GHood – Graphical Visualisation and Animation of Haskell Object Observations

Claus Reinke [1]

Computing Laboratory, University of Kent
Canterbury, UK

Abstract

As a possible extension to his Haskell Object Observation Debugger Hood [7], Andy Gill has described the "dynamic viewing of structures", stepping through observations instead of accumulating them into a static view. Starting from this idea, we have implemented and released an animation back-end for Hood, called GHood. Instead of the dynamic textual visualisation based on pretty-printing proposed in [7], our back-end features a dynamic graphical visualisation, based on a simple tree layout algorithm. This paper reviews the main aspects of Hood, gives a brief introduction to GHood's features and summarises our experience so far.

The visualisation of program behaviour via animations of data structure observations has uses for program comprehension and exposition, in development, debugging and education. We find that the graphical structure facilitates orientation even when textual labels are no longer readable due to scaling, suggesting advantages over a purely textual visualisation. A novel application area is opened by the use of GHood as an applet on web pages – discussions of Haskell program behaviour, e.g., in educational online material or in explanations of functional algorithms, can now easily be augmented with graphical animations of the issues being discussed.

1 Well-typed programs don't go anywhere – or do they?

The war-cry of static typing is that "well-typed programs don't go wrong", but sometimes the question is "where does this well-typed program go?", requiring a more detailed understanding of program behaviour.

For a surprisingly long time, Haskell programmers have been deprived of tools that would enable them to investigate the behaviour of their programs at a suitable level of abstraction. This lack of tool support, especially in the areas of debugging and profiling, has been quoted as one of the reasons "why no one uses functional languages" [18]. In the context of Haskell profiling, the lack has

[1] `mailto:c.reinke@ukc.ac.uk` `http://www.cs.ukc.ac.uk/people/staff/cr3/`

(accepted for Haskell workshop)

not been felt quite so urgently, because increasingly sophisticated lower-level tools have continued to appear (support still varies between implementations, though, and tools are implementation-specific). Still, there is a discrepancy: if programs are written in a nice high-level language, why do their dynamic aspects have to be studied in low-level terms of stack- and heap-usage? And in the area of debugging, the situation has only just started to improve.

A recent survey [3] compares three tools for tracing and debugging of lazy functional programs: Hat [20], Freja [13], and Hood [7]. All of these systems offer inspection facilities at a level close to the programming language, based on different forms of execution traces, and can be characterised on the basis of the questions they help to answer. Hat [2] takes wrong program output as starting points, enabling users to trace backwards through reduction sequences ("where did this result or output come from?"). Freja supports a technique known as declarative debugging, involving users in a dialogue that narrows down to the source of errors ("this part of your program gives the following result. Is this correct (yes/no)?"). For Hood, it is useful to imagine a data-flow model of functional program execution, with parameters flowing into operators or functions and results flowing out. On this basis, programmers can use Hood to insert probes into their programs to monitor or observe the flow of data at runtime ("what kind of data structure is flowing through here?").

Tracing tools offer high-level views into Haskell program executions. Focusing on different aspects of program behaviour, the existing tools complement each other, but it turns out that they all provide essentially static views of program execution traces, highlighting logical connections between intermediate terms instead of execution dynamics. As a possible extension to Hood, Andy Gill described the "dynamic viewing of structures", stepping through observations using a textual form of visualisation based on pretty-printing [7]. Gill implemented and demonstrated a browser back-end for Hood, based on this idea (the back-end itself is available from the Haskell CVS repository, but it is not supported by the Hood observation library, as released in July 2000; that Haskell library implements the observation combinator by accumulating observations and printing a static view at the end of program runs).

We are here concerned with extending the usefulness of Hood (the most recent of these tools, and also the only implementation-independent one) by adding dynamic views of observation traces. Starting from Gill's idea, and building on the Hood observation library, we have implemented and released a graphical animation back-end for Hood, called GHood. Instead of a dynamic textual visualisation based on pretty-printing, our back-end features a dynamic graphical visualisation, based on a simple tree layout algorithm. After reviewing the main aspects of Hood, this paper gives a brief introduction to GHood's features, demonstrates some of the new applications enabled by GHood by way of two small examples, and summarises our experience so far.

[2] Hat has since been extended considerably, and now supports several models of tracing, implemented on top of a single program execution trace (cf. Section 5.1, as well as [19,20]).

2 Hood – goodbye trace, hello observe

The pseudo-function `trace :: String -> a -> a` – not part of any Haskell language definition, but supported by all Haskell implementations – is supposed to be acting as an identity with a `String`-label. When evaluated, it returns its second parameter, but also prints its label as a side-effect. Reminiscent of the print-statements with which imperative programmers inspect their programs in the absence of proper debuggers, side-effecting output can thus be used to generate a trace of the execution of a Haskell program.

But in the end, unconstrained use of side-effecting input/output operations is no more suitable for debugging than for any other kind of input/output in a lazy functional language. Functional input/output has moved on to more systematic, declarative means of expression, which require to make effects visible in the structure, and thus in the type of programs (Chapter 3 of [16] aims to give a logical reconstruction of the main lines in this development). But this is exactly what prevents the use of these more structured means of input/output for debugging purposes, where one wants to inspect the behaviour of a given program, without having to restructure it into something else first.

Enter Hood (Haskell Object Observation Debugger). One way of understanding Hood is via a line of reasoning similar to that which led to today's functional input/output systems – it is not the idea of side-effecting operations that is at fault, it is their undisciplined use that causes problems. As the requirements of debugging differ from those of standard input/output, a similar line of reasoning will not necessarily lead to similar solutions. In standard usage, input/output is part of the program and should be reflected in its type structure whereas, for debugging purposes, the input/output-operations are part of the workbench used to inspect the program, and the original program should be disturbed as little as possible.

Developing this idea, Hood consists of a fairly complex library with a relatively simple interface. In fact, the type of the major function has not changed much: `observe :: Observable a => String -> a -> a`. Similar to `trace`, `observe` acts as an identity with a `String` label. But the similarities end here – calls to `trace` effectively imitate *imperative* print-statements, whereas calls to `observe` capture the intention behind print-style-debugging (indicating interest in intermediate values) in a *declarative* way, leaving the "how" of capturing and presenting information to the implementation. The combination of `observe` and its observation and presentation library eliminates all the major deficiencies of `trace`:

(i) (a) With `trace`, all information is communicated via the `String` parameter. Programmers have to add code to inspect parts of their program, and to incorporate the inspection results into the `String` labels.

(b) With `observe`, instances of the `Observable` class handle all aspects of program inspection, offering a much more convenient high-level interface. The `String` parameter is just used as a label.

(ii) (a) The extra inspection code needed to feed information into `trace` labels implies non-trivial program modifications, which run the risk of introducing bugs and changing strictness properties in the process.

(b) Predefined instances for most standard types and a combinator approach to user-defined instances of `Observable` imply smaller program modifications and ensure that strictness properties of the program under inspection are not affected by the use of `observe`.

(iii) (a) When evaluated, `trace` immediately attempts to output its label. Under a lazy evaluation strategy, this may cause other traced expressions to be evaluated, and the order of output can be confusing.

(b) Evaluation of `observe` causes information to be captured, but this is decoupled from presentation and output. In Hood, the observation events are post-processed when the observed program has terminated – observations are grouped by their labels into comprehensive summaries, which are pretty-printed as partially-known data structures.

For the full details, readers are referred to the Hood paper and documentation [7,8], but for a two-parameter constructor C in an algebraic data type, the general mechanism can be illustrated by the following pseudo-code:

```
observer (C x y) = λposition -> unsafePerformIO $
  do  sendEvent <observed constructor C at position position>
      return (C (observer x position.0) (observer y position.1))
```

where `observer` is a helper function called by `observe` (initialising *position*), and *position* records the position of the current subexpression in the observed data structure. The definition is strict in the observed (sub-)structure, forcing its evaluation to weak head normal form, but *only if* the weak head normal form of the whole expression is required by the evaluation context. On this occasion, the `observer` generates an observation event, tagged with the position information, wraps any constructor parameters in new observers, and returns the observed constructor to the evaluation context.

All those implementation details are hidden behind suitable monads and combinators, offering a simple user-level interface, and observers for most standard types are predefined. The (predefined) instance of `Observable` for lists may serve to illustrate that it is straightforward, if somewhat tedious, to make new types observable:

```
instance (Observable a) => Observable [a] where
  observer (a:as) = send ":"  (return (:) << a << as)
  observer []     = send "[]" (return [])
```

Using `observe` is equally straightforward (`runO :: IO a -> IO ()` runs an `IO`-script while taking care of observation event processing):

```
import Observe
main = runO $ print $ observe "just a list" [1..4::Int]
```

3 GHood – seeing what your program does

Using a small set of commonly implemented extensions to standard Haskell, Hood instruments existing Haskell implementations to generate observation data during program evaluation, and when the observed program terminates, the stream of observation events is postprocessed and pretty-printed. The result is a portable library that can be used with the full Haskell language.

However, there is more information in the stream of observation events than is utilised in the vanilla version of Hood. Each observation event conveys three kinds of information:

(i) *what* constructor or constant is observed?

(ii) *where* is this part of a data structure located?

(iii) *when* is this part of a data structure observed?

Hood uses location information (*where*) to collate related observations and then pretty-prints the collection of partial information (*what*) about the data structures under observation. The original Hood publication [7] mentions "We have an extension to the released version of HOOD, that includes a browser that allows dynamic viewing of structures." and includes screenshots showing dynamic pretty-printing, but this combination has yet to be released [3].

For GHood, we have taken Gill's idea of using the *when* information of observation events as a basis for animating observations as our point of departure. GHood can be characterised as a new back-end for Hood's observation library – instead of textual visualisation, based on pretty-printing, we have chosen a graphical form of visualisation, based on a simple tree-layout algorithm. The visualisation consists of displaying the structure under observation as a tree, and the animation refines the display whenever an observation event adds information. With the potential exception of functions (see section 4.2), all Haskell types are of the (recursive) sum-of-products kind, and thus have a simple mapping to a tree representation. This is not always the most natural mapping – e.g., GHood currently renders `String`s as binary lists of characters.

3.1 Implementation

We have added extension hooks in the Hood observation library: apart from initialisation and finalisation, these hooks enable additional processing of observation events, either individually, as each observation occurs (extending the `sendEvent` used in `observer`), or on the event stream as a whole, between program termination and Hood's pretty-printing. These hooks give fairly good control over the production and formatting of observation logs and could be used by other postprocessing tools. No further modifications of Hood's obser-

[3] nhc98 comes bundled with pre-release versions of the browser (from the Haskell CVS repository) and the Hood observation library, the latter modified to produce the XML-based input expected by the browser (referred to as THood in section 5.1).

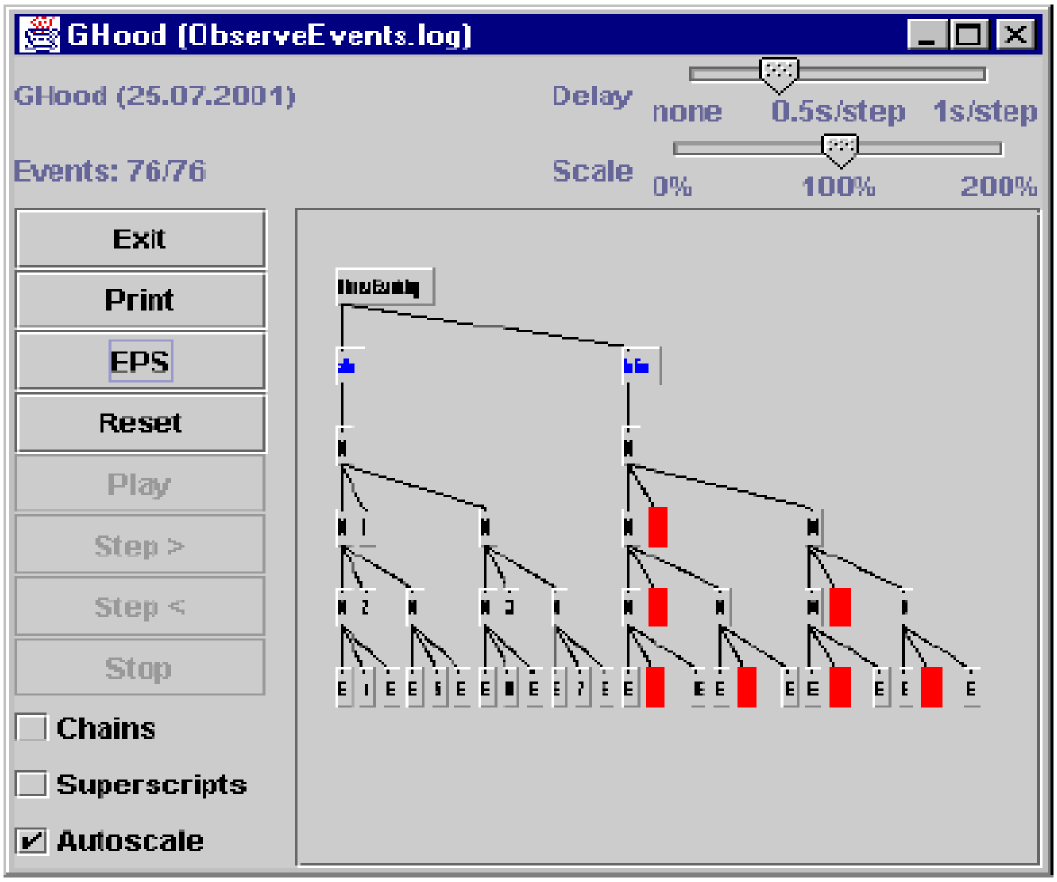

Fig. 1. GHood screenshot

vation library are necessary – the Haskell interface remains unchanged.

Using these hooks, the observation log is made available in a text file. To keep parsing of these logs in our back-end simple, log files consist of one line of plain text per observation event, giving position information and type of observation (observation label, demand for evaluation, constructor or function) for each event, as well as observation-type-specific information (arity and constructor name for observations of constructors, label text for observation labels). Observation logs can then be processed, visualised and animated in our graphical back-end GHood. The hooks give a choice between online and offline generation of external logs, with associated trade-offs: On current machines, the slow-down of programs by file i/o during evaluation in the online variant appears to be more substantial than the extra space usage by the offline version, so the latter is the default. The online version remains useful when GHood is used to debug programs that do not terminate successfully: on ghc, Hood manages to process the observation log anyway, capturing abnormal termination via exceptions, but on other Haskell implementations, only our online version of Hood generates an external log in these cases.

The GHood viewer itself is Java-based, ensuring availability on most platforms that support Haskell implementations, and it can be used with any

Haskell implementation that supports Hood (plus hooks). The graphical user interface (figure 1) is straightforward, comprising a drawing panel in which partially observed structures are displayed using a tree-layout algorithm, and a few buttons to play, stop, reset, and single-step the animation (forwards or backwards), or to print snapshots (printing produces bitmap-style Postscript, so export of vectorised encapsulated Postscript was added for use in print publications). When observation trees get large, they can be scaled down, or the panel can be scrolled, providing survey views or access to parts of the structures under observation. To provide for comprehensible automatic stepping on different platforms, controlable delays have be added between observation events in automatic animation. In the following, we focus on the observation trees, as shown in the drawing panel, but produced by the EPS export.

The main reason for implementing our own viewer was that existing graph drawing tools -as far as they have not gone commercial- appear to be limited to certain platforms or specialised towards pretty, reasonably fast (a few seconds) layout, whereas our application required portability and a quick and simple tree layout for an incrementally updated tree. The only complication resulted from the single-threaded design of Java's GUI libraries (event handlers are scheduled non-preemptively). Fortunately, GHood can be decomposed into two threads (observation tree update and GUI), only one of which requires access to the GUI, but both threads operate on the observation tree. Synchronising the threads on a per-node basis, with an atomic transaction corresponding to the processing of each observation event, appears to give a reasonable compromise between GUI responsiveness and animation progress while avoiding erroneous displays of partially updated trees.

GHood can be used as a standalone Java application or as a Java applet in web pages, and the production and visualisation of observation event logs can be decoupled. This means that online course material, documentation and publications of functional algorithms can be enhanced with dynamic visualisations without requiring a Haskell implementation on the browser side.

3.2 Observations about `unsafePerformIO` *and extension hooks*

In the implementation of `observe`, the non-standard, but commonly implemented, pseudo-function `unsafePerformIO :: IO a -> a` is used to turn an effect (logging an observation event), documented in the type of an expression, into a side-effect, so that the expression tagged with a call to `observe` can be used just as the original expression.

Traditionally, `unsafePerformIO` is seen as a means to *extend programs* with impure operations in such a way that their use, as seen from the evaluating context, can be shown to be uncritical (the prefix `unsafe` is meant to document this proof obligation). In the case of observers, however, the idea is to leave the program under observation entirely undisturbed while *extending the implementation* that runs the program. In other words, `unsafePerformIO` can also be seen as a hook provided in the Haskell evaluation mechanism.

This hook is used in `observe` to instrument the evaluator so that it performs useful logging functions when evaluating structures under obervation. And just as Hood uses an implementation hook to reuse and extend the functionality of existing Haskell implementations, GHood uses hooks in Hood to reuse the observation functionality while extending it for purposes of graphical visualisation. Such implementation extension hooks enormously simplify the implementation of portable tools, and it would seem worthwhile to create and standardise a catalogue of such hooks across Haskell implementations, moving towards portable tools that can plug into different implementations, using only the standardised extension interfaces.

Once it is understood that `unsafePerformIO` functions as an extension hook in the underlying implementation, other uses become possible as well. Instead of just logging the evaluation of some expression, the hook could be used to wait for user input before continuing the evaluation. Such user input could even be used to modify the structure under observation before passing it on to the evaluation context, enabling interactive debugging.

In the specific context of GHood, another useful implementation hook would be to the memory manager, permitting GHood to show when structures become unobservable. According to the documentation (module `Weak` in HsLibs), `addFinalizer :: a -> IO () -> IO ()` should do just that. This operation should associate an IO-script with an expression, so that the script is guaranteed to be run after the expression gets garbage collected. Unfortunately, implementation optimisations currently subvert this operation for most types, rendering it unusable in the general form.

4 GHood applications, by examples

To demonstrate the opportunities opened by GHood, we choose two examples that display non-obvious behaviour but have either been analysed recently (the breadth-first numbering problem) or can be assumed to be well-known to Haskell programmers (the interaction of non-strict evaluation with the use of `foldl` as a pattern for tail recursion). We can thus focus on the visualisation and on the information that can be derived from it. Both of the following subsections can also be seen as examples of how descriptions of functional algorithms can be augmented with animations of program behaviour. To avoid page-filling series of snapshots, we occasionally resort to radio-style textual commentaries of animations that do not easily fit into the static publication format here. Online versions of the examples discussed here are provided on the GHood home page[4], and readers are strongly encouraged to use the online animations side by side with the text here (for completeness, and to give a rough impression of the graphical animations, samples of reduced-size

[4] `http://www.cs.ukc.ac.uk/people/staff/cr3/toolbox/haskell/GHood/`

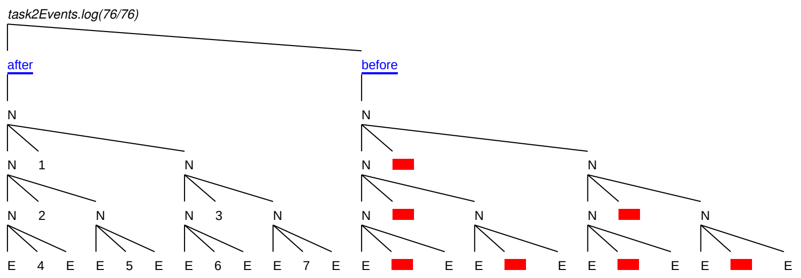

Fig. 2. End-of-run observation of breadth-first numbering

snapshot series are provided in the appendix of this paper).

4.1 Breadth-first numbering revisited

As a first small example, consider the breadth-first numbering problem proposed in a recent functional pearl [14] as "an interesting toy problem that exposes a blind spot common to many –perhaps most– functional programmers". The problem is stated as follows:

> Given a tree T, create a new tree of the same shape, but with the values at the nodes replaced by the numbers 1... |T| in breadth-first order.

Readers who have not come across this problem before are encouraged to try finding a solution for themselves before reading on (our Haskell code is in Appendix A). Originally, we tried to animate our solutions more to gain insight into the practicalities of visualisation than in the expectation to learn anything new about the problem. As a first illustration, figure 2 shows observations of two trees, one `before` and one `after` breadth-first numbering, in the final state of the animation. All observations are grouped under a root node, which also gives the name of the observation file. Below the root node come observation labels (the `String` parameters to the function `observe`), followed by tree-representations of the observed Haskell structures.

The observation labels are underlined and coloured blue[5], constructors and constants are coloured black, unobserved subexpressions (thunks) are shown as red boxes. Thunks under observation are represented as orange boxes with red outlines until their weak head normal form becomes available, and the thunk is replaced by some constructor. The typical lifecycle of a node is from "not yet inspected" (red, closed box) to "under observation, but weak head normal form not yet available" (orange, open box) to some constructor (black constructor label).

Trees are either empty (`E`) or nodes (`N`) with left and right subtree and some label, so the display in figure 2 gives the information expected from the

[5] Presentation scheme changed for publication, to facilitate readability of both colour and greyscale renderings (red and orange appear as dark and light shades of grey, respectively).

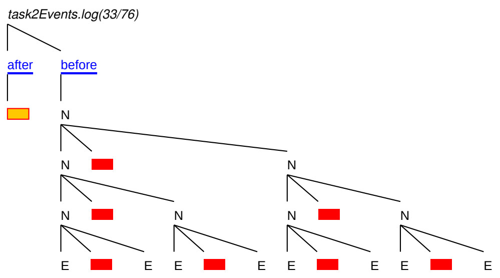

Fig. 3. A middle-of-run strictness problem

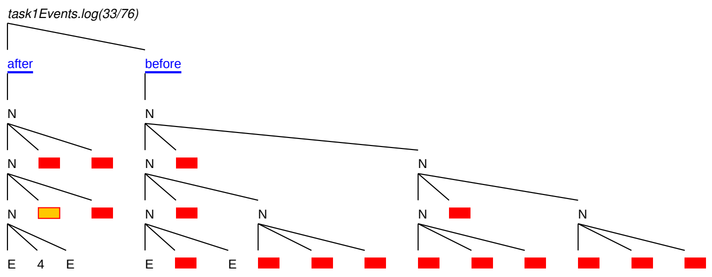

Fig. 4. Strictness problem solved?

problem specification, in that only the shape, but not the node labels of the input tree need to be inspected to construct the resulting tree, in which nodes are labeled with positive integers in breadth-first order.

The surprise came while inspecting intermediate stages of the animation – figure 3 shows an extreme situation in the middle of the run. The thunk which will evaluate to the tree after renumbering is represented as an opened box, indicating that it is being inspected by the evaluation context, but that its weak-head normal form has not yet become available. It has been in that state all the way from just after the start, while more and more of the shape of the input tree has been observed. In other words, this solution has an extreme strictness problem, inspecting parts of the input long before they should be needed! Only the very next step will replace the thunk under inspection by a node labeled `N`, with three unobserved thunks as subnodes, so no part of the result tree becomes available for observation until after all observations of the input tree shape have taken place.

Once the animation had so drastically brought this strictness problem to our attention, improving the program was not too difficult. Choosing roughly the same stage in a run of the modified program, the intermediate observation in figure 4 shows the difference quite clearly (watching the observed structures unfold dynamically during animation, it is almost impossible not to notice the difference between the two programs): parts of the resulting tree have become available for observation, right down to the first complete non-trivial sub-tree at the left, while still not all of the input tree shape has been observed.

In spite of the drastic improvement, a careful inspection of the animation for the new version shows that it still does not behave as one might expect. The relabeled tree is observed in depth-first order, whereas the input tree is observed in breadth-first order. At first, that looks reasonable: the problem specification calls for a breadth-first traversal of the input tree, and the printing routine traverses the result in depth-first order. On second thought, though, *only the computation of the new labels should depend on a breadth-first traversal of the input*, and printing the result should give the whole leftmost branch of the tree before inspecting any node labels.

At this point, we need to explain our approach to the problem and the differences between the versions. In our earliest attempts, we did indeed experience the blind spot discussed by Okasaki, though not for the reasons listed by him. Instead, our road-block was that any solution seems to involve two different views of the input trees: whereas the problem specification clearly calls for a breadth-first traversal, the easy way to describe a recursive algorithm over the trees follows their recursive structure – in depth-first order! Our very first solution side-stepped the issue in an overcautiously systematic approach, restructuring the input tree into a list of levels, then doing the relabeling (straightforward in this form), and finally rebuilding a tree of the original structure, with the new labels. But once we had managed to find at least one solution to Okasaki's problem, and identified our own blind spot on the way, we then sought to get rid of the blind spot by constructing a more suitable solution. This led to the variants described in the present paper (the original brute-force solution had similar strictness problems).

The new approach does not impose a breadth-first traversal on the input tree, but instead follows its natural recursive structure, generating a pool of "things to do" on the way. The tasks -one for each subtree- are connected by data-dependencies which represent the breadth-first traversal constraint, and it is left to the inspection of the result tree to actually cause those tasks to be evaluated, in a co-routine-like fashion. In other words, the producer of relabeled trees consumes the input trees in a depth-first traversal, and any consumer of the result tree will implicitly (by the virtues of lazy evaluation and the data dependencies set up by the producer) cause a breadth-first traversal to take place. This decoupling of the two conflicting traversals solves our blind-spot problem and gives a concise first variant of a solution, called `task1` (figures A.2, A.6, 4).

After reading Okasaki's comments [14], we noticed that his suggestion about replacing two-way queues by unidirectional queues in languages that do not support matching on both ends applied to our task pool (represented as a list, with an awkward use of `splitAt` to pattern-match at its back end). So `task1` became `task2` (figures A.3, A.5, 3) – and acquired the extreme strictness problem described earlier: Okasaki's workaround maintains queues

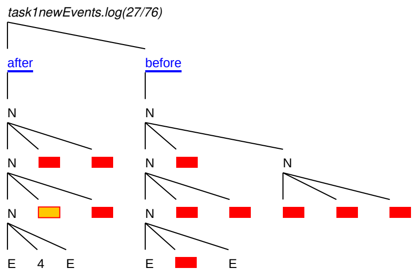

Fig. 5. Strictness problem solved!

in reversed order (so that elements can be taken from the output ends using pattern-matching), which happens to put the relabeled top node at the very end of the queue, so that the whole task queue has to be processed -and the whole input tree be observed- to get to the very first node of the result tree.

Switching back to our original variant got rid of this problem, but left another, only slightly more subtle strictness problem: to show the result tree up to the first label, as in figure 4, it should not be necessary to observe three levels of nodes in the input tree. The node labeled `4` in the result is the first at level three, so observing two levels of the input tree should suffice to compute the label! Perusing the animation again gives the embarrassing insight: just traversing the structure of the result tree seems to trigger the breadth-first traversal of the input tree, even before any labels are inspected. And indeed, this variant takes the result structure from the task pool that was set up to enforce the breadth-first traversal. Separately passing the structure of the input tree and filling in the labels computed on demand solves this problem, and the animation of our final variant, `task1new` (figures A.4, A.7, 5), exhibits a nice, demand-driven pattern of observations.

Note that this kind of dynamic strictness problem, where parts of inputs are demanded too early, differs from the kind of problems that could be investigated using static strictness information (is a part of input ever demanded or not at all?). If the iteration bounds that guarantee termination of a strictness inference system can be increased in cases where termination is obvious for other reasons, the best information such a system could give corresponds to that deducible from figure 2. But that information is the same for all variants of the solution!

4.2 A well-known strictness problem

Recursive algorithms over lists can often be expressed more concisely as folds, avoiding explicitly recursive definitions. For lists, there are two standard fold operators, `foldr` and `foldl`, which combine the list elements by right- and left-associative operators, respectively. More generally, a fold operator replaces constructors in a parameter structure by operators of appropriate arity, thus

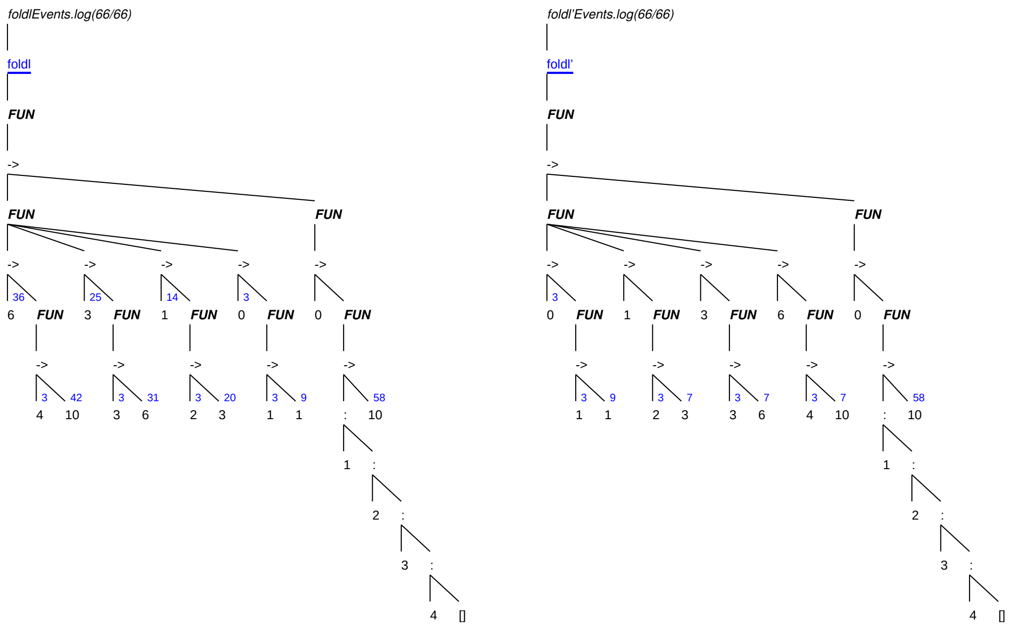

Fig. 6. `foldl` versus `foldl'` – tail recursion with (non-)strict accumulator

expressing the recursive structure of the algorithm in terms of the recursive structure of its input. Viewed in these more general terms, `foldr` expresses a standard recursion along the list structure, whereas `foldl` expresses a tail recursion with an accumulator. Such tail recursions are usually associated with constant stack-usage.

```
foldr op c []     = c
foldr op c (x:xs) = x `op` (foldr op c xs)

foldl op c []     = c
foldl op c (x:xs) = foldl op (op c x) xs
```

As many Haskell programmers discover for the first time in more complex programs, this idea does not quite work – for large inputs their programs can run out of stack space in spite of the careful use of tail recursion! This is quite a common experience, and so it seems worthwhile to see how much of the problem reveals itself by careful analysis of an example, using only the graphical animation of observations. The reader should keep in mind that this subsection is not concerned with new aspects of folds – rather, it serves to illustrate the novel ways of explaining more or less well-known properties of functional algorithms, made possible by visualisation tools such as GHood.

Figure 6 (left) shows an end-of-animation snapshot of the call:

```
observe "foldl" foldl (+) 0 [1..4::Int]
```

To make up for the lack of animation here, nodes in this figure are annotated with superscripts giving the number of observation events between the beginning of their observation and the availability of their weak head normal form (shown only if that number exceeds one). As in Hood, the observed part of a function is presented as a finite map of input/output pairs. Those pairs are labeled with arrows here, so `FUN{6->FUN{4->10},3->FUN{3->6},..}` represents a function f that, when applied to `6`, returned a function that, when applied to `4`, returned `10` (f was also applied to `3`, and returned a function that, when applied to `3`, returned `6`). The overall picture tells us that `foldl` is a ternary function, mapping a binary function (itself applied four times, as there are four pairs in its map) to a function, that maps the integer `0` to a function, that maps the list `[1,2,3,4]` to the integer `10`.

In the animation, several phases can be distinguished. First, `foldl` itself is observed to reveal its arity, then evaluation demands that its result be observed (the box corresponding to this thunk is opened). Before this becomes available, the spine of the input list is observed in full, which in itself is a stumbling block in many programs operating on lists of substantial size: the whole length of the input list is created in memory before any other computations take place (the spine of the list can be collected immediately, but the thunks for its elements take up space, even though these elements are not yet about to be inspected). Using `foldr` would avoid this problem, at the expense of linear stack usage.

Next, observation of the result of applying the binary operator is demanded, leading to a demand for the first parameter of this application. This, in turn, demands observation of the result of another application of the operator, and so on, creating a chain of thunks under observation until the demand for the first parameter of the fourth application is fulfilled by observing the second parameter to `foldl`. After that point, the chain unwinds step by step, demanding successive observations of all input list elements before, finally, the result of the call to `foldl` becomes observable.

Returning to the annotated snapshot in figure 6 (left), we see that some 58 events passed during observation of the final result, `10`, and that the chain consisted of computing, starting in this sequence `6+4->10`, `3+3->6`, `1+2->3`, and `0+1->1`, and terminating in reversed order, taking 42, 31, 20, and 9 observed steps, respectively. In summary, the call to `foldl` was indeed tail recursive, but it only observed the spine of the input list and delivered a thunk involving the list elements as an interim result. Evaluating this thunk then unfolded another, implicit recursion (corresponding to the evaluation of a nested arithmetical expression) with just the kind of linearly growing stack-usage (the chain of opened boxes) we wanted to avoid.

The obvious countermeasure is to force evaluation of the accumulator to avoid this split into a tail-recursive thunk construction and a not tail-recursive evaluation of that thunk, e.g., by using the call-by-value applicator `$!`:

```
foldl' op c []     = c
foldl' op c (x:xs) = (foldl' op $! (op c x)) xs
```

The new annotated end-of-animation snapshot in figure 6 (right) already indicates a major change. With the exception of the final result, no more than 9 observation events occur between the beginning of a node observation and the availability of its weak head normal form. As those delays roughly correspond to stack usage, getting rid of the ghost-recursion has established the bound on stack usage that was the original goal. The order of applications of the binary operator seems to have changed as well.

Going through the full animation sequence shows further differences: the spine and elements of the input list are now inspected in a stepwise fashion, interleaved with applications of the binary operator, now in the sequence `0+1->1`, `1+2->3`, `3+3->6`, and `6+4->10`. This ordering ensures that intermediate results are already available when demanded by the next application and is the result of forcing the evaluation of the accumulator. So, not only has the unbounded use of stack space been avoided, but a space leak (observing the full spine of the input list -thus creating implicit thunks for all elements- long before its elements are inspected) has been plugged as well.

4.3 Summary, and further examples

The examples in this section have been chosen to be small, relatively well-known, yet displaying interesting behaviour and illustrating different aspects of GHood. In the case of breadth-first numbering, animation of observations was used during algorithm development and helped to discover unexpected properties of early program variants, as well as pointing to the source of the problems. In the case of `foldl`, the algorithm and problems are usually considered to be well-known, but resurface with surprising reliability, and the animation was used to demonstrate and explain how a tail-recursive function could still lead to linear resource usage for intermediate structures. The examples differ in another notable aspect: for breadth-first numbering, the tree layout imposed by GHood naturally matches the trees in the problem, whereas the tree layout is rather less natural for `foldl`.

In both examples, observation of unexpected behaviour could be traced back to problems and led to modifications of the programs observed. It would be misleading, though, to assume that the main use of GHood was in debugging – it just happens that understanding what a program does can be a useful asset in debugging (declarative debugging, as in Freja [13], suggests that such an understanding is not always necessary). For a nice example of how animation of observations can aid program comprehension outside of debugging, readers are again referred to the GHood home page: the online examples include an animated observation of Colin Runciman's Haskell implementation of the "wheel sieve" algorithm for generating prime numbers [17]. The program is considerably more complex than the examples discussed here, and the animation provides a nice complement to the discussion in the JFP paper.

5 Evaluation, related and further work

5.1 Experience, feedback, and evaluation

After some internal testing at UKC, first versions of GHood were made available to the Haskell community in January 2001. Since then, we have received a lot of positive feedback, very few feature requests, and problem reports have mostly been limited to problems with the Java 2 runtime installations on which our viewer depends. This suggests that the tool, while far from perfect, is already considered good enough to fill its niche. In other words, while our current users might welcome refinements of the current features, such improvements will not be considered essential unless they reflect changes in the basic approach. Our plans for GHood are thus limited to completion of the modifications currently under development (see below), to be incorporated in a final release later this year.

In March, we also had the opportunity to visit [6] the functional programming group in York and take part in a repetition of the usability study described in [3], with updated variants of the same tools. Though limited to case studies in debugging, the experiment provided a host of useful feedback and ideas. The most important outcome was that the tools (Freja, Hat, and GHood) had actually managed to explore, and partially fill, *different* niches in the area of debugging Haskell programs. Each tool was useful for debugging, but each tool was useful in a different way, and more than once, we would have wanted an easy way to switch from one tool to another – not only with the same Haskell implementation, but in the same debugging session, taking the current debugging state and investigating it from a different perspective. As the Hat trace seems to contain most of the information needed for each of the tools, the York group has now started to move in that direction, and first results are visible in the new Hat toolsuite bundled with the just-released nhc98-1.04 [20,19] (the suite includes a variant of Hood-style observation, implemented on top of Hat's redex trails instead of Hood's observation library).

In the following, we distinguish between Hood -the Haskell library released in July 2000, GHood -the graphical back-end for Hood described in this paper, and THood, by which we refer to the version of Hood that comes bundled with nhc. The latter includes Gill's textual browser from the Haskell CVS repository, and a pre-release version of the Hood library, modified to generate the XML input expected by the browser. In its current pre-release form, THood suffers from differences to the released Hood (this is easily repaired) and from a lack of automated animation (only single-stepping forwards and backwards and jumps to beginning and end of observations are provided).

All Hood backends inherit the core functionality and some limitations from the library. In practice, the most annoying limitation is the need to inspect and modify the source code in order to import the module `Observe` and to

[6] This visit was supported by EPSRC grant number GR/M81953.

define instances of the class `Observable` for all non-standard data types, as far as values of these types need to be observed (this set of types needs to be closed with respect to embedded types). Further modifications include a call to `runO` in `main` and running the implementation with options indicating extensions beyond Haskell 98. In contrast to calls to `observe`, which indicate programmer intentions, these modifications are implied, boring, and error-prone. Even though errors introduced in the process are isolated from the program, easily spotted and fixed, they could be avoided entirely by automating these tasks (Malcolm Wallace suggested using Drift to generate the instances of `Observable`). The main problem with calls to `observe` is to identify program positions where such calls will provide useful information.

The York experiment was limited to debugging, and as far this is concerned, the most useful feature of GHood surprisingly turned out to be information about what is not there: again and again, unevaluated thunks provided shortcuts to spotting bugs (one example was a bugged compiler in which a symboltable lookup managed to return values without the symboltable ever being observed). Both Hood and THood indicate unevaluated thunks as simple underscores, and neither shows temporal relations between different observations (Hood has no animation, THood treats observations under different labels separately). GHood, in contrast, displays unevaluated thunks in clearly visible red, and animates all observations under a single root node, facilitating comprehension of interrelationships. Deriving information from non-available data (thunks) seems to take some getting-used-to, though: the important connection is that Hood-based tools show what the program sees, so if GHood does not show the value of a thunk, there is no need for the debugger to know the value, simply because the program never asks for that value.

Of the tools in the experiment, GHood seemed to cope best with large structures, but it was not entirely without problems in this regard: scaling (both in time and in space) is useful because the graphical structure supports orientation even when textual labels are no longer readable, but because of this graphical structure, small structures are not represented as compactly as in Hood or THood. If THood would be extended with automated animation, it would be at an advantage for small, not inherently tree-like structures, such as the observation of `foldl`. For slightly larger observations, such as the lazy wheel sieve, THood's compact representation can no longer entirely make up for the lack of scaling (scaling the pretty-printed representation to point size would give a graphic represention without much structure, but it would be interesting to compare that representation to GHood's).

GHood extends Hood, so the static pretty-printed observations are still available to complement the dynamic graphic visualisation, but some graphs, especially `String`s, should be represented more compactly, to improve readability. Another problem concerns navigation in large structures: the standard two-scrollbars solution is rather unsuited for concurrently navigating in both dimensions and needs to be replaced, and although both survey views and zooming to details are currently supported, they should not exclude each other. On a related note, we should point out that Hood-based animation tools not only enable programmers to focus on the parts of the program to be observed, decoupling program size from the size of observations. To some extent, the level of abstraction at which to animate program observations can also be controlled: at the level described in section 3, entirely different approaches to the breadth-first numbering problem, such as the brute-force level-and-reconstruct approach, will display similar behaviour, even though their behaviour would differ substantially under more detailed observations.

Other issues include online versus offline generation of observation logs (cf. section 3.1), observability of η-conversion (`observe "f" f` shares a single observation label between all uses of f, whereas `\x->observe "f" f x` creates separate observation labels for each call), the need to remove calls to `observe` to avoid clutter (GHood should be extended to permit selective observation), and the need for "packaging" of observations, preserving the connection between them (for instance, several local variable bindings in a function body).

As mentioned earlier, the approach taken by Hood and GHood does not in principle exclude interactive debugging, and the February 2001 release of Hugs (`www.haskell.org/hugs`) offers support for a built-in variant of Hood, called HugsHood, which heads in this direction by supporting breakpoints. Similarly, there is no fundamental reason against online visualisation (during program execution) but our current offline approach to visualisation has opened new application areas beyond debugging.

5.2 Other related work

The idea to visualise and animate the execution of functional programs in order to gain insights into their behaviour is an old one. For an overview of the problems and opportunities see Sandra Foubister's thesis [5]. We are not aware of a survey covering this area, but various proposals and even implementations have been put forward, including Foubister's "hint" tool and an animation of a G-machine implementation using the graph layout tool daVinci [15], not to mention proposals for specially designed visual functional languages. More recent incarnations of the idea include a graphical debugger/tracer in the Curry Integrated Development EnviRonment CIDER [11], and the Kiel Interactive Evaluation Laboratory [2] for a simple first-order subset of ML. For completeness, text-based navigation through reduction sequences should also be mentioned, as in the DrScheme environment [4] or in the reduction systems in the Berkling and Kluge tradition [10].

Animation of observations in GHood is distinctly different from traditional text- or graphics-based animation or navigation of reduction sequences. Comparing our experience with GHood and with textual single-stepping through reduction sequences, as afforded, e.g., by the reduction systems developed by Kluge et. al. [10,6], we find both disadvantages and advantages.

At first, the disadvantages seem overwhelming: without any extra effort by programmers, reduction systems provide a direct experience of the operational semantics, as well as navigation, editing, and selective reduction of parts of intermediate programs in a reduction sequence. GHood, as a back-end for Hood, only animates observations of intermediate structures. Observations are approximations of weak head normal forms of those intermediates, and the animation shows the sequence in which parts of structures under observation are inspected. This allows only indirect conclusions about the program behaviour. In practice, it can be rather difficult to try and infer the algorithm from the visualisation alone but, starting with a conjecture or some approximate understanding of the program behaviour, it tends to be straightforward to confirm or refute such hypotheses in the visualisation.

On the positive side, graphical visualisation is more suitable for overviews of larger programs and of animation sequences, where textual information is no longer readable. The observational approach also makes it easier to focus visualisation on interesting aspects of program behaviour, excluding both unobserved parts of programs and intermediate expression representations on the way to weak head normal forms. Nevertheless, observation graphs for realistic programs grow quickly, demanding further work on the user interface.

The general problem faced by developers of execution monitoring tools is the need to use (and most likely create) specially instrumented implementations. As a consequence of the efforts involved, such specialised implementations tend to support only small subsets of the original languages, visualisation often takes place at the implementation level, and the specialised implementations do not evolve with the language and its standard implementations. Tools based on specialised implementations are by definition not portable, and if separate implementations are needed for normal and for visualisation use, differences in evaluation mechanisms may occur.

Another alternative is to use a separate evaluator with built-in execution animation facilities and to provide mappings between subsets of that evaluator's language and subsets of the language to be extended with execution monitoring. Wolfram Kahl has demonstrated this approach with his term-graph-based program development and transformation environment HOPS [9], but it means that two evaluators, their languages, and the mapping between them have to be kept in synch, not to mention portability issues.

Hood avoids all these problems by using a commonly implemented implementation hook (`unsafePerformIO`) to instrument existing Haskell implementations, reusing and extending their functionality. The resulting library is portable and can be used with the full Haskell language. GHood uses hooks in Hood to reuse the observation functionality while extending it for purposes of dynamic graphical visualisation, using Java as a widely available implementation platform. Reflecting on the success of these hook-based solutions, implementation hooks turn out to be (application-specific) residues of more general meta-programming infra-structure.

In language communities with successful tool-building traditions, such as Lisp, Prolog, and Smalltalk, tool development seems to rely on well-developed infra-structures for meta-programming and reflection. At a prototype stage, the key idea is to write a meta-interpreter (between a few lines and a page of code for these languages) that reuses existing implementation functionality, and then to instrument the meta-interpreter for purposes of monitoring (animation in our case). Successful prototype tools can then be implemented more efficiently, often using standard techniques. To achieve efficiency, the meta-interpreter should delegate standard functionality to the standard evaluator with as little overhead as possible. In such embedded meta-interpreters, only the extra functionality (e.g., for program monitoring) incurs interpretative overhead, and if suitable extension interfaces to the standard evaluator are available (aka reflection or introspection capabilities), the meta-interpreter *becomes* the standard interpreter, instrumented via its extension hooks.

In the context of declarative debugging, Naish and Barbour [12] have used this idea to design a "portable lazy functional declarative debugger" which could be implemented in the functional language to be debugged, assuming a single impure primitive, called `dirt` (display intermediate reduced term).

Haskell neither supports reflection [7] nor does it offer well-documented interfaces to implementation functionality (cf. the SML/NJ `Compiler` structure [1]), or other typical parts of a meta-programming infra-structure. Its syntax is more complex than Lisp's S-expressions, and reusable parsers for full Haskell have only recently started to appear, but the parsers in the various Haskell implementations remain practically unaccessible; all Haskell implementations internally build up a symbol-table, associating identifiers with attributes, such as types or strictness, but there is no standard interface by which Haskell programs could load a Haskell program and query the symbol-table information.

6 Conclusions

GHood is a new back-end for Hood, providing graphical visualisation and animation of Haskell program execution. Unlike traditional approaches to graph reduction animation, GHood is not based on a special-purpose implementation, but extends and reuses existing Haskell implementations, via Hood. The visualisation itself is also different, in that it does not animate reductions of terms to normal form, but inspection of terms by their evaluation contexts: instead of evolution of a term through intermediate representations, an animation shows refinement of information about a term in a single representation.

Portable tools such as Hood and GHood depend critically on being able to instrument and thus reuse existing Haskell implementations by means of extension hooks, and the ease with which tool implementers can reuse existing

[7] How to do this properly in a statically typed, pure, and non-strict functional language is another research direction that would merit more attention

implementation functionality has an important impact on the development of tools for Haskell. We suggest that a common (implementation-independent) infra-structure for meta-programming and reflection in Haskell, with standard interfaces to implementation functionality, could improve the basis for Haskell tool development, and that both the general framework and specific implementation extension hooks should become a focus of research.

In the present paper, we have focussed on illustrating the way in which GHood can be used to help comprehension of Haskell program behaviour, using small examples from everyday practice. Our own experience and feedback from users shows that dynamic observation of intermediate structures is a useful addition to the Haskell programmer's toolbox. Although the 'd' in Hood stands for "debugger", we prefer to see GHood as a workbench: Haskell programmers can use it to set up and perform experiments involving dynamic aspects of their programs. Such experiments can be used to validate theories of program behaviour or they can deliver the data points from which such theories can be abstracted. For both uses, experiments have to be set up and the data be interpreted carefully, so Hood and GHood are tools that can inform thinking about programs, but they cannot replace such thinking.

We hope to see GHood or similar tools for the visualisation of functional program behaviour used in education (online course material), documentation, and publication (online supplements to articles on functional algorithms). Instructors might want to consider the motivational aspect as well – several correspondents commented the first pre-releases with the words "GHood is cool!". Another correspondent remarked "finally, I can *show* my colleagues what non-strict evaluation means".

References

[1] *Standard ML of New Jersey*, `http://www.smlnj.org`.

[2] Berghammer, R. and M. Tiedt, *Kiel Interactive Evaluation Laboratory*, Technical report, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel (1999), `http://www.informatik.uni-kiel.de/~progsys/kiel.html`.

[3] Chitil, O., C. Runciman and M. Wallace, *Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs*, in: *Proceedings of the 12th International Workshop on Implementation of Functional Languages, Aachen, Germany, September 4th - 7th 2000, LNCS 2011*, 2001, pp. 176–193.

[4] Clements, J., M. Flatt and M. Felleisen, *Modeling an Algebraic Stepper*, in: *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001*, Lecture Notes in Computer Science **2028** (2001).

[5] Foubister, S., "Graphical Application and Visualisation of Lazy Functional Computation ," Ph.D. thesis, Department of Computer Science, University of York (1995).

[6] Gärtner, D. and W. Kluge, *π-RED$^+$: An interactive compiling graph reduction system for an applied λ-calculus*, Journal of Functional Programming **6** (1996).

[7] Gill, A., *Debugging Haskell by Observing Intermediate Data Structures*, in: G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop* (2000).

[8] Gill, A., *The Haskell Object Observation Debugger*, `http://www.haskell.org/hood/` (2000).

[9] Kahl, W., *HOPS – The Higher Object Programming System*, `http://ist.unibw-muenchen.de/kahl/HOPS/`.

[10] Kluge, W. E., *A User's Guide for the Reduction System π-RED$^+$*, Technical Report 9419, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel (1994), `http://www.informatik.uni-kiel.de/~base/`.

[11] Koj, J., "Eine graphische Programmierumgebung für deklarative Programmiersprachen," Master's thesis, RWTH Aachen, Germany (2000), the Curry Integrated Development EnviRonment `http://www.informatik.uni-kiel.de/~pakcs/cider/`.

[12] Naish, L. and T. Barbour, *Towards a portable lazy functional declarative debugger*, Australian Comp. Science Communications **18** (1996), pp. 401–408.

[13] Nilsson, H., "Declarative Debugging for Lazy Functional Languages," Ph.D. thesis, Department of Computer and Information Science, Linköpings Universitet, S-581 83, Linköping, Sweden (1998).

[14] Okasaki, C., *Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design* , in: *International Conference on Functional Programming ICFP'2000, Montreal, Canada*, 2000, pp. 131–136.

[15] Panne, S., *Graph Visualization with daVinci*, `http://www.pms.informatik.uni-muenchen.de/mitarbeiter/panne/haskell_libs/daVinci.html`.

[16] Reinke, C., "Functions, Frames, and Interactions – completing a λ-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments," Ph.D. thesis, Faculty of Engineering, Christian-Albrechts-University, Kiel (1997), Technical Report 9804, Institute of Computer Science, May 1998, `http://www.cs.ukc.ac.uk/people/staff/cr3/publications/phd.html` .

[17] Runciman, C., *Lazy wheel sieves and spirals of primes*, Journal of Functional Programming **7** (1997), pp. 219–225.

[18] Wadler, P., *Functional Programming: Why no one uses functional languages*, SIGPLAN Notices **33** (1998), pp. 23–27.

[19] Wallace, M., O. Chitil, T. Brehm and C. Runciman, *Multiple-View Tracing for Haskell: a New Hat*, in: *ACM SIGPLAN Haskell Workshop, Firenze, Italy*, 2001.

[20] York Functional Programming Group, *Hat - The Haskell Tracer*, `http://www.cs.york.ac.uk/fp/hat/` (2001).

A Source code and animation sequences

A note on the use of animation sequences: online animations for all examples are available on the GHood home page. Snapshot samples of animation sequences are included in this appendix for archival reasons, but as the static medium cannot portray the advantages of dynamic visualisation, the online animations should be preferred, if at all possible. Readers without access to the online animations will find it helpful to print or display this appendix separately from the main text, so that they can see both side by side without having to jump back and forth.

```
import Observe

data Tree a = E | N (Tree a) a (Tree a) deriving (Show)

instance Observable a => Observable (Tree a) where
  observer E         = send "E" (return E)
  observer (N l x r) = send "N" (return N << l << x << r)

main = printO $ observe "after" $ bfnum  $ observe "before" xxx
  where { xxx = N xx 2 xx; xx = N x 1 x; x = N E 0 E }
```

Fig. A.1. task-based breadth-first numbering, common prefix

```
-- for non-empty tree, fork out immediate subtrees (l,r) as
-- new tasks, build result from sub-results (l',r')
task n ~[]      E         = (n  ,[]   ,E)
task n ~[l',r'] (N l x r) = (n+1,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = t':rs'
  where
    (n',tp',t') = task n r t
    ts'         = taskM n' (ts++tp')
    (rs',r)     = splitAt (length ts) ts'

bfnum t = head $ taskM (1::Integer) [t]
```

Fig. A.2. `task1` – task-based breadth-first numbering, first attempt

```
task n ~rs          E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')

bfnum t = head $ taskM (1::Integer) [t]
```

Fig. A.3. `task2` – task-based breadth-first numbering, more elegant?

```
task n ~[]      E         = (n  ,[]   ,E)
task n ~[l',r'] (N l x r) = (n+1,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = t':rs'
  where
    (n',tp',t') = task n r t
    ts'         = taskM n' (ts++tp')
    (rs',r)     = splitAt (length ts) ts'

fillIn E         ~E            = E
fillIn (N l _ r) ~(N l' x' r') = N (fillIn l l') x' (fillIn r r')

bfnum t = fillIn t $ head $ taskM (1::Integer) [t]
```

Fig. A.4. `task1new` – task-based breadth-first numbering, improved!

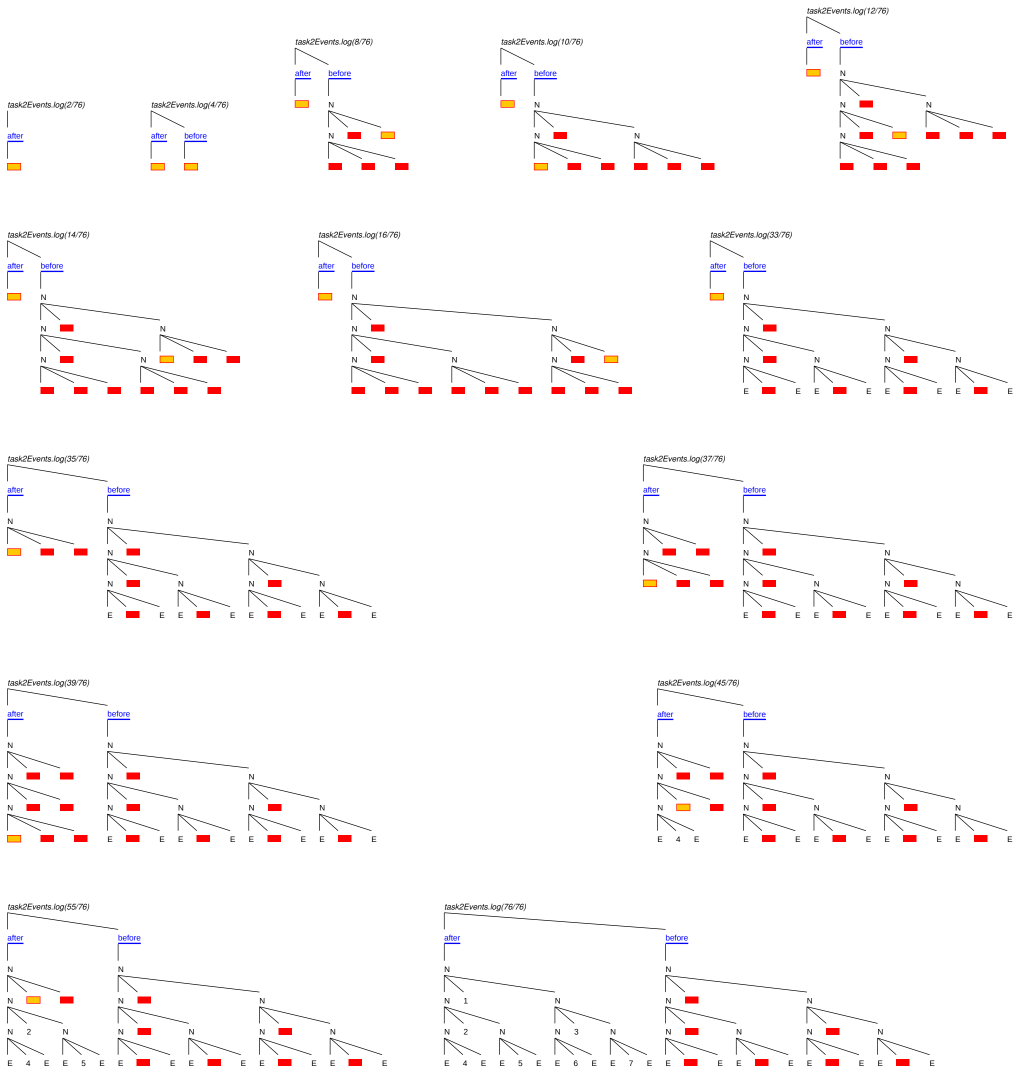

Fig. A.5. `task2`: steps 2, 4, 8, 10, 12, 14, 16, 33, 35, 37, 39, 45, 55, and 76

Fig. A.6. `task1`: steps 2, 4, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, and 76

Fig. A.7. `task1new`: steps 2, 4, 5, 7, 9, 11, 13, 15, 17, 19, 21, . . . and 76

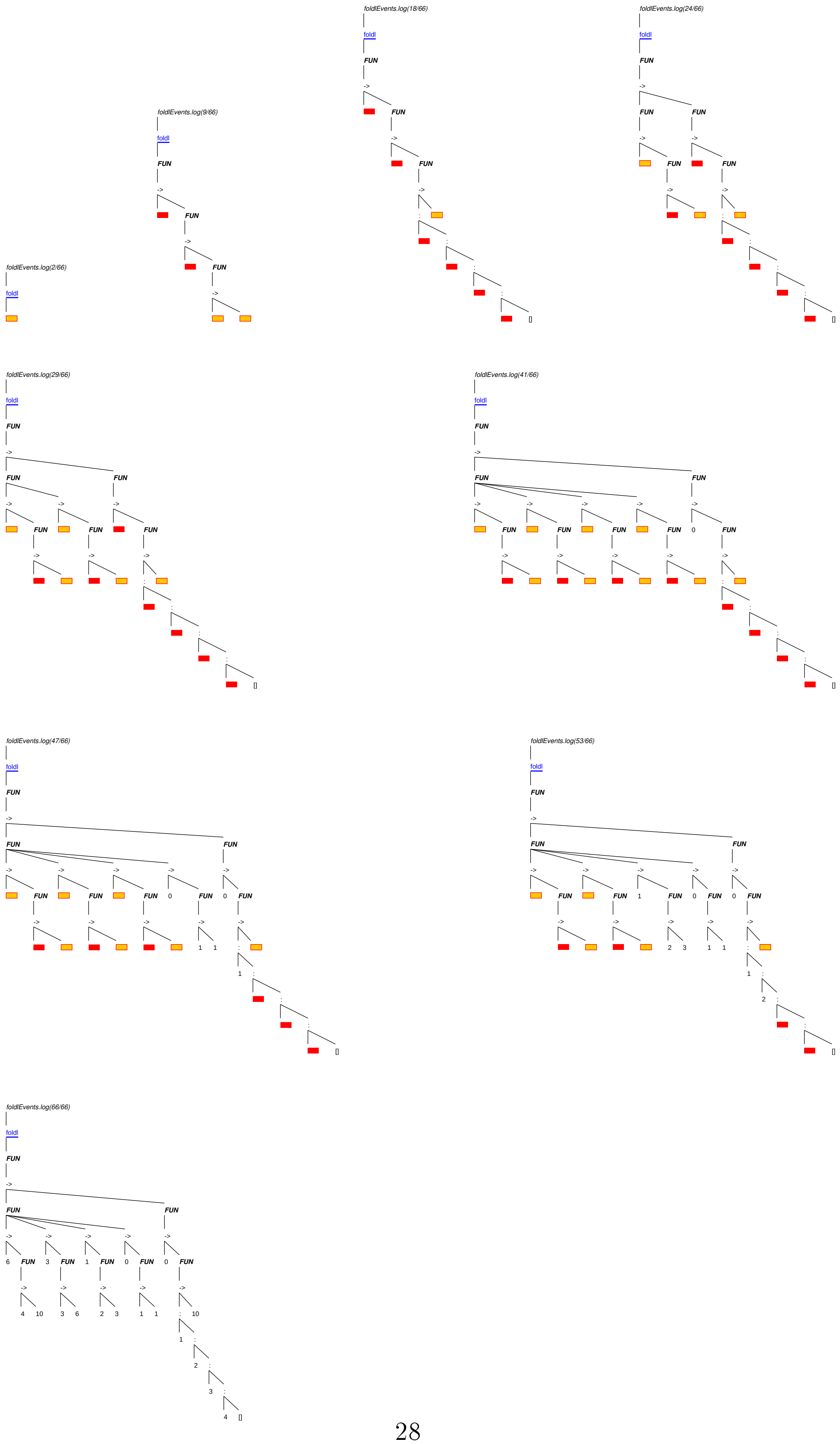

Fig. A.8. `foldl` steps 2, 9, 18, 24, 29, 41, 47, 53 and 66

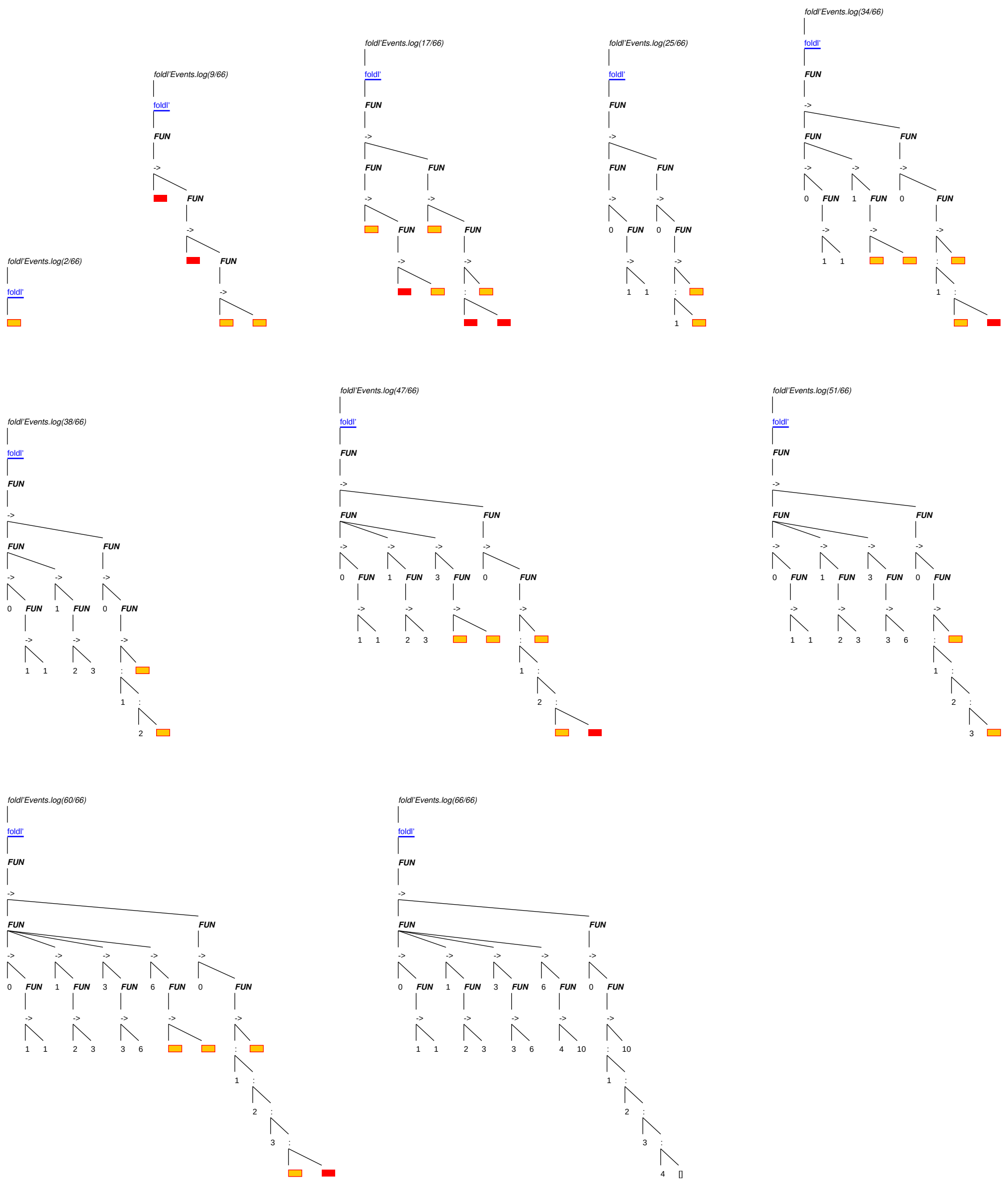

Fig. A.9. `foldl'`: steps 2, 9, 17, 25, 34, 38, 47, 51, 60 and 66