A Proof of Correctness for the STG Machine*

Alberto de la Encina and Ricardo Peña

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

Abstract. J. Launchbury gave an operational semantics for lazy evaluation and showed that it was correct and complete w.r.t. a denotational semantics of the language. P. Sestoft then introduced several abstract machines for lazy evaluation and showed that they were correct and complete w.r.t. Launchbury's operational semantics. We go a step forward and show that the *Spineless Tagless G-machine* is complete and (almost) correct w.r.t. one of Sestoft's machines. In the way to this goal we also prove some interesting properties about the operational semantics and about Sestoft's machines which clarify some minor points on garbage collection and on closures' local environments. Unboxed values and primitive operators are excluded from the study.

1 Introduction

One of the most successful abstract machines for executing lazy functional languages is the *Spineless Tagless G-machine* (STG machine) [5] which is at the heart of the *Glasgow Haskell Compiler* (GHC) [6]. The compiler receives a program written in Haskell [7] and, after some steps and intermediate transformations, produces a program in a very simple functional language called the STG language. This is the input for the STG machine. The back-end then generates imperative code emulating the transitions of the machine.

The STG machine has proved to be efficient compared with some other machines for lazy languages such as the G-machine [2] or the TIM (*Three Instructions Machine*) [1]. But until recently there has been no formal proof of its correctness. This was provided for the first time by J. Mountjoy in [4]. There, the author starts from Launchbury's natural semantics for lazy evaluation [3] and transforms it to successive more elaborated semantics. From these semantics he derives a STG-like machine and proves its correctness w.r.t. the semantics. The machine has a single stack (instead of three) and does not treat constructions as normal forms. Additionally, he proves that the more elaborated semantics are in fact equivalent to Launchbury's.

Launchbury's semantics is a good starting point because it has been accepted *de facto* as *the* reference for defining the meaning of lazy evaluation. It is however debatable whether the right approach to proving the correctness of a machine is to refine the specification so that it gets closer and closer to the desired operational behaviour.

* Work partially supported by the Spanish-British Acción Integrada HB 1999-0102 and Spanish project TIC 2000-0738.

Here we have followed a different strategy. We accept Launchbury's semantics as *the* specification and we continue with the abstract machines developed by Sestoft in [9] which were shown to be correct and complete w.r.t. Launchbury's semantics. The idea is to continue refining one of these machines in order to arrive to the STG. In order to have a common language, we define a language similar to the STG language so that it can be considered a subset of the language used by Sestoft's machines. Then, we define and prove a *bisimulation* between the Sestoft machine called *Mark-2* and the STG.

The bisimulation holds for a single-stack STG machine. The one described in [5] and implemented in the first versions of the GHC compiler had three separate stacks. The recent version of GHC has moved towards a single-stack implementation [8]. Nevertheless, it does not exist yet an operational description for this machine in the sense of the one given in [5]. We show that the three stack machine is *not* correct w.r.t. to the semantics for some ill-typed programs.

The plan of the paper is as follows: After this introduction, in Section 2 Launchbury's semantics is summarized. Sestoft's improvements to this semantics are also presented and further improvements are proposed to solve one small problem in Sestoft's work and to take into account garbage collection. In Section 3, Sestoft's *Mark-2* machine is described and also some theorems about its environments are given. Section 4 defines the STG-like language, the single-stack STG machine and proves the bisimulation with the *Mark-2* machine. Section 5 shows that the three-stack STG machine is complete but not correct w.r.t. the semantics. Finally, Section 6 draws some conclusions.[1]

2 Natural Semantics

2.1 Launchbury's original proposal

A well-known work from Launchbury [3] 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 language used was a normalized λ-calculus, extended with recursive *let*, (saturated) constructor applications and *case* expressions. To ensure sharing, arguments of applications are forced to be variables. A grammar for this language is given in Figure 1 where the overline notation $\overline{A_i}$ denotes a vector $A_1, \ldots, A_n$ of subscripted entities.

To avoid variable capture, the normalized language has the additional restriction that all bound variables (either lambda, let or case bound) in the initial expression must be distinct. (Weak head) normal forms are either lambda abstractions or constructions. Throughout the paper, the symbol w will be used to denote normal forms. The semantic rules are reproduced in Figure 2. There, a judgement $\Gamma : e \Downarrow \Theta : w$ means that expression e, with free variables bound in heap Γ, reduces to normal form w and produces a final heap Θ. The notation $\hat{w}$ means expression w where all bound variables have been replaced by fresh names. We say that $\Gamma : e \Downarrow \Theta : w$ is a (successful) derivation if it can be proved

[1] Propositions proofs can be found at `http://dalila.sip.ucm.es/~albertoe`.

$$
\begin{array}{lll}
e \to x & & \text{-- variable} \\
\quad \mid \ \lambda x.e & & \text{-- lambda abstraction} \\
\quad \mid \ e\ x & & \text{-- application} \\
\quad \mid \ \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e & & \text{-- recursive let} \\
\quad \mid \ C\ \overline{x_i} & & \text{-- constructor application} \\
\quad \mid \ \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} & & \text{-- case expression}
\end{array}
$$

Fig. 1. Launchbury's normalized λ-calculus

$$
\begin{array}{cl}
\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e & \quad \mathit{Lam} \\[1ex]
\Gamma : C\ \overline{x_i} \Downarrow \Gamma : C\ \overline{x_i} & \quad \mathit{Cons} \\[1ex]
\dfrac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : w}{\Gamma : e\ x \Downarrow \Theta : w} & \quad \mathit{App} \\[2ex]
\dfrac{\Gamma : e \Downarrow \Delta : w}{\Gamma \cup [x \mapsto e] : x \Downarrow \Delta \cup [x \mapsto w] : \hat{w}} & \quad \mathit{Var} \\[2ex]
\dfrac{\Gamma \cup [\overline{x_i \mapsto e_i}] : e \Downarrow \Delta : w}{\Gamma : \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e \Downarrow \Delta : w} & \quad \mathit{Letrec} \\[2ex]
\dfrac{\Gamma : e \Downarrow \Delta : C_k\ \overline{x_j} \quad \Delta : e_k[\overline{x_j / y_{kj}}] \Downarrow \Theta : w}{\Gamma : \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{y_{ij}} \to e_i} \Downarrow \Theta : w} & \quad \mathit{Case}
\end{array}
$$

Fig. 2. Launchbury's natural semantics

by using the rules. A derivation can fail to be proved for instance because of entering in a *blackhole*. This would happen in rule *Var* when a reference to variable x appears while reducing expression e and before reaching a normal form. As this is done in a heap Γ not containing a binding for x, no rule can be applied and the derivation cannot be completed. Other forms of failure are those corresponding to ill-typed programs or infinite loops.
The main theorem in [3] is that the operational semantics is correct and complete with respect to a non-strict denotational semantics for the language, i.e. if e is a closed expression, then $[\![e]\!]\ \rho_0 = v \neq \bot$ if and only if there exist Θ and w such that $\{\} : e \Downarrow \Theta : w$ and $[\![w]\!]\ \rho_\Theta = v$, being ρ_0 the empty environment and ρ_Θ an environment defining the free variables of w and obtained from heap Θ.

2.2 Sestoft's improvements

Sestoft introduces in [9] two main changes to the operational semantics of Figure 2: (1) to move the renaming of variables from the *Var* rule to the *Letrec* one, and (2) to make in the *Letrec* rule the freshness condition locally checkable by extending judgements with a set A of *variables under evaluation*. The first modification aims at getting the semantics closer to an implementation in terms of abstract machines. In the usual implementations, fresh variables (i.e. pointers) are created when introducing new closures in the heap in the *Letrec* rule. This is also more economical than renaming all bound variables in a normal form. The second modification makes more precise the definition of *freshness*: a variable is fresh in a judgement $\Gamma\ : e \Downarrow_A \Theta : w$ if it does not belong to either $dom\ \Gamma$ or A, and it is not bound either in $range\ \Gamma$ or e. The modified rules can be seen in Figure 3. In the *Letrec* rule, the notation $\hat{e}$ means the renaming $e[\overline{p_i/x_i}]$ where $\overline{p_i}$ are fresh variables in the judgement $\Gamma : \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e \Downarrow_A \Delta : w$.

The difference between the new rules and the ones in Figure 2 is the place where renaming is done. So, the only thing needed to prove the equivalence between the two sets of rules is that there is neither name capture nor duplicated bindings to the same variable.

Proposition 1 *(Sestoft) Let e be a closed expression and $\{\} : e \Downarrow_{\{\}} \Theta : w$ a successful derivation. Then, in no instance of rule App there can be variable capture in $e'[p/x]$, and in no instance of rule Var is p already bound in Δ.*

$$\Gamma : \lambda x.e \Downarrow_A \Gamma : \lambda x.e \qquad \mathit{Lam}$$

$$\Gamma : C\,\overline{p_i} \Downarrow_A \Gamma : C\,\overline{p_i} \qquad \mathit{Cons}$$

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

$$\frac{\Gamma : e \Downarrow_{A\cup\{p\}} \Delta : w}{\Gamma \cup [p \mapsto e] : p \Downarrow_A \Delta \cup [p \mapsto w] : w} \qquad \mathit{Var}$$

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

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

Fig. 3. Sestoft's natural semantics

Moreover, Sestoft proves that, in every derivation tree, there is a clean distinction between free variables and bound variables in expressions appearing in judgements and in heaps. The first ones are always pointers (in Figure 3 and in what follows, they are denoted by p), and they are either bound in the corresponding heap, or they are under evaluation and belongs to A. The second ones are program variables belonging to the original expression (in Figure 3 and in what follows, they are denoted by x or y).

Unfortunately, the proof of the theorem was done before introducing **case** expressions and constructors and, when the latter were introduced, the theorem was not redone. With the current *Case* rule the freshness property is not locally checkable anymore. While reducing the discriminant in judgement $\Gamma : e \Downarrow_A \Delta : C_k\ \overline{p_j}$, fresh variables may be created with the same names as bound variables in the alternatives, without violating the freshness condition. Then, in the second part of the premise, name capture may occur in expression $e_k[\overline{p_j/x_{kj}}]$.

In the next section we introduce a modification to the rules in order to keep the freshness locally checkable in presence of **case** expressions.

A problem not addressed by Sestoft is garbage collection. One invariant of the derivation of any expression is that heaps always grow with new bindings, i.e. in every judgement $\Gamma : e \Downarrow_A \Delta : w$, it turns out that $\mathit{dom}\ \Gamma \subseteq \mathit{dom}\ \Delta$. We are interested in having a semantics reflecting that garbage collection may happen at any time without altering the result of the evaluation. To this aim, we develop a set of rules in which all heaps are assumed to contain only live bindings. So, the garbage collector must be explicitly activated in certain rules. This forces us to maintain new sets during a derivation in order to keep all the roots of live bindings. This extension is also done in the next section.

2.3 A modified natural semantics

To solve the first problem, i.e. having freshness locally checkable, we introduce a multiset C of *continuations* associated to every judgement. The alternatives of a **case** are stored in this multiset during the evaluation of the discriminant. We say then that a variable is fresh in a judgement $\Gamma : e \Downarrow_{AC} \Delta : w$ if it does not belong either to *dom* Γ or to A, and it is not a bound variable either in e, or in *range* Γ. or in any continuation of C.

To provide for garbage collection, we must first decide which are the roots of live closures. Of course, free variables of the current expression belong to the set of roots. By observing the rules, it is clear that the continuations in set C should also provide additional roots. Otherwise, a minimal heap during the derivation of the discriminant might not include bindings for the free variables of the alternatives. Symmetrically, during the derivation of the normal form of function e in rule *App*, we should include the argument p of an application in the set of roots. So, we introduce an additional multiset B in judgements standing for *arguments of pending applications.* A judgement will have the following form: $\Gamma : e \Downarrow_{ABC} \Delta : w$ where the intention is that Γ be minimal w.r.t. e, B and C, and Δ be minimal w.r.t. w, B and C.

As the knowledgeable reader may have already noticed, set A is *not* an additional source of roots. This set represents bindings currently under evaluation or, in more operational terms, *pending updates.* If the only reference to a pending update is that of set A, this means that the value of the corresponding free variable will not be used anymore in the future. So, the variable can be safely deleted from A, and the corresponding update avoided[2]. Moreover, we want to have also *minimal* sets of pending updates in our derivations. This means that the set A associated to the initial expression of a given judgement needs not be the same anymore than the set A' associated to the final expression. To take this into account, a last modification of judgements is needed. Their final form is the following one:

$$\Gamma A : e \Downarrow_{BC} \Delta A' : w$$

where Γ and A are minimal w.r.t. e, B and C, and Δ and A' are minimal w.r.t. w, B and C. Its meaning is that expression e reduces to normal form w starting with heap Γ and set A, and ending with heap Δ and set A'.

That heaps and sets of pending updates are minimal is just a property that must be proved. To preserve this property in derivations, garbage collections and trimming of pending updates must be activated at certain points. The semantic rules exactly clarify which these points are.

Definition 2 *Given a heap Γ, an expression e, a multiset B of variables, and a multiset C of continuations, we define the set of* live variables *of Γ w.r.t. B, C and e, denoted $live_\Gamma^{BCe}$:*

$$\begin{aligned} live_\Gamma^{BCe} \stackrel{def}{=} fix\ (\lambda L\ .\ & L \cup fv\ e \cup B \cup fv\ C \cup \\ & \bigcup\{fv\ e' \mid p \in L \wedge [p \mapsto e'] \in \Gamma\}) \end{aligned}$$

where $fv\ e$ denotes the set of free variables of expression e, $fv\ C$ is the obvious extension to a continuation and *fix* denotes the least fixed-point.

Definition 3 *Given a heap Γ, a set of pending updates A, an expression e, a multiset B of variables, and a multiset C of continuations, we define the live*

[2] This *trimming* of the set of pending updates is done in the STG machine after each garbage collection. See [5, Section 10.7].

heap of Γ w.r.t. B, C and e, denoted Γ_{gc}^{BCe}, and the subset of live updates of A w.r.t. Γ, B, C and e, denoted $A_{gc}^{\Gamma BCe}$:

$$\begin{array}{l} \Gamma_{gc}^{BCe} \stackrel{def}{=} \{p \mapsto e \mid (p \mapsto e) \in \Gamma \wedge p \in live_{\Gamma}^{BCe}\} \\ A_{gc}^{\Gamma BCe} \stackrel{def}{=} A \cap live_{\Gamma}^{BCe} \end{array}$$

In a judgement $\Gamma A : e \Downarrow_{BC} \Delta A' : w$, if a minimal heap and update set should be ensured before the derivation starts, we will write $\Gamma_{gc} A_{gc} : e \Downarrow_{BC} \Delta A' : w$, meaning that the initial heap and update set should respectively be Γ_{gc}^{BCe} and $A_{gc}^{\Gamma BCe}$. These *gc* annotations exactly mark the points in a derivation where a garbage collection or/and a trimming of the update set may be needed.

The new set of rules is shown in Figure 4. Some explanations are follow:

Maintaining the correct set of roots When evaluating the discriminant of a **case** (see rule *Case*), the pending alternatives must be included in set C in order to avoid losing bindings for the free variables of the alternatives. Also, when evaluating the function of an application (see rules *AppA* and *AppB*), the argument must be included in set B in order to avoid losing the binding for it.

Activating garbage collection in the appropriate points The *gc* annotation, meaning the trimming of a heap or of an update set, must be written in those points where there may be the danger of dead bindings. These are:

- in rule *AppB*, when the parameter of the function does not appear in the body. There is a danger that the binding for p in Δ becomes dead.
- in rules *VarA* and *VarB*, when the reference to p disappear from the current expression. There may be no other reference to p either in e, B or C.
- in rule *Case*, when a particular alternative is chosen. The discarded alternatives may have free variables that now are dead.

Avoiding unnecessary updates This is reflected in rule *VarB*. Assuming that the pair (Δ, A') is minimal w.r.t. w, B and C, and knowing that $p \notin A'$, then the update for variable p may be safely discarded (compare with rule *VarA*).

Assuming no dead code in *letrec* Notice in the antecedent of rule *Letrec* that no garbage collection is launched. So, we are assuming that all the new bindings are live. This is not true if there exists dead code in the **letrec** expression. It is easy for a compiler to eliminate unreachable bindings in a **letrec**. In what follows we will assume that dead code has been eliminated in our programs.

New definition of freshness In the consequent of rule *Letrec* a set $\overline{p_i}$ of fresh variables is created. A variable is *fresh* in judgement $\Gamma A : e \Downarrow_{BC} \Delta A' : w$ if it does not belong to either *dom* Γ or A, and it is not bound either in *range* Γ, e or C.

We will see now that the properties desired for our semantics in fact hold.

Definition 4 *Given a judgement $\Gamma A : e \Downarrow_{BC} \Delta A' : w$, we say that the configuration $\Gamma : e$ is ABC-*good*, if*

1. $A \cap dom\ \Gamma = \emptyset$

$$\mathit{\Gamma} A : \lambda x.e \Downarrow_{BC} \mathit{\Gamma} A : \lambda x.e \qquad \mathit{Lam}$$

$$\mathit{\Gamma} A : C' \, \overline{p_i} \Downarrow_{BC} \mathit{\Gamma} A : C' \, \overline{p_i} \qquad \mathit{Cons}$$

$$\frac{\mathit{\Gamma} A : e \Downarrow_{(B \cup \{p\})C} \mathit{\Delta} A' : \lambda x.e' \qquad x \in \mathit{fv}\ e' \qquad \mathit{\Delta} A' : e'[p/x] \Downarrow_{BC} \mathit{\Theta} A'' : w}{\mathit{\Gamma} A : e\ p \Downarrow_{BC} \mathit{\Theta} A'' : w} \qquad \mathit{AppA}$$

$$\frac{\mathit{\Gamma} A : e \Downarrow_{(B \cup \{p\})C} \mathit{\Delta} A' : \lambda x.e' \qquad x \notin \mathit{fv}\ e' \qquad \mathit{\Delta}_{gc} A'_{gc} : e' \Downarrow_{BC} \mathit{\Theta} A'' : w}{\mathit{\Gamma} A : e\ p \Downarrow_{BC} \mathit{\Theta} A'' : w} \qquad \mathit{AppB}$$

$$\frac{\mathit{\Gamma}\ (A \cup \{p\})_{gc} : e \Downarrow_{BC} \mathit{\Delta}\ (A' \cup \{p\}) : w}{\mathit{\Gamma} \cup [p \mapsto e] A : p \Downarrow_{BC} \mathit{\Delta} \cup [p \mapsto w]\ A' : w} \qquad \mathit{VarA}$$

$$\frac{\mathit{\Gamma}\ (A \cup \{p\})_{gc} : e \Downarrow_{BC} \mathit{\Delta} A' : w \qquad p \notin A'}{\mathit{\Gamma} \cup [p \mapsto e] A : p \Downarrow_{BC} \mathit{\Delta} A' : w} \qquad \mathit{VarB}$$

$$\frac{\mathit{\Gamma} \cup [\overline{p_i \mapsto \hat{e}_i}]\ A : \hat{e} \Downarrow_{BC} \mathit{\Delta} A' : w}{\mathit{\Gamma} A : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow_{BC} \mathit{\Delta} A' : w} \text{ where } \overline{p_i} \text{ fresh} \qquad \mathit{Letrec}$$

$$\frac{\mathit{\Gamma} A : e \Downarrow_{B(C \cup \{alts\})} \mathit{\Delta} A' : C_k\ \overline{p_j} \qquad \mathit{\Delta}_{gc} A'_{gc} : e_k[\overline{p_j / x_{kj}}] \Downarrow_{BC} \mathit{\Theta} A'' : w}{\mathit{\Gamma} A : \textbf{case}\ e\ \textbf{of}\ alts \Downarrow_{BC} \mathit{\Theta} A'' : w} \qquad \mathit{Case}$$

Fig. 4. A natural semantics with minimal heaps and minimal update sets

2. $live_{\mathit{\Gamma}}^{BCe} = A \cup dom\ \mathit{\Gamma}$
3. $(bv\ \mathit{\Gamma} \cup bv\ e \cup bv\ C) \cap (A \cup dom\ \mathit{\Gamma}) = \emptyset$

where $bv\ e$ denotes the bound variables of expression e, $bv\ \mathit{\Gamma}$ its extension to all expressions in *range* $\mathit{\Gamma}$, and $bv\ C$ its extension to a continuation.

The first property has a similar counterpart in Sestoft's semantics and it asserts that variables under evaluation are not at the same time defined in the heap. The second one asserts that every free variable is either defined in the heap or is under evaluation and also that the pair $(\mathit{\Gamma}, A)$ is minimal w.r.t. B, C and e. The third one asserts that free variables are different from bound ones.

Definition 5 *A judgement* $\mathit{\Gamma} A : e \Downarrow_{BC} \mathit{\Delta} A' : w$ *is* promising *if the configuration* $\mathit{\Gamma} : e$ *is* ABC-good.

This definition ensures that the starting point of a derivation already meets the requirements we want for the whole derivation. The following proposition and corollary establish that the desired properties in fact hold.

Proposition 6 *Let* $\mathit{\Gamma} A : e \Downarrow_{BC} \mathit{\Delta} A' : w$ *be a derivation using the rules of the semantics. If it is a promising judgement, then*

1. *The configuration* $\mathit{\Delta} : w$ *is* $A'BC$*-good*
2. $A' \subseteq A$
3. *Every judgement in the derivation is a promising one.*

Corollary 7 *Let* e *be a closed expression and* $\mathit{\Gamma}\{\} : e \Downarrow_{\{\}\{\}} \mathit{\Delta} A : w$ *be a derivation. Then,*

1. *In no instance of rules AppA and Case there can be variable capture in substitutions of the form* $e[p/x]$.
2. *In no instance of rule VarA is* p *already bound in* $\mathit{\Delta}$.

The differences between our semantics and Sestoft's are two:

1. Sestoft's rules *App* and *Var* have been split into two in our semantics. In the first case, the distinction is due to the desire of not launching garbage collection when it is not needed, but in fact both rules could be combined in the following single one:

$$\frac{\Gamma A : e \Downarrow_{(B \cup \{p\})C} \Delta A' : \lambda x.e' \quad \Delta_{gc} A'_{gc} : e'[p/x] \Downarrow_{BC} \Theta A'' : w}{\Gamma A : e\ p \Downarrow_{BC} \Theta A'' : w}$$

   In the second case, our rule *VarB* does not add to the heap a binding $[p \mapsto w]$ that is known to be dead.
2. Our heaps and update sets are minimal in the corresponding judgements.

Otherwise, the semantic rules are the same. Once we have proved that free variables in judgements are either defined in the heap, or they belong to the pending updates set, both semantics produce exactly the same derivations.

3 Sestoft's machine Mark-2

After revising Launchbury's semantics, Sestoft introduces in [9] several abstract machines in sequence, respectively called *Mark-1*, *Mark-2* and *Mark-3*. The one we will use for deriving an STG-like machine is *Mark-2*. There, a configuration consists of a heap Γ, a control expression e possibly having free variables, an environment E mapping free variables to pointers in the heap, and a stack S. The heap Γ is a function from pointers to closures, each one (e, E) consisting of an expression e and an environment E binding its free variables to pointers. The stack contains three kinds of objects: (1) arguments of pending applications, represented by pointers; (2) continuations of pending pattern matchings, each one consisting of a pair $(alts, E)$ where $alts$ is a vector of **case** alternatives and E is an environment binding its free variables; and (3) update markers of the form $\#p$, where p is a pointer.

The reader may have already recognized that stack S represents in fact the union of sets B, C and A we introduced in the revised semantics of Section 2.3. The main difference now is that these entities form a list instead of a set or a multiset, and that they appear ordered from more recent to older ones. In Figure 5 the operational rules of *Mark-2* machine are shown.

We have followed Sestoft's convention that program variables are denoted by x or y, and pointers by p. The machine never makes explicit substitutions of pointers for program (free) variables as the semantics does. Instead, it maintains environments mapping program variables to pointers. If e is a closed expression, the initial configuration is $(\{\}, e, \{\}, [\,])$. The machine stops when no rule can be applied. If the final configuration has the form $(\Gamma, w, E, [\,])$, then w has been successfully derived from e and we write $(\{\}, e, \{\}, [\,]) \Rightarrow^* (\Gamma, w, E, [\,])$.

The main theorem proved by Sestoft —in fact, he formally did it for machine *Mark-1*, and afterwards he reasoned that machine *Mark-2* does exactly the same reduction steps as machine *Mark-1*—, is that successful derivations of the machine are exactly the same as those of the semantics.

Proposition 8 *(Sestoft) For any closed expression e, then*

$$(\{\}, e, \{\}, [\,]) \Rightarrow^* (\Gamma, w, E, [\,]) \text{ if and only if } \{\}\ e :\Downarrow_{\{\}} \Gamma : w$$

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

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

Fig. 5. Abstract machine *Mark-2*

It is worth to note that the correctness and completeness of machine *Mark-2* w.r.t. the operational semantics (by transitivity, w.r.t. the denotational semantics), does not rely on programs being well typed. If a program is ill-typed both the machine and the semantics will be unable to derive a normal form for it.

3.1 Some properties of environments

Mark-2 environments have a complex evolution: they grow with lambda applications, pattern matching and **letrec** execution; they are stored either in closures or in the stack in some transitions, and they are retrieved from there in some other transitions. It is natural to wonder about how much can they grow.

Definition 9 *A closure* (e, E)*, is* consistent *if*

1. $fv\ e \cap bv\ e = \emptyset$ *and all variables in* $bv\ e$ *are distinct.*
2. $fv\ e \subseteq dom\ E$.
3. $bv\ e \cap dom\ E = \emptyset$

This definition can be easily extended to a continuation of the form $(alts, E)$ and to a heap Γ consisting of a set of closures.

Definition 10 *A configuration* (Γ, e, E, S) *of machine Mark-2 is* consistent *if*

1. Γ *is consistent.*
2. *The pair* (e, E) *is consistent.*
3. *All continuations* $(alts, E) \in S$ *are consistent.*

These definitions only take care of program variables being well defined in environments. That pointers are well defined in the heap (or they belong to stack S as update markers) was already proved by Sestoft for all his machines.

$$
\begin{array}{lll}
e & \to x\ \overline{x_i}^n & \text{-- } n \geq 0, \text{ application/variable} \\
 & \mid\ \textbf{letrec}\ \overline{bind_i}\ \textbf{in}\ e & \text{-- recursive let} \\
 & \mid\ C\ \overline{x_i} & \text{-- constructor application} \\
 & \mid\ \textbf{case}\ e\ \textbf{of}\ \overline{alt_i}\ |^t & \text{-- case expression} \\
bind & \to x = \mathit{lf}\ |^t & \\
\mathit{lf} & \to \lambda\ \overline{x_i}^n.e & \text{-- } n \geq 0, \text{ lambda form} \\
alt & \to C\ \overline{x_j} \to e &
\end{array}
$$

Fig. 6. Definition of FUN

Proposition 11 *Let e be a closed expression in which all bound variables are distinct, and $(\{\}, e, \{\}, [\,]) \Rightarrow^* (\Gamma, e', E, S)$ any (possibly partial) derivation of machine Mark-2. Then,*

1. *(Γ, e', E, S) is consistent.*
2. *E exactly binds all variables in scope in expression e'.*
3. *In any closure $(e_i, E_i) \in \Gamma$, E_i exactly binds all variables in scope in expression e_i.*
4. *In any pair $(alts, E) \in S$, E exactly binds all variables in scope in alternatives alts.*

4 The abstract machine STG-1S

4.1 The common language

In order to get closer to the STG machine, firstly we define a common λ-calculus for both machines, *Mark-2* and STG. This is presented in Figure 6 and we call it FUN. It is equivalent to the STG language (STGL) but expressed in a more traditional λ-calculus syntax. The differences with STGL are:

- In FUN there are no unboxed values, primitive operators or primitive **case** expressions. These have been excluded from our study.
- In FUN the default alternative in **case** is missing. This could be added without effort.
- In STGL there is a flag $\backslash\pi$ in lambda forms to indicate that some updates can be safely avoided. This is an efficiency issue. Nevertheless, we modify the *Mark-2* to suppress updates in the obvious cases of bindings to normal forms.
- In STGL there is a non-recursive **let**. This can obviously be simulated by FUN's **letrec**.
- In STGL a program is a list of bindings with a distinguished variable *main* where evaluation starts. This can be simulated by **letrec** $\overline{bind_i}$ **in** *main*.

Compared to the original *Mark-2* λ-calculus (see Figure 1), it is clear that FUN is just a subset of it, having the restrictions: that lambda abstractions may only appear in bindings and that applications have the form $x\ \overline{x_i}^n$ understood by *Mark-2* as $(\ldots(x\ x_1)\ldots)\ x_n$.

The notation $\lambda\ \overline{x_i}^n.e$ is an abbreviation of $\lambda x_n.\cdots.\lambda x_1.e$, where the arguments have been numbered downwards for convenience. In this way, it is easy to accept that $\lambda\ \overline{x_i}^{n-1}.e$ means $\lambda x_{n-1}.\cdots.\lambda x_1.e$. Accordingly, $x\ \overline{x_i}^n$ means $x\ x_1\ \ldots\ x_n$ so that $x\ \overline{x_i}^{n-1}$ will mean $x\ x_1\ \ldots\ x_{n-1}$. When $n = 0$ we will simply write e instead of $\lambda\ \overline{x_i}^n.e$ and x instead of $x\ \overline{x_i}^n$.

A last feature added to FUN is *trimmers*. The notation $\mathit{lf}\ |^t$ means that a lambda form is annotated at compile time with the set t of its free variables. This set t was called a trimmer in [9]. It will be used when constructing a closure in the heap for the lambda form. The environment stored in the closure will only bind the variables contained in the trimmer. This implies a small penalty in terms of execution time but a lot of space saving. Analogously, $\overline{alt_i}\ |^t$ means the annotation of a set of **case** alternatives with the trimmer t of its free variables. When $\overline{alt_i}$ is pushed into the stack, its associated environment will be trimmed according to t. Both optimizations are done in the STG machine, even though the second one is not reflected in the rules given in [5].

4.2 Mark-2 machine for language FUN

In Figure 7, the transition rules of *Mark-2* for language FUN are shown. Let us note that the control expression is in general a lambda form lf. In particular, it can also be an expression e, if $\mathit{lf} = \lambda\overline{x_i}^0.e$. Also, all occurrences of superscripts n are assumed to be $n > 0$. Note that, this is not a different machine, but just the same machine *Mark-2* executed with a restricted input language. Additionally, there are some optimizations which do not essentially affect the original behavior:

- Original rule *var1* has been split into three: the one corresponding to the original *var1* is now called *var1c*; the two other rules are just special cases in which the expression referenced by pointer p is a normal form. The original *Mark-2* machine will execute in sequence either rule *var1* followed by rule *var2*, or rule *var1* followed by rule *var3*. These sequences have been respectively subsumed in the new rules *var1a* and *var1b*.
- Trimmer sets have been added to lambda forms and to continuations. Environments are trimmed to the set of free variables when new closures are created in the heap in rules *letrec* and *var3*, and also when continuations are stored in the stack in rule *case1*. This modification only affects to the set of live closures in the heap which now is smaller. Otherwise, the machine behavior is the same. As in the STG machine, trimmers are not used when making an update with a lambda normal form.

In machine of Figure 7, a new column *Last* has been added recording the last rule executed by the machine. This new field is important to define *stable* configurations. In turn, stable configurations will be used to compare the evolution of machines *Mark-2* and STG-1S (see Section 4.3).

Definition 12 *A configuration* $(\Gamma, \mathit{lf}, E, S, l)$ *of machine Mark-2 is* stable *if one of these two conditions hold:*

1. $\mathit{lf} = e \wedge l \notin \{\mathit{app1}, \mathit{var3}\}$, *or*
2. $S = [\,] \wedge ((\mathit{lf} = \lambda\overline{x_i}^n.e \wedge n > 0) \vee \mathit{lf} = C\ \overline{x_i})$

	Heap	Control	Environment	Stack	Last	rule
	$\varGamma$	$x\ \overline{x_i}^n$	$E \cup [x_n \mapsto p]$	S	l	app1
$\Longrightarrow$	$\varGamma$	$x\ \overline{x_i}^{n-1}$	$E \cup [x_n \mapsto p]$	$p : S$	app1	
	$\varGamma$	$\lambda\overline{x_i}^n.e$	E	$p : S$	l	app2
$\Longrightarrow$	$\varGamma$	$\lambda\overline{x_i}^{n-1}.e$	$E \cup [x_n \mapsto p]$	S	app2	
	$\varGamma \cup [p \mapsto (\lambda\overline{x_i}^n.e', E')]$	x	$E \cup [x \mapsto p]$	S	l	var1a
$\Longrightarrow$	$\varGamma \cup [p \mapsto (\lambda\overline{x_i}^n.e', E')]$	$\lambda\overline{x_i}^n.e'$	E'	S	var1a	
	$\varGamma \cup [p \mapsto (C\ \overline{x_i}, E')]$	x	$E \cup [x \mapsto p]$	S	l	var1b
$\Longrightarrow$	$\varGamma \cup [p \mapsto (C\ \overline{x_i}, E')]$	$C\ \overline{x_i}$	E'	S	var1b	
	$\varGamma \cup [p \mapsto (e', E')]$	x	$E \cup [x \mapsto p]$	S	l	var1c (*)
$\Longrightarrow$	$\varGamma$	e'	E'	$\#p : S$	var1c	
	$\varGamma$	$\lambda\overline{x_i}^n.e$	E	$\#p : S$	l	var2
$\Longrightarrow$	$\varGamma \cup [p \mapsto (\lambda\overline{x_i}^n.e, E)]$	$\lambda\overline{x_i}^n.e$	E	S	var2	
	$\varGamma$	$\mathbf{letrec}\ \overline{x_i = \mathit{lf}_i \vert^{t_i}}\ \mathbf{in}\ e$	E	S	l	letrec (**)
$\Longrightarrow$	$\varGamma \cup [p_i \mapsto (\mathit{lf}_i, E' \vert^{t_i})]$	e	E'	S	letrec	
	$\varGamma$	$\mathbf{case}\ e\ \mathbf{of}\ alts \vert^t$	E	S	l	case1
$\Longrightarrow$	$\varGamma$	e	E	$(alts, E \vert^t) : S$	case1	
	$\varGamma$	$C_k\ \overline{x_i}$	$E \cup [\overline{x_i \mapsto p_i}]$	$(alts, E') : S$	l	case2 (***)
$\Longrightarrow$	$\varGamma$	e_k	$E' \cup [\overline{y_{ki} \mapsto p_i}]$	S	case2	
	$\varGamma$	$C_k\ \overline{x_i}$	E	$\#p : S$	l	var3
$\Longrightarrow$	$\varGamma \cup [p \mapsto (C_k\ \overline{x_i}, E \vert^{\{\overline{x_i}\}})]$	$C_k\ \overline{x_i}$	E	S	var3	

(*) $e' \neq C\ \overline{x_i}$ y $e' \neq \lambda\overline{x_i}^n.e$

(**) Variables $\overline{p_i}$ are distinct and fresh w.r.t. $\varGamma$, **letrec** $\overline{x_i = \mathit{lf}_i \vert^{t_i}}$ **in** e, and S, $E' = E \cup [\overline{x_i \mapsto p_i}]$

(***) Expression e_k corresponds to alternative $C_k\ \overline{y_{ki}} \to e_k$ in *alts*

Fig. 7. Abstract machine *Mark-2* for FUN

In the STG machine, lambda abstractions never appear in the control expression, so it seems natural to exclude lambda abstractions from stable configurations. If the last rule executed is *app1*, then *Mark-2* is still pushing arguments in the stack and it has not yet evaluated the variable x corresponding to the function to be applied. In the STG all these intermediate states do not exist. If the last rule applied is *var3*, then the STG is probably still doing updates and, in any case, pattern matching has not been done yet. As we want to compare configurations in which a FUN expression appears in the control, all these states must be regarded as 'internal'. The second possibility is just a termination state.

Definition 13 *Let us assume that m and m' are stable configurations of Mark-2 machine, and $m \Rightarrow^+ m'$, (i.e. there must be at least one transition) and there is no other stable configuration between m and m'. We will say that m* evolves *to m' and will denote it by $m \Rightarrow_s^+ m'$.*

4.3 The machine STG-1S

In this section we define an abstract machine very close to the STG[5] and show that it is correct and complete w.r.t. *Mark-2* of Figure 7. We call it STG-1S because the main difference with the actual STG is that it has one stack instead of three. The single stack of STG-1S, contains the three usual kind of objects: arguments of applications, continuations and update markers. Being faithful to STGL, the control expression of STG-1S may have three different forms:

	Heap	Control	Environment	S	rule
	Γ	$Eval\ (x\ \overline{x_i}^n)$	$E \cup [x \mapsto p, \overline{x_i \mapsto p_i}]$	S	1
$\Longrightarrow$	Γ	$Enter\ p$	$\{\}$	$\overline{p_i} : S$	
	$\Gamma \cup [p \mapsto (\lambda \overline{x_i}^n.e, E)]$	$Enter\ p$	$\{\}$	$\overline{p_i}^n : S$	2
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\lambda \overline{x_i}^n.e, E)]$	$Eval\ e$	$E \cup [\overline{x_i \mapsto p_i}]$	S	
	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E)]$	$Enter\ p$	$\{\}$	S	$2'$
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E)]$	$Eval\ (C\ \overline{x_i})$	E	S	
	Γ	$Eval\ (\mathbf{letrec}\ \{\overline{x_i = lf_i \vert^{t_i}}\}\ \mathbf{in}\ e)$	E	S	3 (1)
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto (lf_i, E' \vert^{t_i})}]$	$Eval\ e$	E'	S	
	Γ	$Eval\ (\mathbf{case}\ e\ \mathbf{of}\ alts \vert^{t})$	E	S	4
$\Longrightarrow$	Γ	$Eval\ e$	E	$(alts, E \vert^{t}) : S$	
	Γ	$Eval\ (C\ \overline{x_i})$	$E \cup [\overline{x_i \mapsto p_i}]$	S	5
$\Longrightarrow$	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	S	
	Γ	$ReturnCon\ C_k\ \overline{p_i}$	$\{\}$	$(alts, E) : S$	6 (2)
$\Longrightarrow$	Γ	$Eval\ e_k$	$E \cup [\overline{y_{ki} \mapsto p_i}]$	S	
	$\Gamma \cup [p \mapsto (e, E)]$	$Enter\ p$	$\{\}$	S	15 (3)
$\Longrightarrow$	Γ	$Eval\ e$	E	$\#p : S$	
	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	$\#p : S$	16 (4)
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, [\overline{x_i \mapsto p_i}])]$	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	S	
	$\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E)]$	$Enter\ p$	$\{\}$	$\overline{p_i}^k : \#p' : S$	17 (5)
$\Longrightarrow$	Γ'	$Enter\ p$	$\{\}$	$\overline{p_i}^k : S$	

(1) Variables $\overline{p_i}$ are distinct and fresh w.r.t. Γ, **letrec** $\overline{x_i = lf_i |^{t_i}}$ **in** e, and S, $E' = E \cup [\overline{x_i \mapsto p_i}]$
(2) Expression e_k corresponds to alternative $C_k\ \overline{y_{ki}} \to e_k$ in *alts*
(3) Expression $e \neq C\ \overline{x_i}$ y $e \neq \lambda \overline{x_i}^n.e'$
(4) In rule 16, $\overline{x_i}$ are arbitrary distinct variables.
(5) $k < n$ and $\Gamma' = \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E), p' \mapsto (\lambda^{n-k} \overline{x_i}.e, E \cup [\overline{x_i \mapsto p_i}^k])]$

Fig. 8. Abstract machine *STG-1S*

- *Eval e E*, where e is a FUN expression (we recall that this excludes lambda forms) and E is an environment mapping e's free variables to heap pointers.
- *Enter p*, where p is a heap pointer. Notice that there is no environment.
- *ReturnCon C* $\overline{p_i}$, where C is a data constructor and $\overline{p_i}$ are its arguments given as a vector of heap pointers. Also, there is no environment here.

We will call each of these expressions an *instruction*, and use the symbol i to denote them. In order to better compare it with the *Mark-2* machine, we will consider a configuration of the STG-1S to be a 4-tuple (Γ, i, E, S), where Γ is a heap mapping pointers to closures, i is the control instruction, E is the environment associated to instruction i in case the instruction is of the form *Eval e*, and the empty environment $\{\}$ otherwise, and S is the stack. In Figure 8, the transition rules of STG-1S are shown.

We have numbered the rules with the same numbers used in [5] for easy reference. As there is no explicit flag $\backslash\pi$ in FUN lambda forms in order to avoid unnecessary updates, rules 2 and $2'$ reflect that no update frame is pushed in the stack when explicit normal forms in the heap are referenced. Rule $2'$ does not appear in [5], but it is implicit in rule 2.

Now we proceed with the comparison. As in *Mark-2*, we first define *stable* configurations in STG-1S. A stable configuration corresponds either to the evaluation of a FUN expression or to a termination state.

Definition 14 *A configuration $s = (\Gamma, i, E, S)$ of machine STG-1S is* stable *if either*

1. $i = Eval\ e$ *for some* e, *or*
2. $s = (\Gamma, ReturnCon\ C\ \overline{p_i}, \{\}, [\,])$, *or*
3. $s = (\Gamma \cup [p \mapsto (\lambda \overline{x_i}^n.e, E')], Enter\ p, \{\}, [p_1, \ldots, p_k]) \wedge n > k \geq 0.$

Configurations 2 and 3 correspond to termination states. Notice in 3 that the STG-1S may successfully stop with a non-empty stack. This would happen when the initial expression evaluates to a lambda abstraction. As in *Mark-2* machine, we will use $s \Rightarrow_s^+ s'$ to denote the evolution between two stable configurations in STG-1S with no intermediate stable ones, and say that s *evolves* to s'. The notion of *consistent* configuration for the STG-1S machine is the same given in Definition 10 for machine *Mark-2*.

We will now compare two evolutions, one in each machine starting from equivalent states, and show that they exactly pass through the same number of stable configurations and that the corresponding configurations are equivalent. This amounts to saying that there exists a *bisimulation* between the machines. To simplify to notion of configuration equivalence, we will assume that both machines use exactly the same fresh name generator in rule *letrec*. So, if the generator is given the same inputs (i.e. the same control expression, heap and stack), it will generate the same set of fresh variables.

Definition 15 *A configuration $m = (\Gamma, lf, E, S, l)$ in a stable state of machine Mark-2 and a configuration $s = (\Gamma', i, E', S')$ in a stable state of machine STG-1S are said to be* equivalent, *written $m \equiv s$, if*

- $\Gamma = \Gamma'$, *and*
- *one of the following possibilities holds:*
  1. $i = Eval\ e \wedge lf = e \wedge E = E' \wedge S = S'$
  2. $i = ReturnCon\ C\ \overline{p_i}\ \wedge\ lf = C\ \overline{x_i}\ \wedge\ \overline{p_i} = E\ \overline{x_i}\ \wedge S = S' = [\,]$
  3. $i = Enter\ p\ \wedge \Gamma'\ p = (\lambda \overline{x_i}^n.e, E'')\ \wedge\ S' = [p_1, \ldots, p_k]\ \wedge\ n > k \geq 0\ \wedge\ lf = \lambda \overline{x_i}^{n-k}.e\ \wedge\ E\ [x_n, \ldots, x_{n-k+1}] = [p_1, \ldots, p_k]$

The following proposition and corollary establish that STG-1S and *Mark-2* machines bisimulate each other. By transitivity this shows that STG-1S is correct and complete w.r.t. Launchbury's natural semantics.

Proposition 16 *Given two stable and consistent configurations m and s in respectively Mark-2 and STG-1S machines such that $m \equiv s$,*

1. *If $m \Rightarrow_s^+ m'$, then there exists a stable and consistent configuration s' such that $s \Rightarrow_s^+ s'$ and $m' \equiv s'$.*
2. *If $s \Rightarrow_s^+ s'$, then there exists a stable and consistent configuration m' such that $m \Rightarrow_s^+ m'$ and $m' \equiv s'$.*

Corollary 17 *If e is a closed FUN expression, then $(\{\}, e, \{\}, [\,], \perp) \Rightarrow^* m_f$ in Mark-2 machine with $m_f = (\Delta, w, E, [\,])$ if and only if there exists a stable configuration s_f such that $(\{\}, Eval\ e, \{\}, [\,]) \Rightarrow^* s_f$ in STG-1S and $m_f \equiv s_f$.*

5 The abstract machine STG

It has three stacks: the *argument* stack as containing arguments for pending applications; the *return* stack rs containing pairs $(alts, E)$; and the *update* stack us containing update frames. An update frame is a triple (as, rs, p) consisting of an argument stack, a return stack and a pointer p to the closure to be updated. We do not show the STG rules as they can be easily derived from those of STG-1S. A configuratión will be a 6-tuple $(\Gamma, i, E, as, rs, us)$.

The two differences with the STG-1S machine of previous section are:

- Pushing and popping is done in the appropriate stack according to the rule.
- Instead of pushing update markers, the STG machine pushes update frames and leaves empty argument and return stacks in the configuration. When a normal form is reached with empty stacks, an update is triggered.

Apparently, these differences are not essential and one may think that the behavior of both machines is the same. This is not the case as we will see in a moment. The splitting of the single stack into three has the unfortunate consequence of losing the temporal order of events between stacks as and rs. Then, a continuation pushed into rs *before* an argument is pushed into as can be retrieved also *before* the argument is retrieved form as instead of *after*, as it would be the case in the STG-1S machine. Consider the following ill-typed program:

$$e = \textbf{ letrec } y_1 = Nil;\ id = \lambda x.x \textbf{ in case } y_1\, y_1 \textbf{ of } Nil \rightarrow id$$

which has no semantics. The STG machine reduces it as follows:

$$\begin{array}{l}
\quad (\{\},\ e,\ \{\},\ [\,],\ [\,],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Eval\ (\textbf{case}\ y_1\ y_1\ \textbf{of}\ Nil \rightarrow id),\ E_1,\ [\,],\ [\,],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Eval\ (y_1\ y_1),\ E_1,\ [\,],\ [(Nil \rightarrow id,\ E_1)],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Enter\ p_1,\ \{\},\ [p_1],\ [(Nil \rightarrow id,\ E_1)],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Eval\ Nil,\ \{\},\ [p_1],\ [(Nil \rightarrow id,\ E_1)],\ [\,]) \\
\Rightarrow (\Gamma_1,\ ReturnCon\ Nil\ \{\},\ [p_1],\ [(Nil \rightarrow id,\ E_1)],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Eval\ id,\ E_1,\ [p_1],\ [\,],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Enter\ p_2,\ \{\},\ [p_1],\ [\,],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Eval\ x,\ [x \mapsto p_1],\ [\,],\ [\,],\ [\,]) \\
\Rightarrow (\Gamma_1,\ Eval\ Nil,\ \{\},\ [\,],\ [\,],\ [\,]) \\
\Rightarrow (\Gamma_1,\ ReturnCon\ Nil\ \{\},\ [\,],\ [\,],\ [\,])
\end{array}$$

where $\Gamma_1 = [p_1 \mapsto (Nil,\ \{\}),\ p_2 \mapsto (\lambda x.x,\ \{\})]$ and $E_1 = [y_1 \mapsto p_1,\ id \mapsto p_2]$.

So, the correctness of the STG machine with three stacks relies on programs being well-typed. This was not the case with the STG-1S machine: if a program is ill-typed, both Launchbury's semantics and STG-1S will be unable to derive a normal form for it.

However, the STG is complete in the sense that every successful derivation done by the STG-1S can obviously be done by the STG. For every configuration of the STG-1S we can compute a single equivalent configuration in the STG machine. The only thing needed is to split the stack into three algorithm which is very easy to define. The opposite is not true, i.e. given the three stacks as, rs and us of STG, many different stacks for the STG-1S can be constructed by interleaving the contents of the corresponding sections of stacks as and rs.

6 Conclusions

The paper has followed all the way from an abstract operational semantics for lazy evaluation such as Launchbury's, to a very concrete and efficient abstract machine such as the STG. Part of that way had already been followed by Sestoft in [9]. We have started at one of his machines, the *Mark-2*, and have shown that a STG machine with one stack can be derived from it, and that a bisimulation can be defined between both.

We have solved a small problem of Sestoft's semantics regarding freshness of variables and also added some garbage collection considerations to his semantics. As a result, the stack of Sestoft's machines appears very naturally as a transformation of some sets A, B and C needed by the semantics in order to have a complete control over freshness and over live closures. It is interesting to note that the optimization of not using the set of update markers as roots for the garbage collector can be easily understood at the semantic level.

We have also shown that the correctness of the STG machine as described in [5] relies on program being well-typed. This was an underlying assumption which was not explicitly stated in that description.

The obvious solution to this 'problem' is to come back to a single stack machine, and this seems to be the option recently chosen by GHC's implementors (although probably for different reasons). Having only one stack complicates the garbage collector task because pointers and non-pointers must be clearly distinguished. The presence of unboxed primitive values in the stack makes the problem even worse. In compensation, update markers are smaller than update frames and, most important of all, the temporal order of events is preserved.

References

1. J. Fairbairn and S. C. Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In *Proc. of the 1987 Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.
2. T. Johnsson. Efficient Compilation of Lazy Evaluation. *ACM SIGPLAN Notices*, 19(6):58–69, June 1984.
3. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. Conference on Principles of Programming Languages, POPL'93*. ACM, 1993.
4. J. Mountjoy. The Spineless Tagless G-machine, Naturally. In *Third International Conference on Functional Programming, ICFP'98, Baltimore, USA*. ACM Press, 1998.
5. S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine, Version 2.5. *Journal of Functional Programming*, 2(2):127–202, April 1992.
6. S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.
7. S. L. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. URL http://www.haskell.org, February 1999.
8. S. L. Peyton Jones, S. Marlow, and A. Reid. The STG Runtime System (revised). http://www.haskell.org/ghc/docs, 1999.
9. P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.