Journal of Universal Computer Science, vol. 15, no. 14 (2009), 2836-2880
submitted: 8/4/08, accepted: 30/7/09, appeared: 1/8/09 © J.UCS

A Debugging System Based on Natural Semantics[1]

Alberto de la Encina
(Universidad Complutense de Madrid, Spain
albertoe@sip.ucm.es)

Luis Llana
(Universidad Complutense de Madrid, Spain
llana@sip.ucm.es)

Fernando Rubio
(Universidad Complutense de Madrid, Spain
fernando@sip.ucm.es)

Abstract: Due to the absence of side effects, reasoning about functional programs is simpler than reasoning about their imperative counterparts. However, because of the absence of practical debuggers, finding bugs in lazy functional languages has been more complex until quite recently. One of the easiest to use Haskell debuggers is Hood. Its 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 behavior to explain better what should be expected, and also to allow to check whether the different implementations work properly.

In this paper, we formalize the behavior of the Hood debugger by extending Sestoft's natural semantics. Moreover, we also show how to derive an abstract machine including such debugging information. By doing so, we do not only provide a formal foundation, but we also provide an alternative method to implement debuggers. In fact, we have already made a prototype of the abstract machine presented in this paper.

Key Words: Parallel functional programming, debugging, semantics, abstract machines.
Category: F.3.2, D.2.5, D.3.1, D.3.2

1 Introduction

In lazy functional languages, values are only evaluated when it is completely sure that the value will be needed for the final result of the programs. This interesting feature complicates the task of debugging lazy functional programs. Although not much attention was paid to it in the past (see e.g. [Wad98]), during the last years there have been several proposals for incorporating execution traces into lazy functional languages. In particular, we can highlight the work done with Hat [WCBR01, Chi05, DC06], HsDebug [EP03b, EP03c], Rectus [MK06], the declarative debuggers Freja [Nil98, Nil01] and Buddha [PN03, Pop05], the last approach implemented in GHCi [Him06, MIBPG07] and specially the work done with the Haskell Object Observation Debugger (Hood) [Gil01, Rei01]. All

[1] Work partially supported by the MCYT project TIN2006-15578-C02-01, the JCLM project PAC06-0008-6995, the UCM project CCG08-UCM/TIC-4124 and the UCM-BSCH programme to fund research groups (GR58/08 - group number 910606).

of them are designed to be used with the language Haskell [Pey03], 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 user and the implementation points 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 direct 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 the incorrect value. In this paper we will not concentrate on those differences (the interested reader can find a detailed comparison between Freja, Hat and Hood in [CRW01], while [PPRS01] presents a common framework to describe all of them). In contrast, we will concentrate on how to provide a formal foundation for one of them.

Among all of the Haskell debuggers, Hood has an interesting advantage over the rest, as it 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 [PHH$^+$93], Hugs98 [JP99], the Yhc [Tea97], and also with nhc98 [Röj95]. Due to its portability, Hood has become one of the most used Haskell debuggers.

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 (lambda abstractions, constructor applications, and primitive values), but evaluation to the degree required by the lazy 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 [Gil01], the semantics of `observe` (the principal debugging combinator of Hood) should be clearly defined to help understanding its behavior.

In this paper, we continue our previous work [ELR06] to propose a formalization of the Hood debugger allowing both to reason about it and to implement it in a different and systematic way. What we propose is an extension of Sestoft's natural semantics[2] [Ses97] that incorporates new rules to deal with Hood observations. Moreover, in this paper we introduce an equivalent abstract machine with respect to the semantic defined, and we prove their equivalence. By doing so, we obtain two main benefits. First, the semantics of the Hood *observe* operator is clearly defined. Second, we can reuse the work done in [EP02, EP03a, EP09] to effectively implement a debugging system. In fact, we have already made a prototype of the abstract machine presented in this paper.

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. Then, in Section 3 we briefly review

[2] Sestoft's semantics is an extension of the original natural semantics introduced by Launchbury in [Lau93].

the main characteristics of Sestoft's semantics. Next, in Section 4 we show how to modify the original semantics to include debugging information equivalent to that obtained by Hood. Afterwards, in Section 5 we introduce an abstract machine equivalent to our new semantics. Then, in Section 6 we explain some details about the implementation of the debugging system. Finally, in Section 7, in addition to presenting our conclusions and lines for future work, we briefly describe some details about our current implementation of the debugger. Finally, in the Appendix of the paper we present the proofs of the propositions introduced in the paper.

2 An Introduction to Hood

In this section we show the basic ideas behind Hood. The interested reader is referred to [Gil01, Gil00] for more details about it.

When debugging programs written in an imperative language (such as: Pascal, C, etc.), 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 at any moment of the computation. Moreover, it is simple to follow how the value of each variable changes along time.

Unfortunately, this task is not that simple when dealing with lazy functional languages. However, Hood allows the programmer to observe something similar to an imperative environment. In fact, Hood provides a way to observe any intermediate structure appearing in a program. Moreover, by using GHood [Rei01], that is a graphical interface for the debugger Hood, 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 the example introduced in [Gil01]. It will be complex enough to highlight important aspects of Hood, but also relatively simple to be easily understandable without requiring deep knowledge about Haskell. Given a natural number, the following Haskell function returns the digits base 10 of that number:

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

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, being `reverse` the last one to be applied. In order to better understand the previous source code, let us consider an application example. For instance, `natural 3408` returns the list `3:4:0:8:[]`, where `[]` denotes the empty list and `:` denotes the list constructor. 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 because the `iterate` function produces an infinite list applying `(‘div‘ 10)` recursively to the last calculated number. Notice also that only the first five elements are computed. Since it is not necessary to evaluate the rest of the list, 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 with `a` will be written, attaching to it 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 inserting the `observe` function before each function we want to observe:

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

After executing `natural 3408`, we will obtain the desired result. Remember that the first function that will be applied to the integer value `3408` is `iterate (‘div‘ 10)`, the second is `observe "after iterate"` and so on. Then, by introducing the previous three observations we observe the result of the intermediate values. With the observation `observe "after iterate"` we will get the list produced after applying `iterate (‘div‘ 10)` and so on.

Hood does not only observe simple structures like those shown in the previous example. 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 (that is, the list `4:2:5:[]`), 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 in order to obtain the length of a list it is only relevant the number of elements that the list has, but not the value of the *concrete* elements. The function `length` is a polymorphic function that can be applied to lists of any type (i.e. list of integers, list of chars, etc.).

As it can be expected, higher-order functions (functions that take one or more functions as an input or output parameter) 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`:

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

This higher-order function `iterate` applies infinite times the first function it receives. For instance, applying `iterate (+3) 1` returns the following infinite list: `1:4:7:10:13:...`. Now, `observe` only affects to the function `iterate`, so in this case we are observing the behavior of this function. In this situation, when we apply the new definition of `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.

It is important to remark that Hood has to analyze who was responsible for evaluating each data. 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[3]:

```
  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` where actually computed (due to function `sum`), they were not needed at all to compute any application of the function `length`.

3 A Semantics for Lazy Evaluation

We begin by reviewing the language and semantics given by Sestoft [Ses97] as an improvement to Launchbury's semantics [Lau93]. A well-known work from Launchbury defines a big-step operational semantics for lazy evaluation. The only machinery needed is an explicit heap where bindings are kept. A heap is considered to be a finite mapping from variables to expressions, i.e., duplicated bindings to the same variable are disallowed. The proposals of Launchbury and Sestoft share the language given in Figure 1, where $\overline{A_i}$ denotes a vector $A_1, \ldots, A_n$ of subscripted entities. It is a normalized λ-calculus, extended with recursive **let**, constructor applications and **case** expressions. Sestoft's normalization process forces constructor applications to be saturated and all applications to only have variables as arguments. Weak head normal forms are either lambda abstractions (i.e. functions that given an input variable x returns as a result the evaluation of expression e) or constructions (i.e. data types). Throughout this section, w will denote (weak head) normal forms.

Sestoft's semantic rules are given in Figure 2. There, a judgement is represented as $\Gamma : e \Downarrow \Delta : w$ and denotes that expression e, with its free variables bound in heap Γ, reduces to normal form w and produces the final heap Δ. Let us remark that, if the configuration $\Gamma : e$ reduces to normal form, then Δ and w are unique, because the semantics rules are deterministic (that is, mutually exclusive). Then, in each derivation it is only possible to apply one rule. Thus, in the rest of the paper, we will assume this fact to avoid introducing extra quantifiers in our formalizations. Let us also remark that the notation $\hat{e}$ in rule *Letrec* means the replacement of the variables x_i by the fresh pointers p_i. This is the only rule where new bindings are created and added to the heap.

[3] `let` expression stores a new binding in the heap and then goes on computing the expression appearing after the keyword `in`. In this case, `xs` is bound with the value of the expression `take 5 (1:2:3:4:5:6:7:[])`, and then it is computed the body of the expression, in this case `(observe "length" length xs) + (sum xs)`

$e \rightarrow x$	-- variable
$\mid\ \lambda x.e$	-- lambda abstraction
$\mid\ e\ x$	-- function application
$\mid\ \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e$	-- recursive let
$\mid\ C\ \overline{x_i}$	-- constructor application
$\mid\ \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i}$	-- case expression

Figure 1: Sestoft's normalized λ-calculus

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textit{Lam}$$

$$\Gamma : C\,\overline{p_i} \Downarrow \Gamma : C\,\overline{p_i} \qquad \textit{Cons}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda x.e' \quad \Delta : e'[p/x] \Downarrow \Theta : w}{\Gamma : e\ p \Downarrow \Theta : w} \qquad \textit{App}$$

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma \cup [p \mapsto e] : p \Downarrow \Delta \cup [p \mapsto w] : w} \qquad \textit{Var}$$

$$\frac{\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \Downarrow \Delta : w}{\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow \Delta : w}\ \text{where } \overline{p_i} \text{ are fresh} \qquad \textit{Letrec}$$

$$\frac{\Gamma : e \Downarrow \Delta : C_k\ \overline{p_j} \quad \Delta : e_k[\overline{p_j/x_{kj}}] \Downarrow \Theta : w}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i} \Downarrow \Theta : w} \qquad \textit{Case}$$

Figure 2: Sestoft's natural semantics

We use the term *pointers* to refer to dynamically created free variables, bound to expressions in the heap, and the term *variables* to refer to (lambda-bound, let-bound or case-bound) program variables. We consistently use $p, q, \ldots$ to denote pointers and $x, y, \ldots$ to denote program variables. The notation $\Gamma[p \mapsto e]$ means that $(p \mapsto e) \in \Gamma$, and $\Gamma \cup [p \mapsto e]$ represents the disjoint union of Γ and $(p \mapsto e)$.

The first two rules (*Lam*, and *Cons*, in Figure 2) only establish that normal forms are λ-abstractions or constructions, as they reduce to themselves. The *App* rule reduces the function application $e\ p$ to w if the body of the application e reduces to the lambda form $\lambda x.e'$, and this λ-form applied to p (that is $e'[p/x]$) reduces to the normal form w. If the expression bound in the heap with the variable p (that is, e) reduces to the normal form w, the rule *Var* reduces the variable p to w. Moreover, in order to avoid re–evaluation of e, the rule also updates the heap with the reduced value w, The rule *Letrec* reduces the **letrec** expression to w if the body of the **letrec** expression (that is, e) reduces to w in a heap where the new bindings have been introduced. Finally, the rule *Case* reduces the **case** expression to w if the discriminant of the **case** expression (that is, e) reduces to the constructor $C_k\ \overline{p_j}$ and the body of the corresponding alternative (that is, e_k) reduces to w.

4 Semantics with Debugging Features

4.1 Low level details of the real behavior of Hood

In order to better understand how we should define the rules of our semantics, it is convenient to describe some details of the implementation of Hood. When Hood is in action, it produces internal annotations that have this form: $(portId,\ parent,\ change)$. The $portId$ corresponds to a pointer to the place where the annotation is made: in the implementation it corresponds to a line number in the file of annotations. In order to be able to post-process the file, when a function or a constructor is evaluated, its arguments need to know the place where they were invoked. That is, we need to access the *parent* of the arguments; formally, parent is a tuple $(observeParent,\ observePort)$, where $observeParent$ is the $portId$ of the parent and the $observePort$ is the position of the argument. Finally, parameter *change* corresponds to the type of observation carried out, and it can have one of the following forms:

– $Observe\ String$ is generated when we enter in a binding annotated with the corresponding string. This kind of observation has no parent because it is the first observation produced, actually it has the general parent, that corresponds to $(0,\ 0)$. This is the first annotation generated when we start the evaluation of an annotated binding.

– $Enter$ is generated when the evaluation of a binding starts.

– $Cons\ Int\ String$ is generated when the evaluation arrives at a constructor. The integer appearing in the annotation is the arity of the constructor, and the string is the name of the constructor. The arguments of this constructor that is under observation will be annotated with $(parentPortId, 1)$, $\ldots (parentPortId, arity)$, where $parentPortId$ is the place where the annotation $Cons$ has been written. In this way, it is easy to reconstruct the evaluation tree.

– Fun is generated when an observed lambda expression is applied. In the observations of Hood, lambdas have only one argument and one result. The argument of the lambda is annotated with the parent $(parentPortId,\ 0)$ and the result of the lambda is annotated with $(parentPortId,\ 1)$, where $parentPortId$ is the place where the annotation Fun has been written.

Therefore, in Hood it is not only possible to observe the normal forms of the bindings, but also when the bindings start the evaluation. Using this, it is easy to observe which bindings are demanded by another one. These annotations are processed and are shown in a useful way to the user.

4.2 Hood semantics

Let us consider now how to introduce Hood-like observations in the semantics. Let us remind that Hood users can annotate their programs to mark which structures have to be observed. Thus, we have to be able to annotate any structure.

$$
\begin{array}{lll}
e \rightarrow & x & \text{-- variable} \\
\mid & x^{@str} & \textbf{-- observed variable} \\
\mid & \lambda x.e & \text{-- lambda abstraction} \\
\mid & e\ x & \text{-- function application} \\
\mid & \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e & \text{-- recursive let} \\
\mid & C\ \overline{x_i} & \text{-- constructor application} \\
\mid & \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i} & \text{-- case expression} \\
\mid & p^{@(r,s)} & \textbf{-- observed pointer (internal)} \\
\mid & \lambda^{@[\overline{(r_i,s_i)}]}x.e & \textbf{-- observed lambda abstraction (internal)}
\end{array}
$$

Figure 3: Sestoft's normalized λ-calculus extended

Besides we need to write these annotations in a structure. In order to simplify the semantics we have decided to use a *file*. This file has to be post-processed to show the flattened observations.

So a semantic rule take the form $\Gamma : e \looparrowright f \Downarrow \Delta : w \looparrowright f'$, which is read as "the evaluation of the expression e reduces to the normal form w, transforms the heap Γ into Δ and adds observations to f generating a new file f'."

To achieve this, the judgments will have the form $\Gamma : e \looparrowright f \Downarrow \Delta : w \looparrowright f'$. As in the previous section, that means that expression e is evaluated in the heap Γ, we obtain as a result the expression w and the new heap is Δ. The difference is that now we have added the file f where we write the annotations that are produced during the reduction. The information in the file is added sequentially. Thus, we will write $f \circ \langle ann \rangle$ to indicate that we add the annotation ann at the end of the file f. The annotations that we will make will have the following form:

$$
\begin{array}{ll}
ann \rightarrow & (observeParent\ observePort)\ Observe\ str \\
\mid & (observeParent\ observePort)\ Enter \\
\mid & (observeParent\ observePort)\ Cons\ arity\ nameConstr \\
\mid & [\overline{(observeParent_i\ observePort_i)}]\ Fun
\end{array}
$$

They are similar to the ones Hood makes, there exist only two differences. The first difference, is that we omit the *portId* of Hood's annotations because, in our case, it corresponds to the line number in the file where the annotation has been produced. Then, *observeParent* and *observePort* are integers that correspond to the *parent* of Hood's annotations, that is the line number. To handle this we will need the function *length f* that returns the total number of lines in the file f. We will consider that the first line in the file is the 0 line. The second difference is produced in the λ-abstractions annotations: now they have a list of pairs *observeParent* and *observePort* corresponding with the list of closures that are observing the λ-abstraction behavior. We have decided to introduce this modification because λ-abstractions can be observed from different points and our aim is to show in a simple way the semantic rules.

We also have to be able to annotate any structure. This can be trivially done by allowing to annotate as *observable* any variable. Thus, we only need to slightly modify the language presented in Figure 1 to include an extra construction as shown in Figure 3. The expression $x^{@str}$ is the equivalent to the Hood expression `observe str x`. Note that, according to the syntax, these observations cannot appear directly in *applications* or *constructor applications*. However, this is not a drawback, since they may appear in a *recursive let*. Once the language allows to include observations, we have to deal with them in the rules. Besides we need a new kind of normal form $\lambda^{@(r,s)}x.e$: an observed lambda expression; and a new kind of observed pointers $p^{@(r,s)}$: pointers that are observed and refer to their parents. It is important to note that the programmer is not allowed to write this kind of expressions, as they only appear as auxiliary expressions in the rules. The notation (r, s) means that the parent of the pointer or lambda observed is r and this pointer or lambda is the s^{th}-child of r.

The rules in the original Sestoft's natural semantics (Figure 2) do not deal with observations. Thus, they are rewritten with the natural modification to include the annotation file. This file stores the observation in order to post-process it. For instance, our new *Case* rule is

$$\frac{\Gamma : e \hookrightarrow f \ \Downarrow \ \Delta : C_k\ \overline{p_j} \hookrightarrow f' \quad \Delta : e_k[\overline{p_j/x_{kj}}] \hookrightarrow f' \ \Downarrow \ \Theta : w \hookrightarrow f''}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \hookrightarrow f \ \Downarrow \ \Theta : w \hookrightarrow f''}$$

The rest of the rules in Figure 2 should be modified in the same way, that is, adding the file argument in all the configurations. However, in addition to rewriting these rules, it is also necessary to write completely new rules to deal with the new expressions, that is: observed variables, observed pointers, and observed lambda abstractions. The new rules we add to the system are those shown in Figure 4. Let us briefly describe their meaning:

– Rule *Var*@*S*. When we have to evaluate a binding annotated with the string *str*, we have to generate an annotation in the file $\langle 0\, 0\, \textit{Observe str} \rangle$ and we have to continue to evaluate that binding but with an annotation that indicates its parent, in this case $p^{@(n,0)}$ ($n = \textit{length}\ f$ is the length of the file at that point of the evaluation).

– Rule *Var*@*C*. When evaluating an expression such as $p^{@(r,s)}$, a new annotation $\langle r\, s\, \textit{Enter} \rangle$ is generated indicating that we enter to evaluate that binding. Then $p^{@(r,s)}$ evaluates to a constructor, so the observation $\langle r\ s\ \textit{Cons}\ k\ C \rangle$ is generated. This indicates that the binding whose parent is (r, s) has been reduced to the constructor C (whose arity is k). New bindings pointing to each argument of that constructor are generated. These bindings are annotated to indicate that they are being observed. Moreover, in this annotation we must indicate the argument number of the constructor and that its parent is in the corresponding line in the file.

– Rule *Var*@*F* and *Var*@*FO*. Now we have to evaluate $p^{@(r,s)}$. As in the previous case, we generate the annotation $\langle r\, s\, \textit{Enter} \rangle$, but in this case p reduces to a function. So we need to annotate this function saying that is being observed from (r, s). The difference between the rules depends on whether the function was previously annotated with an observation or not, rules

$$\frac{\mathit{\Gamma} : p^{@(\mathit{length}\ f,0)} \looparrowright f \circ \langle 0\,0\ \mathit{Observe\ str}\rangle \ \Downarrow\ \mathit{\Delta} : w \looparrowright f'}{\mathit{\Gamma} : p^{@\mathit{str}} \looparrowright f \ \Downarrow\ \mathit{\Delta} : w \looparrowright f'} \qquad \mathit{Var@S}$$

$$\frac{\mathit{\Gamma} : p \looparrowright f \circ \langle r\ s\ \mathit{Enter}\rangle \ \Downarrow\ \mathit{\Delta} : C\ \overline{p_i}^{k} \looparrowright f'}{\mathit{\Gamma} : p^{@(r,s)} \looparrowright f \ \Downarrow\ \mathit{\Delta} \cup [\overline{q_i \mapsto p_i^{@(\mathit{length}\ f',i)}}] : C\ \overline{q_i} \looparrowright f' \circ \langle r\ s\ \mathit{Cons}\ k\ C\rangle}\ \overline{q_i}\ \text{fresh} \qquad \mathit{Var@C}$$

$$\frac{\mathit{\Gamma} : p \looparrowright f \circ \langle r\ s\ \mathit{Enter}\rangle \ \Downarrow\ \mathit{\Delta} : \lambda x.e \looparrowright f'}{\mathit{\Gamma} : p^{@(r,s)} \looparrowright f \ \Downarrow\ \mathit{\Delta} : \lambda^{@(r,s)} x.e \looparrowright f'} \qquad \mathit{Var@F}$$

$$\frac{\mathit{\Gamma} : p \looparrowright f \circ \langle r\ s\ \mathit{Enter}\rangle \ \Downarrow\ \mathit{\Delta} : \lambda^{@obs} x.e \looparrowright f'}{\mathit{\Gamma} : p^{@(r,s)} \looparrowright f \ \Downarrow\ \mathit{\Delta} : \lambda^{@(r,s):obs} x.e \looparrowright f'} \qquad \mathit{Var@FO}$$

$$\mathit{\Gamma} : \lambda^{@obs} x.e \looparrowright f \ \Downarrow\ \mathit{\Gamma} : \lambda^{@obs} x.e \looparrowright f \qquad \mathit{Lam@}$$

$$\frac{\begin{array}{c} \mathit{\Gamma} : e \looparrowright f \ \Downarrow\ \mathit{\Delta} : \lambda^{@[\overline{(r_i,s_i)}]} x.e' \looparrowright f' \\ \mathit{\Delta} \cup \begin{bmatrix} q \mapsto e'[q'/x], \\ q' \mapsto p^{@(\mathit{length}\ f',0)} \end{bmatrix} : q^{@(\mathit{length}\ f',1)} \looparrowright f' \circ \langle\ [\overline{(r_i,s_i)}]\ \mathit{Fun}\rangle \ \Downarrow\ \mathit{\Theta} : w \looparrowright f'' \\ \text{where } q,\ q' \text{ fresh} \end{array}}{\mathit{\Gamma} : e\ p \looparrowright f \ \Downarrow\ \mathit{\Theta} : w \looparrowright f''} \qquad \mathit{App@}$$

Figure 4: Hood's natural semantics

Var@FO and *Var@F* respectively. In both cases, we add the observation to the function and continue to evaluate a new kind of normal form, an observed λ-abstraction.

– Rule *Lam@* establishes that $\lambda^{@obs}x.e$ is actually a normal form.

– Rule *App@* is the fundamental part of the new semantics. We are evaluating the application of an observed function. First, we generate the annotation in the file indicating that we are applying an observed function (note that *length* f' is the line where the annotation is made). Then we mark its argument as observable, and we use $(\mathit{length}\ f', 0)$ as its parent. In order to observe the result, we create a new observed binding whose parent is $(\mathit{length}\ f', 1)$. The ports are different to remember that one is the argument and the other is the result of the lambda.

  Note that it is not necessary to specify the application to an observed pointer $e\ p^{@(r,s)}$. The reason is that, in the syntax, we have restricted the places where an observed variable may appear, and in the rules we never substitute a variable by an observed pointer.

4.3 Correctness and equivalences between semantics

One important thing we must prove is that the observation marks do not change the meaning of an expression. That is, if we evaluate a marked expression and the equivalent one without marks, we should obtain the same normal form. Let us remark that this property must be satisfied because Hood observations do not modify the normal computation of Haskell programs.

The first difference of our semantics with respect to the original one consists in the observation marks. Thus, in order to compare them we need to provide a function to *remove* the observations. Thus, we define the following simple function that transforms any Sestoft's expression with observations, that we call Sestoft$^{@}$, into an expression without observations.

Definition 1 *We define the function that removes the observations as* $\mathbb{R}$: Sestoft$^{@}$ $\rightarrow$ Sestoft*. It is recursively defined, and all cases are trivial but the case of observed expressions:*

$$
\begin{array}{ll}
\mathbb{R}\ x & \overset{def}{=}\ x \\
\mathbb{R}\ x^{@str} & \overset{def}{=}\ x \\
\mathbb{R}\ (\lambda\ x.e) & \overset{def}{=}\ \lambda\ x.\mathbb{R}\ e \\
\mathbb{R}\ (e\ x) & \overset{def}{=}\ (\mathbb{R}\ e)\ x \\
\mathbb{R}\ (\textbf{letrec}\ \overline{x_i\ =\ e_i}^n\ \textbf{in}\ e) & \overset{def}{=}\ \textbf{letrec}\ \overline{x_i\ =\ \mathbb{R}\ e_i}^n\ \textbf{in}\ \mathbb{R}\ e \\
\mathbb{R}\ (C\ \overline{x_i}) & \overset{def}{=}\ C\ \overline{x_i} \\
\mathbb{R}\ (\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \rightarrow e_i}) & \overset{def}{=}\ \textbf{case}\ \mathbb{R}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \rightarrow \mathbb{R}\ e_i} \\
\mathbb{R}\ p^{@(r,s)} & \overset{def}{=}\ p \\
\mathbb{R}\ \lambda^{@obs}x.e & \overset{def}{=}\ \lambda x.\mathbb{R}\ e
\end{array}
$$

This function is extended to work with heaps, and configurations. Basically, $\mathbb{R}\ \Gamma$ *corresponds to* $\{p \mapsto \mathbb{R}\ e \mid (p \mapsto e) \in \Gamma\}$ *and* $\mathbb{R}\ (\Gamma : e) = \mathbb{R}\ \Gamma : \mathbb{R}\ e$*.*

But the most difficult problem is that in our rules we introduce new pointers and the expressions appearing in the rules contain pointers. Thus, we have to prove that the expressions appearing in both formalisms are equivalent. The pointers are kept in a heap; our new rules *Var*@*C* and *App*@ add new pointers to the heap. These pointers point to the original ones, but they are marked remembering that they are under observation. We would like to define $\Gamma : w \sqsubseteq \Gamma' : w'$ if w in Γ has the same value as w' in Γ', that is, if we obtain the same expressions by following the pointers. We can do that as follows: if p is a pointer in w and $(p \mapsto e) \in \Gamma$, let us substitute the occurrences of p in w by e. By doing so, we obtain new expressions that may have pointers; in that case, we iterate the process. Analogously, we perform the same process to deal with w'. If both processes end, then we look at the final expressions: if both expressions are the same, we can say that w and w' have the same value. The problem appears when one of the processes does not end. In that case, we have to take the *limit* of both sequences: w and w' have the same value if one sequence is a subsequence of the other.

Definition 2 *Let* e *be an expression and* Γ *be a heap.*

– *We denote by* $rp\ e$ *the substitution of all pointers in* e *by the symbol* $\perp$*.*

– *We denote $\mathit{\Gamma}\ e$ as the application of $\mathit{\Gamma}$ to the expression e. It is defined recursively, and all cases are trivial but the case of a pointer:*

$$\begin{array}{lll}
\mathit{\Gamma}\ p & \overset{def}{=}\ rp\ e & \textit{if } (p \mapsto e) \in \mathit{\Gamma} \\
\mathit{\Gamma}\ p & \overset{def}{=}\ \bot & \textit{if } (p \mapsto e) \notin \mathit{\Gamma} \\
\mathit{\Gamma}\ p^{@str} & \overset{def}{=}\ (\mathit{\Gamma}\ p)^{@str} & \\
\mathit{\Gamma}\ \lambda\ x.e & \overset{def}{=}\ \lambda\ x.\mathit{\Gamma}\ e & \\
\mathit{\Gamma}\ e\ p & \overset{def}{=}\ \mathit{\Gamma}\ e\ \mathit{\Gamma}\ p & \\
\mathit{\Gamma}\ \textbf{letrec}\ \overline{x_i\ =\ e_i}\ \textbf{in}\ e & \overset{def}{=}\ \textbf{letrec}\ \overline{x_i\ =\ \mathit{\Gamma}\ e_i}\ \textbf{in}\ \mathit{\Gamma}\ e & \\
\mathit{\Gamma}\ (C\ \overline{q_i}) & \overset{def}{=}\ C\ \overline{\mathit{\Gamma}\ q_i} & \\
\mathit{\Gamma}\ (\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to e_i}) & \overset{def}{=}\ \textbf{case}\ \mathit{\Gamma}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to \mathit{\Gamma}\ e_i} & \\
\mathit{\Gamma}\ p^{@(r,s)} & \overset{def}{=}\ (\mathit{\Gamma}\ p)^{@(r,s)} & \\
\mathit{\Gamma}\ \lambda^{@obs}\ x.e & \overset{def}{=}\ \lambda^{@obs}\ x.\mathit{\Gamma}\ e &
\end{array}$$

Definition 3 *Let e, e' be expressions and $\mathit{\Gamma}, \mathit{\Gamma}'$ be heaps. Let us consider the possibly infinite sequences*

$$s = [e, \mathit{\Gamma}\ e, \mathit{\Gamma}^2\ e, \mathit{\Gamma}^3\ e, \ldots] \qquad \textit{and} \qquad s' = [e', \mathit{\Gamma}'\ e', \mathit{\Gamma}'^2\ e', \mathit{\Gamma}'^3\ e', \ldots]$$

where the superscripts indicate the number of times that the heap $\mathit{\Gamma}$ is applied to the expression e.
We say that:

– $\mathit{\Gamma} : e \sqsubseteq \mathit{\Gamma}' : e'$ *if*
  1. $rp\ e = rp\ e'$
  2. $\forall i\, \exists j \geq i,\ rp\ \mathit{\Gamma}^i\ e = rp\ \mathit{\Gamma}'^j\ e'$
  3. $\forall j,\ rp\ \mathit{\Gamma}'^j\ e' \neq rp\ \mathit{\Gamma}'^{j+1}\ e' \Rightarrow \exists i \leq j,\ rp\ \mathit{\Gamma}^i\ e = rp\ \mathit{\Gamma}'^j\ e'$

– $\mathit{\Gamma} : e \sqsubseteq_{\mathbb{R}}\ \mathit{\Gamma}' : e'$ *if* $\mathit{\Gamma} : e \sqsubseteq \mathbb{R}\ (\mathit{\Gamma}' : e')$

According to the previous definition, if $\mathit{\Gamma} : e \sqsubseteq \mathit{\Gamma}' : e'$ we have that e and e' are equivalent, and the only differences may appear in the pointers that the expressions have. First we require that $rp\ e = rp\ e'$: if e is a lambda expression, an application, a recursive let, a constructor or a case expression, so must be e', and vice versa; the constructors and variables appearing at the top level must be the same. We do not require that the pointers be the same or that they point to the same expressions; what we require is that whenever there is a pointer in e, $(p \mapsto e_1) \in \mathit{\Gamma}$, then there must be a sequence of pointers $[q_1 \mapsto q_2, \ldots q_n \mapsto e_n] \subseteq \mathit{\Gamma}'$ such that q_1 appears in e', and if we apply all the corresponding substitutions in e and e' and then remove the pointers, we obtain the same expression. This is expressed in Definition 3.2 and Definition 3.3.

We need to prove the equivalence between the evaluation of a marked expression and the corresponding one without marks:

Theorem 1 *For all $e \in Sestoft$ and all $e^{@} \in Sestoft^{@}$ such that $e =_{\mathbb{R}} e^{@}$ then:*

$$\{\,\} : e^{@} \looparrowright \langle\,\rangle \Downarrow \Delta^{@} : w^{@} \looparrowright f \textit{ iff } \{\,\} : e \Downarrow \Delta : w$$

$$\textit{and } \Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} : w^{@}$$

In order to prove this theorem we need to take into account some considerations. First, in order to simplify the proof, we substitute the rule *Var* with a new one:

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma[p \mapsto e] : p \Downarrow \Delta \diamond [p \mapsto w] : w} \qquad Var'$$

The equivalence between the evaluation is maintained. The only difference with the original one is that now we do not remove the binding $(p \mapsto e)$, that is under evaluation, to evaluate the expression e. Besides, in this case $\Delta \diamond [p \mapsto w]$ means to update in Δ the expression corresponding to the pointer p with the expression w. It is very easy to prove the equivalences between both rules. If in the evaluation of $\Gamma : e$ the binding $(p \mapsto e)$ had been used, we would have entered in a "black hole," as we need p to evaluate p. In that case, with the new rule *Var′* the evaluation would not have finished and with the rule *Var* the evaluation would have stopped without finishing.

Proposition 1 *$\Gamma : e \Downarrow \Delta : w$ iff $\Gamma : e \Downarrow \Delta : w$ with the rule Var′.*

Proof. The proof is made by rule induction. All rules are trivial except the proof of rule *Var*. The relevant implication is $\Leftarrow$.

We know that $\Gamma : e \Downarrow \Delta : w$, so two alternatives can occur. In the first case, if binding $p \mapsto e$ was not used in this derivation, then we can apply induction hypothesis and the proof finishes. The second possibility is that it was used, then it is impossible that $\Gamma : e \Downarrow \Delta : w$; because at some point in the derivation we have to evaluate $\Gamma[p \mapsto e] : p$ again, but in that case we would obtain an infinite derivation and we would never reach the normal form. □

From now on, in the rest of this section, we will consider that we use rule *Var′* instead of rule *Var*. We also need to observe some properties that are invariant during the evaluation.

Definition 4 *$\Gamma : e$ is a* good *configuration if all the reachable pointers from e are bound in the heap.*

Proposition 2 *Let $\Gamma : e$ be a good configuration. If $\Gamma : e \Downarrow \Delta : w$ then $\Delta : w$ and $\Delta : e$ are good configurations.*

Proof. The proof is made by rule induction. The proof is very easy, as we only need to take care of rules that removes or creates new bindings. The only rule removing bindings was *Var*, but with the new rule *Var′* no binding is removed from the heap. Moreover, rules *Letrec* and *App@* are the only rules that creates bindings, but both rules maintain the property. □

In order to prove the main theorem we take advantage of some properties that the relations $\sqsubseteq$ and $\sqsubseteq_{\mathbb{R}}$ satisfy. It is very easy to prove them by considering the definition and, in some cases, by applying rule induction.

Property 1 *Let be $\Gamma : e$ a good configuration then $\forall p$ fresh, $\Gamma : e \sqsubseteq \Gamma \cup [p \mapsto e] : p$*

1. *If $\Gamma : e \sqsubseteq \Gamma' : e'$ and $\Gamma' : e' \sqsubseteq \Gamma'' : e''$ then $\Gamma : e \sqsubseteq \Gamma'' : e''$*
2. *If $\Gamma : e \sqsubseteq \Gamma' : e'$ then $\mathbb{R}\ (\Gamma : e) \sqsubseteq_{\mathbb{R}} \Gamma' : e'$*
3. *If $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma' : e'$ and $\Gamma' : e' \sqsubseteq \Gamma'' : e''$ then $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma'' : e''$*
4. *If $\Gamma : e \sqsubseteq \Gamma' : e'$ and $\Gamma' : e' \sqsubseteq_{\mathbb{R}} \Gamma'' : e''$ then $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma'' : e''$*
5. *If $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma' : e'$ then $\forall (q \mapsto e') \in \Gamma'$, $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma' \cup [q' \mapsto q^{@}] :\ e'\ [q'/q]$*
6. *If $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma' : e'$ then $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma' \cup [\overline{q_i \mapsto q^{@}_{i-1}}^{n}] :\ q^{@}_{n}$, $n \geq 0$ and $q_0 = e'$*
7. *Let be $\Gamma[\overline{q_i \mapsto q^{@}_{i-1}}^{n}] :\ q_n$, $n \geq 0$ and $q_0 = e$*
   If $\Gamma : e \Downarrow \Delta : w$ then $\Gamma : q_n \Downarrow \Delta \cup [\overline{q_i \mapsto w^i}^{n}, \overline{q'_i \mapsto q''^{@}_{i}}] :\ w^0$, $\forall i\ \exists j,\ w = (\mathbb{R}\ \Delta)^j w^i$ and $\overline{q'_i}$ are fresh.

Let us briefly describe them: the Property 1.1 reflects the fact that the relation $\sqsubseteq$ is transitive. Property 1.2 reflects the relation between $\sqsubseteq$ and $\sqsubseteq_{\mathbb{R}}$. Property 1.3 and Property 1.4 explain how to combine both equivalences, $\sqsubseteq$, and $\sqsubseteq_{\mathbb{R}}$. Property 1.5 reflects the fact that the relation $\sqsubseteq_{\mathbb{R}}$ is not affected if we add in the second configuration intermediate pointers. Property 1.6 reflects the fact that it is equivalent for relation $\sqsubseteq_{\mathbb{R}}$ to have an expression in the configuration or a pointer that points to it. Finally, Property 1.7 reflects that the semantic reduction $\Downarrow$ is equivalent if we have an expression in the configuration or a pointer that points to it.

Using these properties, we can prove a proposition that is more general than the original Theorem 1. In addition to the equivalence between both semantics, we have to prove a companion property $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} : e'^{@}$ for any expression e'. This is an auxiliary result needed in the proof.

Proposition 3 *Let be $e, e' \in Sestoft$, all $e^{@}, e'^{@} \in Sestoft^{@}$, $\Gamma : e$, $\Gamma^{@} : e^{@}$, $\Gamma : e'$ and $\Gamma^{@} : e'^{@}$ good configurations, such that $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e^{@}$ and $\Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e'^{@}$ then:*

$$\Gamma^{@} : e^{@} \hookrightarrow f \Downarrow \Delta^{@} : w^{@} \hookrightarrow f' \text{ iff } \Gamma : e \Downarrow \Delta : w,\ \Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} : w^{@}$$
$$\text{and } \Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} : e'^{@}$$

Proof. Here we only present a sketch of the proof. The interested reader can find the complete proof in the appendix of the paper. The proof is made by rule induction. In order to make the proof easier to read, we will drop the observation file from the rules since it does not participate in the evaluation of the expressions. This file is only a side effect of the evaluation. Notice that, if

Heap	Control	Environment	Stack	rule
Γ	$(e\ x)$	$E \cup [x \mapsto p]$	S	app1
$\Longrightarrow \Gamma$	e	$E \cup [x \mapsto p]$	$p : S$	
Γ	$\lambda y.e$	E	$p : S$	app2
$\Longrightarrow \Gamma$	e	$E \cup [y \mapsto p]$	S	
$\Gamma \cup [p \mapsto (e', E')]$	x	$E \cup [x \mapsto p]$	S	var1
$\Longrightarrow \Gamma$	e'	E'	$\#p : S$	
Γ	$\lambda y.e$	E	$\#p : S$	var2
$\Longrightarrow \Gamma \cup [p \mapsto (\lambda y.e, E)]$	$\lambda y.e$	E	S	
Γ	**letrec** $\{\overline{x_i = e_i}\}$ **in** e	E	S	letrec (1)
$\Longrightarrow \Gamma \cup \overline{[p_i \mapsto (e_i, E')]}$	e	E'	S	
Γ	**case** e **of** $alts$	E	S	case1
$\Longrightarrow \Gamma$	e	E	$(alts, E) : S$	
Γ	$C_k\ \overline{x_i}$	$E \cup [\overline{x_i \mapsto p_i}]$	$(alts, E') : S$	case2 (2)
$\Longrightarrow \Gamma$	e_k	$E' \cup [\overline{y_{ki} \mapsto p_i}]$	S	
Γ	$C\ \overline{x_i}$	E	$\#p : S$	var3
$\Longrightarrow \Gamma \cup [p \mapsto (C\ \overline{x_i}, E)]$	$C\ \overline{x_i}$	E	S	

(1) $\overline{p_i}$ are distinct and fresh w.r.t. Γ, **letrec** $\{\overline{x_i = e_i}\}$ **in** e, and S. $E' = E \cup [\overline{x_i \mapsto p_i}]$
(2) Expression e_k corresponds to alternative $C_k\ \overline{y_{ki}} \rightarrow e_k$ in $alts$

Figure 5: Abstract machine *Mark-2*

configurations are good, the last case considered in Definition 2 ($(p \mapsto e) \notin \Gamma$) cannot occur. However, the definition must take the case into account for the sake of completeness. $\Box$

Finally, as an immediate corollary of the previous proposition, we have that Theorem 1 holds.

5 Abstract Machine

In this section we introduce an abstract machine equivalent to the semantics shown in the previous section. In order to do that, we extend one of the abstract machines defined by Sestoft in [Ses97]. In that work, he introduced several abstract machines in sequence, respectively called *Mark-1*, *Mark-2* and *Mark-3*. The principal difference between those machines are the bindings in the heap. The *Mark-1* machine binds pointers with expressions, while the *Mark-2* machine binds pointers with closures, that is a pair of expression an environment, and the *Mark-3* machine maps pointers with expressions where the variables have been changed by the De Bruijn index [DB72, DB78]. We will use the machine *Mark-2* for deriving the new rules for our semantics because it is close enough to reality and it does not have many low–level details as *Mark-3* machine.

Definition 5 *A* configuration *of Mark-2 is a quadruple of the form* (Γ, e, E, S) *where* Γ *represents the heap,* e *is the expression that is currently under evaluation (denoted by* 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.*

– *The heap Γ binds pointers to closures which, in turn, is a pair (e, E), where e is an expression and E represents the environment which maps the free variables of e to the corresponding pointers.*

– *The stack S stores three kinds of objects: arguments p_i of pending applications, case alternatives $(alts, E)$ of pending pattern matchings, and marks $\#p$ of pending updates.*

In Figure 5 the operational rules of the *Mark-2* machine are shown. The rules $app1$ and $app2$ are needed to evaluate an application, and they correspond to semantic rule App. Rule $app1$ begins the computation corresponding to the left subtree, while $app2$ begins the computation corresponding to the right subtree. Rules $var1$, $var2$ and $var3$ reduce the evaluation of a variable, like semantic rule *Var*. Rule $var1$ starts the computation of the variable. Note that when the evaluation reach the normal form, it is necessary to update the heap. Thus, rule $var1$ pushes in the stack the update mark to remember this fact. When it is reached the corresponding normal form, λ-abstraction or constructor, the update is produced by rules $var2$ or $var3$ respectively. Rule *letrec* evaluates a **letrec** expression, and it corresponds to semantic rule *Letrec*. Finally, rule $case1$ begins the computation corresponding to the left subtree, while $case2$ begins the computation corresponding to the right subtree.

The main theorem proved by Sestoft, is that successful derivations of the machine are exactly the same as those of the semantics. The reason why the environments are needed is because control expressions, lambda expressions and alternatives keep their original variables, and in execution we need to know their associated pointers. Basically, the machine consists of a flattening of the semantic tree.

Following the same ideas, we have derived new machine rules for the new semantic rules. These new rules are presented in Figure 6. To include observations in our machine we need to add some modifications to the original machine. First, we need a new column of side effects for the rules containing the observations. In fact, we have to include the side effect column in all the rules appearing in Figure 5, but we do not show it because it is quite straight forward (it is necessary to include that column with no modifications in the file) since these rules do not have to deal with observations and they do not generate observations. Second, we need to add a new type of objects in the stack. This kind of objects $@(r, s)$ corresponds to pending observations. Let us remark the following facts:

– The side effects are produced at the same time as the evaluation of the program takes place.

– Observations can be obtained even if the program does not finish its computation, because they are written to a file.

Let us briefly describe the new rules introduced into *Mark-2* machine to deal with the new expressions:

Heap	Control	Environment	Stack	Side Effect	rule
Γ	$x^{@str}$	E	S	f	var@S
$\Longrightarrow \Gamma$	$x^{@(length\ f,0)}$	E	S	$f \circ \langle 0\,0\ Observe\ str \rangle$	
Γ	$x^{@(r,s)}$	E	S	f	var1@
$\Longrightarrow \Gamma$	x	E	$@(r,s) : S$	$f \circ \langle r\ s\ Enter \rangle$	
Γ	$\lambda y.e$	E	$@(r,s) : S$	f	var2@
$\Longrightarrow \Gamma$	$\lambda^{@[(r,s)]} y.e$	E	S	f	
Γ	$\lambda^{@obs} y.e$	E	$@(r,s) : S$	f	var2@@
$\Longrightarrow \Gamma$	$\lambda^{@(r,s):obs} y.e$	E	S	f	
Γ	$C\ \overline{x_i}$	E	$@(r,s) : S$	f	var3@ (1)
$\Longrightarrow \Gamma \cup \overline{[q_i \mapsto (x_i^{@(length\ f,i)}, E)]}$	$C\ \overline{x_i}$	$[\overline{x_i \mapsto q_i}]$	S	$f \circ \langle r\ s\ Cons\ k\ C \rangle$	
Γ	$\lambda^{@[\overline{(r_i,s_i)}]} y.e$	E	$p : S$	f	app2@ (2)
$\Longrightarrow \Gamma \cup \begin{bmatrix} q \mapsto (e, E \cup [y \mapsto q_1]), \\ q_1 \mapsto (arg^{@(length\ f,0)}, [arg \mapsto p]) \end{bmatrix}$	$ap^{@(length\ f,1)}$	$[ap \mapsto q]$	S	$f \circ \langle\ [\overline{(r_i, s_i)}]\ Fun \rangle$	

(1) $\overline{q_i}$ are distinct and fresh w.r.t. Γ, $C\ \overline{x_i}$, and S.

(2) q, q_1 are distinct and fresh w.r.t. Γ, $\lambda^{@[\overline{(r_i,s_i)}]} y.e$, and S.

Figure 6: Abstract machine *Mark-2*$^{@}$, rules for Hood

$var@S$ When the control expression is a variable under observation with the string str, we add in the file the corresponding observation saying that the reduction of the observation has started and modify consequently the control expression. This rule corresponds to the semantic rule $Var@S$.

$var1@$ When the control expression is a closure under the internal observation $@(r, s)$, we add in the file the observation saying that the machine has started to reduce that closure. This rule corresponds to the first part of the premises of the semantics rules $Var@F$ and $Var@C$.

$var2@$ **and** $var2@@$ When we arrive at a λ-abstraction and on the top of the stack there is a pending observation, we only annotate the λ-abstraction with the corresponding observation mark. Rule $var2@$ corresponds to the premises of the semantic rule $Var@F$ and rule $var2@@$ corresponds to the premises of the semantic rule $Var@FO$. The same applies to the semantic rules $Var@F$ and $Var@FO$: the only difference between these rules depends on whether the lambda abstraction was previously observed or not, rules $var2@$ or $var2@@$ respectively.

$var3@$ When we arrive at a constructor and on the top of the stack there is a pending observation, we perform the observation, that indicates that the closure has been reduced to a constructor, and continue the evaluation considering that now we are observing the constructor arguments. This rule corresponds to the second part of the premises of the semantic rule $Var@C$.

$app2@$ When the evaluation arrives at a function that is being observed, first the observation is done and afterwards the body of the function is evaluated. And so it is possible to remember that not only the argument of the function is being observed but also the result. This is done by adding the corresponding bindings in the heap. This rule corresponds to the semantic rule $App@$.

5.1 Correctness and equivalences between abstract machines

Now we have to prove the correctness of the new machine, that we call *Mark-2*$^{@}$, with respect to the new semantics (Figure 4). Since we have proved that our semantics and the original one are equivalent, as a corollary we will also have that *Mark-2*$^{@}$ and *Mark-2* are equivalent.

In order to prove the correctness, we need to take into account that the semantic heap, and the *Mark-2*$^{@}$ heap are structurally different. The *Mark-2*$^{@}$ heap has environments that map the free variables of the expression associated to the environment with its corresponding pointers; whereas, in the semantic heap, the free variables are substituted with their corresponding pointers. Thus, we need to define an equivalence between them, applying the corresponding environment to the corresponding expression, that is what the following definition establish.

Definition 6 *Let $\mathit{\Gamma}$ be a heap of the Mark-2$^{@}$, $\mathit{\Gamma}_{env}$ denotes the following heap* $[p \mapsto E\ e \mid (p \mapsto (e, E)) \in \mathit{\Gamma}]$

Theorem 2 *For all $e \in Sestoft^{@}$ then:*

$$\{\,\} : e \hookrightarrow \langle\,\rangle \ \Downarrow\ \mathit{\Delta} : w \hookrightarrow f \ \textit{iff}\ (\{\,\}, e, \{\,\}, [\,], \langle\,\rangle) \Longrightarrow^{*} (\mathit{\Delta}', w', E', [\,], f')$$

$$\textit{and}\ \mathit{\Delta} = \mathit{\Delta}'_{env},\ w = E'\ w',\ f = f'$$

To prove this theorem and to separate the ideas over the environments from the principal proof, we define an auxiliary machine, *Mark-1*$^{@}$, which is similar to *Mark-2*$^{@}$. The technical difference is that it does not have the environment component (see Figure 7) in the control expression, in the stack and in the heap's bindings. For example, rule *app*1 now has as control expression $e\ p$, while in *Mark-2*$^{@}$ machine the expression was $e\ x$ and the environment contained the binding $\{x \mapsto p\}$. In both cases, they push in the stack the pointer p and follows with the evaluation of the expression e. Remember that the environment can be considered as a delayed substitution, so if we apply it to each corresponding expression we get the *Mark-1*$^{@}$ machine. Now, we need to prove that *Mark-1*$^{@}$ is equivalent to the Hood's natural semantics (Figure 4) and to the machine *Mark-2*$^{@}$. As a corollary we get Theorem 2.

First, we prove the equivalence between both machines. To do that we define an equivalence between the configuration of the machine *Mark-1*$^{@}$ and *Mark-2*$^{@}$.

Definition 7

1. *Let S be a stack of the Mark-2$^{@}$ then S_{env} denotes the following stack:*

$$\begin{cases} p : S'_{env} & \text{if } S = p : S'; \\ \#p : S'_{env} & \text{if } S = \#p : S'; \\ @(r, s) : S'_{env} & \text{if } S = @(r, s) : S'; \\ E\ alts' : S'_{env} & \text{if } S = (alts', E) : S'; \\ [\,] & \text{if } S = [\,]; \end{cases}$$

Heap	Control	Stack	Side Effect	rule	
Γ	$(e\ p)$	S	f	app1	
$\Longrightarrow \Gamma$	e	$p : S$	f		
Γ	$\lambda y.e$	$p : S$	f	app2	
$\Longrightarrow \Gamma$	$e[p/y]$	S	f		
$\Gamma \cup [p \mapsto e]$	p	S	f	var1	
$\Longrightarrow \Gamma$	e	$\#p : S$	f		
Γ	$\lambda y.e$	$\#p : S$	f	var2	
$\Longrightarrow \Gamma \cup [p \mapsto \lambda y.e]$	$\lambda y.e$	S	f		
Γ	$\textbf{letrec}\ \{\overline{x_i = e_i}\}\ \textbf{in}\ e$	S	f	letrec	(1)
$\Longrightarrow \Gamma \cup \overline{[p_i \mapsto \hat{e}_i]}$	$\hat{e}$	S	f		
Γ	$\textbf{case}\ e\ \textbf{of}\ alts$	S	f	case1	
$\Longrightarrow \Gamma$	e	$alts : S$	f		
Γ	$C_k\ \overline{p_i}$	$alts : S$	f	case2	(2)
$\Longrightarrow \Gamma$	$e_k[\overline{p_i/y_{ki}}]$	S	f		
Γ	$C_k\ \overline{p_i}$	$\#p : S$	f	var3	
$\Longrightarrow \Gamma \cup [p \mapsto C_k\ \overline{p_i}]$	$C_k\ \overline{p_i}$	S	f		
Γ	$p^{@str}$	S	f	var@S	
$\Longrightarrow \Gamma$	$p^{@(n,0)}$	S	$f \circ \langle 0\ 0\ \mathit{Observe}\ str\rangle$		
Γ	$p^{@(r,s)}$	S	f	var1@	
$\Longrightarrow \Gamma$	p	$@(r,s) : S$	$f \circ \langle r\ s\ \mathit{Enter}\rangle$		
Γ	$\lambda y.e$	$@(r,s) : S$	f	var2@	
$\Longrightarrow \Gamma$	$\lambda^{@[(r,s)]} y.e$	S	f		
Γ	$\lambda^{@obs} y.e$	$@(r,s) : S$	f	var2@@	
$\Longrightarrow \Gamma$	$\lambda^{@(r,s):obs} y.e$	S	f		
Γ	$C\ \overline{p_i}^k$	$@(r,s) : S$	f	var3@	(3)
$\Longrightarrow \Gamma \cup \overline{[q_i \mapsto p_i^{@(n',i)}]}$	$C\ \overline{q_i}$	S	$f \circ \langle r\ s\ \mathit{Cons}\ k\ C\rangle$		
Γ	$\lambda^{@[\overline{(r_i,s_i)}]} y.e$	$p : S$	f	app2@	(4)
$\Longrightarrow \Gamma \cup \begin{bmatrix} q \mapsto e[q_1/y], \\ q_1 \mapsto p^{@(n',0)} \end{bmatrix}$	$q^{@(n',1)}$	S	$f \circ \langle\ [\overline{(r_i,s_i)}]\ \mathit{Fun}\rangle$		

(1) $\overline{p_i}$ are distinct and fresh w.r.t. Γ, **letrec** $\{\overline{x_i = e_i}\}$ **in** e, and S. $\hat{e} = e[\overline{p_i/x_i}]$
(2) Expression e_k corresponds to alternative $C_k\ \overline{y_{ki}} \to e_k$ in *alts*
(3) $\overline{q_i}$ are distinct and fresh w.r.t. Γ, $C\ \overline{p_i}$, and S.
(4) q, q_1 are distinct and fresh w.r.t. Γ, $\lambda^{@[\overline{(r_i,s_i)}]} y.e$, and S.

Figure 7: Abstract machine *Mark-1*$^{@}$

2. *Let* (Γ, e, E, S, f) *be a configuration of the Mark-2*$^{@}$ *then:*

   $(\Gamma, e, E, S, f)_{env}$ *denotes the Mark-1*$^{@}$ *configuration* $(\Gamma_{env}, E\ e, S_{env}, f)$

3. *Let* (Γ, e, S, f) *be a configuration of the Mark-1*$^{@}$*,* $(\Gamma', e', E', S', f')$ *be a configuration of the Mark-2*$^{@}$ *then* $(\Gamma, e, S, f) \equiv_{env} (\Gamma', e', E', S', f')$ *if:*

$$(\Gamma, e, S, f) = (\Gamma', e', E', S', f')_{env}$$

Definition 7 expresses how to convert a stack (Definition 7.1), and a configuration (Definition 7.2) of *Mark-2*$^{@}$ machine into a corresponding stack, or configuration of *Mark-1*$^{@}$ machine, basically applying the environments, and establishing an equivalence between the configurations of *Mark-1*$^{@}$ and *Mark-2*$^{@}$ machines (Definition 7.3), that is that *Mark-1*$^{@}$ machine configuration is equal to remove the environments in *Mark-2*$^{@}$ machine configuration. The following property establishes the equivalence between both machines.

Proposition 4 *Given a configuration* (Γ, e, S, f) *of the Mark-1*$^{@}$ *machine and a configuration* $(\Gamma', e', E', S', f')$ *of the Mark-2*$^{@}$ *machine such that* $(\Gamma, e, S, f) \equiv_{env} (\Gamma', e', E', S', f')$ *then:*

$$(\Gamma, e, S, f) \Longrightarrow^{*} (\Delta, w, S, f) \text{ iff } (\Gamma', e', E', S', f') \Longrightarrow^{*} (\Delta', w', E', S', f'')$$

$$\text{and } (\Delta, w, S, f) \equiv_{env} (\Delta', w', E', S', f'').$$

Proof. It is proved by induction over the rules. It is trivial, as all the rules are equivalent via $\equiv_{env}$.

Now, following the ideas presented by Sestoft, in order to prove the equivalence between the machine and the semantic rules, we will need the idea of *balanced traces*. First, let us concentrate in the abstract machine without observations. Intuitively, a balanced trace corresponds to the evaluation of a complete expression but starting with a non-empty stack. In this context the *trace* is a sequence of rules of the abstract machine. The main characteristics of a balanced trace are:

– The original stack remains untouched. None of the rules applied in a balanced trace change or even consult a value in that stack. The rules may put new values on top of the original stack only.

– At the end of the execution of the trace the final stack is the original one. No new values are on top of the stack when the execution of the trace ends, the final stack is the same as the original one.

Since a balanced trace is the execution of a complete expression, any non-empty balanced trace must begin with $app1$, $var1$ or let. A balanced trace cannot start with $app2$ nor $var2$ because in both cases at the top of the stack there is a *pending* pointer that indicates that evaluation of a expression had already been started.

If the trace begins with $app1$, it produces an intermediate stack of the form $p : S$. Since the only rule that uses the pointer and restores the stack is $app2$, it must appear later in the trace. Now, in the control we have to evaluate expression e, so the subtrace between $app1$ and its corresponding $app2$ is a balanced one. Analogously, if the trace starts with $var1$ it should end with either $var2$ or $var3$; if it starts with $case1$, $case2$ eventually appears (in this case the evaluation may continue with the corresponding expression in the alternatives).

So, in the abstract machine *Mark-1*, the balanced traces are the ones derived from the following grammar:

$$\begin{aligned} bal ::= \epsilon \mid app1\ bal\ app2\ bal \mid var1\ bal\ var2 \mid let\ bal \mid \\ var1\ bal\ var3 \mid case1\ bal\ case2\ bal \end{aligned}$$

Now let's get back to our machine *Mark-1*$^{@}$. The new rule $var@S$ does not modify the stack, then a balanced trace can be of the form $var@S\ bal$ that corresponds to the natural semantic rule $Var@S$. Our rule $var@1$ follows the same reasoning as the rule $var1$. Thus, $var1@\ bal\ var2@$ and $var1@\ bal\ var3@$ are also balanced traces that correspond to the semantic rules $Var@F$ and $Var@C$ respectively. And the ϵ trace corresponds to the *Cons*, *Lam* or *Lam*@ rule.

Definition 8

– *A* balanced trace *is a sequence of rules that can be derived from:*

$$\begin{aligned} bal ::= {} & \epsilon \mid app1\ bal\ app2\ bal \mid var1\ bal\ var2 \mid var1\ bal\ var3 \mid let\ bal \mid \\ & case1\ bal\ case2\ bal \mid app1\ bal\ app2@\ bal \mid var@S\ bal \mid \\ & var1@\ bal\ var2@ \mid var1@\ bal\ var2@@ \mid var1@\ bal\ var3@ \end{aligned}$$

– *Let* Γ_1, Γ_2 *be two heaps,* $e_1, e_2 \in Sestoft^{@}$*,* S_1, S_2 *be two stacks, and* f_1, f_2 *be two files; we say that the computation*

$$(\Gamma_1, e_1, S_1, f_1) \Longrightarrow^* (\Gamma_2, e_2, S_2, f_2)$$

is balanced *if* $S_1 = S_2$ *and no rules involved in the computation consult any value in* S_1*.*

Then it is easy to prove that a balanced computation of the machine *Mark-1*$^{@}$ corresponds to a balanced trace.

Proposition 5 *Let* Γ_1, Γ_2 *be two heaps,* $e_1, e_2 \in Sestoft^{@}$*,* S_1, S_2 *be two stacks, and* f_1, f_2 *be two files. The computation*

$$(\Gamma_1, e_1, S_1, f_1) \Longrightarrow^* (\Gamma_2, e_2, S_2, f_2)$$

is balanced iff the sequence of rules applied is a balanced trace.

Proof.

$\Longrightarrow$ The discussion above shows that any balanced computation must start with one of the following rules:

$app1$. Then any balanced computation must have one of the following forms: $app1\ bal\ app2\ bal$ or $app1\ bal\ app2@\ bal$.

let. The balanced computation must have the form $let\ bal$.

$case1$. The balanced computation must be $case1\ bal\ case2\ bal$.

$var@S$. The balanced computation must be $var@S\ bal$.

$var1$ or $var1@$. In this case, we put an element on the top of the stack that must be removed by one of the following rules: $var2$, $var3$, $var2@$, $var2@@$, or $var3@$. The computation in between must be balanced and then generated by a balanced trace. The trace might be followed by another balanced computation. But since the control after $var2$, $var2@$, and $var2@@$ is a lambda, and after $var3$ and $var3@$ is a constructor, only a rule that reads the top of the stack can be applied. If that were the case, the computation would not be any longer balanced. So the balanced trace must end at the rule $var2$, $var3$, $var2@$, $var2@@$, or $var3@$.

$\Longleftarrow$ The proof is made by structural induction on the form of the balanced trace. The induction base is when the trace is ϵ which is trivial, since there is no computation. All the induction cases are similar, so let us concentrate on $app1\ bal\ app2@\ bal$.

In this case we have $e_1 = e'\ x$, so

$$(\varGamma_1, e\ x, S_1, f_1) \Longrightarrow (\varGamma', e', p : S, f')$$

that meets the requirements of this Proposition. Then, by structural induction, we have a computation that verifies the thesis of the current Proposition. Thus, we have

$$(\varGamma', e', p : S, f') \Longrightarrow^* (\varGamma'', e'', p : S, f'')$$

Then we have an application of $app2@$:

$$(\varGamma'', e'', p : S, f'') \Longrightarrow (\varGamma''', e''', S, f''')$$

which meets this Proposition. Finally, by structural induction, we have a computation that meets the thesis of this proposition:

$$(\varGamma''', e''', S, f''') \Longrightarrow^* (\varGamma_2, w_2, S_2, f_2)$$

The following proposition proves the equivalence between the semantics and the *Mark-1*$^@$ machine.

Proposition 6 *Given $\varGamma$ a heap, $e \in Sestoft^@$, S a stack, f a file then:*

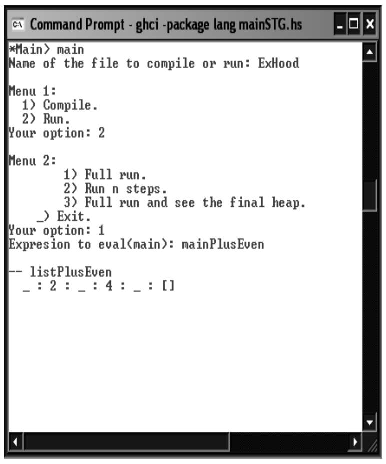
$$\varGamma : e \looparrowright f \ \Downarrow\ \varDelta : w \looparrowright f' \text{ iff } (\varGamma, e, S, f) \Longrightarrow^* (\varDelta, w, S, f') \text{ is balanced.}$$

Proof. The complete proof can be found in the appendix of this paper and it consists on an extension of the proof that appears in [Ses97], but taking into account the new rules and the file used to store the observations.

As a corollary of Proposition 6 and Proposition 4 we get Theorem 2. Also, as a corollary of Theorem 1, Theorem 2 and Sestoft's equivalence theorems (see [Ses97]) we get the equivalence between the original *Mark-2* and our *Mark-2*$^@$ with observations.

Corollary 1 *For all $e \in Sestoft$, $e^@ \in Sestoft^@$ such that $e = \mathbb{R}\ e^@$ then:*

$$\begin{gathered}(\{\,\}, e, \{\,\}, [\,]) \Longrightarrow^* (\varDelta, w, E, [\,]) \textit{ derivation of Mark-2} \\ \textit{iff} \\ (\{\,\}, e^@, \{\,\}, [\,], \langle\rangle) \Longrightarrow^* (\varDelta^@, w^@, E^@, [\,], f) \textit{ derivation of Mark-2}^@ \\[1ex] \textit{and } \varDelta_{env} : E\ w \sqsubseteq_{\mathbb{R}}\ \varDelta^@_{env} : E^@\ w^@.\end{gathered}$$



Command Prompt - ghci -package lang mainSTG.hs

```
*Main> main
Name of the file to compile or run: ExHood

Menu 1:
  1) Compile.
  2) Run.
Your option: 2

Menu 2:
        1) Full run.
        2) Run n steps.
        3) Full run and see the final heap.
    _) Exit.
Your option: 1
Expresion to eval(main): mainPlusEven

-- listPlusEven
  _ : 2 : _ : 4 : _ : []
```

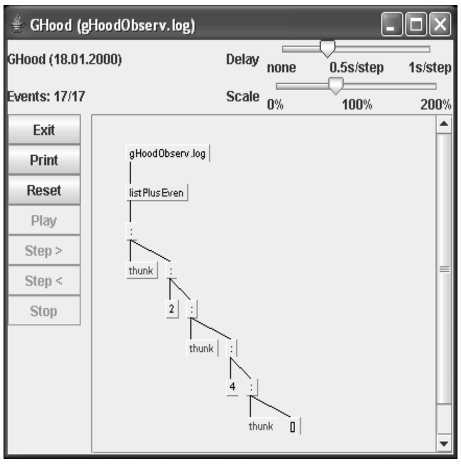


Figure 8: Debugging environment: Produces textual output (left) or graphical output (right)

6 Implementation Issues and Reduction Example

In this section we briefly describe some details of our implementation. As we said in the introduction, we have implemented a prototype of the abstract machine presented in this paper. In order to do that, we have followed the work done in [EP09], where we presented a very simple imperative machine. That machine was used as an intermediate step between the STG abstract machine and the C code generated by the GHC compiler. By using that intermediate machine, a translation scheme was provided allowing to translate the language expressions presented in that paper into executable code. Moreover, in [EP09] a proof of correctness was also provided for the translation and two different models of implementations were analyzed: the so-called push-enter model, and the new one based on an eval-apply style. In the case of the current paper, the implementation is simpler than there because we need to implement a simple machine (the *Mark-2* machine) and the language being used has not so many optimizations as there. The main difference between this machine and the STG machine is that the λ-abstraction application in *Mark-2* machine is made one by one to its arguments, while in the STG a λ-abstraction is applied to all its arguments in a single step, so all the steps proposed in [EP09] can be followed in a similar way to get the final implementation.

The only feature of our current machine that is more difficult to handle than in [EP09] is dealing with the observations file. Let us remind that the observation annotations have been produced in an external file. This has the advantage that even when the computation does not finish we get the observations. The observation file of the implementation has exactly the same structure presented in this paper. In order to obtain a flattened observation equivalent to that of

Hood, the file needs to be post-processed. This post-process is very simple, and it can be done analyzing the lines in the file in a sequential way. For each *str* observation mark, it is produced a tree with the observations that depends on that *str*. These trees are then flattened to show a Hood observation. For the sake of clarity, this process is shown in the next example (see Section 6.1).

In order to asses the usefulness of a debugger, it is important to obtain experimental results by using the system to debug a benchmarks collection. Fortunately, such work has already been done in [CRW01], where errors were introduced in several programs (ranging form 100 to 900 lines of source code) and a set of programmers used the debugger to find those errors. The results were quite satisfactory, certifying that Hood is a useful debugger.

Finally, we would like to comment that in our environment (see Figure 8) 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 [Rei01] 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 them.

In order to better understand the behavior of the implementation, we present an example of execution of an expression with observations at *Mark-2*$^@$ level.

6.1 Example of reduction of and expression in the *Mark-2*$^@$

As we have proved that the *Mark-2*$^@$ machine and the semantics rules are equivalent, the reduction of an expression produces the same result. Thus, we will only show the details of the example at *Mark-2*$^@$ level. In order to better understand how the evaluation produces the annotations in the file and how this can be post-processed to obtain a flattened observation, we will show an example where an expression with observation marks is reduced in the *Mark-2*$^@$ machine.

Example 1 In this example we observe a number with two different observation marks. We start with the following initial expression that we call e_0:

```
letrec
  ten    = 10
  tenO   = ten@{obs1}
  tenOO  = tenO@{obs2}
in tenOO
```

Now we present the reduction steps of this expression in the *Mark-2*$^@$ machine. Note that in the *Mark-2*$^@$ machine there is no optimization of the environments related with each expression. Thus, they have all the variables instead of only having the free variable of each expression. In the reduction of e_0 only one environment is produced that we will call it E_0, and it is the following:

$$E_0 = \left\{ \begin{array}{l} ten \mapsto p_1 \\ tenO \mapsto p_2 \\ tenOO \mapsto p_3 \end{array} \right\}$$

In order to simplify the presentation, in the steps where some of the configuration components do not change, we will show them with: "...".

Heap	Control	Env.	Stack	Side effect	rule
$\{\}$	e_0	$\{\}$	$[\,]$	$\{\}$	*letrec*
$\Longrightarrow \left\{\begin{array}{l} p_1 \mapsto (10, E_0) \\ p_2 \mapsto (ten^{@\mathrm{obs1}}, E_0) \\ p_3 \mapsto (tenO^{@\mathrm{obs2}}, E_0) \end{array}\right\}$	$tenOO$	E_0	$[\,]$	$\{\}$	*var1*
$\Longrightarrow \left\{\begin{array}{l} p_1 \mapsto (10, E_0) \\ p_2 \mapsto (ten^{@\mathrm{obs1}}, E_0) \end{array}\right\}$	$tenO^{@\mathrm{obs2}}$	E_0	$[\#p_3]$	$\{\}$	*var@S*
$\Longrightarrow \{\ldots\}$	$tenO^{@(0,0)}$	E_0	$[\#p_3]$	$\{0\ 0\ \mathit{Observe}\ \mathrm{obs2}\}$	*var1@*
$\Longrightarrow \{\ldots\}$	$tenO$	E_0	$@(0,0) : [\ldots]$	$\left\{\begin{array}{l} \ldots \\ 0\ 0\ \mathit{Enter} \end{array}\right\}$	*var1*
$\Longrightarrow \{p_1 \mapsto (10, E_0)\}$	$ten^{@\mathrm{obs1}}$	E_0	$\#p_2 : [\ldots]$	$\{\ldots\}$	*var@S*
$\Longrightarrow \{\ldots\}$	$ten^{@(2,0)}$	E_0	$[\ldots]$	$\left\{\begin{array}{l} \ldots \\ 0\ 0\ \mathit{Observe}\ \mathrm{obs1} \end{array}\right\}$	*var1@*
$\Longrightarrow \{\ldots\}$	ten	E_0	$@(2,0) : [\ldots]$	$\left\{\begin{array}{l} \ldots \\ 2\ 0\ \mathit{Enter} \end{array}\right\}$	*var1*
$\Longrightarrow \{\}$	10	E_0	$\#p_1 : [\ldots]$	$\{\ldots\}$	*var3*
$\Longrightarrow \{p_1 \mapsto (10, E_0)\}$	10	E_0	$@(2,0) : [\ldots]$	$\{\ldots\}$	*var3@*
$\Longrightarrow \{\ldots\}$	10	E_0	$\#p_2 : [\ldots]$	$\left\{\begin{array}{l} \ldots \\ 2\ 0\ \mathit{Cons}\ 0\ 10 \end{array}\right\}$	*var3*
$\Longrightarrow \left\{\begin{array}{l} p_1 \mapsto (10, E_0) \\ p_2 \mapsto (10, E_0) \end{array}\right\}$	10	E_0	$@(0,0) : [\ldots]$	$\{\ldots\}$	*var3@*
$\Longrightarrow \{\ldots\}$	10	E_0	$[\#p_3]$	$\left\{\begin{array}{l} \ldots \\ 0\ 0\ \mathit{Cons}\ 0\ 10 \end{array}\right\}$	*var3*
$\Longrightarrow \left\{\begin{array}{l} p_1 \mapsto (10, E_0) \\ p_2 \mapsto (10, E_0) \\ p_3 \mapsto (10, E_0) \end{array}\right\}$	10	E_0	$[\,]$	$\left\{\begin{array}{l} 0\ 0\ \mathit{Observe}\ \mathrm{obs2} \\ 0\ 0\ \mathit{Enter} \\ 0\ 0\ \mathit{Observe}\ \mathrm{obs1} \\ 2\ 0\ \mathit{Enter} \\ 2\ 0\ \mathit{Cons}\ 0\ 10 \\ 0\ 0\ \mathit{Cons}\ 0\ 10 \end{array}\right\}$	

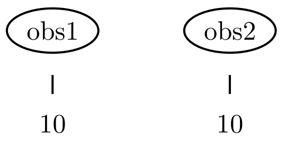
The first configuration in the previous table corresponds with the initial configuration where the heap, environment, stack, and observation file are empty. The name of the rule in that line (*letrec*) corresponds to the rule that is applied and the result of this application is shown in the next row, and so on. As it can be seen, the reduction of this simple expression involves a lot of rules, so that showing the complete reduction is cumbersome. Anyway, it is very interesting to analyze the stack. It can be considered that it has a set of *continuations*. In this case, when the control expression reach a normal form it is needed to analyze the top of the stack to follow with the computation.

Now, adding the line numbers to the observation file (shown in column *side effect*) we get the following file:

$$\left\{\begin{array}{ll} \text{Line} & \text{Observation} \\ 0 & 0\ 0\ \mathit{Observe}\ \mathrm{obs2} \\ 1 & 0\ 0\ \mathit{Enter} \\ 2 & 0\ 0\ \mathit{Observe}\ \mathrm{obs1} \\ 3 & 2\ 0\ \mathit{Enter} \\ 4 & 2\ 0\ \mathit{Cons}\ 0\ 10 \\ 5 & 0\ 0\ \mathit{Cons}\ 0\ 10 \end{array}\right\}$$

This file has more information than Hood shows to us. It is ordered in the same way as the reduction of the closures has been produced. As previously said, this extra information can be used by GHood to produce graphical animations. The first observed closure that has been demanded is the one that refers to the observation mark *obs2*, that is tenOO. The following annotation in the file indicates that we have entered to reduce that closure. Before this closure has reduced to normal form, it has been demanded the evaluation of the closure annotated with *obs1*, that is tenO. The machine has entered to reduce tenO (line 3), and has reached its normal form (line 5). Notice that in order to generate the Hood observations it is not needed the *Enter* annotations.

Now we will analyze how to generate the Hood annotations from this file. First, we will generate a tree with the observations. The way to generate this tree with the observations is considering first the observations with the form *Observe str*. Each *Observe str* will be considered as a root of a different tree annotated with the string *str*. From that point it is only needed to analyze in a sequential way the annotations in the file. In this case, there are only two interesting marks (those starting with *Cons*): the annotation in line 4 refers to the line 2, so its parent is in that line. It is the same for the annotation in line 5, this refers to the line 0, so its parent is there. Therefore, this simple file produces the following two independent observation trees:



The observation that Hood produces is trivially obtained by flattening these observation trees, and the result is the following:

```
—— obs1
        10
—— obs2
        10
```

7 Conclusions

In this paper we have presented a new view of the Hood debugger allowing both to clarify its formal foundations and to implement different variations of it. In particular, we have described how to embed Hood inside Sestoft's natural semantics. Note that the approach we have followed to codify Hood inside the natural semantics can also be used to provide a formal foundation for any other Haskell debugger. In this sense, it could be used as a common framework for describing (and also implementing) all of them.

In order to obtain an implementation of our modified semantics, we have used the abstract machine *Mark-2*$^{@}$. To derive an implementation of it, we have reused the work done in [EP09].

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 [KLPR01, LOP$^+$02]. In fact, in [ERR06] we have already implemented a parallel extension of Hood to deal with Eden programs. Our extension includes a parallelization of the basic Hood library and also a set of tools to allow checking the amount of speculative work that the parallel programs are performing. However we have not provided yet a semantics for our parallel observations. In this sense, we plan to extend the work done in the present paper to deal with parallel semantics. In order to do that, the best choice is to try to embed the debugging method inside the Jauja language [HO03], a very simple parallel functional language whose semantics are clearly defined in terms of an extension of Launchbury's natural semantics, and that has already been used as a common framework to describe three different languages, namely GpH, Eden and pH.

References

[Chi05] Chitil, O.: Source-based trace exploration; In *Implementation of Functional Languages (IFL'04)*, LNCS 3474, pages 126–141. Springer-Verlag, 2005.

[CRW01] Chitil, O., Runciman, C., and Wallace, M.: 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.

[DB72] De Bruijn, N.: Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation.; *Indag. Math.*, 34(5):381–392, 1972.

[DB78] De Bruijn, N.: Lambda Calculus Notation with Namefree Formulas Involving Symbols that Represent Reference Transforming Mapping.; *Indag. Math.*, 40:348–356, 1978.

[DC06] Davie, T. and Chitil, O.: Display of functional values for debugging; In *Draft Proceedings of Implementation of Functional Languages (IFL'06)*, pages 326–337, 2006.

[ELR06] Encina, A., Llana, L., and Rubio, F.: Introducing debugging capabilities to natural semantics; In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI'06)*, LNCS 4378, pages 191–204. Springer-Verlag, 2006.

[EP02] Encina, A. and Peña, R.: Proving the correctness of the STG machine; In *Implementation of Functional Languages (IFL'01)*, LNCS 2312, pages 88–104. Springer-Verlag, 2002.

[EP03a] Encina, A. and Peña, R.: Formally deriving an STG machine; In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 102–112. ACM, 2003.

[EP03b] Ennals, R. and Peyton Jones, S.: HsDebug: Debugging lazy programs by not being lazy; In *Proceedings of the 7th Haskell Workshop*, pages 84–87. ACM, 2003.

[EP03c] Ennals, R. and Peyton Jones, S.: Optimistic evaluation: an adaptive evaluation strategy for non-strict programs; In *International Conference on Functional Programming (ICFP'03)*, pages 287–298. ACM, 2003.

[EP09] Encina, A. and Peña, R.: From natural semantics to C: A formal derivation of two STG machines; *Journal of Functional Programming*, 19(1):47–94, 2009.

[ERR06] Encina, A., Rodríguez, I., and Rubio, F.: Testing speculative work in a lazy/eager parallel functional language; In *Languages and Compilers for Parallel Computing (LCPC'05)*, LNCS 4339, pages 274–288. Springer-Verlag, 2006.

[Gil00] Gill, A.: Hood homepage; `http://www.haskell.org/hood`, 2000.

[Gil01] Gill, A.: Debugging Haskell by observing intermediate data structures; In *Proceedings of the 4th Haskell Workshop*, volume 41(1) of *ENTCS*. Elsevier, 2001.

[Him06] Himmelstrup, D.: Interactive debugging with GHCi; In *Proceedings of the 10th Haskell Workshop (Haskell 2006)*, pages 107–107. ACM, 2006.

[HO03] Hidalgo-Herrero, M. and Ortega-Mallén, Y.: Continuation semantics for parallel Haskell dialects; In *First Asian Symposium on Programming Languages and Systems (APLAS'03)*, LNCS 2895, pages 303–321. Springer-Verlag, 2003.

[JP99] Jones, M. P. and Peterson, J. C.: The Hugs 98 User Manual; `http://www.haskell.org/hugs`, 1999.

[KLPR01] Klusik, U., Loogen, R., Priebe, S., and Rubio, F.: Implementation skeletons in Eden: Low-effort parallel programming; In *Implementation of Functional Languages (IFL'00)*, LNCS 2011, pages 71–88. Springer-Verlag, 2001.

[Lau93] Launchbury, J.: A Natural Semantics for Lazy Evaluation; In *Principles of Programming Languages (POPL'93)*. ACM, 1993.

[LOP$^+$02] Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S., and Rubio, F.: Parallelism Abstractions in Eden; In Rabhi, F. A. and Gorlatch, S., editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pages 95–128. Springer, 2002.

[MIBPG07] Marlow, S., Iborra, J., B. Pope, B., and Gill, A.: A lightweight interactive debugger for Haskell; In *Proceedings of the 11nd Haskell Workshop (Haskell 2007)*, pages 13–24. ACM, 2007.

[MK06] Mürk, O. and Kolmodin, L.: Rectus - locally eager Haskell debugger; Technical report, Göteborg University, 2006.

[Nil98] Nilsson, H.: Declarative debugging for lazy functional languages; PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 1998.

[Nil01] Nilsson, H.: 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.

[Pey03] Peyton Jones, S.: *Haskell 98 language and libraries: The Revised Report* Cambridge University Press, April 2003.

[PHH$^+$93] Peyton Jones, S., Hall, C., Hammond, K., Partain, W., and Wadler, P.: The Glasgow Haskell compiler: A technical overview; In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.

[PN03] Pope, B. and Naish, L.: Practical aspects of declarative debugging in Haskell 98; In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 230–240. ACM, 2003.

[Pop05] Pope, B.: Declarative debugging with Buddha; In *Advanced Functional Programming*, LNCS 3622, pages 273–308. Springer-Verlag, 2005.

[PPRS01] Pareja, C., Peña, R., Rubio, F., and Segura, C.: Adding traces to a lazy functional evaluator; In *Eurocast 2001*, LNCS 2178, pages 627–641. Springer-Verlag, 2001.

[Rei01] Reinke, C.: GHood — graphical visualization and animation of Haskell object observations; In *Proceedings of the 5th Haskell Workshop*, volume 59 of *ENTCS*. Elsevier, 2001.

[Röj95] Röjemo, N.: Highlights from nhc - a space-efficient haskell compiler; In *FPCA*, pages 282–292, 1995.

[Ses97] Sestoft, P.: Deriving a Lazy Abstract Machine; *Journal of Functional Programming*, 7(3):231–264, May 1997.

[Tea97] Team, T. Y.: The York Haskell Compiler - users guide; `http://www.haskell.org/haskellwiki/Yhc`, 1997.

[Wad98] Wadler, P.: Functional programming: Why no one uses functional languages; *SIGPLAN Notices*, 33(8):23–27, August 1998 Functional Programming Column.

[WCBR01] Wallace, M., Chitil, O., Brehm, T., and Runciman, C.: Multipleview tracing for Haskell: a new Hat; In *Proceedings of the 5th Haskell Workshop*, pages 151–170, 2001.

A Proofs of the propositions

In this appendix we will use $q^@$ and $\lambda^@$ to denote, respectively, a pointer and a lambda that could be observed. We will also use the non-standard abbreviation *i.h.* that will stand for *induction hypothesis.*

Proof. Proposition 3

The proof is made by rule induction.

As we have said in the sketch of the proof, we will not consider the observations file, since it does not participate in the evaluation of the expressions. We will also consider that the pointers and the program variables belong to disjoints sets. With this consideration, the freshness of variables can be locally checked, as it is only necessary to create a new pointer that does not belong to the heap.

Proof of the first implication: $\Rightarrow$

Lam **or** *Cons*

Hypothesis:

H1 $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma^@ : e^@$

H2 $\Gamma : e \Downarrow \Delta : w$

H3 By rule *Lam* or *Cons*:

a) $\Delta : w = \Gamma : e$

b) $e = \lambda x.e'$ or $e = C\ \overline{p_i}$

H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^@ : e'^@$

Thesis:

T1 $\Gamma^@ : e^@ \Downarrow \Delta^@ : w^@$

T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^@ : w^@$

T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^@ : e'^@$

Proof:

P1 By H1 and H3(b):
$e^@ = \lambda^@ x.e'^@$ or $e^@ = C\ \overline{q_i}$

P2 By rule *Lam*, *Lam@* or *Cons*:
$\Gamma^@ : e^@ \Downarrow \Gamma^@ : e^@$ (T1)

P3 By H3(a) and H1:
$\Delta : w = \Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma^@ : e^@$ (T2)

P4 By H3(a) and H4:
$\Delta : e' = \Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^@ : e'^@$ (T3)

Letrec

Hypothesis:

H1 $\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \sqsubseteq_{\mathbb{R}} \Gamma^@ : e_0^@$

H2 $\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow \Delta : w$

H3 By rule *Letrec*:

a) $\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \Downarrow \Delta : w$

b) $\overline{p_i}$ fresh in Γ

H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^@ : e'^@$

Thesis:

T1 $\Gamma^@ : e_0^@ \Downarrow \Delta^@ : w^@$

T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^@ : w^@$

T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : e'^{@}$

Proof:

P1 By H1:
$e_0^{@} = \textbf{letrec}\ \overline{x_i = e_i^{@}}\ \textbf{in}\ e^{@}$

P2 By H1, H3(b), H4, P1 and Proposition 2:

a) $\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} \cup [\overline{q_i \mapsto \widehat{e_i^{@}}}] : \widehat{e^{@}}$ if $\overline{q_i}$ are fresh

b) $\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : e' \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} \cup [\overline{q_i \mapsto \widehat{e_i^{@}}}] : e'^{@}$ if $\overline{q_i}$ are fresh

P3 By P2 and i.h. on H3(a):

a) $\Gamma^{@} \cup [\overline{q_i \mapsto \widehat{e_i^{@}}}] : \widehat{e^{@}} \Downarrow \Delta^{@} : w^{@}$
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : w^{@}$ (T2)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : e'^{@}$ (T3)

P4 By P3(a) and rule *Letrec*:
$\Gamma^{@} : \textbf{letrec}\ \overline{x_i = e_i^{@}}\ \textbf{in}\ e^{@} \Downarrow \Delta^{@} : w^{@}$ (T1)

Case

Hypothesis:

H1 $\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e_0^{@}$
H2 $\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \Downarrow \Delta : w$
H3 By rule *Case*:

a) $\Gamma : e \Downarrow \Theta : C_k\ \overline{p_j}$
b) $\Theta : e_k[\overline{p_j / x_{kj}}] \Downarrow \Delta : w$

H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma^{@} : \; e_0^{@} \Downarrow \Delta^{@} : w^{@}$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : w^{@}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : e'^{@}$

Proof:

P1 By H1:
$e_0^{@} = \textbf{case}\ e^{@}\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i^{@}}$

P2 By H1 and P1:
$\Gamma : e \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e^{@}$

P3 By H1, P2, H4 and i.h. on H3(a):

a) $\Gamma^{@} : e^{@} \Downarrow \Theta^{@} : w_0^{@}$
b) $\Theta : C_k\ \overline{p_j} \sqsubseteq_{\mathbb{R}} \; \Theta^{@} : w_0^{@}$
c$_1$) $\Theta : e' \sqsubseteq_{\mathbb{R}} \; \Theta^{@} : e'^{@}$
c$_2$) $\Theta : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}} \sqsubseteq_{\mathbb{R}} \; \Theta^{@} : e_0^{@}$

P4 By P3(b):
$w_0^{@} = C_k\ \overline{q_j}$

P5 By P3(b), P4 and P3(c$_2$):
$\Theta : e_k[\overline{p_j / x_{kj}}] \sqsubseteq_{\mathbb{R}} \; \Theta^{@} : e_k^{@}[\overline{q_j / x_{kj}}]$

P6 By P5, P3(c$_1$) and i.h. on H3(b):

a) $\Theta^{@} : e_k^{@}[\overline{q_j / x_{kj}}] \Downarrow \Delta^{@} : w^{@}$
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : w^{@}$ (T2)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : e'^{@}$ (T3)

P7 By P3(a), P4, P6(a) and rule $Case$:
$\Gamma^@ : \textbf{case } e^@ \textbf{ of } \overline{C_i\ \overline{x_{ij}} \to e_i^@} \Downarrow \Delta^@ : w^@$ (T1)

App

Hypothesis:

H1 $\Gamma : e\ p \sqsubseteq_{\mathbb{R}} \Gamma^@ : e_0^@$
H2 $\Gamma : e\ p \Downarrow \Delta : w$
H3 By rule App:
a) $\Gamma : e \Downarrow \Theta : \lambda x.e_1$
b) $\Theta : e_1[p/x] \Downarrow \Delta : w$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^@ : e'^@$

Thesis:

T1 $\Gamma^@ : e_0^@ \Downarrow \Delta^@ : w^@$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^@ : w^@$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^@ : e'^@$

Proof:

P1 By H1:
$e_0^@ = e^@\ q$
P2 By H1 and P1:
a) $\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma^@ : e^@$
b) $\Gamma : p \sqsubseteq_{\mathbb{R}} \Gamma^@ : q$
P3 By P2, H4 and i.h. on H3(a):
a) $\Gamma^@ : e^@ \Downarrow \Theta^@ : w_0^@$
b) $\Theta : \lambda x.e_1 \sqsubseteq_{\mathbb{R}} \Theta^@ : w_0^@$
c$_1$) $\Theta : e' \sqsubseteq_{\mathbb{R}} \Theta^@ : e'^@$
c$_2$) $\Theta : p \sqsubseteq_{\mathbb{R}} \Theta^@ : q$
P4 By P3(b):
$w_0^@ = \lambda^@ x.e_1^@$
P5 By P3(b), P4 and P3(c$_2$):
$\Theta : e_1[p/x] \sqsubseteq_{\mathbb{R}} \Theta^@ : e_1^@[q/x]$
P6 By P5, P3(c$_1$) and i.h. on H3(b):
a) $\Theta^@ : e_1^@[q/x] \Downarrow \Delta^@ : w^@$
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^@ : w^@$ (T2, case $w_0^@ = \lambda x.e_1^@$)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^@ : e'^@$ (T3, case $w_0^@ = \lambda x.e_1^@$)

Now by cases on P4:
$w_0^@ = \lambda x.e_1^@$
P7 By P3(a), P4, P6(a) and rule App:
$\Gamma^@ : e^@\ q \Downarrow \Delta^@ : w^@$ (T1)

$w_0^@ = \lambda^{@(r,s)} x.e_1^@$
P7 Property 1.5 applied to P5:
$\Theta : e_1[p/x] \sqsubseteq_{\mathbb{R}} \Theta^@ \cup [q'' \mapsto q^{@(l,0)}] : (e_1^@[q/x])[q''/q]$
P8 Property 1.6 applied to P7:
$\Theta : e_1[p/x] \sqsubseteq_{\mathbb{R}} \Theta^@ \cup [q' \mapsto e_1^@[q''/x],\ q'' \mapsto q^{@(l,0)}] : q'^{@(l,1)}$
P9 By P6(c), Proposition 2 and q', q'' fresh:
$\Theta : e' \sqsubseteq_{\mathbb{R}} \Theta^@ \cup [q' \mapsto e_1^@[q''/x],\ q'' \mapsto q^{@(l,0)}] : e'^@$
P10 By P8, P9 and i.h. on H3(b):
a) $\Theta^@ \cup [q' \mapsto e_1^@[q''/x], q'' \mapsto q^{@(l,0)}] : q'^{@(l,1)} \Downarrow \Delta'^@ : w'^@$

b) $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta'^{@} : w'^{@}$ (T2)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta'^{@} : e'^{@}$ (T3)
P11 Applying rule $App@$ to P3(a) and P10(a):
$\Gamma^{@} : e^{@}\, q \Downarrow \Delta'^{@} : w'^{@}$ (T1)

Var'

Hypothesis:
H1 $\Gamma : p \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e_0^{@}$
H2 $\Gamma : p \Downarrow \Delta \cup [p \mapsto w] : w$
H3 By rule Var':
a) $(p \mapsto e) \in \Gamma$
b) $\Gamma : e \Downarrow \Delta : w$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e'^{@}$
Thesis:
T1 $\Gamma^{@} : e_0^{@} \Downarrow \Delta^{@} : w^{@}$
T2 $\Delta \cup [p \mapsto w] : w \sqsubseteq_{\mathbb{R}} \Delta^{@} : w^{@}$
T3 $\Delta \cup [p \mapsto w] : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} : e'^{@}$
Proof: By cases on e:
$e = p'$
P1 By H3(a):
$\Gamma : p' \sqsubseteq \Gamma : p$
P2 Property 1.4 applied to P1 and H1:
$\Gamma : p' \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e_0^{@}$
P3 By P2, H4 and i.h. on H3(b):
a) $\Gamma^{@} : e_0^{@} \Downarrow \Delta^{@} : w^{@}$ (T1)
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} : w^{@}$ (T2)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} : e'^{@}$ (T3)

$e \neq p'$
P1 By H1:
a) $\Gamma^{@} : e_0^{@} = \Gamma_0^{@} \cup [\overline{q_i \mapsto q_{i-1}^{@}}^{n}] : q_n$ with $n \geq 0$
b) $q_0 = e^{@}$
c) $rp\ (e) = \mathbb{R}\ (rp\ (e^{@}))$
P2 By H1 and P1:
$\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e^{@}$
P3 By P2, H4 and i.h. on H3(b):
a) $\Gamma^{@} : e^{@} \Downarrow \Delta_0^{@} : w^{@}$
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta_0^{@} : w^{@}$
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta_0^{@} : e'^{@}$
P4 Property 1.6 applied to P3(a) and P1:
a) $\Gamma^{@} : q_n \Downarrow \Delta^{@} \cup [\overline{q_i \mapsto w^{@i}}^{n}, \overline{q'_i \mapsto q''^{@}_i}] : w^{@0}$ (T1)
b) $\forall i \exists j | w = (\mathbb{R}\ \Delta)^j w^i$
c) $\overline{q'_i}$ are fresh
P5 By H1, H3(a), P1, P3(b) and P4(b):
$\Delta \cup [p \mapsto w] : w \sqsubseteq_{\mathbb{R}} \Delta^{@} \cup [\overline{q_i \mapsto w'^{@i}}^{n}, \ldots] : w'^{@0}$ (T2)
P6 By P4(b,c), P3(c), P1 and H1:
$\Delta \cup [p \mapsto w] : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} \cup [\overline{q_i \mapsto w'^{@i}}^{n}, \ldots] : e'^{@}$ (T3)

Proof of the second implication: $\Leftarrow$

Lam , *Lam@* **or** *Cons*

Hypothesis:

H1 $\Gamma : e \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e^{@}$

H2 $\Gamma^{@} : e^{@} \Downarrow \Delta^{@} : w^{@}$

H3 By rule *Lam*, *Lam@* or *Cons*:

a) $\Delta^{@} : w^{@} = \Gamma^{@} : e^{@}$

b) $e^{@} = \lambda^{@} x.e'^{@}$ or $e = C\ \overline{q_i}$

H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma : e \Downarrow \Delta : w$

T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : w^{@}$

T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : e'^{@}$

Proof:

P1 By H1 and H3(b):

$e = \lambda x.e'$ or $e = C\ \overline{p_i}$

P2 By P1 and rule *Lam* or *Cons*:

$\Gamma : e \Downarrow \Gamma : e$ (T1)

P3 By H1 and H3(a):

$\Gamma : e \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e^{@} = \Delta^{@} : w^{@}$ (T2)

P4 By H4 and H3(a):

$\Gamma : e' \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e'^{@} = \Delta^{@} : e'$ (T3)

Letrec

Hypothesis:

H1 $\Gamma : e_0 \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : \textbf{letrec}\ \overline{x_i = e_i^{@}}\ \textbf{in}\ e^{@}$

H2 $\Gamma^{@} : \textbf{letrec}\ \overline{x_i = e_i^{@}}\ \textbf{in}\ e^{@} \Downarrow \Delta^{@} : w^{@}$

H3 By rule *Letrec*:

a) $\Gamma^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : \hat{e^{@}} \Downarrow \Delta^{@} : w^{@}$

b) $\overline{q_i}$ fresh in $\Gamma^{@}$

H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma :\ e_0 \Downarrow \Delta : w$

T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : w^{@}$

T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : e'^{@}$

Proof:

P1 By H1:

$e_0 = \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e$

P2 By H1, H3(b), H4, P1 and Proposition 2:

a) $\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : \hat{e^{@}}$ if $\overline{p_i}$ fresh

c) $\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : e' \sqsubseteq_{\mathbb{R}} \; \Gamma^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : e'^{@}$ if $\overline{p_i}$ fresh

P3 By P2 and i.h. on H3(a):

a) $\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \Downarrow \Delta : w$

b) $\Delta : w \sqsubseteq_{\mathbb{R}} \; \Delta^{@} : w^{@}$ (T2)

c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$ (T3)

P4 By P3(a) and rule *Letrec*:

$\Gamma :$ **letrec** $\overline{x_i = e_i}$ **in** $e \Downarrow \Delta : w$ (T1)

Case

Hypothesis:

H1 $\Gamma : e_0 \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} :$ **case** $e^{@}$ **of** $\overline{C_i \ \overline{x_{ij}} \to e_i^{@}}$

H2 $\Gamma^{@} :$ **case** $e^{@}$ **of** $\overline{C_i \ \overline{x_{ij}} \to e_i^{@}} \Downarrow \Delta^{@} : w^{@}$

H3 By rule *Case*:

a) $\Gamma^{@} : e^{@} \Downarrow \Theta^{@} : C_k \ \overline{q_j}$

b) $\Theta^{@} : e_k^{@}[\overline{q_j/x_{kj}}] \Downarrow \Delta^{@} : w^{@}$

H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma : \ e_0 \Downarrow \Delta : w$

T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$

T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$

Proof:

P1 By H1:

$e_0 =$ **case** e **of** $\overline{C_i \ \overline{x_{ij}} \to e_i}$

P2 By H1 and P1:

$\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e^{@}$

P3 By H1, P2, H4 and i.h. on H3(a):

a) $\Gamma : e \Downarrow \Theta : w_0$

b) $\Theta : w_0 \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : C_k \ \overline{q_j}$

c$_1$) $\Theta : e' \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : e'^{@}$

c$_2$) $\Theta : e_0 \sqsubseteq_{\mathbb{R}} \ \Theta^{@} :$ **case** $e^{@}$ **of** $\overline{C_i \ \overline{x_{ij}} \to e_i^{@}}$

P4 By P3(b):

$w_0 = C_k \ \overline{p_j}$

P5 By P3(b), P4 and P3(c$_2$):

$\Theta : e_k[\overline{p_j/x_{kj}}] \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : e_k^{@}[\overline{q_j/x_{kj}}]$

P6 By P5, P3(c$_1$) and i.h. on H3(b):

a) $\Theta : e_k[\overline{p_j/x_{kj}}] \Downarrow \Delta : w$

b) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$ (T2)

c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$ (T3)

P7 By P3(a), P4, P6(a) and rule *Case*:

$\Gamma :$ **case** e **of** $\overline{C_i \ \overline{x_{ij}} \to e_i} \Downarrow \Delta : w$ (T1)

App

Hypothesis:

H1 $\Gamma : e_0 \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e^{@} \ q$

H2 $\Gamma^{@} : e^{@} \ q \Downarrow \Delta^{@} : w^{@}$

H3 By rule *App*:

a) $\Gamma^{@} : e^{@} \Downarrow \Theta^{@} : \lambda x.e_1^{@}$

b) $\Theta^{@} : e_1^{@}[q/x] \Downarrow \Delta^{@} : w^{@}$

H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma : \ e_0 \Downarrow \Delta : w$

T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$

Proof:

P1 By H1:
$e_0 = e\ p$

P2 By H1 and P1:
a) $\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e^{@}$
b) $\Gamma : p \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q$

P3 By P2, H4 and i.h. on H3(a):
a) $\Gamma : e \ \Downarrow \ \Theta : w_0$
b) $\Theta : w_0 \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : \lambda x.e_1^{@}$
c$_1$) $\Theta : e' \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : e'^{@}$
c$_2$) $\Theta : p \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : q$

P4 By P3(b):
$w_0 = \lambda x.e_1$

P5 By P3(b), P4 and P3(c$_2$):
$\Theta : e_1[p/x] \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : e_1^{@}[q/x]$

P6 By P5, P3(c$_1$) and i.h. on H3(b):
a) $\Theta : e_1[p/x] \ \Downarrow \ \Delta : w$
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$ (T2)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$ (T3)

P7 By P3(a), P4, P6(a) and rule *Var*:
$\Gamma : \ e\ p \ \Downarrow \ \Delta : w$ (T1)

App@

Hypothesis:

H1 $\Gamma : e_0 \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e^{@}\ q$
H2 $\Gamma^{@} : e^{@}\ q \ \Downarrow \ \Delta^{@} : w^{@}$
H3 By rule *App*@:
a) $\Gamma^{@} : e^{@} \ \Downarrow \ \Theta^{@} : \lambda^{@obs}x.e_1^{@}$
b) $\Theta^{@} \cup [q' \mapsto e_1^{@}[q''/x],\ q'' \mapsto q^{@(l,0)}] : q'^{@(l,1)} \ \Downarrow \ \Delta^{@} : w^{@}$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma : \ e_0 \ \Downarrow \ \Delta : w$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$

Proof:

P1 By H1:
$e_0 = e\ p$

P2 By H1 and P1:
a) $\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e^{@}$
b) $\Gamma : p \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q$

P3 By P2, H4 and i.h. on H3(a):
a) $\Gamma : e \ \Downarrow \ \Theta : w_0$
b) $\Theta : w_0 \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : \lambda^{@obs}x.e_1^{@}$
c$_1$) $\Theta : e' \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : e'^{@}$
c$_2$) $\Theta : p \sqsubseteq_{\mathbb{R}} \ \Theta^{@} : q$

P4 By P3(b):
$w_0 = \lambda x.e_1$
P5 By P3(b), P4 and P3(c$_2$):
$\Theta : e_1[p/x] \sqsubseteq_{\mathbb{R}} \Theta^{@} : e_1^{@}[q/x]$
P6 Property 1.5 applied to P5:
$\Theta : e_1[p/x] \sqsubseteq_{\mathbb{R}} \Theta^{@} \cup [q'' \mapsto q^{@(l,0)}] : (e_1^{@}[q/x])[q''/q]$
P7 Property 1.6 applied to P6:
$\Theta : e_1[p/x] \sqsubseteq_{\mathbb{R}} \Theta^{@} \cup [q' \mapsto e_1^{@}[q''/x],\ q'' \mapsto q^{@(l,0)}] : q'^{@(l,1)}$
P8 By P7 and i.h. on H3(b):
a) $\Theta : e_1[p/x] \Downarrow \Delta : w$
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} : w^{@}$ (T2)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} : e'^{@}$ (T3)
P9 By P3(a), P4, P8(a) and rule *App*:
$\Gamma :\ e_0 \Downarrow \Delta : w$ (T1)

Var'

Hypothesis:
H1 $\Gamma : e_0 \sqsubseteq_{\mathbb{R}} \Gamma^{@} : q$
H2 $\Gamma^{@} : q \Downarrow \Delta^{@} \cup [q \mapsto w^{@}] : w^{@}$
H3 By rule *Var'*:
a) $(q \mapsto e^{@}) \in \Gamma^{@}$
b) $\Gamma^{@} : e^{@} \Downarrow \Delta^{@} : w^{@}$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e'^{@}$

Thesis:
T1 $\Gamma :\ e_0 \Downarrow \Delta : w$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} \cup [q \mapsto w^{@}] : w^{@}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} \cup [q \mapsto w^{@}] : e'^{@}$

Proof:
P1 By H1 and H3(a):
$\Gamma : e_0 = \Gamma[p \mapsto e] : p$
Now we proceed by analyzing cases on $e^{@}$ and e. Notice that it is impossible that $e^{@} \neq q'^{@}$ and $e = p'$, by definition of $\sqsubseteq_{\mathbb{R}}$:

$e^{@} = q'^{@}$ **and** $e \neq p'$
P2 By H1 and P1:
$\Gamma : e_0 \sqsubseteq_{\mathbb{R}} \Gamma^{@} : q'^{@}$
P3 By P2, H4 and i.h. on H3(b):
a) $\Gamma : e_0 \Downarrow \Delta : w$ (T1)
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} : w^{@}$
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} : e'^{@}$
P4 By P3(b,c), Proposition 2, H1 and P1:
a) $\Delta : w \sqsubseteq_{\mathbb{R}} \Delta^{@} \cup [q \mapsto w^{@}] : w^{@}$ (T2)
b) $\Delta : e' \sqsubseteq_{\mathbb{R}} \Delta^{@} \cup [q \mapsto w^{@}] : e'^{@}$ (T3)

($e^{@} \neq q'^{@}$ **and** $e \neq p'$) **or** ($e^{@} = q'^{@}$ **and** $e = p'$)
P2 By P1, H1 and H3(a):
$\Gamma : e \sqsubseteq_{\mathbb{R}} \Gamma^{@} : e^{@}$
P3 By P2, H4 and i.h. on H3(b):
a) $\Gamma : e \Downarrow \Delta : w$

b) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$

P4 By P3(a) and applying rule Var':
$\Gamma[p \mapsto e] : p \ \Downarrow \ \Delta \cup [p \mapsto w] : w$ (T1)

P5 By P3(b), H1 and P1:
a) $\Delta \cup [p \mapsto w] : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} \cup [q \mapsto w^{@}] : w^{@}$ (T2)
b) $\Delta \cup [p \mapsto w] : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} \cup [q \mapsto w^{@}] : e'^{@}$ (T3)

Var@*S*

Hypothesis:
H1 $\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q^{@str}$
H2 $\Gamma^{@} : q^{@str} \ \Downarrow \ \Delta^{@} : w^{@}$
H3 By rule $Var@S$:
$\Gamma^{@} : q^{@(r,s)} \ \Downarrow \ \Delta^{@} : w^{@}$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e'^{@}$

Thesis:
T1 $\Gamma : \ e \ \Downarrow \ \Delta : w$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$

Proof:
P1 By H1:
$\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q^{@(r,s)}$
P2 By P1, H4 and i.h. on H3:
a) $\Gamma : e \ \Downarrow \ \Delta : w$ (T1)
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : w^{@}$ (T2)
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$ (T3)

Var@*C*

Hypothesis:
H1 $\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q^{@(r,s)}$
H2 $\Gamma^{@} : q^{@(r,s)} \ \Downarrow \ \Delta^{@} \cup [\overline{q'_i \mapsto q_i^{@(l,i)}}] : C \ \overline{q'_i}$
H3 By rule $Var@C$:
a) $\Gamma^{@} : q \ \Downarrow \ \Delta^{@} : C \ \overline{q_i}$
b) $\overline{q'_i}$ fresh in $\Delta^{@}$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e'^{@}$

Thesis:
T1 $\Gamma : e \ \Downarrow \ \Delta : w$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} \cup [\overline{q'_i \mapsto q_i^{@(l,i)}}] : C \ \overline{q'_i}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} \cup [\overline{q'_i \mapsto q_i^{@(l,i)}}] : e'^{@}$

Proof:
P1 By H1:
$\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q$
P2 By P1, H4 and i.h. on H3(a):
a) $\Gamma : e \ \Downarrow \ \Delta : w$ (T1)
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : C \ \overline{q_i}$
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$
P3 By P2(b) and H3(b):

a) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} \cup \overline{[q'_i \mapsto q_i^{@(l,i)}]} : C\ \overline{q'_i}$ (T2)
b) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} \cup \overline{[q'_i \mapsto q_i^{@(l,i)}]} : e'^{@}$(T3)

Var@F

Hypothesis:

H1 $\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q^{@(r,s)}$
H2 $\Gamma^{@} : q^{@(r,s)} \ \Downarrow \ \Delta^{@} : \lambda^{@[(r,s)]} x.e^{@}$
H3 By rule *Var@F*:
$\Gamma^{@} : q \ \Downarrow \ \Delta^{@} : \lambda x.e^{@}$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma : e \ \Downarrow \ \Delta : w$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : \lambda^{@[(r,s)]} x.e^{@}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$

Proof:

P1 By H1:
$\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q$
P2 By P1, H4 and i.h. on H3:
a) $\Gamma : e \ \Downarrow \ \Delta : w$ (T1)
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : \lambda x.e^{@}$
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$ (T3)
P3 By P2(b):
$\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : \lambda^{@[(r,s)]} x.e^{@}$ (T2)

Var@FO

Hypothesis:

H1 $\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q^{@(r,s)}$
H2 $\Gamma^{@} : q^{@(r,s)} \ \Downarrow \ \Delta^{@} : \lambda^{@(r,s):obs} x.e^{@}$
H3 By rule *Var@F*:
$\Gamma^{@} : q \ \Downarrow \ \Delta^{@} : \lambda^{@obs} x.e^{@}$
H4 $\Gamma : e' \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : e'^{@}$

Thesis:

T1 $\Gamma : e \ \Downarrow \ \Delta : w$
T2 $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : \lambda^{@(r,s):obs} x.e^{@}$
T3 $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$

Proof:

P1 By H1:
$\Gamma : e \sqsubseteq_{\mathbb{R}} \ \Gamma^{@} : q$
P2 By P1, H4 and i.h. on H3:
a) $\Gamma : e \ \Downarrow \ \Delta : w$ (T1)
b) $\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : \lambda x.e^{@}$
c) $\Delta : e' \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : e'^{@}$ (T3)
P3 By P2(b):
$\Delta : w \sqsubseteq_{\mathbb{R}} \ \Delta^{@} : \lambda^{@(r,s):obs} x.e^{@}$ (T2)

□

Proof. Proposition 6

Proof of the first implication: $\Rightarrow$

The proof is made by semantic rule induction. It is very easy, as it is only necessary to apply induction hypothesis and to compound the results.

Lam, *Lam@* **or** *Cons*

Hypothesis:

H1 $\Gamma : e \looparrowright f \Downarrow \Gamma : e \looparrowright f$

Proof:

$(\Gamma, e, S, f) \Longrightarrow^0 (\Gamma, e, S, f)$

Letrec

Hypothesis:

H1 $\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \looparrowright f \Downarrow \Delta : w \looparrowright f'$

H2 By rule *Letrec*:

a) $\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \looparrowright f \Downarrow \Delta : w \looparrowright f'$

b) $\overline{p_i}$ fresh in Γ

Proof:

$$\begin{array}{lll} & (\Gamma, \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e, S, f) & \{\text{let}\} \\ \Longrightarrow & (\Gamma \cup [\overline{p_i \mapsto \hat{e_i}}], \hat{e}, S, f') & \{\text{i.h. on H2(a)}\} \\ \Longrightarrow^* & (\Delta, w, S, f') & \surd \end{array}$$

Case

Hypothesis:

H1 $\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i} \looparrowright f \Downarrow \Delta : w \looparrowright f''$

H2 By rule *Case*:

a) $\Gamma : e \looparrowright f \Downarrow \Theta : C_k\ \overline{p_j} \looparrowright f'$

b) $\Theta : e_k[\overline{p_j / x_{kj}}] \looparrowright f' \Downarrow \Delta : w \looparrowright f''$

Proof:

$$\begin{array}{lll} & (\Gamma, \textbf{case}\ e\ \textbf{of}\ \overline{alt_i}, S, f) & \{\text{case1}\} \\ \Longrightarrow & (\Gamma, e, \overline{alt_i} : S, f') & \{\text{i.h. on H2(a)}\} \\ \Longrightarrow^* & (\Theta, C_k\ \overline{p_j}, \overline{alt_i} : S, f') & \{\text{case2}\} \\ \Longrightarrow & (\Theta, e_k[\overline{p_j / x_{kj}}], S, f'') & \{\text{i.h. on H2(b)}\} \\ \Longrightarrow^* & (\Delta, w, S, f'') & \surd \end{array}$$

App

Hypothesis:

H1 $\Gamma : e\ p \looparrowright f \Downarrow \Delta : w \looparrowright f''$

H2 By rule *App*:

a) $\Gamma : e \looparrowright f \Downarrow \Theta : \lambda x.e' \looparrowright f'$

b) $\Theta : e'[p/x] \looparrowright f' \Downarrow \Delta : w \looparrowright f''$

Proof:

$$\begin{array}{lll} & (\Gamma, e\ p, S) & \{\text{app1}\} \\ \Longrightarrow & (\Gamma, e, p : S, f) & \{\text{i.h. on H2(a)}\} \\ \Longrightarrow^* & (\Theta, \lambda x.e', p : S, f') & \{\text{app2}\} \\ \Longrightarrow & (\Theta, e'[p/x], S, f') & \{\text{i.h. on H2(b)}\} \\ \Longrightarrow^* & (\Delta, w, S, f'') & \surd \end{array}$$

Var

Hypothesis:

H1 $\Gamma \cup [p \mapsto e] : p \looparrowright f \Downarrow \Delta \cup [p \mapsto w] : w \looparrowright f'$

H2 By rule *Var*:

$\Gamma : e \looparrowright f \Downarrow \Delta : w \looparrowright f'$

Proof:

$$\begin{array}{lll}
 & (\Gamma \cup [p \mapsto e], p, S) & \{\text{var1}\} \\
\Longrightarrow & (\Gamma, e, \#p : S, f) & \{\text{i.h. on H2}\} \\
\Longrightarrow^* & (\Delta, w, \#p : S, f') & \{\text{var2 or var3}\} \\
\Longrightarrow^* & (\Delta \cup [p \mapsto w], w, S, f') & \surd
\end{array}$$

Var@S

Hypothesis:

H1 $\Gamma : q^{@str} \looparrowright f \Downarrow \Delta : w \looparrowright f'$

H2 By rule *Var@S*:

$\Gamma : q^{@(r,s)} \looparrowright f \circ \langle 0\,0\ \mathit{Observe\, str} \rangle \Downarrow \Delta : w \looparrowright f'$

Proof:

$$\begin{array}{lll}
 & (\Gamma,\ q^{@str}, S, f) & \{\text{var@S}\} \\
\Longrightarrow & (\Gamma, q^{@(n,0)}, S, f \circ \langle 0\,0\ \mathit{Observe\, str} \rangle) & \{\text{i.h. on H2}\} \\
\Longrightarrow^* & (\Delta, w, S, f') & \surd
\end{array}$$

Var@C

Hypothesis:

H1$\Gamma :\ p^{@(r,s)} \looparrowright f \Downarrow \Delta \cup [\overline{q_i' \mapsto q_i^{@(length\ f',i)}}] : C\ \overline{q_i'} \looparrowright f' \circ \langle r\ s\ \mathit{Cons}\ k\ C \rangle$

H2 By rule *Var@C*:

$\Gamma : p \looparrowright f \circ \langle r\, s\, \mathit{Enter} \rangle \Downarrow \Delta : C\ \overline{q_i} \looparrowright f'$

Proof:

$$\begin{array}{lll}
 & (\Gamma,\ p^{@(r,s)}, S, f) & \{\text{var1@}\} \\
\Longrightarrow & (\Gamma, p, @(r,s) : S, f \circ \langle r\, s\, \mathit{Enter} \rangle) & \{\text{i.h. on H2}\} \\
\Longrightarrow^* & (\Delta, C\ \overline{q_i}, @(r,s) : S, f') & \{\text{var3@}\} \\
\Longrightarrow^* & (\Delta \cup [\overline{q_i' \mapsto q_i^{@(length\ f',i)}}], C\ \overline{q_i'}, S, f' \circ \langle r\ s\ \mathit{Cons}\ k\ C \rangle) & \surd
\end{array}$$

Var@F

Hypothesis:

H1 $\Gamma : q^{@(r,s)} \looparrowright f \Downarrow \Delta : \lambda^{@[(r,s)]} x.e \looparrowright f'$

H2 By rule *Var@F*:

$\Gamma : q \looparrowright f \circ \langle r\, s\, \mathit{Enter} \rangle \Downarrow \Delta : \lambda x.e \looparrowright f'$

Proof:

$$\begin{array}{lll}
 & (\Gamma,\ q^{@(r,s)}, S, f) & \{\text{var1@}\} \\
\Longrightarrow & (\Gamma, q, @(r,s) : S, f \circ \langle r\, s\, \mathit{Enter} \rangle) & \{\text{i.h. on H2}\} \\
\Longrightarrow^* & (\Delta, \lambda x.e, @(r,s) : S, f') & \{\text{var2@}\} \\
\Longrightarrow^* & (\Delta, \lambda^{@[(r,s)]} x.e, S, f') & \surd
\end{array}$$

Var@FO

Hypothesis:

H1 $\Gamma : q^{@(r,s)} \looparrowright f \Downarrow \Delta : \lambda^{@(r,s):obs} x.e \looparrowright f'$

H2 By rule *Var@FO*:

$\Gamma : q \looparrowright f \circ \langle r\, s\, \mathit{Enter} \rangle \Downarrow \Delta : \lambda^{@obs} x.e \looparrowright f'$

Proof:

$$\begin{array}{lll} & (\Gamma,\ q^{@(r,s)}, S, f) & \{\text{var1@}\} \\ \Longrightarrow & (\Gamma, q, @(r,s) : S, f \circ \langle r\, s\, \mathit{Enter}\rangle) & \{\text{i.h. on H2}\} \\ \Longrightarrow^* & (\Delta, \lambda^{@obs} x.e, @(r,s) : S, f') & \{\text{var2@@}\} \\ \Longrightarrow^* & (\Delta, \lambda^{@(r,s):obs} x.e, S, f') & \surd \end{array}$$

App@

Hypothesis:

H1 $\Gamma : e\ p \hookrightarrow f \Downarrow \Delta : w \hookrightarrow f''$

H2 By rule *App*@:

a) $\Gamma : e \hookrightarrow f \Downarrow \Theta : \lambda^{@(r,s)} x.e' \hookrightarrow f'$

b) $\Theta \cup [q' \mapsto e'[q''/x],\ q'' \mapsto p^{@(length\ f',0)}] : q'^{@(length\ f',1)} \hookrightarrow f' \circ \langle r\, s\, \mathit{Fun}\rangle \Downarrow \Delta : w \hookrightarrow f''$

Proof:

$$\begin{array}{lll} & (\Gamma, e\ p, S, f) & \{\text{app1}\} \\ \Longrightarrow & (\Gamma, e, p : S, f) & \{\text{i.h. on H2(a)}\} \\ \Longrightarrow^* & (\Theta, \lambda^{@(r,s)} x.e', p : S, f') & \{\text{app2@}\} \\ \Longrightarrow & \left(\Theta \cup \begin{bmatrix} q \mapsto e'[q_1/y], \\ q_1 \mapsto p^{@(n',0)}]) \end{bmatrix}, q^{@(n',1)}, S, f' \circ \langle r\, s\, \mathit{Fun}\rangle\right) & \\ & & \{\text{i.h. on H2(b)}\} \\ \Longrightarrow^* & (\Delta, w, S, f'') & \surd \end{array}$$

Proof of the second implication: $\Leftarrow$

Since balanced computations correspond to balanced traces, we can make the proof by cases over the balanced traces.

ϵ

Hypothesis:

H1 $(\Gamma, e, S, f) \Longrightarrow^0 (\Delta, w, S, f')$

Proof:

P1 By H1:

a) $(\Gamma, e, S, f) = (\Delta, w, S, f')$

b) $w = C\ \overline{p_i}$ or $w = \lambda^{@} x.e'$

P2 By P1 and applying rule *Cons*, *App* or *App*@:

$\Gamma : e \hookrightarrow f \Downarrow \Gamma : e \hookrightarrow f\ \surd$

*app*1 *bal app*2 *bal*

Hypothesis:

$$\begin{array}{lll} & (\Gamma, e\ p, S, f) & \{\text{app1}\} \\ \Longrightarrow & (\Gamma, e, p : S, f) & \{\text{bal}_1\} \\ \Longrightarrow^* & (\Theta, \lambda x.e', p : S, f') & \{\text{app2}\} \\ \Longrightarrow & (\Theta, e'[p/x], S, f') & \{\text{bal}_2\} \\ \Longrightarrow^* & (\Delta, w, S, f'') & \end{array}$$

Proof:

P1 By i.h. on bal_1:

$\Gamma : e \hookrightarrow f \Downarrow \Theta : \lambda x.e' \hookrightarrow f'$

P2 By i.h. on bal_2: $\Theta : e'[p/x] \hookrightarrow f' \Downarrow \Delta : w \hookrightarrow f''$

P3 By P1, P2 and rule *App*:

$\Gamma : e \hookrightarrow f \Downarrow \Delta : w \hookrightarrow f''\ \surd$

$var1$ bal $var2$ **or** $var1$ bal $var3$

Hypothesis:

$$\begin{array}{lll} & (\Gamma \cup [p \mapsto e], p, S, f) & \{\text{var1}\} \\ \Longrightarrow & (\Gamma, e, \#p : S, f) & \{\text{bal}\} \\ \Longrightarrow^* & (\Delta, w, \#p : S, f') & \{\text{var2 or var3}\} \\ \Longrightarrow^* & (\Delta \cup [p \mapsto w], w, S, f') & \end{array}$$

Proof:

P1 By i.h. on bal:
$\Gamma : e \looparrowright f \ \Downarrow \ \Delta : w \looparrowright f'$

P2 By P1 and rule *Var*:
$\Gamma \cup [p \mapsto e] : p \looparrowright f \ \Downarrow \ \Delta \cup [p \mapsto w] : w \looparrowright f' \ \surd$

let bal

Hypothesis:

$$\begin{array}{lll} & (\Gamma, \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e, S, f) & \{\text{let}\} \\ \Longrightarrow & (\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}], \hat{e}, S, f) & \{\text{bal}\} \\ \Longrightarrow^* & (\Delta, w, S, f') & \end{array}$$

Proof:

P1 By i.h. on bal:
a) $\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \looparrowright f \ \Downarrow \ \Delta : w \looparrowright f'$
b) $\overline{p_i}$ fresh in Γ

P2 By P1 and rule *Letrec*:
$\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \looparrowright f \ \Downarrow \ \Delta : w \looparrowright f' \ \surd$

$case1$ bal_1 $case2$ bal_2

Hypothesis:

$$\begin{array}{lll} & (\Gamma, \textbf{case}\ e\ \textbf{of}\ \overline{alt_i}, S, f) & \{\text{case1}\} \\ \Longrightarrow & (\Gamma, e, \overline{alt_i} : S, f) & \{\text{bal}_1\} \\ \Longrightarrow^* & (\Theta, C_k\ \overline{p_j}, \overline{alt_i} : S, f') & \{\text{case2}\} \\ \Longrightarrow & (\Theta, e_k[\overline{p_j / x_{kj}}], S, f') & \{\text{bal}_2\} \\ \Longrightarrow^* & (\Delta, w, S, f'') & \end{array}$$

Proof:

P1 By i.h. on bal$_1$:
$\Gamma : e \looparrowright f \ \Downarrow \ \Theta : C_k\ \overline{p_j} \looparrowright f'$

P2 By i.h. on bal$_2$:
$\Theta : e_k[\overline{p_j / x_{kj}}] \looparrowright f' \ \Downarrow \ \Delta : w \looparrowright f''$

P3 By P1, P2 and rule *Case*:
$\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \rightarrow e_i} \looparrowright f \ \Downarrow \ \Delta : w \looparrowright f'' \ \surd$

$app1$ bal $app2@$ bal

Hypothesis:

$$\begin{array}{lll}
 & (\mathit{\Gamma}, e\ p, S, f) & \{\text{app1}\} \\
\Longrightarrow & (\mathit{\Gamma}, e, p : S, f) & \{\text{bal}_1\} \\
\Longrightarrow^* & (\mathit{\Theta}, \lambda^{@(r,s)}x.e', p : S, f') & \{\text{app2@}\} \\
\Longrightarrow & (\mathit{\Theta} \cup \begin{bmatrix} q \mapsto e'[q_1/y], \\ q_1 \mapsto p^{@(n',0)}]) \end{bmatrix}, q^{@(n',1)}, S, f' \circ \langle r\, s\, \mathit{Fun}\rangle) & \{\text{bal}_2\} \\
\Longrightarrow^* & (\mathit{\Delta}, w, S, f'') &
\end{array}$$

Proof:

P1 By i.h. on bal$_1$:
$\mathit{\Gamma} : e \looparrowright f \ \Downarrow\ \mathit{\Theta} : \lambda^{@(r,s)}x.e' \looparrowright f'$

P2 By i.h. on bal$_2$:
$\mathit{\Theta} \cup [q' \mapsto e'[q''/x],\ q'' \mapsto p^{@(n',0)}] : q'^{@(n',1)} \looparrowright f' \circ \langle r\, s\, \mathit{Fun}\rangle \ \Downarrow\ \mathit{\Delta} : w \looparrowright f''$

H1 By P1, P2 and rule $\mathit{App}@$:
$\mathit{\Gamma} : e\ p \looparrowright f \ \Downarrow\ \mathit{\Delta} : w \looparrowright f''\ \surd$

var@*S bal*

Hypothesis:

$$\begin{array}{lll}
 & (\mathit{\Gamma},\ q^{@str}, S, f) & \{\text{var@S}\} \\
\Longrightarrow & (\mathit{\Gamma}, q^{@(n,0)}, S, f \circ \langle 0\, 0\, \mathit{Observe\, str}\rangle) & \{\text{bal}\} \\
\Longrightarrow^* & (\mathit{\Delta}, w, S, f') &
\end{array}$$

Proof:

P1 By i.h. on bal
$\mathit{\Gamma} : q^{@(r,s)} \looparrowright f \circ \langle 0\, 0\, \mathit{Observe\, str}\rangle \ \Downarrow\ \mathit{\Delta} : w \looparrowright f'$

P2 By P1 and rule $\mathit{Var}@S$:
$\mathit{\Gamma} : q^{@str} \looparrowright f \ \Downarrow\ \mathit{\Delta} : w \looparrowright f'\ \surd$

*var*1@ *bal var*2@

Hypothesis:

$$\begin{array}{lll}
 & (\mathit{\Gamma},\ q^{@(r,s)}, S, f) & \{\text{var1@}\} \\
\Longrightarrow & (\mathit{\Gamma}, q, @(r,s) : S, f \circ \langle r\, s\, \mathit{Enter}\rangle) & \{\text{bal}\} \\
\Longrightarrow^* & (\mathit{\Delta}, \lambda x.e, @(r,s) : S, f') & \{\text{var2@}\} \\
\Longrightarrow^* & (\mathit{\Delta}, \lambda^{@[(r,s)]}x.e, S, f') &
\end{array}$$

Proof:

P1 By i.h. on bal:
$\mathit{\Gamma} : q \looparrowright f \circ \langle r\, s\, \mathit{Enter}\rangle \ \Downarrow\ \mathit{\Delta} : \lambda x.e \looparrowright f'$

P2 By P1 and rule $\mathit{Var}@F$:
$\mathit{\Gamma} : q^{@(r,s)} \looparrowright f \ \Downarrow\ \mathit{\Delta} : \lambda^{@[(r,s)]}x.e \looparrowright f'\ \surd$

*var*1@ *bal var*2@@

Hypothesis:

$$\begin{array}{lll}
 & (\mathit{\Gamma},\ q^{@(r,s)}, S, f) & \{\text{var1@}\} \\
\Longrightarrow & (\mathit{\Gamma}, q, @(r,s) : S, f \circ \langle r\, s\, \mathit{Enter}\rangle) & \{\text{bal}\} \\
\Longrightarrow^* & (\mathit{\Delta}, \lambda^{@obs}x.e, @(r,s) : S, f') & \{\text{var2@@}\} \\
\Longrightarrow^* & (\mathit{\Delta}, \lambda^{@(r,s):obs}x.e, S, f') &
\end{array}$$

Proof:

P1 By i.h. on bal:
$\mathit{\Gamma} : q \looparrowright f \circ \langle r\, s\, \mathit{Enter}\rangle \ \Downarrow\ \mathit{\Delta} : \lambda x.e \looparrowright f'$

P2 By P1 and rule $\mathit{Var}@F$:

$$\Gamma : q^{@(r,s)} \looparrowright f \;\Downarrow\; \Delta : \lambda^{@(r,s):obs} x.e \looparrowright f' \;\surd$$

$var1@\ bal\ var3@$

Hypothesis:

$$\begin{array}{lll} & (\Gamma,\ p^{@(r,s)}, S, f) & \{\text{var1@}\} \\ \Longrightarrow & (\Gamma, p, @(r,s) : S, f \circ \langle r\, s\, Enter\rangle) & \{\text{bal}\} \\ \Longrightarrow^* & (\Delta, C\ \overline{q_i}, @(r,s) : S, f') & \{\text{var3@}\} \\ \Longrightarrow^* & (\Delta \cup \overline{[q_i' \mapsto q_i^{@(length\ f',i)}]}, C\ \overline{q_i'}, S, f' \circ \langle r\, s\, Cons\, k\, C\rangle) & \end{array}$$

Proof:

P1 By i.h. on bal:

$\Gamma : p \looparrowright f \circ \langle r\, s\, Enter\rangle \;\Downarrow\; \Delta : C\ \overline{q_i} \looparrowright f'$

P2 By P1 and rule $Var@C$:

$\Gamma : p^{@(r,s)} \looparrowright f \Downarrow \Delta \cup \overline{[q_i' \mapsto q_i^{@(length\ f',i)}]} : C\ \overline{q_i'} \looparrowright f' \circ \langle r\, s\, Cons\, k\, C\rangle$
$\surd$

$\square$