Formalizing the Debugging Process in Haskell

Alberto de la Encina, Luis Llana, and Fernando Rubio

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
e-mail: {albertoe,llana,fernando}@sip.ucm.es

Abstract. Due to its absence of side effects, it is usually claimed that reasoning about functional programs is simpler than reasoning about their imperative counterparts. Unfortunately, due to the absence of practical debuggers, finding bugs in lazy functional languages has been much more complex until quite recently. One of the easiest to use Haskell debuggers is Hood, whose behavior is based on the concept of observation of intermediate data structures. However, although using Hood can be simple when observing some structures, it is known that it can be hard to understand how it works when dealing with complex situations. In fact, the author of Hood recognizes that it is necessary to formalize its behaviour to better explain what should be expected, and also to allow to check whether the different implementations are working properly.
In this paper, we introduce debugging facilities in the STG abstract machine. Our goal is to obtain debugging information as close to the one obtained by the Hood debugger as possible. By extending the STG abstract machine, we do not only provide a formal framework to the debugging process, but we also provide an alternative method to implement debuggers.

Keywords: Functional programming, debugging, abstract machines.

1 Introduction

The debugging of lazy functional programs is currently an active area of research. Although not much attention was paid to it in the past (see e.g. [20]), during the last years there have been several proposals for incorporating execution traces to lazy functional languages. In particular, we can highlight the work done with Hat [19, 21], HsDebug [5], the declarative debuggers Freja [10, 11] and Buddha[17], and specially the work done with the Haskell Object Observation Debugger (Hood) [6, 18]. All of them are designed to be used with the language Haskell [15], the *de facto* standard in the lazy-evaluation functional programming community.

The approaches followed in each of the previous debuggers are quite different, both from the point of view of the user of the system and from the implementation point of view. For instance, from an implementation point of view, most of them strongly depend on the compiler being used, while that is not the case in Hood. From the user point of view, Freja and Buddha are question-answer systems that directs the programmer to the cause of an incorrect value, while Hat allows the user to travel backwards from a value along the redex history leading to it. In this paper we will not concentrate on those differences (the interested reader can find a detailed comparison between Freja, Hat and Hood in [2], while [12] presents a common framework to describe all of them). In contrast, we will concentrate on how to improve one of them.

Among all of the Haskell debuggers, Hood has an interesting advantage over the rest, as it is the only one that can be used with different Haskell compilers. The reason is that it is implemented as an independent library that can be used from any Haskell compiler, provided that the compiler implements some quite common extensions. Hood can currently be used with the Glasgow Haskell Compiler, Hugs98, the Snowball Haskell compiler, and also with nhc98. Due to its portability, Hood has become one of the most used Haskell debugger.

The way Hood works is relatively simple. First, the programmer instruments the program marking the variables he wants to observe and, after finishing the execution of the program, the system produces a printing of their final value. Let us remark that *final value* does not necessarily mean normal form, but evaluation to the degree required by the computation. Unfortunately, it is sometimes tricky to understand what should be observed by using Hood in special situations. In fact, as the author recognizes in [6], the semantics of `observe` (the main debugging combinator of Hood) should be clearly defined to help understanding these situations.

The problem can get more complicated when we consider different implementations of Hood. For instance, the Hugs interpreter currently incorporates an `observe` combinator that differs from that of the original version of Hood. Both the original debugging strategy and the debugging strategy implemented in Hugs are useful and have advantages over each other. However, both of them lack a formalization allowing to reason what should be expected about concrete executions. Moreover, although both strategies are interesting, only one of them can be used in each environment.

In this paper we propose an extension of the STG abstract machine [16, 14] for it to have debugging capabilities. We present three extensions in order to approach gradually the behavior of the Hood debugger. By doing so, we obtain several benefits.

– First, the semantics of the observations are clearly defined in terms of the semantics of the STG abstract machine.
– Second, the implementation can be trivially derived from the modified rules of the STG machine, overcoming some limitations of the current Hood implementation.[1] In this sense, we have reused the work done in [4] to implement our versions of the Hood debugger.
– Third, by encoding the observations inside the STG machine, we can easily study different types of observations. Thus, variations of the Hood `observe` combinator can be introduced. Let us remark that, when introducing such variations, we will trivially obtain its semantics. Moreover, the implementation will also be obtained by modifying accordingly the one of the STG machine. In fact, we have already implemented all the variations appearing in the paper (and also others not shown).

Summarizing, we propose a cleaner and more modular approach to the trace problem in lazy functional programming, allowing to easily provide both implementations and formal foundations for them.

The rest of the paper is structured as follows. In the next section we introduce the main characteristics of Hood. In Section 3 we briefly review the main characteristics of the STG machine. Then, in Section 4 we present how to modify the STG machine to include Hood-like observations. In sections 5 and 6 we study variations of the Hood observations that can be obtained by modifying the STG rules: First, sharing of closures is introduced, like in the Hood version implemented in Hugs; Second, in Section 6 we also keep the sharing of closures, but keeping also the type of observations obtained using the original implementation of Hood. Afterwards, in Section 7 we present some details about our current implementations of the debuggers. Finally, in Section 8 we present our conclusions and lines for future work.

2 An Introduction to Hood

In this section we show the basic ideas behind Hood. The interested reader is referred to [6, 7] for more details about it.

When debugging programs written in an imperative language, the programmer can explore not only the final result of the computation, but also the intermediate values stored in the variables being used by the program. Moreover, it is simple to follow how the value of each variable changes over time.

[1] The sharing of closures is partially lost when using the original implementation of Hood.

Unfortunately, this task is not that simple when dealing with lazy functional languages. However, Hood allows the programmer to observe something similar to it. In fact, Hood allows the programmer to observe any intermediate structure appearing in a program. Moreover, by using GHood [18] we can also observe the evolution in time of the evaluation of the structures under observation.

In order to illustrate what kind of observations can be obtained by using Hood, let us consider an example. It will be complex enough to highlight important aspects of Hood, but also relatively simple to be easily understandable without requiring knowledge about Haskell. Given a natural number, the following Haskell function returns the list of digits of that number:[2]

```
natural :: Int -> [Int]
natural = reverse
          . map (‘mod‘ 10)
          . takeWhile (/= 0)
          . iterate (‘div‘ 10)
```

That is, `natural 3408` returns the list `3:4:0:8:[]`, where `[]` denotes the empty list and `:` denotes the list constructor. Let us remark that, in order to compute the final result, three intermediate lists were produced in the following order:

```
-- after iterate
3408:340:34:3:0:_
-- after takeWhile
3408:340:34:3:[]
-- after map
8:0:4:3:[]
```

Notice that the first intermediate list is infinite, although only the first five elements are computed. As the rest of the list does not need to be evaluated, it is represented as _ (the underscore char).

By using Hood we can annotate the program in order to obtain the output shown before. In order to do that, we have to use the `observe` combinator that is the core of Hood. The type declaration of this combinator is: `observe :: String -> a -> a` From the evaluation point of view, `observe` only returns its second value. That is, `observe s a = a`. However, as a side effect, the value associated to `a` will be squirrelled away, using the label `s`, in a file that will be analyzed after the evaluation finishes. It is important to remark that `observe` returns its second parameter in a completely lazy, demand driven manner. That is, the evaluation degree of `a` is not modified by introducing the observation, in the same way that it is not modified when applying the identity function `id`. Thus, as the evaluation degree is not modified, Hood can deal with infinite lists like the one appearing after applying `iterate (‘div‘ 10)`.

If we consider again our previous example, we can observe all of the intermediate structures by introducing three observations as follows:

```
natural :: Int -> [Int]
natural = reverse
          . observe "after map"
          . map (‘mod‘ 10)
          . observe "after takeWhile"
          . takeWhile (/= 0)
          . observe "after iterate"
          . iterate (‘div‘ 10)
```

[2] The first line of the definition only provides the type declaration of the function: given an integer it returns a list of integers. The other four lines define the sequence of functions to be applied to obtain the overall effect, `reverse` being the last one to be applied.

After executing `natural 3408`, we will obtain the desired result. Hood does not only observe simple structures like those shown before. In fact, it can observe anything appearing in a Haskell program. In particular, we can observe functions. For instance,

```
observe "sum" sum (4:2:5:[])
```

will observe the application of function `sum` to its parameter, returning

```
-- sum
  { \ (4:2:5:[]) -> 11
  }
```

Notice that what we observe can be read as *when the function receives as input the list 4:2:5:[], it returns as output the value 11*. The elements `4`, `2` and `5` appear explicitly because they were really demanded to evaluate the output. However, when observing something like

```
observe "length" length (4:2:5:[])
```

we will obtain the following observation:

```
-- length
   { \ (_:_:_:[]) -> 3
   }
```

That is, we are observing a function that when it receives a list with three elements it returns the number 3 without evaluating the concrete elements appearing in the list. Note that only the number of elements is relevant , but not the *concrete* elements.

As it can be expected, higher-order functions can also be observed. This is done in a similar way as in the previous cases. For instance, in our initial example, instead of observing the intermediate structures, we can observe the higher-order function `iterate`:[3]

```
natural :: Int -> [Int]
natural = reverse
           . map (‘mod‘ 10)
           . takeWhile (/= 0)
           . observe "iterate" iterate (‘div‘ 10)
```

In this situation, when applying `natural` to `3408`, Hood returns

```
-- iterate
   { \ { \ 3 -> 0
       , \ 34 -> 3
       , \ 340 -> 34
       , \ 3408 -> 340
       } 3408
        -> 3408 : 340 : 34 : 3 : 0 : _
   }
```

That is, it observes that it is a function that returns `3408:340:34:3:0:_` when it receives as second parameter `3408` and as first parameter a function (`‘div‘ 10`) that has been observed with four different input values: 3408, 340, 34 and 3.

Let us finally comment a drawback of Hood implementation. Even though it is guaranteed that `observe` does not modify the evaluation degree of the observed values, it does not completely preserve the sharing of closures. That is, each time a structure is observed, the

[3] This higher-order function applies infinite times the first function it receives. For instance, applying `iterate (+3) 1` returns the infinite list `1:4:7:10:13:...`

corresponding closure is cloned, and its computation takes place without sharing the closure with the rest of the program. Obviously, this is an inefficiency in terms of memory and also in terms of runtime. However, by doing so, the implementation was easier.

It is important to remark that, as expressions under observation do not share closures with other parts of the program, it is easier to decide who was the responsible of evaluating each thing. That is, if we are observing a structure in a given environment, we are not interested in the parts of the structure that were evaluated due to other environments. For instance, if we are observing function `length` in the following example

```
let xs = take 5 (1:2:3:4:5:6:7:[])
in (observe "length" length xs) + (sum xs)
```

we will obtain the output

```
-- length
   { \ (_:_:_:_:_:[]) -> 5
   }
```

That is, even though all the elements of the list `xs` were actually computed (due to function `sum`), they were not needed at all to compute any application of the function `length`

However, the Hugs interpreter provides a modified version of Hood whose behavior is quite different in this situation. In fact, when observing function length in exactly the same situation as before, the observation that it returns is

```
-- length
   { \ (1:2:3:4:5:[]) -> 5
   }
```

That is, Hugs does not observe what was really demanded by function `length`. The main advantage of using Hugs-like observations instead of Hood is that the sharing of closures is not lost in Hugs. Moreover, we obtain information about the actual degree of evaluation of each structure. However, no information is provided about what function was responsible for each evaluation, while Hood does obtain it.

Summarizing, both the original version of Hood and the version implemented in Hugs have advantages over each other. So, it is interesting to try to obtain the best of both worlds.

3 The STG Abstract Machine

In this section we briefly describe a slightly modified version of the STG abstract machine. The interested reader is referred to [14] for a very detailed description of it.

3.1 The Basic Language

Before starting the description of the rules governing our STG machine, let us firstly present the core functional language used in the machine. The syntax of this language, called FUN, is shown in Figure 1, where we assume n, k and m to be integers such that $n > 0$, $k \geq 0$ and m is an unboxed integer. We also assume $\overline{x_i}^n$ to represent n variables.

As it can be seen, FUN is a normalized λ-calculus, extended with recursive **let**, constructor applications, **case** expressions and unboxed integers. The normalization process forces constructor applications to be saturated and all applications to only have variables or unboxed values as arguments. Weak head normal forms are lambda abstractions, constructions or unboxed integers.

Let us note that **case** expressions are not required to be saturated. In this sense, two types of default alternatives are introduced. The difference between them is that in the second one

$$
\begin{array}{lll}
e & \rightarrow e\ \overline{atom_i}^{\,n} & \text{-- application} \\
  & \mid\ op\ \overline{atom_i}^{\,n} & \text{-- saturated built-in op} \\
  & \mid\ atom & \text{-- variable or literal} \\
  & \mid\ \mathbf{let}\ \overline{x_i = lf_i}^{\,n}\ \mathbf{in}\ e & \text{-- let} \\
  & \mid\ \mathbf{letrec}\ \overline{x_i = lf_i}^{\,n}\ \mathbf{in}\ e & \text{-- recursive let} \\
  & \mid\ \mathbf{case}\ e\ \mathbf{of}\ alts & \text{-- case expression} \\
 & & \\
alts & \rightarrow \overline{C_i\ \overline{x_j}^{k_i} \mapsto e_i}^{\,k}.default & \text{-- algebraic alternative} \\
  & \mid\ \overline{m_i \mapsto e_i}^{\,k}.default & \text{-- primitive alternative} \\
 & & \\
default & \rightarrow \mathbf{default} \mapsto e & \text{-- default alternative} \\
  & \mid\ v \mapsto e & \\
  & \mid\ \phi & \\
 & & \\
lf & \rightarrow \lambda\ \overline{x_i}^{\,n}.e & \text{-- lambda abstraction} \\
  & \mid\ C\ \overline{x_i}^{\,k} & \text{-- constructor application} \\
  & \mid\ e & \text{-- expression} \\
 & & \\
atom & \rightarrow x & \text{-- variable} \\
  & \mid\ m & \text{-- primitive integers} \\
 & & \\
op & \rightarrow +\# \mid -\# \mid *\# \mid /\# & \text{-- primitive integer op}
\end{array}
$$

Fig. 1. FUN language

the result of the discriminant will be bound to variable v, while in the first case it will be discarded. Let us also note that applications are done to n arguments at once, not one by one. Finally, note that the language includes unboxed integers. We would like to remark that it is not possible to bind an unboxed value.

As it was said before, the language used in this paper is not exactly the same as the core language of the STG abstract machine. The differences are the following:

- In core STG there is a flag $\backslash\pi$ in lambda expressions to indicate that some updates can be safely avoided. This is an efficiency issue, and, for the sake of clarity, we have preferred not to take it into account in our framework.
- We relax applications to have the form $e\ \overline{x_i}^n$ instead of the form $x\ \overline{x_i}^n$, where e is an arbitrary expression (excluding, of course, lambda forms). We have decided to introduce this modification in order to avoid restrictions that are not sufficiently justified (see [3] for details about it.)
- In core STG a program is a list of bindings with a distinguished variable *main* where evaluation starts. This can be trivially simulated by using **letrec** $\overline{bind_i}$ **in** *main*.

Let us remark that these trivial modifications do not change the basics of the STG machine. In fact, in [4] the equivalence with the original version of the STG machine is proved, and the advantages of the new machine are discussed.

3.2 The STG Abstract Machine

A configuration in the STG machine is a quadruple of the form (Γ, e, E, S) where Γ represents the heap, e is the control expression, E is the environment, and S is the stack:

- The environment E binds the free variables of the control expression e with the corresponding pointers.

Heap	Control	Environment	Stack	rule
Γ	$\mathbf{let}\ \{\overline{x_i = lf_i}\}\ \mathbf{in}\ e$	E	S	let (1)
$\Rightarrow \Gamma \cup [\overline{q_i \mapsto (lf_i, E)}]$	e	E'	S	
Γ	$\mathbf{letrec}\ \{\overline{x_i = lf_i}\}\ \mathbf{in}\ e$	E	S	letrec (2)
$\Rightarrow \Gamma \cup [\overline{q_i \mapsto (lf_i, E')}]$	e	E'	S	
Γ	$\mathbf{case}\ e\ \mathbf{of}\ alts$	E	S	case1
$\Rightarrow \Gamma$	e	E	$(alts, E) : S$	
$\Gamma[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts, E') : S$	case2 (3)
$\Rightarrow \Gamma$	e_k	$E' \cup \{\overline{y_{ki} \mapsto p_i}\}$	S	
$\Gamma[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts.\mathbf{default}- > e, E') : S$	case2d (4)
$\Rightarrow \Gamma$	e	E'	S	
$\Gamma[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts.v- > e, E') : S$	case2v (4)
$\Rightarrow \Gamma$	e	$E' \cup \{v \mapsto q\}$	S	
Γ	$e\ \overline{x_i}^n$	$E\{\overline{x_i \mapsto p_i}^n\}$	S	app1
$\Rightarrow \Gamma$	e	E	$\overline{p_i}^n : S$	
$\Gamma[q \mapsto (\lambda \overline{x_i}^n.e, E')]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^n : S$	app2
$\Rightarrow \Gamma$	e	$E' \cup \{\overline{x_i \mapsto p_i}^n\}$	S	
$\Gamma \cup [p \mapsto (e, E')]$	x	$E\{x \mapsto p\}$	S	var1
$\Rightarrow \Gamma$	e	E'	$\#p : S$	
$\Gamma[p \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E')]$	x	$E\{x \mapsto p\}$	$\overline{p_i}^k : \#q : S$	var2 (5)
$\Rightarrow \Gamma \cup [q \mapsto (x\ \overline{x_i}^k, E'')]$	x	E	$\overline{p_i}^k : S$	
$\Gamma[q \mapsto (C_k\ \overline{x_i}, E')]$	x	$E\{x \mapsto q\}$	$\#p : S$	var3
$\Rightarrow \Gamma \cup [p \mapsto (C_k\ \overline{x_i}, E')]$	x	$\{x \mapsto p\}$	S	
Γ	x	$E\{x \mapsto k\}$	S	int#
$\Rightarrow \Gamma$	k	$\{\}$	S	
Γ	k	E	$(alts, E') : S$	case2# (3)
$\Rightarrow \Gamma$	e_k	E'	S	
Γ	k	E	$(alts.\mathbf{default}- > e, E') : S$	case2d# (4)
$\Rightarrow \Gamma$	e	E'	S	
Γ	k	E	$(alts.v- > e, E') : S$	case2v# (4)
$\Rightarrow \Gamma$	e	$E' \cup \{v \mapsto k\}$	S	
Γ	$op\ x_1\ x_2$	E	S	op# (6)
$\Rightarrow \Gamma$	$k_1 op\ k_2$	$\{\}$	S	

(1) $\overline{q_i}$ are distinct and fresh w.r.t. Γ, $\mathbf{let}\ \{\overline{x_i = lf_i}\}\ \mathbf{in}\ e$, and S. $E' = E \cup \{\overline{x_i \mapsto q_i}\}$
(2) $\overline{q_i}$ are distinct and fresh w.r.t. Γ, $\mathbf{letrec}\ \{\overline{x_i = lf_i}\}\ \mathbf{in}\ e$, and S. $E' = E \cup \{\overline{x_i \mapsto q_i}\}$
(3) Expression e_k corresponds to the k-th alternative of $\overline{C_j\ \overline{y_{ji}} \rightarrow e_j}$ in $alts$
(4) C_k not in $alts$
(5) $E'' = \{x \mapsto p, \overline{x_i \mapsto p_i}^k\}$
(6) k_i =if $isInt(x_i)$ then x_i else $E(x_i)$

Fig. 2. The STG abstract machine

- The heap Γ binds pointers to closures which are presented by a pair (lf, E) where lf is a lambda expression and E represents the environment which maps the free variables of lf to the corresponding pointers.
- The stack S stores three kinds of objects: arguments p_i of pending applications (that should be unboxed values or pointers), case alternatives $(alts, E)$ of pending pattern matches, and marks $\#p$ of pending updates.

The reason why we need the environments is that control expressions, lambda expressions and alternatives keep their original variables and in execution we need to know their associated pointers.

The STG machine starts with an expression to be evaluated in an empty heap, environment and stack. Its execution follows the rules shown in Figure 2. The STG machine presented above has some differences from the original one [14] and they are deeply discussed in [4].

We will use the following notation to describe the rules:

- x and y represent variables, while p and q represent pointers in the heap.
- From now on we will use the term *lambda forms* to refer to both λ-abstractions and constructor application.

$$
\begin{array}{lll}
lf & \to \lambda\ \overline{x_i}^n.e & \text{-- lambda abstraction} \\
 & \mid\ C\ \overline{x_i}^k & \text{-- constructor application} \\
 & \mid\ e & \text{-- expression} \\
 & \mid\ x^{@str} & \text{-- observed variable}
\end{array}
$$

Fig. 3. Introducing observations in FUN

- By using $\Gamma \cup [p \mapsto w]$ we denote that we are adding to the heap Γ a new closure w, and we are locating it at position p; we assume that the pointer p is not bound to any closure w in Γ. In contrast, $\Gamma[p \mapsto w]$ means that closure w is already in the position p of the heap.
- The same notation used for the heap is also used with the environments. That is, $E\{x \mapsto p\}$ means that variable x is bound to pointer p in the environment. Moreover, if variable x was not bound, by using $E \cup \{x \mapsto p\}$ we add a new binding so that variable x is now bound to pointer p. In case x was already bound to something else, by using $E \cup \{x \mapsto p\}$ we replace such previous binding with the new one.

As it happens in the STG machine, lambda forms do not appear in control expressions. The main reason for that is efficiency: we do not want to move continuously lambda forms from heap to control and the other way around. Because of that, weak head normal forms corresponds to pointers pointing to lambda forms. It is easy to demonstrate that this is invariant under the application of all rules.

Let us briefly comment the transitions shown in Figure 2. The first two rules (*let*, *letrec*) deal with **let** and **letrec** expressions. In this case, the body of the **let** and **letrec** will be evaluated after adding to the heap all the bindings corresponding to their local definition.

The evaluation of **case** (rules $case1$, $case2$, $case2d$, $case2v$) expressions requires two steps: The first rule stores the list of alternatives in the stack, and goes on evaluating the discriminant. After finishing its computation, a weak head normal form will be obtained. In that moment, it will be compared with the alternatives stored in the stack to decide what alternative of the case matches the result. In order to do that, three different rules are needed ($case2$, $case2d$, $case2v$): one for applying conventional alternatives, and two for applying each kind of default alternative.

The case of the applications is also split into two steps ($app1$, $app2$): first, the arguments are stored in the stack until the function is computed, and then it is applied to those arguments.

We have three cases when evaluating a variable: if it is not evaluated (rule $var1$), we put an update mark on the stack and start to evaluate the corresponding expression; on the other hand if it is in normal form we update the heap with the corresponding normal form (rules $var2$, $var3$).

Finally, the last five rules deal with unboxed integers.

4 Encoding Hood in the STG Abstract Machine

Let us consider now how to introduce Hood-like observations at STG level. Let us remember that Hood users can annotate their programs to mark which structures are to be observed. Thus, in our case we also have to be able to annotate any structure. This can be trivially done by allowing to annotate as *observable* any variable bound in a **letrec**. Thus, we only need to modify slightly the FUN language to include an extra construction as shown in Figure 3. Notice that $x^{@str}$ is the equivalent to the Hood expression `observe str x`.

Once the language allows us to include observations, we have to deal with them in the abstract machine. The main idea is that each rule will have the possibility of generating a side effect to write some observations in a log file. By using this basic idea, we need to rewrite

	Heap	Control	Environment	Stack	Side Effect	rule
	$\mathit{\Gamma}[q \mapsto \mathit{closure}]$	$x^{@str}$	$E\{x \mapsto q\}$	S		var1@$_1$
$\Rightarrow$	$\mathit{\Gamma} \cup [q' \mapsto \mathit{closure}]$	x	$\{x \mapsto q'^{@<str>}\}$	S	$\emptyset$	
	$\mathit{\Gamma}[q \mapsto \mathit{closure}]$	$x^{@str}$	$E\{x \mapsto q^{@<strs>}\}$	S		var1@$_2$
$\Rightarrow$	$\mathit{\Gamma} \cup [q' \mapsto \mathit{closure}]$	x	$\{x \mapsto q'^{@<str+\!\!+strs>}\}$	S	$\emptyset$	
	$\mathit{\Gamma} \cup [q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	S		observer@
$\Rightarrow$	$\mathit{\Gamma} \cup [\overline{p'_i \mapsto \mathit{\Gamma}(p_i)}] \cup [q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i'^{@<strs>}}\})]$	x	$\{x \mapsto q\}$	S	$(strs, q \mapsto C_k\ \overline{p'_i})$	
	$\mathit{\Gamma}[q \mapsto (\lambda \overline{x_i}^n.e, E')]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^n : S$		app2@
$\Rightarrow$	$\mathit{\Gamma} \cup [\overline{p'_i \mapsto \mathit{\Gamma}(p_i)}] \cup [q' \mapsto (e, E' \cup \{\overline{x_i \mapsto p_i'^{@<strs>}}^n\})]$	x	$\{x \mapsto q'^{@<strs>}\}$	S	$(strs, q \mapsto (\lambda \overline{p'_i}^n.q'))$	
	$\mathit{\Gamma}[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E')]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^k : \#q'^{@<strs>} : S$		var2@$_1$ ([1])
$\Rightarrow$	$\mathit{\Gamma} \cup [q' \mapsto (x\ \overline{x_i}^k, E'')]$	x	$\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : S$	$\neg(strs, q \mapsto \overline{p_i}^k)$	
	$\mathit{\Gamma}[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E')]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : \#q'^{@<strs'>} : S$		var2@$_2$ ([2])
$\Rightarrow$	$\mathit{\Gamma} \cup [q' \mapsto (x\ \overline{x_i}^k, E'')]$	x	$\{x \mapsto q^{@<strs+\!\!+strs'>}\}$	$\overline{p_i}^k : S$	$\neg(strs', q \mapsto \overline{p_i}^k)$	

([1]) $E'' = \{x \mapsto q^{@<strs>}, \overline{x_i \mapsto p_i}^k\}$
([2]) $E'' = \{x \mapsto q^{@<strs+\!\!+strs'>}, \overline{x_i \mapsto p_i}^k\}$

Fig. 4. Including the original version of Hood in the STG abstract machine

some of the transitions. In Figure 4, the new transitions needed to model the observations are shown. We will need a completely new rule (*observer*) to deal with the observation of the constructors, and we will also have to add modifications of the rules $var1$, $var2$ and $app2$ of the STG machine.

In Figure 4 it can be seen that there is an extra column containing the side effects of the rules. Let us remark that the side effects are produced at the same time as the evaluation of the program takes place. Thus, observations can be obtained even in the case that the program does not finish its computation.

Before describing the concrete rules, we must introduce some notation. As Hood allows to annotate with a string each observed structure, our pointers can also be annotated with sets of strings (the notation $< strs >$ represents a set of strings). We use sets of strings instead of only one string because the same closure could be observed in different environments with different marks.

As our pointers can now contain annotations, in the stack we can have two different types of pointers: Normal pointers and pointers under observation. As both types of pointers are still pointers, the transitions of the STG machine equally apply to both types. However, in some situations we will be interested in distinguishing both of them. In order to do that in a compact way, we will use as notation that q refers to non-observed pointers, while p refers to any type of pointer (under observation or not).

Let us describe the main differences with the STG machine shown in the previous section. First of all, when a variable under observation is entered, the corresponding closure is cloned, and then this new closure is observed (this is done in the first two rules shown in Figure 4). It is important to remark that what we are doing is to copy the behavior of Hood. Let us remember that, in Hood, closures under observation are not shared with other parts of the program. By doing so, it is easier to decide who was the responsible for evaluating each part of the structure.

In order to better understand the rules, we assume $x^{@str}$ to represent a closure that is to be observed under the name *str*, but that has not already been cloned. It is worth to point out that this kind of closures can only be created by the programmer. That is, once the original program is introduced, it is not possible that new closures of this style appear due to the effects of the transitions of the machine.

In contrast, we would use $p^{@<strs>}$ to represent a cloned closure that is being observed under the set of names *strs*.

Let us describe the modified rules one by one. Rules $var1@_1$ and $var1@_2$ clone an observed closure and annotate the pointer to the cloned closure as an observed pointer. By doing so, we can remember that the closure has already been cloned. Moreover, when the closure is reduced to weak head normal form (whnf), the machine will be able to detect that a side effect has to be performed to observe such a whnf. In addition to that, in the same moment, it will be necessary to clone the closures appearing in such whnf, and also to annotate them as observables.

The rule *observer*@ corresponds to the observation of a constructor applied to a list of arguments. When obtaining the constructor, a side effect is produced. Then, the closures corresponding to the arguments of the constructor are cloned. Finally, those cloned closures are marked as observables.

From that moment, the evaluation goes on, but what is being evaluated is not necessarily a variable associated to a pointer under observation. The reason is that the observation has already been performed. Thus, the variable under evaluation will only be under observation again when entering a closure that has been marked appropriately.

Let us remark that the rule *observer*@ does not modify the stack, as its only purpose is to perform an observation, not an evaluation.

As it can be seen, observing constructors is relatively simple. However, observing applications of functions will be a little bit more complex. In fact, such observations are done at the same time as the normal computation, by using rules $app2@$, $var2@_1$ and $var2@_2$.

Rule $app2@$ is used to observe a total application, that is, the application of a function to all the arguments it requires. Let us remember that, due to efficiency issues, the STG machine assumes that partial applications are never done.

In this $app2@$ rule, we have something of the form $\lambda\overline{x_i}^n.e$ (where $e \neq \lambda$) that is being observed under the set of strings *strs*. Moreover, this function is going to be applied to its arguments $\overline{p_i}^n$. Notice that in this situation what we need to do to behave as Hood is to observe both the arguments of the function and the result of the application. Thus, we proceed to observe both things: We clone the closures of the arguments marking the new closures as observables; and we create a new closure corresponding to the body of the lambda form, but marking as observables all the arguments. Notice that, in case any of the p_i are already being observed due to another mark, then the corresponding p'_i will have both observation marks.

Rules $var2@_1$ and $var2@_2$ correspond to updates of partial applications. In this case, closure q has been reduced to a partial application to its first k parameters of the λ-abstraction stored at position p of the heap. In this situation, an observation is generated indicating that we are not really observing the first k parameters, and we go on evaluating the λ-abstraction marked as observable. Notice that we need to record that it is an *intermediate* step needed before the actual observation, but we do not want to show this intermediate step to the user (this is represented in the rules by using the $\neg$ symbol). Thus, by using this strategy we are obtaining more information than expected by the Hood user. However, by using this kind of special intermediate marks, after the end of the execution we can trivially postprocess the log file obtaining only those observations corresponding to the actual behavior of Hood.

Let us finally comment that in rules $var2@_2$ and $var@_2$ we deal with the situation where a closure is observed with several marks. Rule $var2@_2$ join the two sets of marks corresponding to the marks that were already attached to the closure and to the marks of the new observation. Similarly, $var@_2$ adds the new mark to the previous set of marks. For the sake of simplicity, we use the same symbol ($+\!\!+$) to denote both the union of sets and adding an element to a set.

5 Sharing Closures: Hugs-like Debugging

As we said in Section 2, when observing a structure in Hood, the sharing of closures is completely lost. This implies that more memory is needed to run the program, and it also implies that some computations can be duplicated. Thus, both the runtime and the space efficiency are reduced.

In Section 2 we commented that the Haskell interpreter Hugs includes a different version of Hood. One of the advantages of this version is that the sharing of closures is not lost due to the observations. However, the results obtained when using Hugs are not the same as

Heap	Control	Environment	Stack	Side Effect	rule
Γ	$x^{@str}$	$E\{x \mapsto q\}$	S		var1@$_1$
$\Rightarrow \Gamma$	x	$\{x \mapsto q^{@<str>}\}$	S	$\emptyset$	
Γ	$x^{@str}$	$E\{x \mapsto q^{@<strs>}\}$	S		var1@$_2$
$\Rightarrow \Gamma$	x	$\{x \mapsto q^{@<str +\!\!+ strs>}\}$	S	$\emptyset$	
$\Gamma \cup [q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	S		observerH@
$\Rightarrow \Gamma \cup [q \mapsto (C\ \overline{x_i}, \{\overline{x_i \mapsto p_i^{@<strs>}}\})]$	x	$\{x \mapsto q\}$	S	$(strs, q \mapsto C\ \overline{p_i})$	
$\Gamma[q \mapsto (\lambda\overline{x_i}^n.e, E')]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^n : S$		app2@
$\Rightarrow \Gamma \cup [q' \mapsto (e, E' \cup \{\overline{x_i \mapsto p_i^{@<strs>}}^n\})]$	x	$\{x \mapsto q'^{@<strs>}\}$	S	$(strs, q \mapsto (\lambda\overline{p_i}^n.q'))$	
$\Gamma[q \mapsto (\lambda\overline{x_i}^k.\lambda\overline{y_i}^n.e, E')]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^k : \#q'^{@<strs>} : S$		var2@$_1$ (^{1})
$\Rightarrow \Gamma \cup [q' \mapsto (x\ \overline{x_i}^k, E'')]$	x	$\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : S$	$\neg(strs, q \mapsto \overline{p_i}^k)$	
$\Gamma[q \mapsto (\lambda\overline{x_i}^k.\lambda\overline{y_i}^n.e, E')]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : \#q'^{@<strs'>} : S$		var2@$_2$ (^{2})
$\Rightarrow \Gamma \cup [q' \mapsto (x\ \overline{x_i}^k, E'')]$	x	$\{x \mapsto q^{@<strs +\!\!+ strs'>}\}$	$\overline{p_i}^k : S$	$\neg(strs', q \mapsto \overline{p_i}^k)$	

(^{1}) $E'' = \{x \mapsto q^{@<strs>}, \overline{x_i \mapsto p_i}^k\}$
(^{2}) $E'' = \{x \mapsto q^{@<strs +\!\!+ strs'>}, \overline{x_i \mapsto p_i}^k\}$

Fig. 5. The Hugs-Hood abstract machine

when using Hood with other compilers. The difference is that Hugs does not record who was responsible for each evaluation, it only records whether a value was computed or not.

We can easily embed this variant of Hood by slightly modifying the rules we introduced in the previous section. The new rules are shown in Figure 5. Notice that rules $var1@_1$ and $var1@_2$ are nearly the same as in the previous implementation, the difference being that now we do not clone any closure. Rule $observerH@$ has also been modified to remove the duplication of the closure under evaluation. By removing such duplications, we keep the sharing of closures, like Hugs does.

Rule $app2@$ is also slightly modified to avoid cloning the closures corresponding to the parameters of the application. However, it is necessary to use a new pointer to bind the body of the lambda abstraction. This is done to be able to observe the lambda form it evaluates to. However, we would like to remark that we are not cloning the closure. That is, sharing is kept.

Finally, rules $var2@_1$ and $var2@_2$ are not changed at all. That is, they are exactly the same as in the previous section. The reason is that the only difference with the previous implementation is that we do not want to clone closures. However, we do not care about keeping the behavior of the original version of Hood. The reason is that we are marking all the closures that are under observation, but we are not taking care about the closures that are observing those observed closures. Thus, in case function `length` is observed in a program like

```
let xs = take 5 (1:2:3:4:5:6:7:[])
in (observe "length" length xs) + (sum xs)
```

we have that `xs` is being observed, being function `length` its observer. Moreover, `xs` is completely evaluated due to the demand of function `sum`. Then, although function `length` only demands the spine of `xs`, the observation obtained would be

```
-- length
   { \ (1:2:3:4:5:[]) -> 5
   }
```

That is, we obtain the same type of observations as Hugs, not that of the original Hood. The reason is that the abstract machine is not considering who is the observer of the observed structures. Thus, in case an observed closure is demanded, its evaluation will be observed even if the reason of the computation is not related with the observer function.

6 Hood Sharing Closures

As we showed in Section 4, by encoding Hood in the STG abstract machine we can do exactly the same as in Hood. In fact, we have followed an approach that was also losing the sharing of closures. Thus, it had the same efficiency disadvantages.

In the previous section we have shown another implementation keeping the sharing of closures. However, its behavior was not equivalent to that of the original version of Hood.

Fortunately, when working at the level of the abstract machines we can easily overcome the previous problems. In fact, in this section we show how we can embed Hood in the STG abstract machine without reducing the sharing of closures. That is, the observed programs will run using the same space as it would be required to run the program without observations.

In Figure 6, the new transitions needed to model the observations are shown. Before explaining them in detail, we would like to remark that there is no rule where closures are cloned. That is, sharing of closures will be preserved.

As closures are not being cloned, it is now very important to clearly understand not only what closure is being observed, but also what other closure is observing it. For instance, in the expression **letrec** $x = y^{@str}\ \ldots$ **in** e, we have that y is a closure that is being observed, while x is the closure that is observing it. That is, x is the observer of y. In this situation, even though we must remember that y is being observed with the mark str, when updating a value the observation marks should be attached to the pointer corresponding to variable x, and not to the pointer corresponding to variable y. That is, what we need to do is to associate the observations to the entity that is observing, not to the one that is being observed.

Following the previous ideas, $x^{@str}$ means that closure x is being observed by other closure. Similarly, $q^{@<strs>}$ means that the closure located at position q of the heap is being observed from other closures by using the set of marks $strs$. As a consequence, closure q must be stored in the heap without any observation mark.

Rules $var2@_1$ and $var2@_2$ are exactly the same as in Section 4, that is, when sharing was not preserved, except that now it is not necessary to add $@<strs>$ to the closure q' added to the heap because annotation $@<strs>$ means that another closure is observing q'. However, rules $var1@_1$, $var1@_2$ and $app2@$ need to be slightly rewritten, the only difference being that now we do not have to clone any closure. That is, we have to use exactly the same rules as in the Hugs-like machine.

Notice that in the new abstract machine we can not use the rules $observer@$ or $observerH@$. Rule $observer$ is incorrect because closure p corresponds to the closure that is being observed, not to the closure that is observing it. In the first machine, as closure p was a copy of the original closure, we could mark it without any risk, as it was only possible to enter this closure from the closures that were observing it. However, now p is not cloned, and it can be entered both from a closure that is observing it or from any other closure. Due to this reason, now we need more rules to be able to distinguish both situations. Let us remark that rule $observerH$ is also incorrect in this situation, as it was not taking into account who was the observer of each observed closure.

Let us describe the rest of the new rules, that are needed to substitute rule $observer$. As we need to take care of both the observer and the observed closures, we need to track the evolution of the computation transmitting the marks corresponding to the closures under observation. Notice that rules $case2@$, $case2d@$, $case2v@$ and $var3@$ are exactly the same as in the original STG machine, but transmitting the marks and generating side effects.

Rules $case2@$, $case2d@$ and $case2v@$ deal with closures that has been reduced to a constructor application, and that are being observed with the set of strings $strs$. Thus, a side effect has to be generated to observe the constructor, and then the machine has to go on evaluating the corresponding alternative of the outer case expression. Notice that all the pointers appearing in the application of the constructor are also marked as observable, so that we can actually observe them when appropriate.

Heap	Control	Environment	Stack	Side Effect	rule
$\mathit{\Gamma}$	$x^{@str}$	$E\{x \mapsto q\}$	S		var1@$_1$
$\Rightarrow \mathit{\Gamma}$	x	$\{x \mapsto q^{@<str>}\}$	S	$\emptyset$	
$\mathit{\Gamma}$	$x^{@str}$	$E\{x \mapsto q^{@<strs>}\}$	S		var1@$_2$
$\Rightarrow \mathit{\Gamma}$	x	$\{x \mapsto q^{@<str+\!\!+strs>}\}$	S	$\emptyset$	
$\mathit{\Gamma}[q \mapsto (\lambda \overline{x_i}^n.e, E')]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^n : S$		app2@
$\Rightarrow \mathit{\Gamma} \cup [q' \mapsto (e, E' \cup \{\overline{x_i \mapsto p_i^{@<strs>}}^n\})]$	x	$\{x \mapsto q'^{@<strs>}\}$	S	$(strs, q \mapsto (\lambda \overline{p_i}^n.q'))$	
$\mathit{\Gamma}[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E')]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^k : \#q'^{@<strs>} : S$		var2@$_1$ ([1])
$\Rightarrow \mathit{\Gamma} \cup [q' \mapsto (x\ \overline{x_i}^k, E'')]$	x	$\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : S$	$\neg(strs, q \mapsto \overline{p_i}^k)$	
$\mathit{\Gamma}[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E')]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : \#q'^{@<strs'>} : S$		var2@$_2$ ([2])
$\Rightarrow \mathit{\Gamma} \cup [q' \mapsto (x\ \overline{x_i}^k, E'')]$	x	$\{x \mapsto q^{@<strs+\!\!+strs'>}\}$	$\overline{p_i}^k : S$	$\neg(strs', q \mapsto \overline{p_i}^k)$	
$\mathit{\Gamma}[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$(alts, E') : S$		case2@ ([3])
$\Rightarrow \mathit{\Gamma}$	e_k	$E' \cup \{\overline{y_{ki} \mapsto p_i^{@<strs>}}\}$	S	$(strs, q \mapsto C_k\ \overline{p_i})$	
$\mathit{\Gamma}[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$(alts.\mathbf{default}- > e, E') : S$		case2d@ ([4])
$\Rightarrow \mathit{\Gamma}$	e	E'	S	$(strs, q \mapsto C_k\ \overline{p_i})$	
$\mathit{\Gamma}[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$(alts.v- > e, E') : S$		case2v@ ([4])
$\Rightarrow \mathit{\Gamma}$	e_k	$E' \cup \{v \mapsto q^{@<strs>}\}$	S	$(strs, q \mapsto C_k\ \overline{p_i})$	
$\mathit{\Gamma}[q \mapsto (C\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\#p : S$		var3@
$\Rightarrow \mathit{\Gamma} \cup [p \mapsto (C\ \overline{x_i}, \{\overline{x_i \mapsto p_i^{@<strs>}}\})]$	x	$\{x \mapsto p\}$	S	$(strs, q \mapsto C\ \overline{p_i})$	

([1]) $E'' = \{x \mapsto q, \overline{x_i \mapsto p_i}^k\}$
([2]) $E'' = \{x \mapsto q^{@<strs>}, \overline{x_i \mapsto p_i}^k\}$
([3]) Expression e_k corresponds to alternative $\overline{C_j\ \overline{y_{ji}} \to e_j}$ in *alts*
([4]) C_k not in *alts*

Fig. 6. The Hood abstract machine sharing closures

Rule $var3@$ deals with the update of closures. In this case, p is updated recording the appropriate observation, the corresponding side effect is produced, and the machine goes on evaluating p.

Let us finally remark that, although the new machine is not so simple as the one presented in Section 4, it is still presented in a relatively simple an compact way.

6.1 Equivalence with respect to the STG machine

Let us finally prove that the previous machine (that we call in the following STG$^@$) is equivalent to the original STG machine. That is, our goal is to prove that both machines perform the same computations. Obviously, the same can be done with the two machines presented in the previous sections, but we do not include it due to lack of space. Then, let us prove that STG$^@$ and STG have the same computations. Since both machines are not identical, we will have to define an equivalence notion between configurations in both machines $\equiv^@$:

$$C \equiv^@ C^@,\ C \Longrightarrow^* D \qquad \text{iff} \qquad C^@ \Longrightarrow^* D^@,\ D \equiv^@ D^@$$

The STG$^@$ machine introduces new pointers in rule *app*2@ that STG does not. These new pointers point to the body of an observed λ-expression. We consider these new pointers as belonging to a set of pointers disjoint to the original ones. We call V_1 the set of the *normal* pointers and V_2 the set of pointers introduced in rule *app*2@. Finally, we recall that the column of Side Effects does not have any influence in the computations of a STG$^@$, so we can ignore it in the proof. We will need the following notation:

Definition 1

- *Let* $e^@$, $p^@$ *and* $\#p^@$ *be an expression, a pointer and an updated mark of* STG$^@$. *We denote by* $\mathrm{rmo}(e^@)$, $\mathrm{rmo}(p^@)$ *and* $\mathrm{rmo}(\#p^@)$ *the same expression, pointer and updated mark but without observation marks. This is trivially extended to stacks, heaps and environments:* $\mathrm{rmo}(S)$, $\mathrm{rmo}(\mathit{\Gamma})$, $\mathrm{rmo}(E)$.
- *Let* $\mathit{\Gamma}$ *be a heap of* STG *and* $\mathit{\Gamma}^@$ *be a heap of* STG$^@$, *then we say that* $\mathit{\Gamma} \subseteq_{\mathbf{V_1}} \mathit{\Gamma}^@$ *if* $\mathit{\Gamma} = \mathit{\Gamma}^@|_{V_1}$*: the closures* $p \mapsto e$ *where* $p \in V_1$ *are the same in* $\mathit{\Gamma}$ *and* $\mathit{\Gamma}^@$ *and the pointers* p *such that* $p \mapsto e \in \mathit{\Gamma}^@ \setminus \mathit{\Gamma}$ *verify that* $p \in V_2$.

– *Let S be a stack of* STG *and $S^{@}$ be a stack of* $\mathrm{STG}^{@}$*, then we say that $S \subsetneq_{\mathbf{V_1}} S^{@}$ if $S = \mathrm{rmo}(S^{@})|_{V_1}$: the stack S is the same as the stack $S^{@}$ when we remove observations, and pointers and updated marks to pointers in V_2. Recall that, besides pointers, in the stack there can also appear pending alternatives. One can easily prove that the pointers in V_2 never appear in those alternatives.*
– *Let (Γ, e, E, S) be a configuration of* STG *and $(\Gamma^{@}, e^{@}, E^{@}, S^{@})$ be a configuration of* $\mathrm{STG}^{@}$*, we say that they are equivalent $(\Gamma, e, E, S) \equiv^{@} (\Gamma^{@}, e^{@}, E^{@}, S^{@})$ iff*

$$e = \mathrm{rmo}(e^{@}), \quad E = \mathrm{rmo}(E^{@}), \quad \Gamma \subsetneq_{\mathbf{V_1}} \mathrm{rmo}(\Gamma^{@}), \quad \text{and} \quad S \subsetneq_{\mathbf{V_1}} \mathrm{rmo}(S^{@})$$

We need to prove that both machines have the same computations. However, the new machine performs more evaluation steps, since it has to deal with observations. So, there are configurations that do not have a match in the STG machine. We will call these configurations unstable.

Definition 2 *A configuration (Γ, e, E, S) of machine* $\mathrm{STG}^{@}$ *is unstable if the expression is a variable bound to a pointer in V_2: $(\Gamma, x, E\{x \mapsto q\}, S)$ with $q \in V_2$. The rest are called* stable *configurations.*

ow we can prove the following proposition that will allow us to achieve our final goal. This proposition states that the stable configurations of $\mathrm{STG}^{@}$ correspond to regular configurations of STG.

Proposition 1 *Let $(\Gamma^{@}, e^{@}, E^{@}, S^{@})$ be a stable configuration of* $\mathrm{STG}^{@}$*, (Γ, e, E, S) be a configuration of* STG *such that $(\Gamma, e, E, S) \equiv^{@} (\Gamma^{@}, e^{@}, E^{@}, S^{@})$ then $(\Gamma^{@}, e^{@}, E^{@}, S^{@}) \Longrightarrow^{*} (\Delta^{@}, w^{@}, E'^{@}, S'^{@})$ iff $(\Gamma, e, E, S) \Longrightarrow^{*} (\Delta, w, E', S')$ and $(\Delta, w, E', S') \equiv^{@} (\Delta^{@}, w^{@}, E'^{@}, S'^{@})$*

Proof. Sketch The proof is made by induction on the length of the computations from $(\Gamma^{@}, e^{@}, E^{@}, S^{@})$ to $(\Delta^{@}, w^{@}, E'^{@}, S'^{@})$ and from (Γ, e, E, S) to (Δ, w, E', S'). The inductive basis (length 0) is trivial. In the inductive step we distinguish cases depending on the first rule applied. The most complex rule is *app*2@, that gets to an unstable configuration. Fortunately, due to syntactical restrictions, in the body of a λ-expression there cannot appear a constructor nor another λ-expression. Therefore, the next rule (*var*) will get us back to an stable configuration. Thus, as a result of applying rule *var*, the stack has a new update mark that does not appear in the evaluation of STG, but fortunately, this update mark is a pointer in V_2. Dealing with the rest of the rules is quite straightforward.

As a corollary of this proposition, we obtain the theorem we were looking for

Theorem 1 *Let be $e^{@}$ an expression with observable marks and e an expression such that $\mathrm{rmo}(e^{@}) = e$ then, $(\{\,\}, e^{@}, \{\,\}, [\,]) \Longrightarrow^{*} (\Delta^{@}, w^{@}, E^{@}, S^{@})$ iff $(\{\,\}, e, \{\,\}, [\,]) \Longrightarrow^{*} (\Delta, w, E, S)$ and $(\Delta, w, E, S) \equiv^{@} (\Delta^{@}, w^{@}, E^{@}, S^{@})$*

Since in each evaluation step the heaps are equivalent and the pointers V_2 are not in the same connected component as w, we have a final corollary:

Corollary 1 *The new rules keeps the laziness of the computation.*

7 Implementation Issues

In this section we briefly describe some details of our implementation. As we said in the introduction, we have implemented all the machines presented in the previous sections. In order to do that, we have followed the work done in [4]. In that work we presented a very simple imperative machine that is used as an intermediate step between the STG machine an the C code generated by the GHC compiler. By using that intermediate machine, we provided

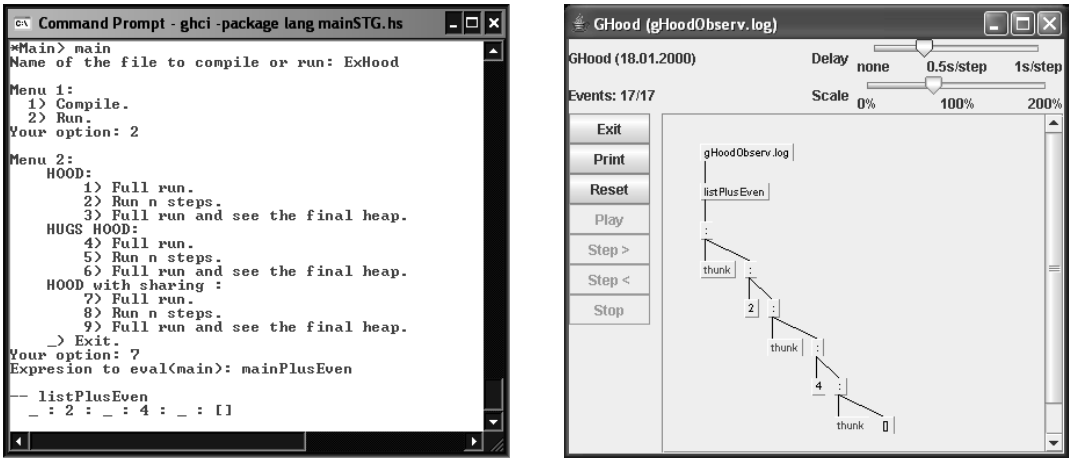

Fig. 7. Debugging environment: Produces text output (left) or graphical output (right)

a translation scheme allowing to translate FUN expressions into executable code. Moreover, in [4] we already provided a proof of correctness for the translation.

One of the advantages of embedding the debugger at STG level is that we can easily use different debugging techniques, as we only need to use the proper rules at STG level. Then, the corresponding implementations are obtained by following the steps described in [4]. Taking into account this advantage, we have implemented a programming environment (see Figure 7) that allows to select in each situation what type of debugger is to be used. That is, for a concrete situation we can use Hugs-like observations, while in another execution we can use Hood-like observations.

Finally, we would like to comment that in our environment we do not only generate plain text as observations. In fact, the user can choose between observing the output of the observations by using plain text (with the same style as in Hood) or by using the GHood[18] graphical environment. By using GHood, we do not only observe the degree of evaluation of the observed structures, but also the order of evaluation of it.

8 Conclusions and Future Work

In this paper we have presented a new view of the Hood debugger allowing both to clarify its formal foundations and to easily implement different variations of it. In particular, we have described how to embed Hood inside the STG abstract machine, showing different possible implementations of it. In fact, by encoding it at STG level we have shown how to improve the efficiency of the implementation, as observations can now be done without cloning closures.

Note that the approach we use to embed Hood inside the STG machine can also be done to embed any other Haskell debugger. In this sense, it could be used as a common framework for describing (and also implementing) all of them.

As future work, we plan to extend our framework to deal with parallel extensions of Haskell. In this sense, we will pay special attention to the language Eden[9, 13], whose abstract machine (DREAM [1]) is an extension of the STG machine. An interesting feature of Eden is that it uses eager evaluation when sending values from one process to another process. Due to this reason, unnecessary computations are sometimes done, and it can be hard to discover how much data was produced speculatively. By using an extension of Hood we could detect the amount of speculation, as we could see how much data was produced and how much was actually demanded.

A more general framework to deal with different parallel extensions of Haskell could also be used. Unfortunately, there does not exist a common abstract machine shared by different parallel languages. In this sense, the best choice is to try to embed the debugging method inside the Jauja language[8], a very simple parallel functional language that has already been used as a common framework to describe three different languages, namely GpH, Eden and pH.

References

1. S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: the distributed Eden abstract machine. In *Implementation of Functional Languages (IFL'97)*, LNCS 1467, pages 250–269. Springer-Verlag, 1998.
2. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood — a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Implementation of Functional Languages (IFL'00)*, LNCS 2011, pages 176–193. Springer-Verlag, 2001.
3. A. Encina and R. Peña. Proving the correctness of the STG machine. In *Implementation of Functional Languages (IFL'01)*, LNCS 2312, pages 88–104. Springer-Verlag, 2001.
4. A. Encina and R. Peña. Formally deriving an STG machine. In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 102–112. ACM, 2003.
5. R. Ennals and S. Peyton Jones. HsDebug: Debugging lazy programs by not being lazy. In *Proceedings of the 7th Haskell Workshop*, pages 84–87. ACM, 2003.
6. A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop*. Technical Report of the University of Nottingham, 2000.
7. A. Gill. Hood homepage. http://www.haskell.org/hood, 2005.
8. M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel Haskell dialects. In *First Asian Symposium on Programming Languages and Systems (APLAS'03)*, LNCS 1058, pages 303–321. Springer-Verlag, 2003.
9. U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation skeletons in Eden: Low-effort parallel programming. In *Implementation of Functional Languages (IFL'00)*, LNCS 2011, pages 71–88. Springer-Verlag, 2001.
10. H. Nilsson. Declarative debugging for lazy functional languages. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 1998.
11. H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, Nov 2001.
12. C. Pareja, R. Peña, F. Rubio, and C. Segura. Adding traces to a lazy functional evaluator. In *Eurocast 2001*, LNCS 2178, pages 627–641. Springer-Verlag, 2001.
13. R. Peña and F. Rubio. Parallel functional programming at two levels of abstraction. In *Principles and Practice of Declarative Programming (PPDP'01)*, pages 187–198. ACM, 2001.
14. S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine, version 2.5. *Journal of Functional Programming*, 2(2):127–202, 1992.
15. S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. URL http://www.haskell.org, February 1999.
16. S. L. Peyton Jones and J. Salkild. The Spineless Tagless G-machine. In *Functional Programming Languages and Computer Arquitecture*, pages 184–201. Addison Wesley, 1989.
17. B. Pope and L. Naish. Practical aspects of declarative debugging in Haskell 98. In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 230–240. ACM, 2003.
18. C. Reinke. GHood — graphical visualization and animation of Haskell object observations. In *Proceedings of the 5th Haskell Workshop*, volume 59 of *ENTCS*. Elsevier Science, 2001.
19. J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *Programming Languages, Implementations, Logics and Programs (PLILP'97)*, LNCS 1292, pages 291–308. Springer-Verlag, 1997.
20. P. Wadler. Functional programming: Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, August 1998. Functional Programming Column.
21. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multipleview tracing for Haskell: a new Hat. In *Proceedings of the 5th Haskell Workshop*, pages 151–170, 2001.