions (often including type class information)

Therefore we cannot just show all information of the heap. Simply hiding them doesn't work either because they may help in understanding sharing, as we can see in Figure 3.5b, where *sum x* is referencing *x* through its BCO. Instead we traverse the BCO's pointers to see whether they point to an object we would visualize anyway, and only connect it in that case, as can be seen in Figure 3.5c.

3.3.4 Loading and Reloading Files

When a file gets `:load`ed or `:reload`ed in GHCi all defined variables get removed. It also causes the global communication channel ghc-vis uses to be reinitialized. The thread listening to it would not receive any further commands. Therefore the listening thread has a timeout that handles this by resetting our global variables and listening to the new signal channel:
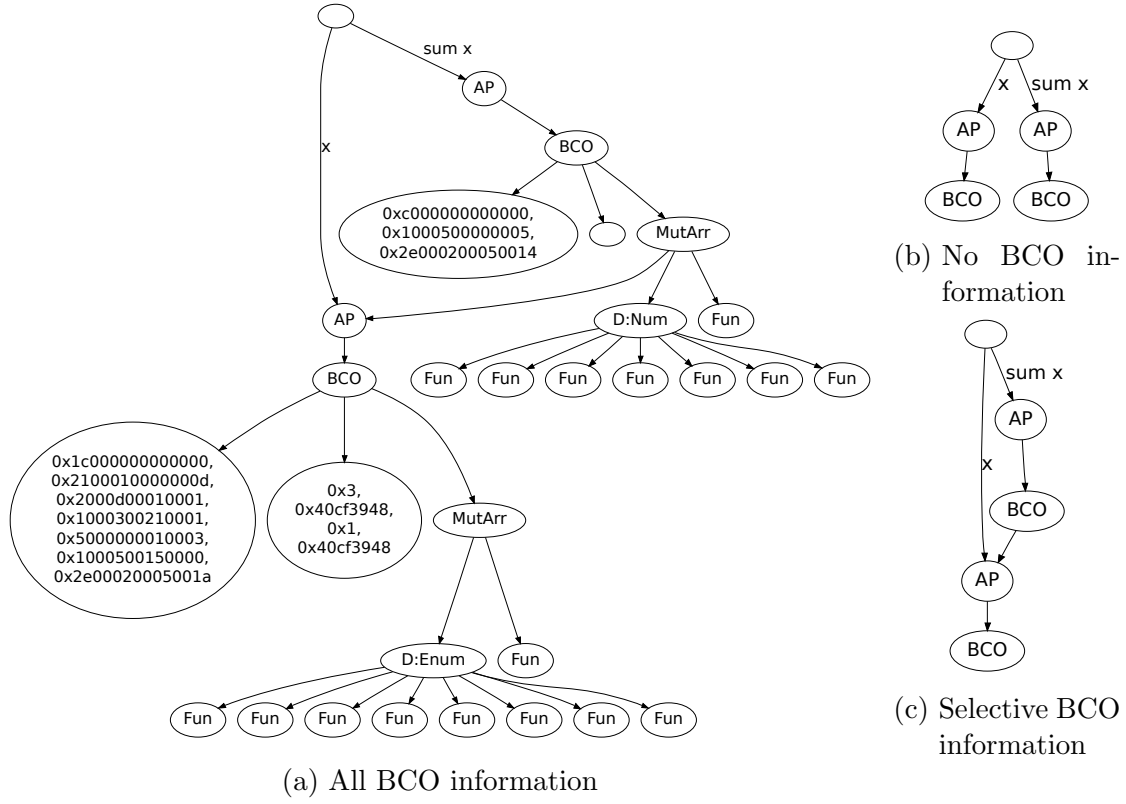
(a) All BCO information

(b) No BCO information

(c) Selective BCO information

Figure 3.5: Sharing through BCO pointers

$react\ canvas\ window = \mathbf{do}$
$\quad mbSignal \leftarrow timeout\ signalTimeout\ (takeMVar\ visSignal)$
$\quad \mathbf{case}\ mbSignal\ \mathbf{of}$
$\qquad Nothing \quad \rightarrow \mathbf{do}$
$\qquad\quad running \leftarrow readMVar\ visRunning$
$\qquad\quad \mathbf{if}\ running$
$\qquad\qquad \mathbf{then}\ react\ canvas\ window$
$\qquad\qquad \mathbf{else\ do} \quad \text{-- :r caused visRunning to be reset}$
$\qquad\qquad\qquad swapMVar\ visRunning\ True$
$\qquad\qquad\qquad timeout\ signalTimeout\ (putMVar\ visSignal\ UpdateSignal)$
$\qquad\qquad\qquad react\ canvas\ window$
$\qquad Just\ signal \rightarrow \mathbf{do}$
$\qquad\quad \text{-- Normal signal handling}$

CHAPTER 4

Evaluation

4.1 Comparison

Let us return to the example we used in Section 2.6.1 to illustrate the shortcomings of *:print* and *vacuum* in visualizing lazy evaluation and sharing and compare those results to *ghc-vis*:

```
ghci> let a = "foo"
ghci> let b = a ++ a
ghci> head b
'f'
```

```
ghci> :print a
a = 'f' : (_t1::[Char])
ghci> :print b
b = 'f' : (_t2::[Char])
```


(a) :print
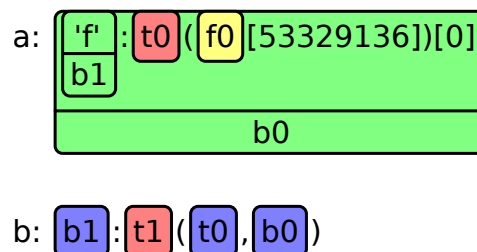
(b) ghc-vis: Linear view

Figure 4.1: Comparing visualization of lists in *:print* and the linear view of *ghc-vis*

In Figure 4.1 we see the output of *:print* and the linear view of *ghc-vis*. The linear view contains the following information which is missing from *:print*:

- The first character of *a* and *b*, `'f'`, is shared.

- The thunk for the remaining list in *b* references *a* and a function that is also refrenced in *a*.

:print can not be used to analyze cyclic data structures, like *repeat* 1. It keeps recursing on the data structure to build the textual representation of it until it runs out of memory. They pose no problem for ghc-vis, as can be seen in Figure 4.2, which shows how ghc-vis visualizes *repeat* 1.

In Figure 4.3 we see the output of *vacuum* and the graph view of *ghc-vis*. While the sharing of `'f'` is visible in *vacuum* as well, it is missing the referencing of *a* and a
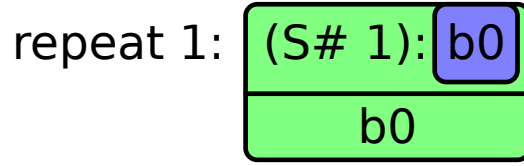
Figure 4.2: Visualizing *repeat* 1 using ghc-vis

(a) vacuum   (b) ghc-vis:
                                  Graph view
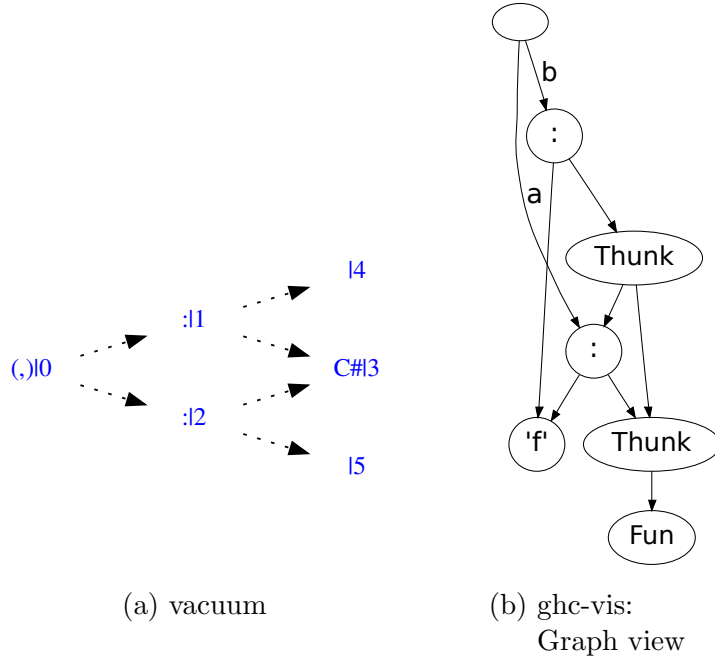
Figure 4.3: Comparing visualization of lists in *vacuum* and the graph view of *ghc-vis*

function inside of *a* in *b*'s thunk. The reason for this is that vacuum does not inspect the insides of thunks. Vacuum does not support any interactivity, while ghc-vis can evaluate thunks interactively and more data structures can easily be added to its visualization.

4.2 Visualizations

As the main focus of ghc-vis is visualizing Haskell data structures and furthering the understanding of lazy evaluation and sharing we will now look at examples using ghc-vis and evaluate what we can learn from them. No new results will be presented in this section, instead we look at well known phenomenons.

4.2.1 Compiled and Interpreted Code

Compiled and interpreted code may lead to different representations on the heap. Consider the following example:

$$bigSquares :: [Integer]$$
$$bigSquares = filter\ (>10)\ squares$$
$$squares\ \ \ \ :: [Integer]$$
$$squares\ \ \ \ = map\ (\uparrow 2)\ [1\mathbin{..}]$$

By default all code is interpreted in GHCi. Using `:set -fobject-code` GHCi can

(a) Compiled code (-fobject-code)   (b) Interpreted code (-fbyte-code)
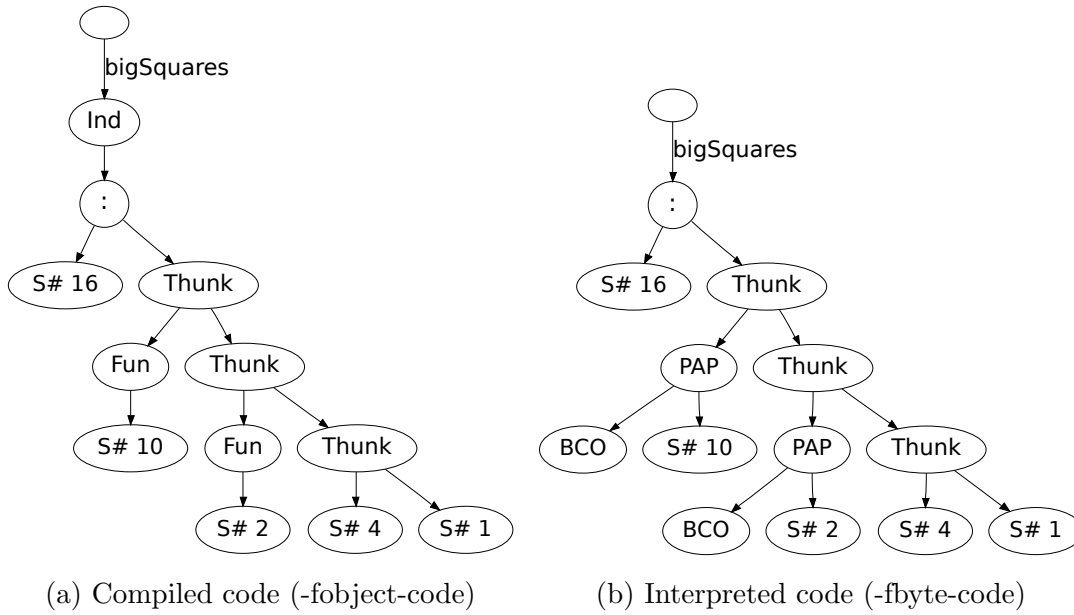
Figure 4.4: Squares

be instructed to compile :**load**ed files instead. In Figure 4.4 we see that the compiled version created special functions, whereas the interpreted version uses more general functions, to which some arguments were already supplied.

In the following examples the visualization is mostly the same for compiled and interpreted code. A notable exception is the Sieve of Eratosthenes in Subsection 4.2.4 which is visualized in compiled form because the type class information in the interpreted data structures causes the graph to be significantly bigger.

Another notable difference between compiled and interpreted code is the representation of data constructors with no arguments:

data *Maybe a = Just a | Nothing*

The data constructor for *Just* looks the same in compiled and interpreted code. But data constructors without arguments, like *Nothing* in this example, get a bogus argument in interpreted code, while in compiled code they have no such argument.

The reason for this is that all closures on the heap have a minimum size of two words, so that they can be replaced by pointers that temporarily get created during garbage collection.[12] All interpreted data constructors are allocated on the heap, and therefore those without arguments are filled with a dummy argument to reach the minimum size.

Compiled data constructors are not allocated on the heap, but instead reside in the data section of the compiled object code. They can not be replaced and therefore they can be smaller than the minimum size for heap objects and no bogus argument is added.

4.2.2 Sharing

This simple example shows how sharing may be used:

$complexOperation \quad :: Double \rightarrow Double$
$complexOperation \ x = x + 1$

$f \qquad :: Double \rightarrow Int \rightarrow [Double]$
$f \ x \ 0 = []$
$f \ x \ y = complexOperation \ x : f \ x \ (y - 1)$

$g \qquad :: Int \rightarrow [Double]$
$g \ y \quad = f \ 235 \ y$

$f' \qquad :: Double \rightarrow Int \rightarrow [Double]$
$f' \ x \quad = go \ (complexOperation \ x) \ y$
$\quad \textbf{where} \ go \ x' \ 0 = []$
$\qquad \qquad go \ x' \ y = x' : go \ x' \ (y - 1)$

$g' \qquad :: Int \rightarrow [Double]$
$g' \ y \quad = f' \ 235 \ y$

Whereas g does not use sharing g' only calculates the *complexOperation* once and
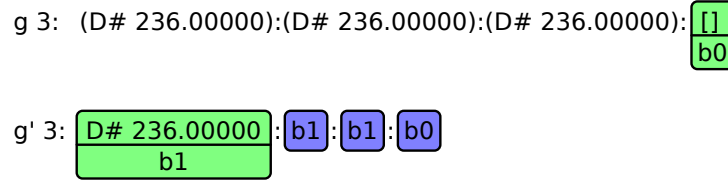
(a) Linear view
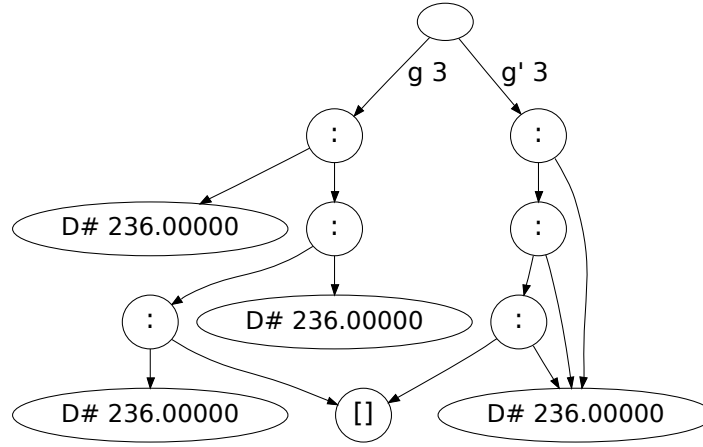
(b) Graph view

Figure 4.5: Sharing

then shares the resulting value across the entire list. This can be seen in Figure 4.5: *g 3* consists of three seperate list members, while *g' 3* only uses a single one three times.

Expression	Time spent	Memory used
$sum \ (g \ 1000000)$	5.39 s	1367.92 MiB
$sum \ (g' \ 1000000)$	1.55 s	429.46 MiB

Table 4.1: Effect of sharing on performance

Table 4.1 shows how time and memory performance are influenced in this case by the

use of sharing. Note that for the performance measurements the following definition of *complexOperation* was used instead:

$$complexOperation\ x = x ** 9 - sqrt\ (x ** 5) + sqrt\ (sqrt\ (x ** 3))$$

A more complex operation increases the effect of sharing.

4.2.3 Infinite Data Structures

We already looked at two definitions of the infinite list of ones in the Introduction. Now we look at the following definition for the infinite list of Fibonacci numbers:

$$fib\ ::\ [\,Integer\,]$$
$$fib = 1 : 1 : [\,a + b \mid (a, b) \leftarrow zip\ fib\ (tail\ fib)\,]$$

The state of *fib* after evaluating the first three members can be seen in Figure 4.6.

(a) Linear view   (b) Graph view
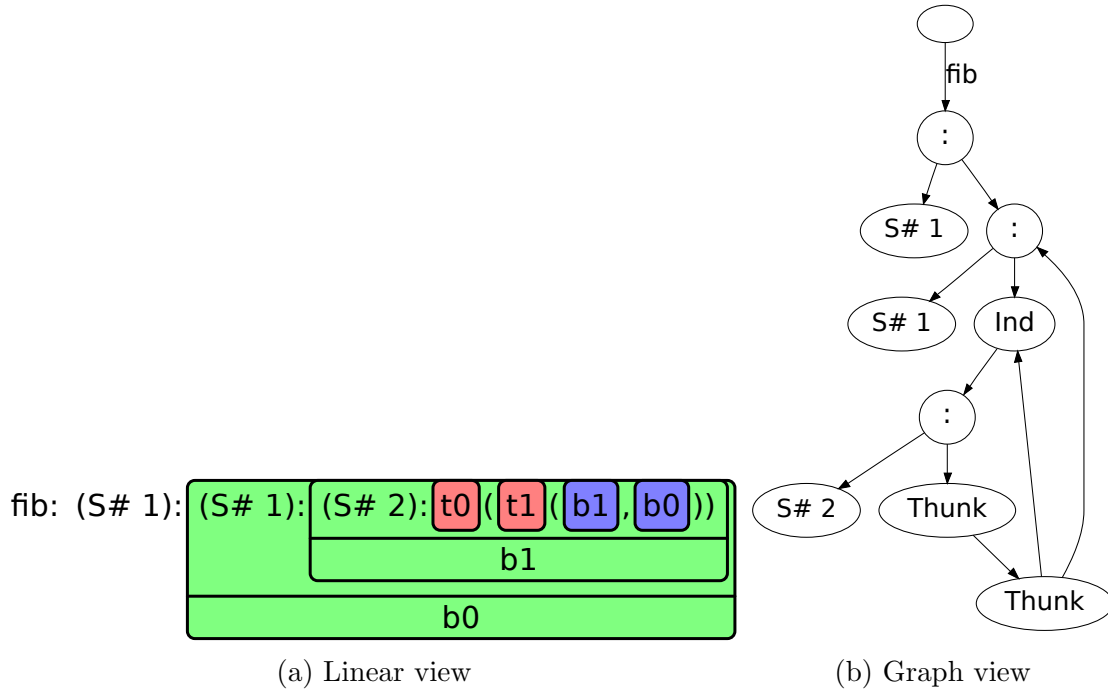
Figure 4.6: Fibonacci numbers

It becomes apparent how the function that constructs member n references members $n - 1$ and $n - 2$.

4.2.4 Sieve of Eratosthenes

The sieve of Eratosthenes is a simple algorithm to calculate all prime numbers. It is often used as an example of lazy evaluation.

$$primes \qquad ::\ [\,Integer\,]$$
$$primes \qquad = sieve\ [\,2\,..\,]$$
$$sieve \qquad ::\ [\,Integer\,] \rightarrow [\,Integer\,]$$
$$sieve\ (p : xs) = p : sieve\ [\,x \mid x \leftarrow xs, x\ `mod`\ p > 0\,]$$

What happens in the sieve of Eratosthenes in Figure 4.7 is that once a number is

(a) Early state   (b) Later state
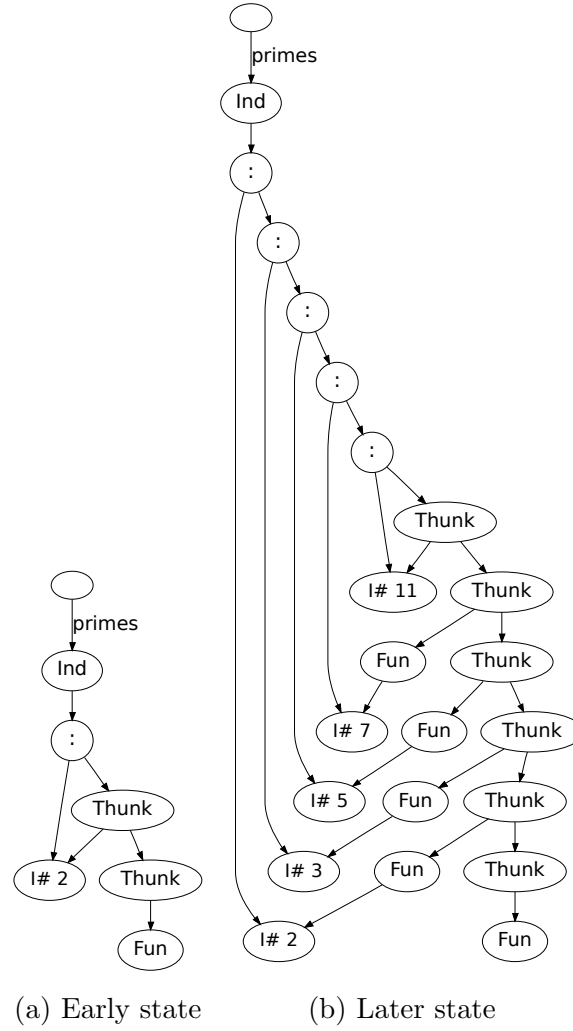
Figure 4.7: Sieve of Eratosthenes, see Appendix A for a larger example

visited by the top thunk, it is declared a prime, added to the final list of primes and a thunk is created for it. This thunk traverses down the infinite list of numbers and filters out all multiples of itself. Likewise the top thunk wanders down the resulting list, spawning new thunks for all numbers it encounters. Because the numbers have been filtered by the thunks spawned before they must be primes.

4.2.5 Unboxed Values

Some basic Haskell data types are defined as follows in GHC:

$$
\begin{aligned}
\textbf{data } Int &= I \# \quad Int \# \\
\textbf{data } Integer &= S \# \quad Int \# \\
&\mid J \# \quad Int \# \; ByteArray \# \\
\textbf{data } Word &= W \# \; Word \# \\
\textbf{data } Float &= F \# \quad Float \# \\
\textbf{data } Double &= D \# \quad Double \#
\end{aligned}
$$

We have already seen $S\#$, which is used to define small integers and $D\#$ for double precision floating point numbers.

The difference between *Int* and *Int#* is that the first is a boxed value, the second is unboxed. We can easily visualize what that means:

```
ghci> data Point = Point Int Int
ghci> :view Point 2 3
ghci> data PointU = PointU Int# Int#
ghci> :view PointU 2# 3#
```
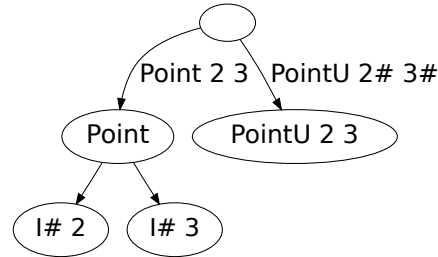

Figure 4.8: Boxed and unboxed values

In Figure 4.8 we see that boxed values are not actually stored in the data constructor, but instead are seperate closures which the data constructor points to. Unboxed values are stored inside of the data constructor directly. This makes them interesting for performant code as the overhead of an indirection and the space for another closure are saved.

4.2.6 Double Linked List

A nice example of the power of lazy evaluation is using it to create a double linked list[27]:

$$\textbf{data } DList\ \alpha = DLNode\ (DList\ \alpha)\ \alpha\ (DList\ \alpha)\ \textbf{deriving } Show$$
$$mkDList \quad :: [\alpha] \rightarrow DList\ \alpha$$
$$mkDList\ [\,] = error\ \texttt{"must have at least one member"}$$
$$mkDList\ xs = \textbf{let } (first, last) = go\ last\ xs\ first$$
$$\textbf{in } first$$
$$\textbf{where } go :: DList\ \alpha \rightarrow [\alpha] \rightarrow DList\ \alpha \rightarrow (DList\ \alpha, DList\ \alpha)$$
$$go\ prev\ [\,] \qquad next = (next, prev)$$
$$go\ prev\ (x : xs)\ next = \textbf{let } this \qquad = DLNode\ prev\ x\ rest$$
$$(rest, last) = go\ this\ xs\ next$$
$$\textbf{in } (this, last)$$

The real work is done in the *go* function: A *DLNode* is created using *prev*, which is the previous list member, and *rest*, which is only defined in the next line and is the result of calling *go* for the next node. Thanks to lazy evaluation we can supply *this* to *go* and at the same time define *this* using the result of the exact same call to *go*.

Connecting the last and the first members of the double linked list also makes heavy use of lazy evaluation: In *mkDList last* is defined as the second tuple element of a call to *go*, which takes *last* as a parameter. This technique is called *tying the knot.*

Trying to print the created double linked list using the *show* function does not work, because it is cyclic and *show* does not know when to stop. The structure of the final list can be seen using ghc-vis, as is shown in Figure 4.9.
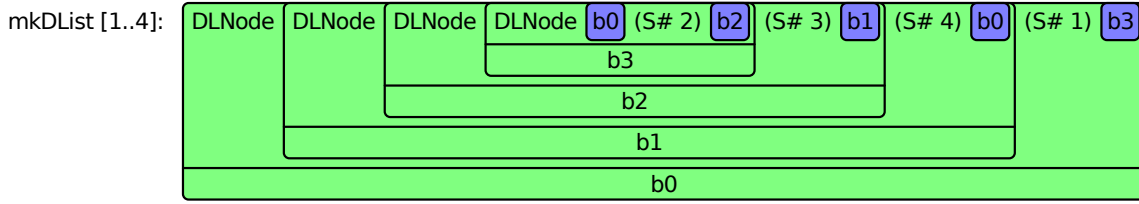
(a) Linear view
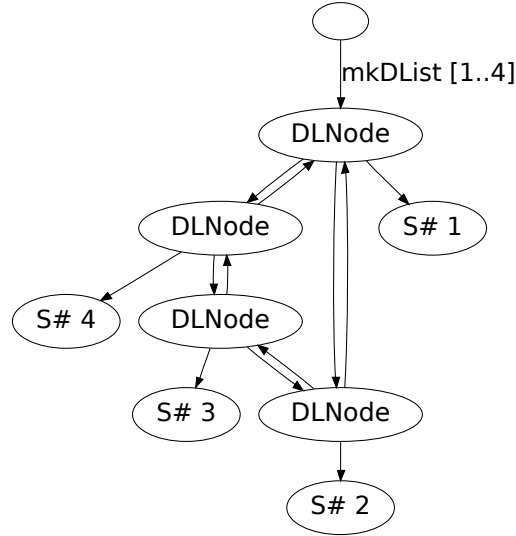
(b) Graph view

Figure 4.9: Double linked list

4.2.7 IntMap

Sharing is used in many Haskell libraries to achieve high performance. Consider for example *IntMap*s:

```
ghci> let x = Data.IntMap.fromList $ zip [1..5] [1..]
ghci> x
fromList [(1,1),(2,2),(3,3),(4,4),(5,5)]
ghci> let y = Data.IntMap.fromList $ zip [6..10] [6..]
ghci> y
fromList [(6,6),(7,7),(8,8),(9,9),(10,10)]
ghci> :view x
ghci> :view y
ghci> :view Data.IntMap.union x y
```

The resulting graph is shown in Figure 4.10. Only some internal nodes get newly created for the union of x and y, most of the original data structures are reused.

4.2.8 Nexuses and Dynamic Programming

This and the next subsection contain examples from [8]. In *dynamic programming* a memo table is usually used to avoid recomputing solutions to subproblems. In Haskell one can use another approach to achieve the same: *Nexus trees*, which are trees with shared nodes.

We define a data type *Tree* for binary trees:
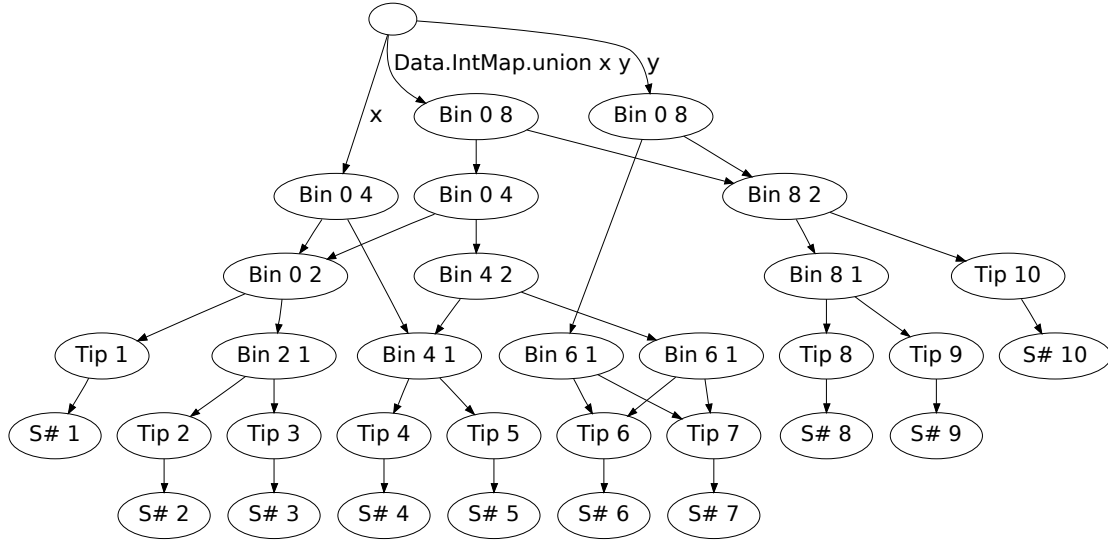
Figure 4.10: Sharing in *IntMap*s

data *Tree a = Empty*
                 | *Node* { *left* :: *Tree a, info* :: *a, right* :: *Tree a* }
        deriving (*Eq, Show*)

leaf   :: $\forall \alpha. \alpha \rightarrow Tree\ \alpha$
leaf x   = *Node Empty x Empty*

The definition of insertion into a binary tree:

insert   :: $\forall \alpha. Ord\ \alpha \Rightarrow \alpha \rightarrow Tree\ \alpha \rightarrow Tree\ \alpha$
insert x Empty = leaf x
insert x (*Node l k r*)
     | $x \leqslant k$ = *Node* (*insert x l*) *k r*
     | *otherwise*   = *Node l k* (*insert x r*)

Now we can create a full binary tree of depth n containing the value x using the function *full*:

full   :: $\forall \alpha. Integer \rightarrow \alpha \rightarrow Tree\ \alpha$
full 0 *x*   = *leaf x*
full n x   = *Node t x t*
     where $t = full\ (n-1)\ x$

Figure 4.11 shows how sharing is used in the evaluation of *full* 4 2.0: The child node t is created once and then shared as the left and right child, which makes the resulting tree a nexus. Because of this a full binary tree can be created in $O(\log n)$ time and stored in $O(\log n)$ space, where n is the number of nodes.

Once we change a seemingly slight detail of the definition of *full*, as we did in *full'* by typing out the identical left and right child of the created node, the sharing disappears:

full'   :: $\forall \alpha. Integer \rightarrow \alpha \rightarrow Tree\ \alpha$
full' 0 *x*   = *leaf x*
full' n x   = *Node* (*full'* (*n* − 1) *x*) *x* (*full'* (*n* − 1) *x*)

Because of referential transparency this refactoring is allowed and does not change
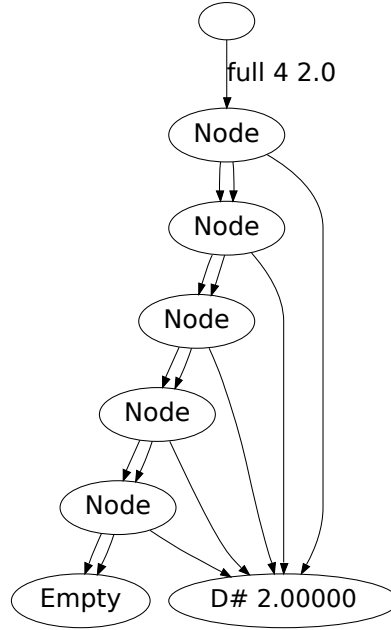
Figure 4.11: Visualization of *full*: Nodes are shared
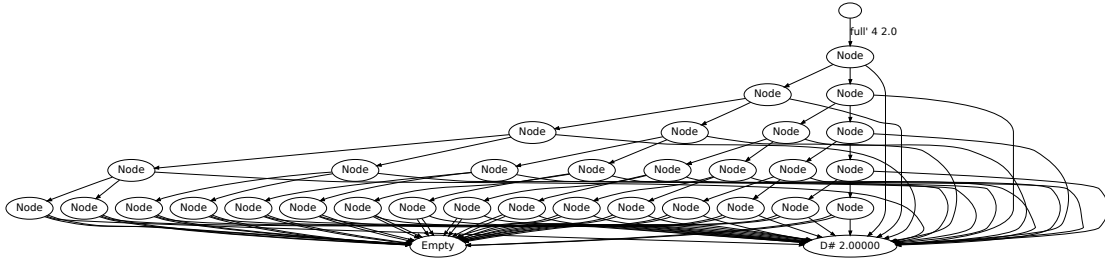
Figure 4.12: Visualization of *full'*: Nodes are *not* shared

the semantics of the function. Figure 4.12 reveals the effect the missing sharing has: We're not getting a nexus anymore. Space and time used are now linear, even though we are clearly expressing the same graph, as *full* 4 2.0 ≡ *full′* 4 2.0 evaluates to *True*.

Nexuses provide an alternative way to the list based one for calculating Fibonacci numbers we visualized in Subsection 4.2.3:

$$
\begin{aligned}
&memofib \quad :: Integer \rightarrow Tree\ Integer \\
&memofib\ 0 = leaf\ 0 \\
&memofib\ 1 = Node\ (leaf\ 0)\ 1\ Empty \\
&memofib\ n = node\ t\ (left\ t) \\
&\quad\quad \textbf{where}\ t = memofib\ (n-1) \\
&node \quad\quad :: Tree\ Integer \rightarrow Tree\ Integer \rightarrow Tree\ Integer \\
&node\ l\ r \quad = Node\ l\ (info\ l + info\ r)\ r
\end{aligned}
$$

The n-th Fibonacci number can be obtained using *info* (*memofib* 8). In Figure 4.13 the sharing used in *memofib* can be seen.
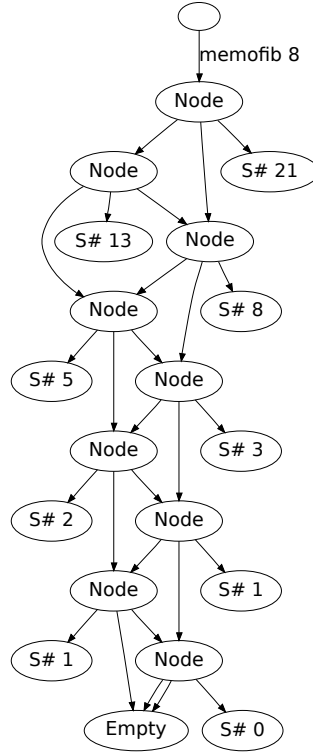
Figure 4.13: Fibonacci numbers using a nexus

4.2.9 Optimal Bracketing

Let us consider the problem of *optimal bracketing*: We want to bracket an expression of the form $x_1 \oplus x_2 \oplus \ldots \oplus x_n$, where $\oplus$ is an associative operator. That means the result will be the same, no matter how we bracket the expression. But computational costs can differ, for example in a chain matrix multiplication.

$$
\begin{aligned}
\textbf{data}\ Expr\ \alpha = {} & Const\ \alpha \\
& |\ Expr\ \alpha \oplus Expr\ \alpha \\
\textbf{deriving}\ & Show
\end{aligned}
$$

We can solve the problem using the recursive function *opt*:

$$
\begin{aligned}
opt & :: [\alpha] \rightarrow Expr\ \alpha \\
opt\ [x] & = Const\ x \\
opt\ xs & = best\ [opt\ s_1 \oplus opt\ s_2 \mid (s_1, s_2) \leftarrow uncat\ xs] \\
uncat & :: \forall \alpha.[\alpha] \rightarrow [([\alpha], [\alpha])] \\
uncat\ [x_1, x_2] & = [([x_1], [x_2])] \\
uncat\ (x : xs) & = ([x], xs) : map\ (\lambda(l, r) \rightarrow (x : l, r))\ (uncat\ xs)
\end{aligned}
$$

In this example we consider the expression with the lowest bracketing depth the best one, as could be desirable for parallel execution:

$$
\begin{aligned}
best & :: [Expr\ \alpha] \rightarrow Expr\ \alpha \\
best\ xs & = snd\ \$\ go\ xs \\
\textbf{where}\ go\ [y] & = (size\ y, y) \\
go\ (y : ys) & \\
\mid sizeY < sys & = (sizeY, y)
\end{aligned}
$$

$$| \; otherwise \quad = gys$$
$$\textbf{where} \; gys@(sys, \_) = go \; ys$$
$$sizeY = size \; y$$

$$size \qquad\qquad :: \; Expr \; a \rightarrow Integer$$
$$size \; ((l \oplus r)) \;\; = 1 + max \; (size \; l) \; (size \; r)$$
$$size \; (Const \; \_) = 1$$

The possible bracketings of $Const1 \oplus Const2 \oplus Const3 \oplus Const4$ are:

1. $((Const1 \oplus Const2) \oplus Const3) \oplus Const4$

2. $(Const1 \oplus Const2) \oplus (Const3 \oplus Const4)$

3. $Const1(\oplus Const2 \oplus (Const3 \oplus Const4))$

The second bracketing is the optimal one by our definiton of *best*, as is verified by $opt \; [1 \mathrel{.\,.} 4]$ evaluating to $(Const \; 1 \oplus Const \; 2) \oplus (Const \; 3 \oplus Const \; 4)$.

But this function takes exponentional time, which is why we are now going to consider a solution using a nexus:

$$leaf \qquad\qquad :: \; \alpha \rightarrow Tree \; (Expr \; \alpha)$$
$$leaf \; x \qquad\qquad = Node \; Empty \; (Const \; x) \; Empty$$

leaf x creates a Leaf node storing the value x.

$$lspine, rspine \qquad :: \; \forall \alpha. \, Tree \; \alpha \rightarrow [\alpha]$$
$$lspine \; (Empty) \quad = [\,]$$
$$lspine \; (Node \; l \; x \; r) = lspine \; l \mathbin{+\!\!+} [x]$$
$$rspine \; (Empty) \quad = [\,]$$
$$rspine \; (Node \; l \; x \; r) = [x] \mathbin{+\!\!+} rspine \; r$$

The functions *lspine* and *rspine* traverse the left and right side of the tree respectively, returning the nodes visited.

$$combine \qquad\qquad :: \; [(Expr \; \alpha, Expr \; \alpha)] \rightarrow Expr \; \alpha$$
$$combine \qquad\qquad = best \circ map \; (uncurry \; (\oplus))$$
$$node \qquad\qquad :: \; Tree \; (Expr \; \alpha) \rightarrow Tree \; (Expr \; \alpha) \rightarrow Tree \; (Expr \; \alpha)$$
$$node \; l \; r \qquad\qquad = Node \; l \; (combine \; (zip \; (lspine \; l) \; (rspine \; r))) \; r$$

The function *node* constructs a new node for our nexus.

$$step \qquad\qquad :: \; [\, Tree \; (Expr \; \alpha)] \rightarrow [\, Tree \; (Expr \; \alpha)]$$
$$step \; [t] \qquad\qquad = [\,]$$
$$step \; (t_1 : t_2 : ts) \qquad = node \; t_1 \; t_2 : step \; (t_2 : ts)$$

step creates the next layer of nodes in our nexus. Sharing is introduced in the last line by t_2 appearing twice.

$$build \qquad\qquad :: \; [\, Tree \; (Expr \; \alpha)] \rightarrow Tree \; (Expr \; \alpha)$$
$$build \; [t] \qquad\qquad = t$$
$$build \; ts \qquad\qquad = build \; (step \; ts)$$
$$bottomup \qquad\qquad :: \; [\alpha] \rightarrow Tree \; (Expr \; \alpha)$$
$$bottomup \qquad\qquad = build \circ map \; leaf$$

Finally *bottomup* builds the entire nexus using the funtions defined above.
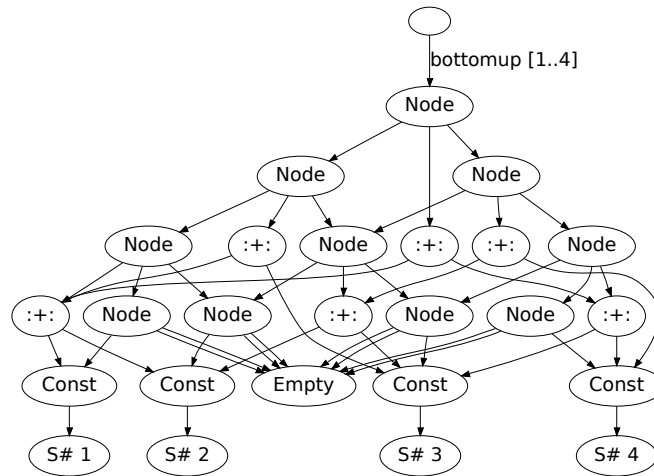
Figure 4.14: Solving the *optimal bracketing* problem using a nexus

When we now evaluate *bottomup* [1 . . 4] the data structure seen in Figure 4.14 gets created. The resulting tree uses extensive sharing to construct the optimal solution of the problem from optimal solutions to subproblems. The solution is obtained by *info* (*bottomup* [1 . . 4]).

Further examples of consciously using sharing in the form of nexuses are available in [28].

4.2.10 Negative Effects of Lazy Evaluation and Sharing

Unfortunately the effects of lazy evaluation and sharing are not always beneficial. Consider the follwing example:

```
ghci> let x = let n = [1..5] in (last n, head n)
ghci> :view x
```

As we can see in Figure 4.15 when the first element of the tuple x gets evaluated the whole list n gets evaluated to determine its last member and thereby stored on the heap. As n is shared between the tuple elements, *head n* now refences the entire list n and forces the garbage collector to leave it on the heap. This would not be necessary, as *head n* can be obtained from the first list member of n, whereas the rest of the list is not needed anymore. A *space leak* has occured. While the space leak in this example is negligibly small, much larger ones can occur in practice. When we *avoid sharing* this problem does not occur in our example:

```
ghci> let y = (last [1..5], head [1..5])
ghci> :vis y
```

The resulting data structure can be seen in Figure 4.16. This would not work anymore when we get passed a list instead. A more general approach that *explicitly un-shares* data structures is provided by the experimental *dup*[29] operation.

(a) Tuple elements are un- (b) First tuple element was (c) Second tuple element
    evaluated   evaluated (*fst x*)   was evaluated (*snd x*)

Figure 4.15: Space leak

(a) Tuple elements are un- (b) First tuple element was (c) Second tuple element
    evaluated   evaluated (*fst x*)   was evaluated (*snd x*)
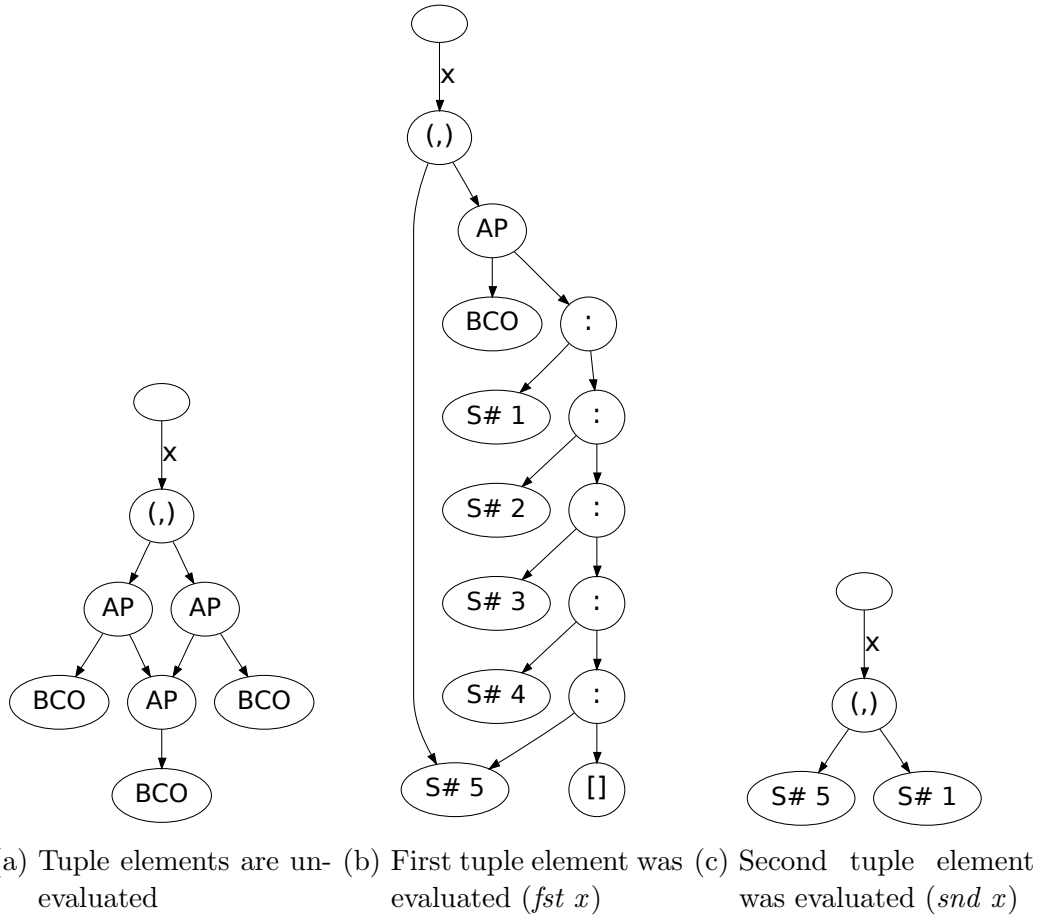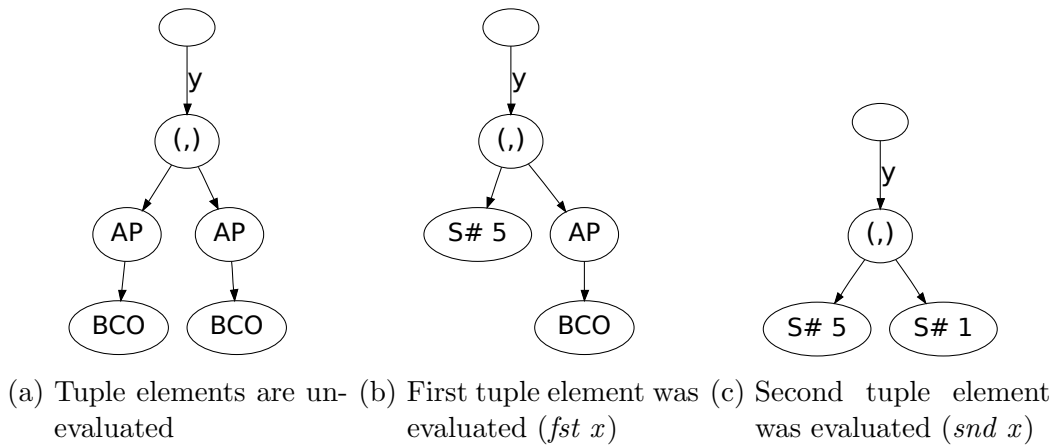
Figure 4.16: No more space leak by avoiding sharing

Another cause for space leaks is lazy evaluation:

$$sum :: \quad (Num\ a) \Rightarrow [a] \rightarrow a$$
$$sum\ l = sum'\ l\ 0$$
where
$$\quad sum'\ [\,] \qquad a = a$$
$$\quad sum'\ (x : xs)\ a = sum'\ xs\ (a + x)$$

This is the definition of *sum* from Haskell's base library. When we evaluate *sum* $[1 \mathinner{\ldotp\ldotp} 10\uparrow$ 7] in GHCi the memory usage quickly grows to hundreds of Megabytes.

What is happening? We can use ghc-vis in combination with GHCi's debugger to find out:[1]

```
ghci> :l sum
[1 of 1] Compiling Main              ( sum.hs, interpreted )
Ok, modules loaded: Main.
ghci> :vis
ghci> :break 6
ghci> sum [1..10^7]
Stopped at sum.hs:6:20-32
_result :: Integer = _
a :: Integer = _
x :: Integer = 1
xs :: [Integer] = _
ghci> :view a
ghci> :clear
ghci> :continue
Stopped at sum.hs:6:20-32
_result :: Integer = _
a :: Integer = _
x :: Integer = 2
xs :: [Integer] = _
ghci> :view a
ghci> :clear
ghci> :continue
Stopped at sum.hs:6:20-32
_result :: Integer = _
a :: Integer = _
x :: Integer = 3
xs :: [Integer] = _
ghci> :view a
```

The status of *a* at every recursion step can be seen in Figure 4.17. The path of unevaluated closures to calculate *a* keeps growing. In the last line of the definition of *sum'* $(a + x)$ is not evaluated and instead the unevaluated thunk gets passed to the next call to *sum'*. This builds up a big amount of thunks, even though it could be reduced to a single value.

---

[1]Note that the debugger of GHCi can only use breakpoints and single-stepping in interpreted code, so we have to load our own file containing the code to be debugged instead of using the *sum* function from the base library.

(a) First stop   (b) Second stop   (c) Third stop
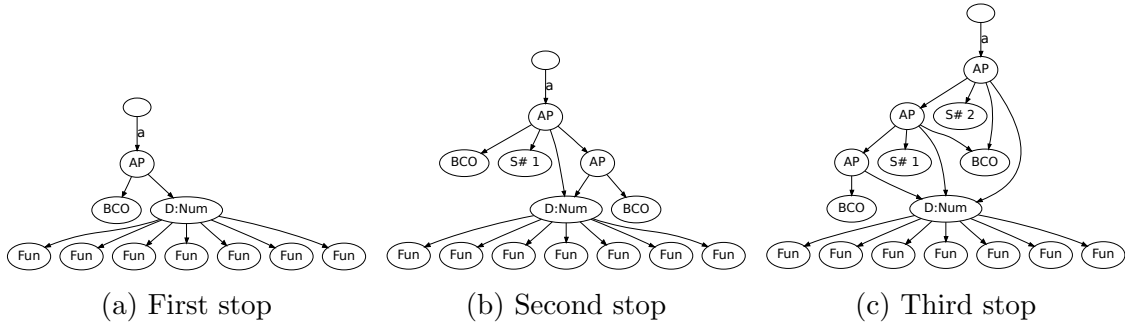
Figure 4.17: Space leak caused by lazy evaluation in *sum*

To fix this we apply the last argument in the call to *sum'* strictly using the $! operator:

$$sum2 :: \quad (Num\ a) \Rightarrow [\,a\,] \rightarrow a$$
$$sum2\ l = sum'\ l\ 0$$
where
$$\quad sum'\ [\,] \qquad a = a$$
$$\quad sum'\ (x:xs)\ a = sum'\ xs\ \$!\ (a+x)$$

This *strict evaluation* causes $a+x$ to be evaluated to weak-head normal form, which

(a) First stop   (b) Second stop   (c) Third stop
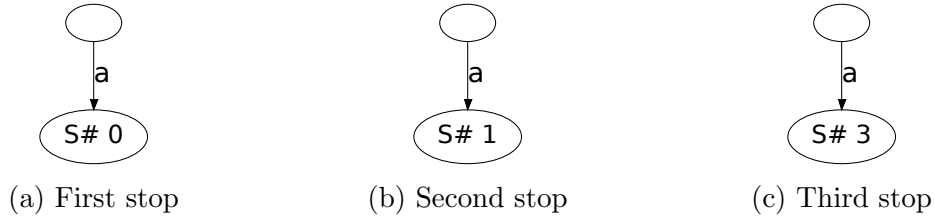
Figure 4.18: Avoiding space leak by strict evaluation in *sum2*

means only the actual value gets passed. Old values can then be removed by the garbage collector. Memory usage is constant with *sum2*, and the correct behavior can be seen in Figure 4.18, which has been recorded in the same way as we just did for *sum*.

GHC uses optimization techniques which can automatically prevent space leaks like this. When optimization is enabled during compilation the space leak disappears.

4.3 Combination with GHCi's Debugger

Evaluating a value to weak-head normal form works very well when visualizing the evaluation of lists or other composite data types. But for elementary data types like Integers even that may be too much and a smaller evaluation may be more appropriate. ghc-vis can be combined with GHCi's debugger using the supplied `:su`, `:asu` and `:tsu` commands, which allow single-stepping through the evaluation of expressions and watching the data structures transform during this.

We are going to use a file called *test.hs* in this example to illustrate this:

$$f \qquad \qquad :: [Integer] \rightarrow [Integer]$$
$$f\ (x:xs) = (2*x):f\ xs$$
$$f\ [\,] \qquad = [\,]$$

```
ghci> :l test.hs
ghci> :vis
ghci> let x = [1..3]
ghci> let y = f x
ghci> let z = sum y
ghci> :view x
ghci> :view y
ghci> :view z
ghci> :su z
```

x: S# 1 : S# 2 : t0 ((f0 (S# 3, S# 1), b1 , b2)   x: (S# 1):(S# 2): S# 3 :[]
   b0   b1   b2   b0

y: t1 [b0]: TSO   y: (S# 2): TSO : t0 [b0]:[]
             b3   b1

z: b3   z: b1

     (a) Early state   (b) Later state
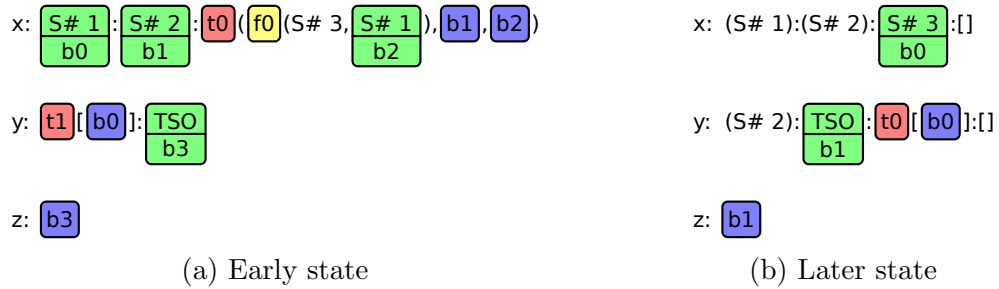
Figure 4.19: Combination with GHCi's debugger, see Appendix B for a larger example

Using :su we can now single-step through the evaluation of z and watch the data structures changing during it. Two intermediate steps can be seen in Figure 4.19.

The available commands for combining ghc-vis with GHCi's debugger are described in Table 4.2.

Command	Description
:su x	Same as :step x: Single-step through evaluation of x, but the visualization gets updated automatically
:cu	Same as :continue: Continue to next breakpoint, but the visualization gets updated automatically
:asu x	Single-step through evaluation of x on any key pressed, quit with q
:tsu t x	Single-step through evaluation of x with one step every t seconds until a key is pressed

Table 4.2: Debugging commands of ghc-vis

4.4 Usage as a Library

Although ghc-vis is mainly meant to be used in GHCi it can also be used as a library in regular Haskell programs which are then run or compiled by GHC to inspect expressions inside of them at runtime[2]:

---

[2]Currently ghc-vis only works correctly in compiled programs when they have been compiled with the *-threaded* flag

```
import GHC.Vis
main = do
  let a = "teeest"
  let b = [1 . . 3]
  let c = b ++ b
  let d = [1 . .]
  putStrLn $ show $ d !! 1

  visualization
  view a "a"
  view b "b"
  view c "c"
  view d "d"

  getChar
  switch

  getChar
```
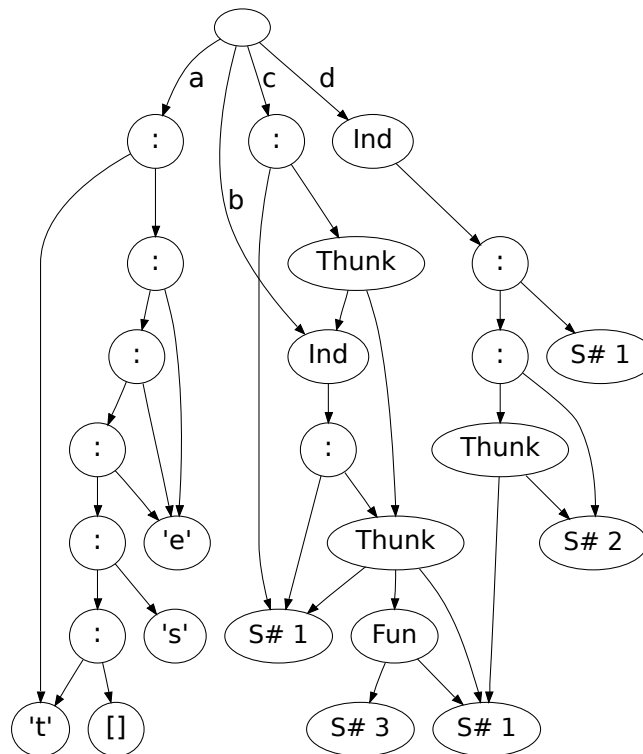

Figure 4.20: ghc-vis as a library

The output on the screen looks exactly like the one we are used from inside of GHCi, but some optimizations stand out, which are not performed in GHCi: Notice for example how letters are shared within a in Figure 4.20.

The GHCi commands of ghc-vis closely resemble the library interface, as is shown in Table 4.3.

Method call	Type	Corresponding GHCi command
visualization	*IO* ()	`:vis`
view x `"x"`	$a \rightarrow String \rightarrow IO$ ()	`:view x`
eval `"t0"`	$String \rightarrow IO$ ()	`:eval t0`
switch	*IO* ()	`:switch`
update	*IO* ()	`:update`
clear	*IO* ()	`:clear`
export `"file.pdf"`	$String \rightarrow IO$ ()	`:export file.pdf`

Table 4.3: Public library interface

4.5 Source Code

Project	File	Code	Comments
ghc-vis	Vis.hs	180	70
	Vis/Internal.hs	393	184
	View/List.hs	276	129
	View/Graph.hs	117	53
	View/Graph/Parser.hs	79	49
	View/Common.hs	43	27
	Types.hs	70	29
	ghci	11	9
		1169	550
xdot	Parser.hs	213	46
	Viewer.hs	156	56
	Types.hs	35	13
	Demo.hs	96	36
		500	151

Table 4.4: Source code statistics

Statistics of the source code for ghc-vis and xdot are presented in Table 4.4. They were created using GHC's *count_lines*.

4.6 Performance

ghc-vis is not written with big data structures in mind, as their visualizations would become two crowded to understand easily. The measurements of ghc-vis' performance in Table 4.5 indicate that the HeapMap data structure is not restricting the performance, but instead the complex drawing operations take most of the time.

Operation	Evaluated members	Runtime [ms]
walkHeap	0	0.02
	25	0.13
	50	0.36
	75	0.70
	100	1.19
xDotParse	0	13.46
	25	22.82
	50	36.52
	75	53.11
	100	69.70
linear view	0	6.68
	25	9.51
	50	11.68
	75	19.71
	100	22.04
graph view	0	36.87
	25	125.68
	50	239.17
	75	353.40
	100	472.54

Table 4.5: Runtime of various operations on a partially evaluated list

These measurements were taken using *Criterion*[30] on a machine with an Intel Core2 processor clocked at 2.5 GHz, running Linux 3.5.3 and GHC 7.4.2 using the x86-64 instruction set.

CHAPTER 5

Conclusion

This thesis described the design and implementation of *ghc-vis*, a tool for visualizing Haskell data structures inside of the Glasgow Haskell Compiler's interactive environment and compiled programs at runtime.

After introducing the Haskell programming language and two of its features, lazy evaluation and sharing, we investigated relevant specifics of the Glasgow Haskell Compiler and considered existing tools, which can be used to visualize lazy evaluation and sharing. We noticed that not all necessary information is available in existing tools, thus justifying the creation of ghc-vis to fill these gaps.

The integration of ghc-vis into GHCi and the extraction of information from the GHC heap were detailed. Problems which were encountered in realizing this had to be solved.

Finally we showed how ghc-vis can help visualizing simple and more complex data structures to understand them and how lazy evaluation and sharing affect them.

5.1 Future Work

Some issues remain with ghc-vis:

As we have seen the performance for big data structures is still low. In order to improve it the graph view's drawing operations should be improved.

To visualize big data structures another function would have to be implemented: Zooming into and moving around the view.

Some information, for example type classes, can be overwhelming in some visualizations, but desirable in others. Allowing the user to hide certain data structures dynamically may be a way to solve this problem.

It might be worth investigating how to gather more information in GHCi debugging sessions to provide better visualizations.

It would be desirable to provide type information about closures in ghc-vis. But Haskell is statically typed and so types are checked at compile-time. Therefore no types are stored in GHC heap closures. Instead the type would have to be reconstructed, as GHCi's debugger does. This would require a deeper integration of

ghc-vis into GHC, as the necessary functions for type reconstruction are not available from outside of GHC.

Another feature is making objects from ghc-vis available as variables in GHCi. Type reconstruction would also be needed for this. There are still some ways to get access to the internals of visualized objects, but they are not as flexible: You can use a regular function, if one exists, for example *head* to access the first element of a list. Alternatively you can use *:print* to get access to some thunks. By adding the variable to ghc-vis you can see that you are accessing the correct one.

Integrating ghc-vis into a web based GHCi interface might be another route of work, that would help newcomers with understanding Haskell better.

Bibliography

[1] G. Hutton, *Programming in Haskell*. Cambridge University Press, Jan. 2007.

[2] A. J. T. Davie, *An introduction to functional programming systems using Haskell*. Cambridge University Press, 1992.

[3] C. Strachey, "Fundamental concepts in programming languages," *Higher Order and Symbolic Computation*, vol. 13, no. 1-2, pp. 11–49, Apr. 2000.

[4] W. Quine, *Word and Object*. The MIT Press, 1960.

[5] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, Sep. 1989.

[6] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Apr. 1989.

[7] S. Marlow, "Haskell 2010 language report."

[8] R. Bird and R. Hinze, "Trouble shared is trouble halved," in *Haskell Workshop*, 2003, pp. 1–6.

[9] "The Glasgow Haskell Compiler," http://www.haskell.org/ghc/, accessed: 2012-09-18.

[10] "The Haskell Platform," http://www.haskell.org/platform/, accessed: 2012-09-18.

[11] S. Peyton Jones, "Implementing lazy functional languages on stock hardware: The spineless tagless g-machine," *Journal of Functional Programming*, vol. 2, no. 2, pp. 127–202, 1992.

[12] "GHC Commentary: The layout of Heap Objects," http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects, accessed: 2012-09-18.

[13] P. Wadler, "Fixing some space leaks with a garbage collector," *Softw. Pract. Exper.*, vol. 17, no. 9, pp. 595–608, Sep. 1987.

[14] R. Jones, "Tail recursion without space leaks," *Journal of Functional Programming*, vol. 2, 1991.

[15] J. Breitner, "ghc-heap-view-0.3.0.2: Extract the heap representation of Haskell values and thunks," http://hackage.haskell.org/package/ghc-heap-view-0.3.0.2, accessed: 2012-09-18.

[16] S. Marlow and S. Peyton Jones, "Making a fast curry: push/enter vs. eval/apply for higher-order languages," in *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, vol. 39, no. 9, Sep. 2004, pp. 4–15.

[17] J. Iborra and S. Marlow, "Examine your laziness. A lightweight procedural debugging technique for Haskell," Tech. Rep., Apr. 2007.

[18] A. Seipp, "Vacuum: visualising the GHC heap," http://thoughtpolice.github.com/vacuum/, accessed: 2012-09-18.

[19] A. Gill, "Debugging Haskell by observing intermediate data structures," in *Proceedings of the 2000 ACM SIGPLAN Workshop on Haskell*, 2000.

[20] C. Reinke, "GHood – Graphical Visualisation and Animation of Haskell Object Observations," in *Proceedings of the 2001 ACM SIGPLAN Workshop on Haskell*, September 2001.

[21] "Gtk2Hs: A GUI Library for Haskell based on Gtk," http://projects.haskell.org/gtk2hs/, accessed: 2012-09-18.

[22] S. Peyton Jones, S. Marlow, and C. Elliot, "Stretching the storage manager: weak pointers and stable names in Haskell," in *Proceedings of the 11th International Workshop on the Implementation of Functional Languages*, September 1999.

[23] A. Gill, "Type-safe observable sharing in Haskell," in *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[24] "Graphviz - Graph Visualization Software," http://www.graphviz.org/, accessed: 2012-09-18.

[25] I. L. Miljenovic, "graphviz - graph visualisation library for Haskell," http://projects.haskell.org/graphviz/, accessed: 2012-09-18.

[26] "Graphviz Output Formats: xdot," http://www.graphviz.org/doc/info/output.html#d:xdot, accessed: 2012-09-18.

[27] "Tying the Knot," http://www.haskell.org/haskellwiki/Tying_the_Knot, accessed: 2012-09-18.

[28] R. Bird, "Hylomorphisms and nexuses," in *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010, ch. 21, pp. 168–179.

[29] J. Breitner, "dup – explicit un-sharing in haskell," *CoRR*, vol. abs/1207.2017, 2012.

[30] B. O'Sullivan, "Criterion: robust, reliable performance measurement," https://github.com/bos/criterion, accessed: 2012-09-18.

Appendix

A Full Examination of the Sieve of Eratosthenes
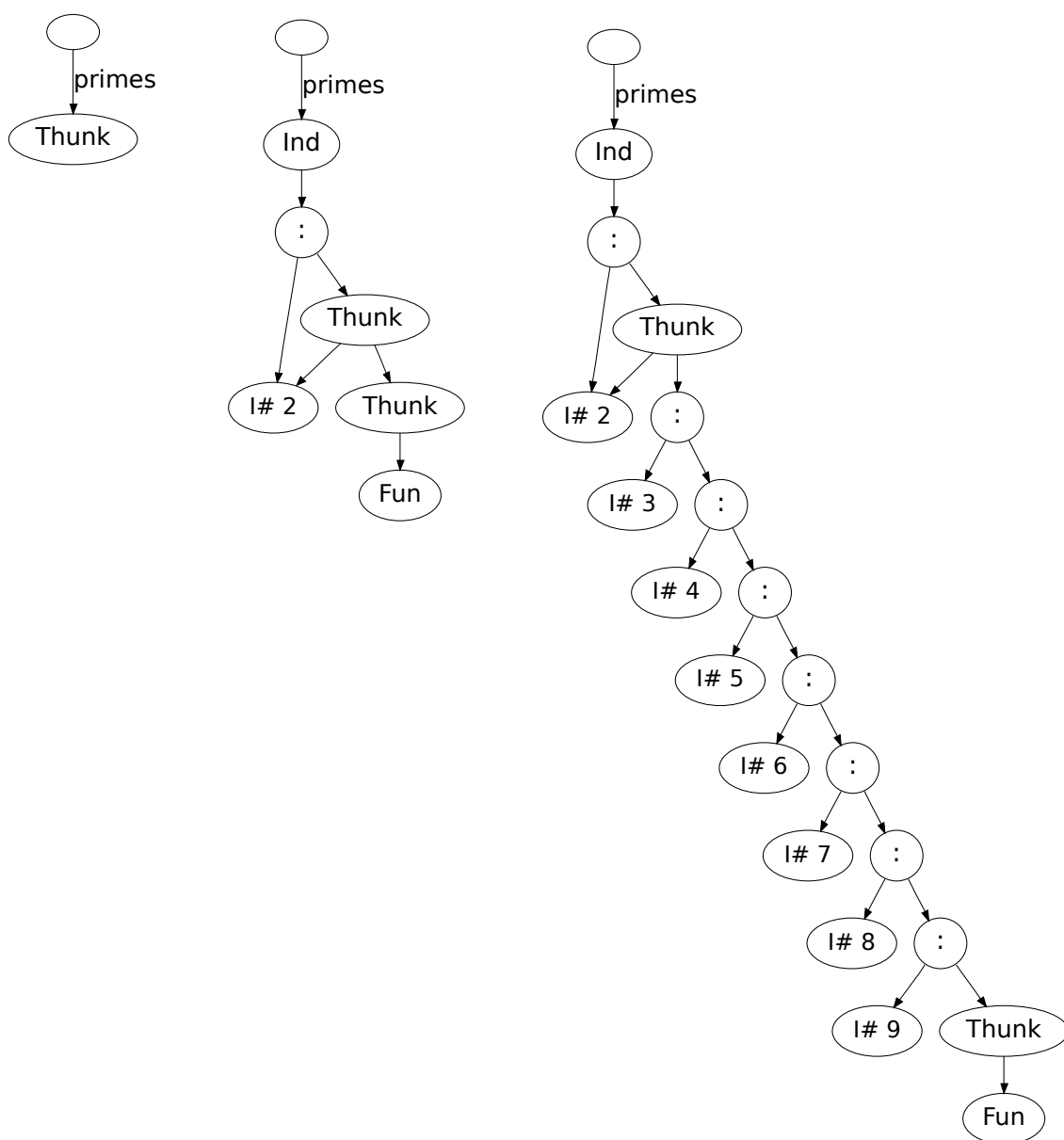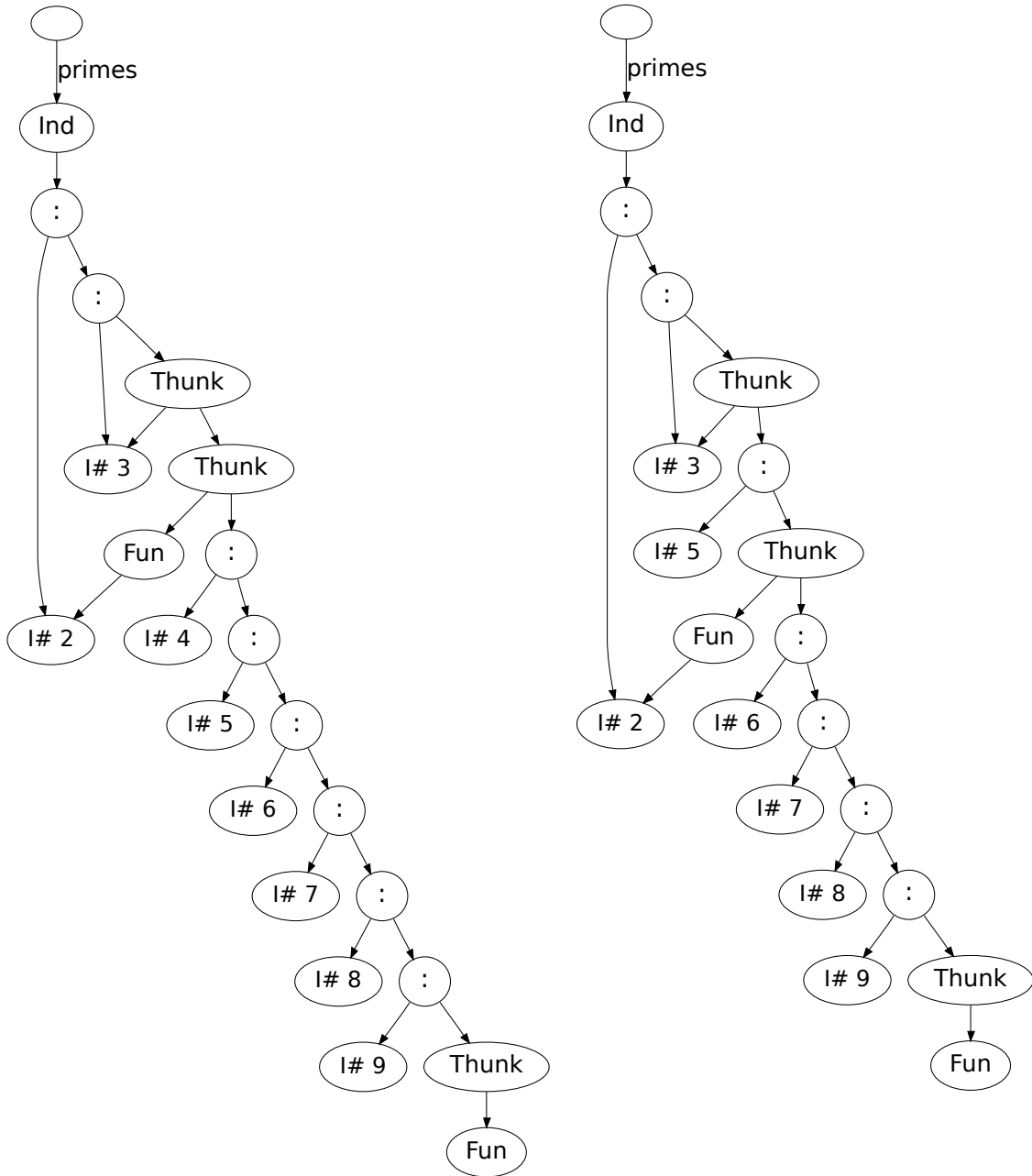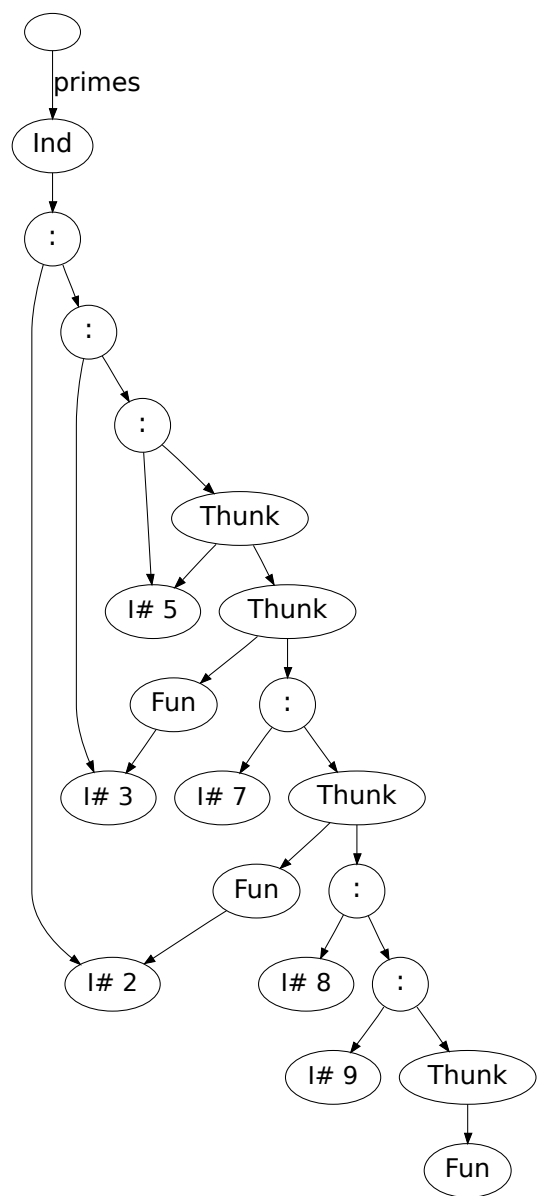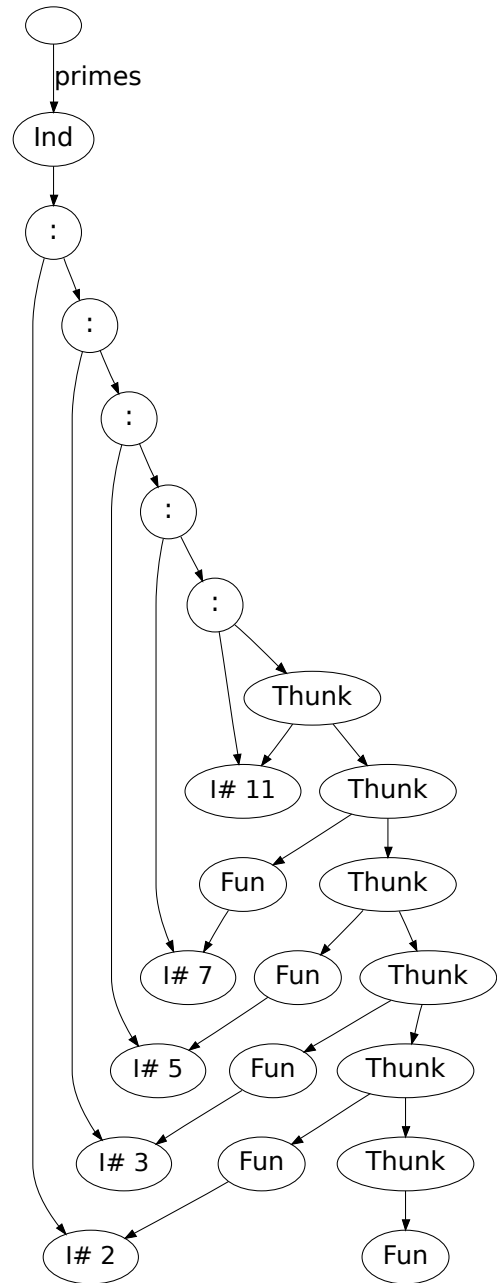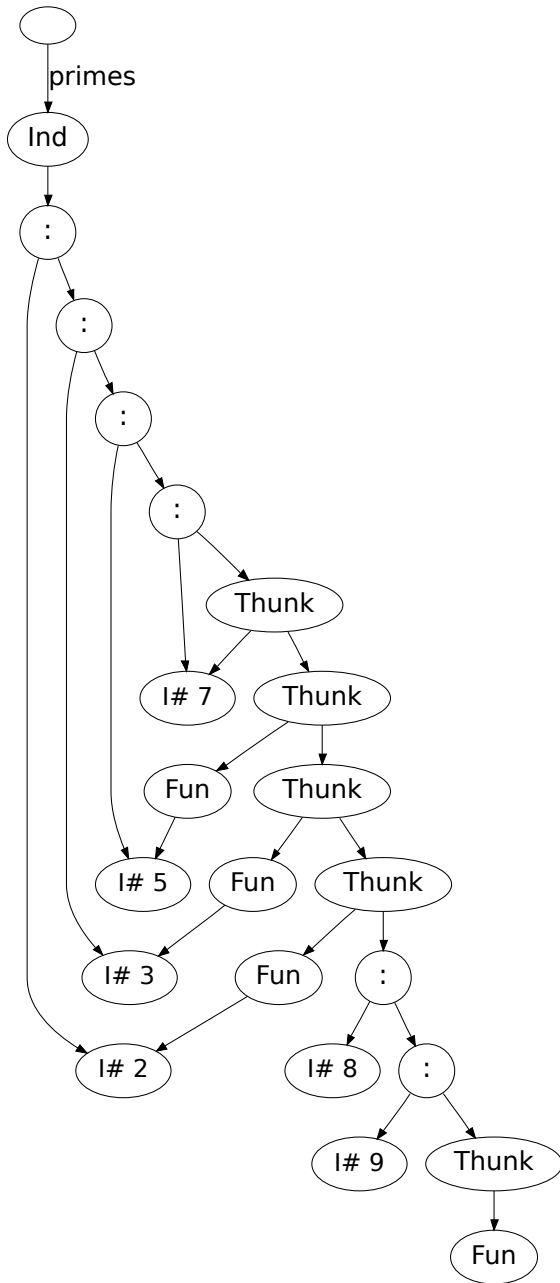
B Full Example of combining ghc-vis with GHCi's Debugger

We are going to use a file called *test.hs* with the following content in this example:

$$
\begin{aligned}
&f &&:: [\mathit{Integer}] \rightarrow [\mathit{Integer}] \\
&f\ (x:xs) = (2*x):f\ xs \\
&f\ [] &&= []
\end{aligned}
$$

```
ghci> :l test.hs
ghci> :vis
ghci> let x = [1..3]
ghci> let y = f x
ghci> let z = sum y
ghci> :view x
ghci> :view y
ghci> :view z
```
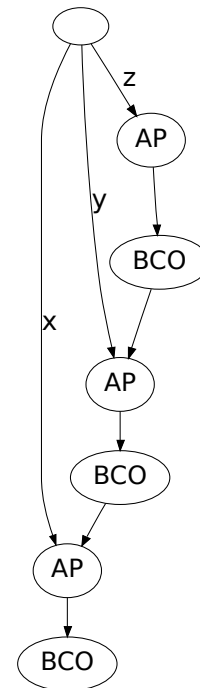


```
ghci> :su z
```

x: t0

y: TSO / b0

z: b0
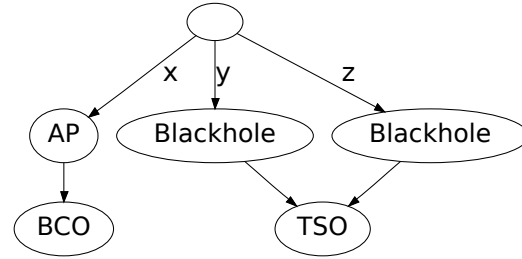
ghci> :su

From here on only *:su* is called so it will be omitted.

x: S# 1 / b0 :t0 (f0 (S# 3, S# 1 / b1), b0 , b1)

y: TSO / b2

z: b2
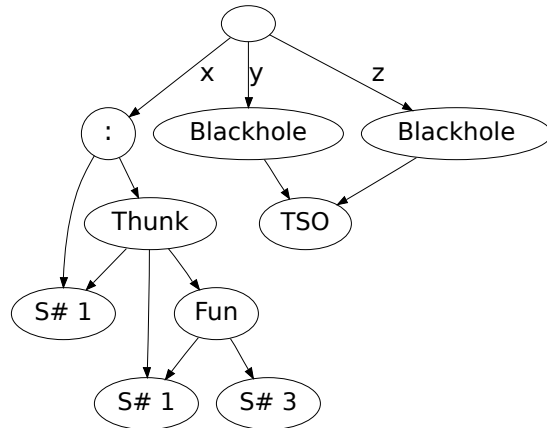
x: S# 1 / b0 :t0 (f0 (S# 3, S# 1 / b1), b0 , b1)

y: t1 [b0]: TSO / b2

z: b2

x: S# 1 / b0 : S# 2 / b1 : S# 3 / b2 : t0 (f0 (S# 3, S# 1 / b3), b2 , b3)
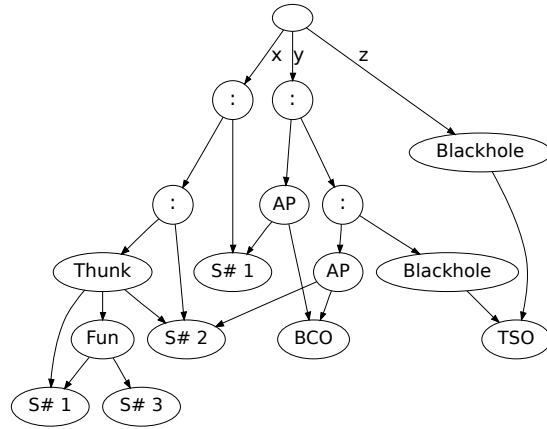
y: t1 BCO / b4 [b0]: t2 b4 [b1]: t3 b4 [b2]: TSO / b5

z: b5
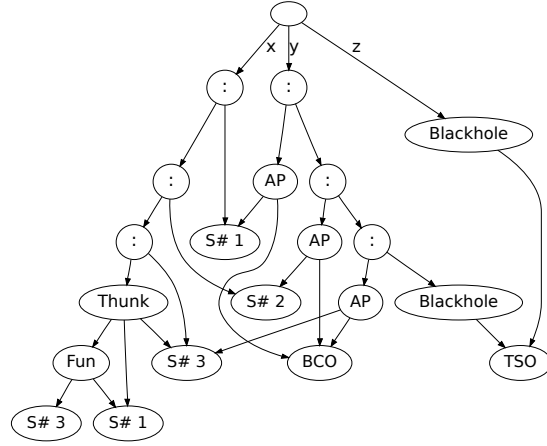
x: S# 1 / b0 : S# 2 / b1 : S# 3 / b2 :[]

y: t0 BCO / b3 [b0]: t1 b3 [b1]: t2 b3 [b2]: TSO / b4

z: b4
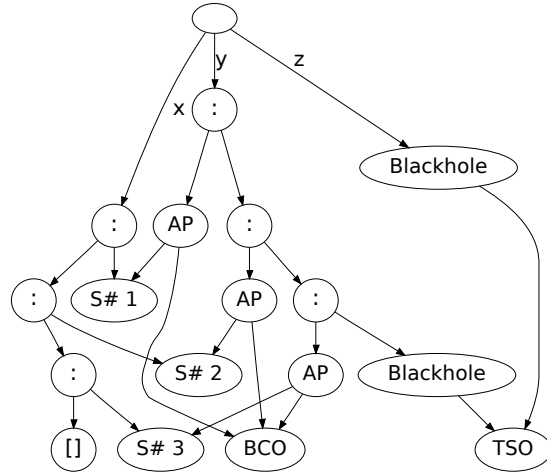
x: (S# 1): S# 2 / b0 : S# 3 / b1 :[]

y: TSO / b2 : t0 BCO / b3 [b0]: t1 b3 [b1]:[]

z: b2

x: (S# 1):(S# 2): [S# 3 / b0] :[]

y: (S# 2): [TSO / b1] :[t0][[b0]]:[]

z: [b1]

x: (S# 1):(S# 2):(S# 3):[]

y: (S# 2):(S# 4): [TSO / b0] :[]

z: [b0]

x: (S# 1):(S# 2):(S# 3):[]

y: (S# 2):(S# 4):(S# 6):[]

z: S# 12