JFP **19** (1): 47–94, 2009.
doi:10.1017/S0956796808006746 First published online 18 March 2008 Printed in the United Kingdom

From natural semantics to C: A formal derivation of two STG machines

ALBERTO DE LA ENCINA and RICARDO PEÑA*

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

Abstract

The *Spineless Tag-less G-machine* (STG machine) was defined as the target abstract machine for compiling the lazy functional language Haskell. It is at the heart of the *Glasgow Haskell Compiler* (GHC) which is claimed to be the Haskell compiler that generates the most efficient code. A high-level description of the STG machine can be found in Peyton Jones (In Journal of Functional programming, 2(2), 127–202, 1992), Marlow & Peyton Jones (In Sigplan Not., 39(9), 4–5, 2004), and Marlow & Peyton Jones (In Journal of Functional Programming, 16(4–5), 415–449, 2006). Should the reader be interested in a more detailed view, then the only additional information available is the Haskell code of GHC and the C code of its runtime system.

It is hard to prove that this machine correctly implements the lazy semantics of Haskell. Part of the problem lies in the fact that the STG machine executes a bare-bones functional language, called STGL, much lower level than Haskell. Therefore, part of the correctness should be—and it is—established by showing that the translation from Haskell to STGL preserves Haskell's semantics.

The other part involves showing that the STG machine correctly implements the lazy semantics of STGL. In this paper we provide a step-by-step formal derivation of the STG machine and of its compilation to C, starting from a natural semantics of STGL. Thus, our starting point is higher level than the descriptions found Peyton Jones (In Journal of Functional programming, 2(2), 127–202, 1992) and Marlow & Peyton Jones (In Sigplan Not., 39(9), 4–5, 2004), and our arrival point is lower level than those works. Additionally, there has been substantial changes between the so-called push/enter model of the STG machine described in Peyton Jones (In Journal of Functional programming, 2(2), 127–202, 1992), and the eval/apply model of the STG machine described in Marlow & Peyton Jones (In Sigplan Not., 39(9), 4–5, 2004). So, in fact, we derive *two* machines instead of one, starting from the same initial semantics.

At each step we provide enough intuitions and explanations in order to understand the refinement, and then the formal definitions and statements proving that the derivation step is sound and complete. The main contribution of the paper is to show that an efficient machine such as the STG can be presented, understood, and formally reasoned about at different levels of abstraction.

1 Introduction

The *Spineless Tag-less G-machine* (STG) (Peyton Jones & Salkild 1989; Peyton Jones 1992; Marlow & Peyton Jones 2004; Marlow & Peyton Jones 2006) is at the heart of

* Work partially supported by the Spanish project TIN2004 07943-C04-04. The first author has also been supported by the MCYT project TIN2006-15578-C02-01 (WEST) and the second author has also been supported by the Madrid Region project S-0505/TIC/0407 (PROMESAS).

the *Glasgow Haskell Compiler* (GHC) (Peyton Jones *et al.* 1993; Peyton Jones 1996) which is probably the Haskell compiler that generates the most efficient code. For a description of the Haskell language see Peyton Jones & Hughes (1999). One of the reasons for this efficiency is the set of analyses and transformations carried out at the intermediate representation level (Peyton Jones & Santos 1998; Peyton Jones & Marlow 2002); another reason is the efficient design and implementation of the STG machine.

A high-level description of the STG can be found at Peyton Jones (1992). Recently, the machine has undergone important modifications (Marlow & Peyton Jones 2004; Marlow & Peyton Jones 2006) in particular, in the way function applications are handled. The so-called *push-enter* model has been replaced by a new one based on an *eval-apply* style; the efficiency of the generated code is approximately the same in both models, while the compiler is much simpler in the new model.

However, the gap between these detailed machine descriptions and the language semantics, is too wide to be brushed aside in a single step. Moreover, if somebody is interested in learning how the language constructions are mapped into the target language C,[1] then the only information available is the Haskell code of GHC and the C code of its different runtime systems, amounting to thousands of lines.

In this paper we provide a step-by-step formal derivation of the two STG machines, the old and the new one, starting from the standard Launchbury semantics for lazy evaluation of the STGL language (Launchbury 1993), passing through descriptions at the same level as those of Peyton Jones (1992), Marlow & Peyton Jones (2006), and finally arriving at a description at the target language level. We structure the presentation in such a way that the common parts of both machines and of both translations to imperative code are highlighted. This, we hope, can help to understand the differences and similarities between the two evaluation models.

There has been a previous attempt at formally deriving the STG machine by Jon Mountjoy (Mountjoy 1998) which, in our opinion, was not completely successful but has strongly inspired the present work. Here, we go beyond his work in the sense that we also provide a translation of the source language to imperative code. Other attempts by ourselves resulted in Encina & Peña (2002) and Encina & Peña (2003). In the first one, we proved the equivalence between the old STG machine and one of Sestoft's machines (Sestoft, 1997). In the second one we followed for the old model a similar approach to the one followed here, but in this paper we also introduce the new model and some important differences in the semantics, in the machines and in the translation. Also, here we present the relevant parts of the proofs. This paper can then be considered as an extended and corrected version of Encina & Peña (2003).

As Mountjoy's, our starting point is the commonly accepted operational semantics for lazy evaluation provided by Peter Sestoft (Sestoft 1997) which in turn was an improvement over John Launchbury's well-known natural semantics (Launchbury 1993). Then, we present the following refinements:

1. A new operational semantics, which we call semantics $S3$—acknowledging that semantics $S1$ and $S2$ were those defined by Mountjoy—where lambdas and data constructions appear only in **let** bindings, and applications of n arguments

[1] GHC targets not only C but also assembly language for some platforms and C^{--}. For the sake of simplicity we will assume in the following that C is the target language.

are done simultaneously. We formally prove the equivalence between semantics $S3$ and Sestoft's.

2. Two machines, called STG_{PE}-1 and STG_{EA}-1, respectively implementing the *push-enter* and the *eval-apply* models, in which explicit substitution of pointers for variables is done in reductions, are derived from $S3$. We prove the soundness and completeness of these two machines with respect to $S3$.
3. Then, two machines STG_{PE}-2 and STG_{EA}-2 are derived by introducing environments in closures, **case** alternatives, and in the control expression. We prove the equivalence between these machines and their counterparts STG_{PE}-1 and STG_{EA}-1.
4. Next, we derive two machines, called $ISTG_{PE}$ and $ISTG_{EA}$ (the 'I' stands for *imperative*) with a very small set of elementary instructions, which can be very easily implemented in a conventional language such as C. The two imperative machines differ only in two instructions, clearly showing the low level differences of the two evaluation models.
5. Finally, two translations from the source functional language to the imperative languages of $ISTG_{PE}$ and $ISTG_{EA}$ are provided. We show how the data structures of STG_{PE}-2 and STG_{EA}-2 are represented (or implemented) by the imperative machines. We also prove the correctness of the translations.

By putting together all the proof steps, the imperative implementations are shown correct with respect to Sestoft's operational semantics.

Our main contribution is to show that an efficient machine such as the STG can be presented, understood, and formally reasoned about at different levels of abstraction. By introducing enough intermediate levels, we highlight where and why the important decisions are taken, and also make the formal proofs relatively simple. As an interesting byproduct, we show that both implementations, the old and the new one, can be derived and proved correct starting from a single semantics of the source language.

The organization of the paper is as follows: after this introduction, in Section 2 we give an overview of Launchbury's and Sestoft's semantics. In Section 3 a new language called *Fun* is introduced and its operational semantics, which we call $S3$, is defined. Two theorems relating Sestoft's original language and semantics to the new ones are proven. Section 4 defines the four machines STG_{PE}-1, STG_{EA}-1, STG_{PE}-2, and STG_{EA}-2. Some propositions show the consistency of the machines and the soundness and completeness of STG_{PE}-1 and STG_{EA}-1 with respect to $S3$. Section 5 defines machines $ISTG_{PE}$ and $ISTG_{EA}$ and the translation from the source constructions to imperative code. Two invariants are proved which show the correctness of the translation. Section 6 discusses the differences between our translation and the actual implementation by GHC, surveys related work, and concludes.

2 Natural semantics

2.1 Launchbury's original proposal

We begin by reviewing the language and semantics given by Sestoft as an improvement over Launchbury's semantics. Both share the language given in Figure 1, that

$$
\begin{array}{lcll}
v & \to & x & \text{-- bound variable} \\
 & | & p & \text{-- free variable} \\
 & & & \\
e & \to & v & \\
 & | & \lambda x.e & \text{-- lambda abstraction} \\
 & | & e\ v & \text{-- application} \\
 & | & \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e & \text{-- recursive let} \\
 & | & C\ \overline{v_i} & \text{-- constructor application} \\
 & | & \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} & \text{-- case expression} \\
 & & & \\
w & \to & \lambda x.e & \text{-- weak head normal forms} \\
 & | & C\ \overline{v_i} &
\end{array}
$$

Fig. 1. Launchbury's and Sestoft's original language *Basic*.

$$
\frac{}{\Gamma : \lambda x.e \Downarrow_A \Gamma : \lambda x.e} \quad \textit{Lam}
$$

$$
\frac{}{\Gamma : C\ \overline{p_i} \Downarrow_A \Gamma : C\ \overline{p_i}} \quad \textit{Cons}
$$

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

$$
\frac{\Gamma : e \Downarrow_{A \uplus \{p\}} \Delta : w}{\Gamma \uplus [p \mapsto e] : p \Downarrow_A \Delta \uplus [p \mapsto w] : w} \quad \textit{Var}
$$

$$
\frac{\Gamma \uplus [\overline{p_i \mapsto e_i[\overline{p_i/x_i}]}] : e[\overline{p_i/x_i}] \Downarrow_A \Delta : w \quad \overline{p_i} \in \mathit{fresh}}{\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow_A \Delta : w} \quad \textit{Letrec}
$$

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

Fig. 2. Sestoft's natural semantics.

from now on we will refer to as *Basic*, where $\overline{A_i}$ denotes a vector $A_1, \ldots, A_n$ of subscripted entities. When n is important, we will write $\overline{A_i}^n$. Unless otherwise stated we will assume throughout the paper that $n > 0$, $m > 0$ and $l \geqslant 0$. Also, $\overline{x_i}$ will denote a non empty vector, except for constructor applications or patterns such as $C\ \overline{x_i}$.

Basic is a normalised λ-calculus, extended with recursive **let**, constructor applications and **case** expressions. The normalisation process forces constructor applications to be saturated, and all applications to only have variables as arguments. Weak head normal forms, denoted by w, are either lambda abstractions or constructions.

Sestoft's semantic rules are given in Figure 2. There, a judgement $\Gamma : e \Downarrow_A \Delta : w$ denotes that expression e, with its free variables bound in heap Γ, reduces to normal form w and produces the final heap Δ. When a fresh variable p is created (see rule *Letrec*), freshness is understood w.r.t. the left configuration $\Gamma : e \Downarrow_A$ of the rule's consequent. More precisely $p \notin \mathit{var}\ \Gamma \cup \mathit{var}\ e \cup A$, where *var* denotes the set of all variables occurring in heap Γ or in expression e. The set A contains the domain variables p of the heap bindings $[p \mapsto e]$ under evaluation (see rule *Var*). We say that freshness is *locally checkable* to indicate that a fresh variable can be created by simply looking into the current rule components. Rule *Letrec* is the only one in

which fresh variables are created, and bindings are added to the heap. By $e[p/x]$ we denote the substitution of p for the free occurrences of x in e. The notation $\Gamma[p \mapsto e]$ means that $[p \mapsto e] \in \Gamma$, while $\Gamma \uplus [p \mapsto e]$ denotes the disjoint union of Γ and $[p \mapsto e]$.

We say that $\Gamma : e \Downarrow \Theta : w$ is a successful derivation if it can be proved by using the rules. For instance, a derivation can fail because a configuration gets stuck. This would happen in rule *Var* when a reference to variable p appears while reducing expression e and before reaching a normal form. As this is done in a heap Γ not containing a binding for p, 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.

2.2 Sestoft's improvements

Sestoft proved in Sestoft (1997) the following proposition:

Proposition 1 (*Sestoft*)
If $\Gamma_0 : e_0 \Downarrow_{\{\}} \Theta : w_0$ is a successful derivation, in all judgements $\Gamma : e \Downarrow_A \Theta : w$ of the derivation tree the following properties hold:

1. $(dom\ \Gamma) \cap A = \emptyset$.
2. $fv\ e \subseteq (dom\ \Gamma) \uplus A$.
3. $bv\ e \cap ((dom\ \Gamma) \uplus A) = \emptyset$
4. For all $[p \mapsto e'] \in \Gamma$ we have $fv\ e' \in (dom\ \Gamma) \uplus A$ and $bv\ e' \cap ((dom\ \Gamma) \uplus A) = \emptyset$

That is, in every judgement of the derivation tree, there is a clear distinction between free variables and bound variables appearing in expressions, heaps and judgements: the former are either bound in the corresponding heap or they are under evaluation and belong to A, while the latter are program variables belonging to the original expression written by the programmer. Occasionally, we will use the term *pointers* to refer to dynamically created fresh variables, bound to expressions in the heap, and the term *program 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.

Unfortunately, the proof of the above theorem was completed before introducing **case** expressions and constructors and, when the latter were introduced, the theorem proof was not rewritten. With the current *Case* rule the freshness property is not locally checkable anymore: while reducing the discriminant in the judgement $\Gamma : e \Downarrow_A \Delta : C_k\ \overline{p_j}$, fresh variables with the same names as those bound in the alternatives may be created without violating the freshness condition. Thus, free variables may be captured.

To make freshness locally checkable, we would need an additional set holding the free and bound variables of the pending **case** alternatives. Adding a new set of variables to the derivation arrow $\Downarrow$ would be enough but instead, in Encina & Peña (2002), we introduced a multi-set C of *continuations* associated to every judgement. The alternatives of a **case** were stored in this multi-set during the evaluation of the discriminant. We then said that a variable p is fresh in a judgement $\Gamma : e \Downarrow_{AC} \Delta : w$ if $p \notin var\ \Gamma \cup var\ e \cup A \cup var\ C$. The modified rules are shown in Figure 3. The

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

$$\Gamma : C\ \overline{p_i} \Downarrow_{AC} \Gamma : C\ \overline{p_i} \qquad \textit{Cons}$$

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

$$\frac{\Gamma : e \Downarrow_{A \uplus \{p\} C} \Delta : w}{\Gamma \uplus [p \mapsto e] : p \Downarrow_{AC} \Delta \uplus [p \mapsto w] : w} \qquad \textit{Var}$$

$$\frac{\Gamma \uplus \overline{[p_i \mapsto e_i[\overline{p_i/x_i}]]} : e[\overline{p_i/x_i}] \Downarrow_{AC} \Delta : w \quad \overline{p_i} \in \textit{fresh}}{\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow_{AC} \Delta : w} \qquad \textit{Letrec}$$

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

Fig. 3. Sestoft's natural semantics corrected.

knowledgeable reader has probably recognised in these two sets A and C part of the stack of an abstract machine, as they represent work remaining to be done after reducing the current expression. In Encina & Peña (2002), we introduced a third set B in which the argument p of an application was stored in rule *App* while reducing the functional part e to normal form. The motivation for that was having available in the sets A, B, and C, the roots of the live part of the heap, and to develop a semantics in which we could reason about garbage collection. After that, the stack of the abstract machine was derived from these sets.

After modifying Launchbury's semantics, Sestoft derives in Sestoft (1997) several abstract machines, respectively called Mark-1, Mark-2, and Mark-3. His technique has inspired us the derivation of machines STG-1 and STG-2 in the present work.

3 A new semantics for lazy evaluation

Before presenting our new semantics $S3$ we modify the input language *Basic* in the following aspects:

1. We force λ-abstractions and constructor applications to appear only in **letrec** bindings. Expressions will include neither λ-abstractions nor constructions. We will use the term *binding expression* to refer either to the latter or to an expression.
2. Correspondingly, normal forms are variables bound in the heap to either λ-abstractions or constructions. During evaluation, a third kind of normal form, *partial application*, may arise. It corresponds to a λ-abstraction applied to a smaller number of actual than formal arguments.
3. λ-abstractions may now have more than one formal argument. Consequently, a function may apply now to more than one actual argument. Moreover, the functional expression is restricted to be a variable.

As in *Basic*, constructor applications are assumed to be *saturated*, i.e. not partially applied.

The syntax of the modified language, called *Fun*, is shown in Figure 4. The main motivation for the changes of *Fun* w.r.t. *Basic* is to make *Fun* as close as

		-- Variables	
v	$\rightarrow$	x	-- bound variable
	$\mid$	p	-- free variable
		-- Expressions	
e	$\rightarrow$	$v\ \overline{v_i}$	-- application
	$\mid$	v	-- variable
	$\mid$	**letrec** $\overline{x_i = be_i}$ **in** e	-- recursive let
	$\mid$	**case** e **of** $\overline{alt_i}$	-- case expression
alt	$\rightarrow$	$C\ \overline{x_j}^{l} \rightarrow e$	-- case alternative
		-- Binding expressions	
be	$\rightarrow$	$\lambda\ \overline{x_i}.e$	-- lambda abstraction
	$\mid$	$C\ \overline{v_i}^{l}$	-- constructor application
	$\mid$	e	-- expression
		-- Normal forms	
w	$\rightarrow$	$\lambda\ \overline{x_i}.e$	-- lambda abstraction
	$\mid$	$C\ \overline{v_i}^{l}$	-- constructor application
	$\mid$	$\mathrm{pap}(p\ \overline{p_i})$	-- partial application **(internal)**

Fig. 4. Language *Fun*.

possible to the STG abstract machine language. The latter is supposed to be an intermediate language which results from desugaring and transforming Haskell. Hence, programmers are not expected to write their programs in such a bare functional language. The actual STG language also includes primitive values such as integers and float numbers, primitive operators, and a **case** expression for primitive values. Also, **case** expressions may provide a default alternative. If this alternative is missing, the compiler will ensure that there is one for every constructor of the data type. Should the programmer have not specified an alternative for a particular constructor, the compiler will provide one with an *error* expression that would cause program abortion at runtime. In order to simplify the derivation of our machines, we have preferred to ignore in *Fun* these 'real life' aspects of the STG language. We believe that they would not add essential insight to the present work. Nevertheless, we will assume in *Fun* that **case** expressions contains exactly one alternative for every data constructor of the data type.

The reader may wonder why the STG and the *Fun* languages have the above described restrictions. Once, in subsequent sections, the STG machines are derived and the imperative translation of *Fun* is done, the reader will discover that having λ-abstractions and constructions in the heap saves much trashing and copying between the heap and the control expression. Rule *Var* in Sestoft's semantics of Figure 2 can give an idea of what we mean: each time a variable p is evaluated, the expression e bound to it in the heap must be installed as the current control expression. If e happens to be a normal form, this implies some kind of copying; also, when a heap binding must be updated with a normal form w this must be copied from the control expression to the heap.

Having applications to several actual arguments at once also leads to more efficient compiled code. The decision of having a variable as the only allowed functional part of applications is a different matter; in previous versions of this work, we allowed arbitrary expressions. We claimed that the resulting machine

$$\Gamma[p \mapsto w] : p \downarrow_S \Gamma : p \qquad \textit{Normal form}_{S3}$$

$$\frac{\Gamma : p \downarrow_{\overline{p_i}:S} \Delta[q \mapsto w] : q \quad \Delta : q\ \overline{p_i} \downarrow_S \Theta[t \mapsto w'] : t}{\Gamma[p \mapsto e] : p\ \overline{p_i} \downarrow_S \Theta : t} \qquad \textit{AppThunk}_{S3}$$

$$\frac{\Gamma : q\ \overline{q_i}\ \overline{p_i} \downarrow_S \Delta[t \mapsto w] : t}{\Gamma[p \mapsto \mathrm{pap}(q\ \overline{q_i})] : p\ \overline{p_i} \downarrow_S \Delta : t} \qquad \textit{AppPap}_{S3}$$

$$\frac{n \geqslant m \quad \Gamma : e[\overline{p_i/x_i}^m] \downarrow_{p_{m+1} \ldots p_n:S} \Delta[q \mapsto w] : q \quad \Delta : q\ p_{m+1} \ldots p_n \downarrow_S \Theta[t \mapsto w'] : t}{\Gamma[p \mapsto \lambda \overline{x_i}^m.e] : p\ \overline{p_i}^n \downarrow_S \Theta : t} \qquad \textit{App}_{S3}$$

$$\frac{n < m \quad q \in \textit{fresh}}{\Gamma[p \mapsto \lambda \overline{x_i}^m.e] : p\ \overline{p_i}^n \downarrow_S \Gamma \uplus [q \mapsto \mathrm{pap}(p\ \overline{p_i}^n)] : q} \qquad \textit{App}'_{S3}$$

$$\frac{\Gamma : e \downarrow_{\#p:S} \Delta[q \mapsto w] : q}{\Gamma \uplus [p \mapsto e] : p \downarrow_S \Delta \uplus [p \mapsto w] : q} \qquad \textit{Var}_{S3}$$

$$\frac{\overline{p_i} \in \textit{fresh} \quad \Gamma \uplus \overline{[p_i \mapsto be_i[\overline{p_i/x_i}]]} : e[\overline{p_i/x_i}] \downarrow_S \Delta[p \mapsto w] : p}{\Gamma : \textbf{letrec}\ \overline{x_i = be_i}\ \textbf{in}\ e \downarrow_S \Delta : p} \qquad \textit{Letrec}_{S3}$$

$$\frac{\Gamma : e \downarrow_{\overline{alt_i}:S} \Delta[p \mapsto C_k\ \overline{p_j}] : p \quad \Delta : e_k[\overline{p_j/y_{kj}}] \downarrow_S \Theta[q \mapsto w] : q}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{alt_i} \downarrow_S \Theta : q} \qquad \textit{Case}_{S3}$$

Fig. 5. Semantics $S3$.

saved constructing some unneeded heap bindings w.r.t. the actual STG machine implementation. The counterpart was having a rather complex *SLIDE* machine instruction in the imperative machine (see Section 5) to remove runtime environments from the stack. Also, the new eval-apply model could not be implemented in such a simple way should we allow any expression in the functional part of applications.

Fun's operational semantics—called $S3$ acknowledging the two previous attempts made by Mountjoy in Mountjoy (1998)—is given in Figure 5. There, a judgement $\Gamma : e \downarrow_S \Theta[p \mapsto w] : p$ expresses that the initial configuration (Γ, e, S) is reduced to the normal form configuration $(\Theta[p \mapsto w], p, S)$, where S is a stack with the following syntax:

$$\begin{array}{rcl} S & \rightarrow & \overline{p_i} : S \\ & | & \overline{alt_i} : S \\ & | & \#p : S \\ & | & [\,] \end{array}$$

The notation $o : S$ means pushing the object o to the stack S. The stack is introduced with two purposes: one is to precisely define freshness in rules $\textit{App}'_{S3}$ and $\textit{Letrec}_{S3}$, and the other to make the correctness of the first two machines easier to prove. In a judgement $\Gamma : e \downarrow_S \Theta[p \mapsto w] : p$, a variable $q \in \textit{fresh}$ if $q \notin \textit{var}\ \Gamma \cup \textit{var}\ e \cup \textit{var}\ S$. The stack contains three kinds of objects: (1) sequences $\overline{p_i}$ of arguments belonging to

pending applications; (2) continuations $\overline{alt_i}$ of **case** expressions whose discriminant expression e is under evaluation; and (3) update marks $\#p$ of bindings $[p \mapsto e]$ whose expression e is under evaluation.

The first rule expresses that normal forms are pointers to either λ-abstractions, partial applications or constructions. The next four take care of function applications. Rule $AppThunk_{S3}$ deals with the case in which the functional part is not in normal form. In rule $AppPap_{S3}$ the functional part is a partial application, while in rules App_{S3} and App'_{S3} the functional part is a λ-abstraction. The latter ones distinguish the case of having enough actual arguments from the case of not having them, where a partial application normal form is generated in the heap. The last three rules are almost identical to the corresponding ones of Sestoft's semantics. Notice that, in rule $Case_{S3}$, expression e_k represents the right-hand side expression of the alternative alt_k.

The reader may wonder why this semantics generates a third kind of normal form (partial applications) instead of generating just λ-abstractions, as Sestoft does. The obvious answer is: because this is what the STG machine we are trying to derive does. A more insightful answer is the following: generating λ-abstractions not corresponding to those of the original program would probably imply generating imperative code at runtime and this is something everyone would like to avoid. Instead, a partial application contains a pointer to a known λ-abstraction whose imperative code already exists. A partial application 'just waits' in the heap until enough arguments to the original λ-abstraction are provided (see rule $AppPap_{S3}$). Then, the lambda is fully applied by using rule App_{S3}. Notice also in rule $AppPap_{S3}$ that, should a partial application be applied to a number of arguments not enough to satisfy the arity of the lambda, then a new partial application would be generated by rule App'_{S3}. This new partial application would contain a copy of the arguments of the first one, plus some additional arguments. A different design option could have been generating an indirection to the previous partial application and not replicating the arguments. At this level, we do not feel the need to be efficient and we have chosen the first option; the second could be taken at the abstract machine design level.

Language *Fun* is at least as expressive as *Basic*. The following normalising function transforms any *Basic* expression into a semantically equivalent *Fun* expression.

Definition 2

The normalising function $\mathbb{N} : Basic \rightarrow Fun\text{Expression}$ is defined as follows:

$$
\begin{array}{lcll}
\mathbb{N}\ x & \stackrel{\text{def}}{=} & x & \\
\mathbb{N}\ (e\ x) & \stackrel{\text{def}}{=} & (\mathbb{N}\ e)\ x & \text{if } e = e'\ x' \\
\mathbb{N}\ (e\ x) & \stackrel{\text{def}}{=} & \textbf{letrec } y\ =\ \mathbb{N}'\ e \textbf{ in } y\ x & \text{if } e \neq e'\ x', y \notin fv\ e \\
\mathbb{N}\ (\lambda\ x.e) & \stackrel{\text{def}}{=} & \textbf{letrec } y\ =\ \mathbb{N}'\ (\lambda\ x.e) \textbf{ in } y & \text{if } y \notin fv\ (\lambda\ x.e) \\
\mathbb{N}\ (C\ \overline{x_i}) & \stackrel{\text{def}}{=} & \textbf{letrec } y\ =\ C\ \overline{x_i} \textbf{ in } y & \text{if } y \notin fv\ (C\ \overline{x_i}) \\
\mathbb{N}\ (\textbf{letrec } \overline{x_i\ =\ e_i} \textbf{ in } e) & \stackrel{\text{def}}{=} & \textbf{letrec } \overline{x_i\ =\ \mathbb{N}'\ e_i} \textbf{ in } \mathbb{N}\ e & \\
\mathbb{N}\ (\textbf{case } e \textbf{ of } \overline{C_i\ \overline{y_{ij}} \rightarrow e_i}) & \stackrel{\text{def}}{=} & \textbf{case } \mathbb{N}\ e \textbf{ of } \overline{C_i\ \overline{y_{ij}} \rightarrow \mathbb{N}\ e_i} &
\end{array}
$$

The auxiliary functions $\mathbb{N}'$, $\mathbb{N}''$: *Basic*→ *Fun*BindingExpression are defined as follows:

$$\begin{array}{lcll} \mathbb{N}'\ (C\ \overline{x_i}) & \stackrel{\text{def}}{=} & C\ \overline{x_i} & \\ \mathbb{N}'\ e & \stackrel{\text{def}}{=} & \mathbb{N}''\ e & \text{if } e \neq C\ \overline{x_i} \\ \mathbb{N}''\ (\lambda\ x.e) & \stackrel{\text{def}}{=} & \lambda\ x.\mathbb{N}''\ e & \\ \mathbb{N}''\ e & \stackrel{\text{def}}{=} & \mathbb{N}\ e & \text{if } e \neq \lambda\ x.e'. \end{array}$$

3.1 Soundness and completeness of semantics S3

To see that both semantics reduce an expression to equivalent normal forms, we first will prove that the normalisation does not change the meaning of an expression within Sestoft's semantics, and then that both semantics, Sestoft's and *S*3, reduce any *Fun* expression to equivalent normal forms, provided that such normal forms exist.

To prove that the normalisation process does not change the meaning of an expression, first we define an equivalence between a normalised expression $e^* = \mathbb{N}\ e$, and the original one e, in Sestoft's semantics. The evaluation of the normalised expression will create more heap bindings, as it has more **letrec** expressions. So, our equivalence should take into consideration the live variables of the transformed expression, the fact that the normalisation process produces a different expression, and the names of the fresh variables, which could be different in the two semantics. In the following, we will consistently use e^* to denote normalised (i.e. *Fun*) expressions, and e to denote *Basic* (i.e. possibly not normalised) expressions. We define the equivalence in three steps:

- First, we define the live part of a heap.
- Then, we define the equivalence between two Sestoft configurations up to α-renaming.
- Finally, we define the equivalence between two Sestoft configurations taking into account the normalisation process.

An α-renaming is a bijection between two finite sets V and V' of variables which, when applied to the bound variables of an expression, or of a binding (both to the left and the right parts of it), does not produce capture of free variables. We are interested in abstracting classes of configurations which are equivalent modulo an α-renaming, as it is frequently done Morrisett *et al.* (1995); Urban *et al.* (2004); Pitts (2005). We will use the notation $\beta\ e$ and $\beta\ \Gamma$ to respectively denote the α-renaming of and expression e and of all the bindings of a heap Γ by the bijection β.

Definition 3

Given a heap Γ, an expression e, and a set P of variables, we define the live part of Γ with respect to e and P, denoted $\Gamma^{P,e}_{live}$, as follows:

$$\Gamma^{P,e}_{live} = fix\ (\lambda L\,.\,L \cup \{[p \mapsto e'] \in \Gamma \mid p \in P \vee p \in fv\ e \vee (\exists [q \mapsto e''] \in L\,.\,p \in fv\ e'')\}$$

where *fix* denotes the least fix point.

Definition 4
Let $\Gamma : e$ and $\Gamma' : e'$ be two Sestoft configurations, P a set of variables, and an α-renaming β. We say that both configurations are P-equivalent, denoted $\Gamma : e \overset{\beta}{\equiv}_P \Gamma' : e'$, if $\Gamma^{P,e}_{live} : e = (\beta\ \Gamma')^{P,\beta\ e'}_{live} : \beta\ e'$.

Definition 5
Let be $\Gamma : e$ and $\Gamma^* : e^*$ two Sestoft configurations, P a set of variables, and an α-renaming β. We say that they are

1. *P-equivalent via normalisation*, denoted $\Gamma : e \overset{\beta}{\equiv}_{P,\mathbb{N}} \Gamma^* : e^*$, if $\mathbb{N}'\ \Gamma : \mathbb{N}\ e \overset{\beta}{\equiv}_P \Gamma^* : e^*$.
2. *P-equivalent normal forms via normalisation*, denoted $\Gamma : w \overset{\beta}{\equiv}_{P,\mathbb{N}'} \Gamma^* : w^*$, if $\mathbb{N}'\ \Gamma : \mathbb{N}'\ w \overset{\beta}{\equiv}_P \Gamma^* : w^*$.

The following theorem proves that the normalisation does not change the meaning of an expression within Sestoft's semantics.

Theorem 6 (*Sestoft* $\Leftrightarrow$ *Sestoft**)
Let $\Gamma : e$ and $\Gamma^* : e^*$ be two Sestoft configurations, and P be a set of variables, such that:

- $fv\ e^* \subseteq P$
- $\exists\beta.\Gamma : e \overset{\beta}{\equiv}_{P,\mathbb{N}} \Gamma^* : e^*$

Let A, A^* (sets of variables) and C, C^* (sets of **case** continuations), such that $P = A \cup fv\ C$, $A^* = \beta\ A$, and $C^* = \beta(\mathbb{N}\ C)$. Then:

1. If $\Gamma : e \Downarrow_{AC} \Delta : w$ then $\Gamma^* : e^* \Downarrow_{A^*C^*} \Delta^* : w^*$ and $\exists\beta'.\Delta : w \overset{\beta'}{\equiv}_{P,\mathbb{N}'} \Delta^* : w^*$.
2. If $\Gamma^* : e^* \Downarrow_{A^*C^*} \Delta^* : w^*$ then $\Gamma : e \Downarrow_{AC} \Delta : w$ and $\exists\beta'.\Delta : w \overset{\beta'}{\equiv}_{P,\mathbb{N}'} \Delta^* : w^*$.

Proof
1. By induction on the depth of e's normal form $\Downarrow$ derivation.
2. By induction on the depth of e^*'s normal form $\Downarrow$ derivation. □

Corollary 7
Given a closed expression $e \in$ *Basic*:

$$\{\} : e \Downarrow_{\{\}\{\}} \Delta : w \ \text{ iff } \ \{\} : \mathbb{N}\ e \Downarrow_{\{\}\{\}} \Delta^* : \mathbb{N}'\ w \ \text{ and } \ \exists\beta.\Delta : w \overset{\beta}{\equiv}_{\{\},\mathbb{N}'} \Delta^* : w^*$$

Before proving the equivalence between the two semantics, we must ensure that there is no variable capture in $S3$. The following proposition, equivalent to Sestoft's Proposition 1, establishes that:

Proposition 8
If $\Gamma : e \downarrow_S \Theta : w$ is a successful $S3$ derivation, and $fv\ e \cap bv\ e = \emptyset$, and for all $[p \mapsto e'] \in \Gamma$, $fv\ e' \cap bv\ e' = \emptyset$ holds, then:

1. $fv\ w \cap bv\ w = \emptyset$, and
2. For all $[p \mapsto e'] \in \Theta$, $fv\ e' \cap bv\ e' = \emptyset$ holds, and

3. The premise of the proposition, and (1), and (2) hold for every intermediate judgement of the derivation.

Proof
By induction on the depth of the $\downarrow_S$ derivation. □

The next theorem proves the equivalence between the two semantics. We consider only *Fun* expressions because it has already been proved that the normalisation does not change the Sestoft's meaning of expressions. Now, we define an equivalence between a Sestoft configuration and an $S3$ configuration. The definition takes into account that Sestoft's and $S3$'s heaps are different: Sestoft's semantics applies functions to the arguments one by one and does not create partial applications. On the other hand, the control expression in $S3$ is never a λ-abstraction or a construction. Also, an α-renaming is needed. In the following, e denotes a *Fun* expression reduced within $S3$ semantics, and e^* a *Fun* expression reduced within Sestoft's semantics. We define the equivalence in two steps:

- First, partial applications in $S3$ heaps are removed and replaced by their equivalent λ-abstractions. It is clear that this operation does not change the meaning of heap expressions.
- Then, an equivalence between Sestoft and $S3$ configurations is defined, by taking into account only live variables and a possible α-renaming.

Definition 9
The partial-application removing function $\mathbb{E} : \text{Heap} \to \text{Heap}$ is defined as follows:

$$\begin{array}{lcll} \mathbb{E}\ \{\} & \stackrel{\text{def}}{=} & \{\} & \\ \mathbb{E}\ (\Gamma[q \mapsto \lambda\ \overline{x_i}^n.\lambda\ \overline{y_i}.e] \uplus [p \mapsto \text{pap}(q\ \overline{q_i}^n)]) & \stackrel{\text{def}}{=} & (\mathbb{E}\ \Gamma) \uplus [p \mapsto \lambda\ \overline{y_i}.e[\overline{q_i/x_i}^n]] & \\ \mathbb{E}\ (\Gamma \uplus [p \mapsto e]) & \stackrel{\text{def}}{=} & (\mathbb{E}\ \Gamma) \uplus [p \mapsto e] & \text{otherwise} \end{array}$$

Notice that function $\mathbb{E}$ does not change the number of bindings in the heap Γ.

Definition 10
Let $\Gamma : e$ be an $S3$ configuration, $\Gamma^* : e^*$ a Sestoft configuration, P a set of variables, and β an α-renaming. We say that they are *P-equivalent via partial application elimination*, denoted $\Gamma : e \stackrel{\beta}{\equiv}_{P,\mathbb{E}} \Gamma^* : e^*$, if

1. $\mathbb{E}\ \Gamma : e \stackrel{\beta}{\equiv}_P \Gamma^* : e^*$, or
2. e is a variable and $\mathbb{E}\ \Gamma : (\mathbb{E}\ \Gamma)\ e \stackrel{\beta}{\equiv}_P \Gamma^* : e^*$.

Some particular cases of this definition are:

$$\begin{array}{lcl} \Gamma[p \mapsto C\ \overline{x_i}] : p & \stackrel{\beta}{\equiv}_{P,\mathbb{E}} & (\mathbb{E}\ \Gamma) : C\ \overline{x_i} \\ \Gamma[p \mapsto C\ \overline{x_i}] : p & \stackrel{\beta}{\equiv}_{P,\mathbb{E}} & (\mathbb{E}\ \Gamma[p \mapsto C\ \overline{x_i}]) : p \\ \Gamma[p \mapsto \lambda\overline{x_i}.e] : p & \stackrel{\beta}{\equiv}_{P,\mathbb{E}} & (\mathbb{E}\ \Gamma) : \lambda\overline{x_i}.e \\ \Gamma[p \mapsto \lambda\overline{x_i}^n\lambda\overline{y_i}.e,\ q \mapsto \text{pap}(p\ \overline{p_i}^n)] : q & \stackrel{\beta}{\equiv}_{P,\mathbb{E}} & (\mathbb{E}\ \Gamma) : \lambda\overline{y_i}.e[\overline{p_i/x_i}] \\ \Gamma[p \mapsto \lambda\overline{x_i}.e] : p & \stackrel{\beta}{\equiv}_{P,\mathbb{E}} & (\mathbb{E}\ \Gamma[p \mapsto \lambda\overline{x_i}.e]) : p \end{array}$$

In order to prove the general theorem we need the following auxiliary lemmas:

Lemma 11
Let n and m satisfy $n \geqslant m > 0$. If $\Gamma : p \Downarrow_{\{\overline{p_i}^n \cup A\}C} \Delta : \lambda \overline{x_i}^m.e$ and $\Delta : e[\overline{p_i/x_i}^m]\ p_{m+1} \ldots p_n \Downarrow_{AC} \Theta : w$, then $\Gamma : p\ \overline{p_i}^n \Downarrow_{AC} \Theta : w$.

Proof
By induction on n. □

Lemma 12
If $\Gamma : p \Downarrow_{\{\overline{p_i}^n \cup A\}C} \Delta : \lambda \overline{x_i}^n.\lambda \overline{y_i}.e$ then $\Gamma : p\ \overline{p_i}^n \Downarrow_{AC} \Gamma : \lambda \overline{y_i}.e[\overline{p_i/x_i}^n]$

Proof
Trivial, by induction on n. □

Lemma 13

$$\text{If } \Gamma : p\ \overline{p_i}^n \Downarrow_{AC} \Theta : w, \text{ then } \begin{cases} \Gamma : p \Downarrow_{\{\overline{p_i}^n \cup A\}C} \Delta : \lambda \overline{x_i}^m.e, \text{ and} \\ \text{if } n \geqslant m \text{ then } \Delta : e[\overline{p_i/x_i}^m]\ p_{m+1} \ldots p_n \Downarrow_{AC} \Theta : w \\ \qquad \text{else } \begin{cases} \lambda \overline{x_i}^m.e = \lambda \overline{y_i}^n.\lambda \overline{z_i}^{m-n}.e \\ w = \lambda \overline{z_i}^{m-n}.e[\overline{p_i/y_i}^n] \\ \Delta = \Theta \end{cases} \end{cases}$$

Proof
By induction on n. □

Lemma 14
If $\Gamma : p \Downarrow_{\{\overline{p_i}^n \cup A\}C} \Delta[q \mapsto w] : w$ and $\Delta : q\ \overline{p_i}^n \Downarrow_{AC} \Theta : w'$ then, $\Gamma : p\ \overline{p_i}^n \Downarrow_{AC} \Theta : w'$

Proof
Trivial, by induction on n □

Now, we can approach the proof of the main equivalence theorem. By *upd S* and *varalts S* we respectively denote the set of variables included in update marks and in **case** alternatives stored in stack S.

Theorem 15 (*Sestoft** $\Leftrightarrow S3$)
Let $\Gamma : e$ be an $S3$ configuration, $\Gamma^* : e^*$ be a Sestoft configuration, and P be a set of variables such that:

1. $fv\ e^* \subseteq P$, and
2. $\exists \beta.\Gamma : e \overset{\beta}{\equiv}_{P,\mathbb{E}} \Gamma^* : e^*$

Then, for all A (set of variables), C (set of **case** continuations), and S (stack), such that $P = A \cup fv\ C$, and $A \cup var\ C = \beta(upd\ S \cup varalts\ S)$, we have:

1. If $\Gamma^* : e^* \Downarrow_{AC} \Delta^* : w^*$ then $\Gamma : e \downarrow_S \Delta[p \mapsto w] : p$ and $\exists \beta'.\Delta : p \overset{\beta'}{\equiv}_{P,\mathbb{E}} \Delta^* : w^*$
2. If $\Gamma : e \downarrow_S \Delta[p \mapsto w] : p$ then $\Gamma^* : e^* \Downarrow_{AC} \Delta^* : w^*$ and $\exists \beta'.\Delta : p \overset{\beta'}{\equiv}_{P,\mathbb{E}} \Delta^* : w^*$

Proof
1. By induction on the depth of e^*'s normal form $\Downarrow$ derivation.
2. By induction on the depth of e's normal form $\downarrow$ derivation.

As an example of the proof technique, we will show in the proof of (2), the case where the initial $S3$ configuration has an application in th econtrol expression:

$\Gamma[p \mapsto e] : p\ \overline{p_i}^n$. By hypothesis, we know:

H1 $\Gamma : p\ \overline{p_i}^n \overset{\beta}{\equiv}_{P,\mathbb{IE}}\ \Gamma^* : e^*$ and $\{p, \overline{p_i}\} \subseteq P$
H2 $\Gamma : p\ \overline{p_i}^n \downarrow_S \Theta[t \mapsto w'] : t$
H3 By H1 and by applying the $S3$ $AppThunk_{S3}$ rule, we also know:

$$\Gamma : p \downarrow_{\overline{p_i}^n : S} \Delta[q \mapsto w] : q \quad \text{and} \quad \Delta : q\ \overline{p_i}^n \downarrow_S \Theta[t \mapsto w'] : t$$

By H1 and by Definition 10, we have $\Gamma^{P,p\ \overline{p_i}^n}_{live} : p\ \overline{p_i}^n = (\beta\ \Gamma^*)^{P,(\beta\ e^*)}_{live} : (\beta\ e^*)$, then $e^* = p^*\ \overline{p_i^*}^n$ and $p\ \overline{p_i}^n = \beta\ (p^*\ \overline{p_i^*}^n)$. From this we can conclude $\Gamma : p \overset{\beta}{\equiv}_{P,\mathbb{IE}}\ \Gamma^* : p^*$.

By induction hypothesis on H3, we have:

$$\Gamma^* : p^* \Downarrow_{AC} \Delta^* : w^* \quad \text{and} \quad \exists \beta'.\Delta[q \mapsto w] : q \overset{\beta'}{\equiv}_{P\mathbb{IE}}\ \Delta^* : w^*$$

Now there are two possibilities: $q \in dom\ (\beta'\ \Delta^*)^{P,(\beta'\ w^*)}_{live}$ or $q \notin dom\ (\beta'\ \Delta^*)^{P,(\beta'\ w^*)}_{live}$ (i.e. $\beta'^{-1}\ q$ is either live or dead in $\Delta^* : w^*$). We will prove the case where $q^* = \beta'^{-1}\ q$ is live in $\Delta^* : w^*$ and will indicate how to prove the other.

By $\Delta[q \mapsto w] : q \overset{\beta'}{\equiv}_{P\mathbb{IE}}\ \Delta^* : q^*$ and q^* live in $\Delta^* : w^*$, we have:

$$\Delta[q \mapsto w] : q\ \overline{p_i}^n \overset{\beta'}{\equiv}_{(P \cup \{q\})\mathbb{IE}}\ \Delta^* : q^*\ \overline{p_i^*}^n \quad \text{and} \quad \Delta^*[q^* \mapsto w^*]$$

Then, by induction hypothesis on H3, we get $\Delta^* : q^*\ \overline{p_i^*}^n \Downarrow_{AC} \Theta^* : w'^*$ and $\exists \beta''.\Theta[t \mapsto w'] : t \overset{\beta''}{\equiv}_{(P \cup \{q\})\mathbb{IE}}\ \Theta^* : w'^*$. So, $\Theta[t \mapsto w'] : t \overset{\beta''}{\equiv}_{P\mathbb{IE}}\ \Theta^* : w'^*$. Then, by Lemma 14 we finally get $\Gamma^* : e^* \Downarrow_{AC} \Theta^* : w'^*$.

The case where $\beta'^{-1}\ q$ is not live in $\Delta^* : w^*$ is slightly more complicated. First, we need to add a new closure to Δ^* pointing to the normal form w^* and then to extend β' in order to get a new equivalence $\overset{\beta''}{\equiv}$. The rest of the proof is very similar to the previous one. □

Corollary 16
Given a closed expression $e \in Fun$:

1. If $\{\} : e \Downarrow_{\{\}\{\}} \Delta^* : w^*$, then $\{\} : e \downarrow_{[]} \Delta[p \mapsto w] : p$, and $\exists \beta.\Delta : p \overset{\beta}{\equiv}_{\{\},\mathbb{IE}}\ \Delta^* : w^*$
2. If $\{\} : e \downarrow_{[]} \Delta[p \mapsto w] : p$, then $\{\} : e \Downarrow_{\{\}\{\}} \Delta^* : w^*$, and $\exists \beta.\Delta : p \overset{\beta}{\equiv}_{\{\},\mathbb{IE}}\ \Delta^* : w^*$

The above theorems allow us to transform *Basic* programs, with the usual Sestoft's semantics, into *Fun* ones with the new—and equivalent—$S3$ semantics. Once adapted the source language and its semantics to the 'STG world', we are ready to derive our first STG-like machines.

4 Deriving two abstract STG machines

During the last years, the STG machine has undergone some changes, the main one being the way it reduces applications. The initial evaluation strategy was a so-called *push/enter* model. It works as follows: in order to reduce the application, the machine first pushes the arguments onto the stack and then enters the closure where the λ-abstraction is. The λ-abstraction is responsible for analysing whether there are enough arguments or not in order to perform the β reduction. The principal

advantages of the model are:

- The machine enters the closure without further analysis or delay. This scheme is also followed for the rest of the closures (constructions, partial applications, and thunks). The closure code is responsible for what must be done upon entering, so closures need not be tagged. In fact, the surname 'tag-less' makes reference to this important feature of the machine.
- In certain cases the machine does not need to create intermediate partial applications.

The main drawback, as the authors explain in Marlow & Peyton Jones (2004), is the difficulty of navigating through the stack. While other stack objects like update frames or **case** continuations have a clear layout and size, easily known by the runtime system, pending arguments represent a complex issue as they must be dealt with one by one. In the several situations in which the stack must be walked through (e.g. garbage collection, stack migration, etc.), pending arguments much complicate the runtime system code or lead to an inefficient one.

Currently the STG machine implements the so-called *eval/apply* model. In this model, the responsibility for checking the correct number of arguments is delegated to the application code: if there are enough arguments, then the λ-abstraction closure is entered; otherwise, a partial application is created. In this model, the closure must be inspected in order the application site to know the exact number of arguments needed by the λ-abstraction. The code of the latter is liberated from the argument satisfaction check that is typical of the push-enter model. The main advantages of eval-apply are:

- When pending arguments must be left in the stack—for instance, when the function to be applied consists of an unevaluated thunk—they are packed in a *regular* stack frame, much in the same way as the update and the **case** continuation frames. Its layout is known by the whole runtime system and the navigation through the stack is much simpler than in the push-enter model.
- A lot of optimisations can be done in the application site when the number of arguments needed by the λ-closure is known at compile time.

The main disadvantages of this model are that in some programs it produces more partial applications closures than the push/enter model, and that it needs closures to be tagged.

At first sight, the eval-apply model looks simpler to implement but leading to a less efficient code. This was probably the reason why the authors decided to implement the push-enter model in the beginning. However, the actual tests published in Marlow & Peyton Jones (2004), and Marlow & Peyton Jones (2006) show that the eval/apply model is even a little more efficient than the push/enter model in the majority of the (large number of) programs tested. Given these results, the current STG machine implements eval-apply.

We consider that both models deserve to be formally derived from our *S*3 semantics. One reason is to show that a single semantics can lead to two different machines, and to different imperative code. The other reason is to highlight

	Heap	Control	Stack	rule
	Γ	**letrec** $\overline{x_i = be_i}$ **in** e	S	letrec ([1])
$\rightarrow$	$\Gamma \uplus [p_i \mapsto be_i[\overline{p_j/x_j}]]$	$e[\overline{p_i/x_i}]$	S	
	Γ	**case** e **of** $\overline{C_i\ \overline{y_{ij}} \rightarrow e_i}$	S	case1
$\rightarrow$	Γ	e	$\overline{C_i\ \overline{y_{ij}} \rightarrow e_i} : S$	
	$\Gamma[p \mapsto C_k\ \overline{p_i}]$	p	$\overline{C_i\ \overline{y_{ij}} \rightarrow e_i} : S$	case2
$\rightarrow$	Γ	$e_k[\overline{p_i/y_{ki}}]$	S	
	$\Gamma \uplus [p \mapsto e]$	p	S	var1 ([2])
$\rightarrow$	Γ	e	$\#p : S$	
	$\Gamma[q \mapsto w]$	q	$\#p : S$	var2
$\rightarrow$	$\Gamma \uplus [p \mapsto w]$	q	S	

([1]) $\overline{p_i}$ *fresh* w.r.t. Γ, **letrec** $\{\overline{x_i = be_i}\}$ **in** e, and S
([2]) $e \neq w$

Fig. 6. The STG-1 common machine.

the similarities and the differences of two successful, industrial-strength, Haskell implementations.

Following an approach similar to Sestoft's derivation of his MARK-1 machine (Sestoft, 1997), we first introduce a very simple STG-1 machine for each model—which we respectively call STG_{PE}-1, and STG_{EA}-1—in which explicit substitutions of pointers for program variables are done. The only difference between the push/enter and the eval/apply models is the way they deal with applications. So, it can be expected that both machines have a (rather large) common part and a specific part. We will present first the rules which are common to both models. These rules are shown in Figure 6.

A configuration in an STG-1 machine is a triple (Γ, e, S), where Γ represents the heap, e is the control expression and S is the stack. As in the $S3$ semantics, the heap Γ binds pointers to either expressions or normal forms which, in turn, may contain other pointers. The expression e is any *Fun* expression. The stack S stores for the moment two kinds of objects: **case** continuations, *alts* of pending pattern matchings, and marks $\#p$ of pending updates.

The common machine rules look very close to the lazy semantics $S3$ rules presented in Section 2. For instance, the machine rule for **letrec** in Figure 6 is a literal transcription of the $Letrec_{S3}$ rule of Figure 5. As there, freshness of variables is understood with respect to the current control expression, heap, and stack. The semantic rule for **case** is split into the two rules *case1* and *case2* in the machine. The semantic rule for a variable is also split into two in order to take care of updating the closure. So, in principle, the execution of the STG-1 machine on an initial expression e_0 could be regarded as the linearisation of the $S3$ semantics derivation tree for e_0, by using the stack as the source of pending work.

In the next two subsections, we will show the specific rules for the push/enter and the eval/apply models.

	Heap	Control	Stack	rule
	Γ	$p\ \overline{p_i}$	S	appPE1
$\rightarrow$	Γ	p	$\overline{p_i} : S$	
	$\Gamma[p \mapsto \lambda\overline{x_i}^n.e]$	p	$\overline{p_i}^n : S$	appPE2
$\rightarrow$	Γ	$e[\overline{p_i/x_i}^n]$	S	
	$\Gamma[p \mapsto \text{pap}(q\ \overline{q_i})]$	p	$\overline{p_i} : S$	appPE3
$\rightarrow$	Γ	q	$\overline{q_i} : \overline{p_i} : S$	
	$\Gamma[p \mapsto \lambda\overline{x_i}^n.\lambda\overline{y_i}.e]$	p	$\overline{p_i}^n : S$	appPE4 ([1])
$\rightarrow$	$\Gamma \uplus [q \mapsto \text{pap}(p\ \overline{p_i}^n)]$	q	S	

([1]) $\overline{p_i}^n$ is the whole set of pointers on top of the stack and q fresh w.r.t. Γ, p, and $\overline{p_i}^n : S$

Fig. 7. The specific part of machine STG_{PE}-1 (push/enter).

4.1 The push/enter machine

As we have previously mentioned, in the push/enter model the arguments of an application are pushed to the stack and the function closure is entered. This is reflected in rule *appPE1* of Figure 7. The function is responsible for testing whether there are enough arguments in the stack. If this is the case, the β reduction is performed (rule *appPE2*). Otherwise, a partial application is created (rule *appPE4*). If after applying rule *appPE4* the stack S is not empty, then a number of update marks will follow. Rule *var2* will perform the necessary updates with the partial application just created and will remove those marks. If the binding itself is a partial application and there is at least one pending argument in the stack, the partial application arguments are pushed on top of the pending arguments (rule *appPE3*), and the λ-abstraction the partial application comes from is entered. Then one of the rules *appPE2* or *appPE4* could be used. The net effect of this combination of rules is that, if the whole set of arguments for a λ-abstraction are separated by update marks, the machine will remove the marks trying to put all arguments together. In the mean time, it will create partial application bindings and use them for updating the marked bindings.

The stack may contain now a third kind of object: a pending argument p of a function under evaluation. Notice that each pending argument is in the stack by itself, not being part of a packet. A set of arguments can be pushed early in a computation and consumed much later and separately by different functions.

4.2 The eval/apply machine

As explained, the eval/apply machine pushes/pops arguments to/from the stack in packets of $n > 1$ arguments, denoted $\bullet\overline{p_i}^n$. So, this is a third kind of objects present in the STG_{EA}-1 stack. Notice that this object is different from n pending arguments $\overline{p_i}^n$ in the STG_{PE}-1 stack.

	Heap	Control	Stack	rule
	$\Gamma[p \mapsto e]$	$p\ \overline{p_i}^n$	S	appEA1 ([1])
$\rightarrow$	Γ	p	$\bullet\overline{p_i}^n : S$	
	$\Gamma[p \mapsto \lambda\overline{x_i}^n.e]$	$p\ \overline{p_i}^n\ \overline{q_i}^l$	S	appEA2
$\rightarrow$	Γ	$e[\overline{p_i/x_i}^n]$	$\bullet\overline{q_i}^l : S$	
	$\Gamma[p \mapsto \text{pap}(q\ \overline{q_i}^m)]$	$p\ \overline{p_i}^n$	S	appEA3
$\rightarrow$	Γ	$q\ \overline{q_i}^m\ \overline{p_i}^n$	S	
	$\Gamma[p \mapsto \lambda\overline{x_i}^n.\lambda\overline{y_i}^m.e]$	$p\ \overline{p_i}^n$	S	appEA4([2])
$\rightarrow$	$\Gamma \uplus [q \mapsto \text{pap}(p\ \overline{p_i}^n)]$	q	S	
	$\Gamma[p \mapsto w]$	p	$\bullet\overline{p_i}^n : S$	appEA5 ([3])
$\rightarrow$	Γ	$p\ \overline{p_i}^n$	S	

([1]) $e \neq w$
([2]) q fresh w.r.t. Γ, $p\ \overline{p_i}^n$, and S
([3]) $w \neq C\ \overline{p_i}^l$

Fig. 8. The specific part of machine STG$_{EA}$-1 (eval/apply).

The specific rules for this model are presented in Figure 8. Given an application in the control expression, five cases may happen when inspecting the functional binding:

1. It is an unevaluated thunk. In this case, a packet with the n arguments is pushed to the stack (rule *appEA1*) and the thunk is entered (by using rule *var1* of the common part).
2. It is a λ-abstraction and there are enough actual arguments. Then, a β reduction is performed (rule *appEA2*). The remaining actual arguments, if any, are pushed to the stack in a packet. It should be understood that $l = 0$ means that no packet is pushed.
3. It is a lambda and there are not enough actual arguments. Then, a partial application is created (rule *appEA4*).
4. It is a partial application. Then it is copied to the control expression and the actual arguments are appended to the partial application ones (rule *appEA3*).
5. The control is a variable pointing either to a lambda or to a partial application and there is a packet on top of the stack. Then, the packet is migrated to the control expression as arguments (rule *appEA5*).

4.3 Soundness and completeness

In this section we prove that every derivation which is possible in semantics $S3$ can be emulated by any of the two STG-1 machines and the other way around. To this aim, we first define an equivalence relation between $S3$ and machine configurations. We prove separately the push/enter and the eval/apply machines.

4.3.1 The push/enter machine

The first observation is that both the $S3$ semantics and the STG$_{PE}$-1 derivations generate partial application bindings in the heap but, due to rule App'_{S3}, the semantics generates more partial applications than the machine. For this very same reason, in some points of the derivation the semantics will have in the control expression a variable pointing to a partial application binding, while the machine will have instead a variable pointing to a λ-abstraction and there will be some pending arguments in the machine stack. In other points of the derivation the semantics will have in the control expression an application, while the machine will have instead a variable pointing to a λ-abstraction and there will be some pending arguments in the machine stack.

Fortunately, the excess partial application bindings in $S3$ heap have a short lifetime. They become dead bindings when rule $AppPap_{S3}$ is applied. So, we only need to take into account the live parts of the heaps and the α-renaming which make these equivalent. We include in the $S3$ configurations the stack S' associated to $\downarrow$ in the judgement.

Definition 17
Let be $\Gamma' : e' \downarrow_{S'}$ an S3 configuration, and $(\Gamma, e, \overline{q_i}^l : S)$ an STG$_{PE}$-1 configuration. We say that they are *equivalent*, denoted $\Gamma' : e' \downarrow_{S'} \equiv (\Gamma, e, \overline{q_i}^l : S)$, if there exists an α-renaming β such that,

- either $l = 0$, and $\Gamma : e \overset{\beta}{\equiv}_{fv\ S} \Gamma' : e'$, and $S' = \beta\ S$,
- or $l > 0$, e is a variable, $S' = \beta\ S$, and
  - — either e' is a variable, $\Gamma'\ e' = \text{pap}(p'\ \overline{p'_i}^l)$, and $\Gamma : e\ \overline{q_i}^l \overset{\beta}{\equiv}_{fv\ S} \Gamma' : p'\ \overline{p'_i}^l$,
  - — or $e' = q'\ \overline{q'_i}^l$ and $\Gamma : e\ \overline{q_i}^l \overset{\beta}{\equiv}_{fv\ S} \Gamma' : e'$.

Now, we establish a correspondence between a derivation subtree in $S3$ and a partial derivation in STG$_{PE}$-1. In $S3$ we notice that the stack associated to $\downarrow$ is the same in the initial and final configurations of any judgement. It may grow and shrink up in the tree, but when the initial expression reaches its normal form, the stack is again the initial one. Moreover, up in the tree it never shrinks below its initial value. The following definitions capture this idea.

Definition 18
Given a fixed stack S, we denote by $(\Gamma[p \mapsto w], p, p^* : S)$ one of the following STG$_{PE}$-1 configurations:

1. $(\Gamma[p \mapsto w], p, S)$, or
2. $(\Gamma[p \mapsto \lambda\overline{y_i}^n.\lambda\overline{x_i}.e], p, \overline{p_i}^n : S)$.

In both cases we say that $(\Gamma[p \mapsto w], p, p^* : S)$ is a *normal form of STG$_{PE}$-1 with respect to S*.

Definition 19
We define a *normal form derivation of STG$_{PE}$-1 with respect to S*, denoted $(\Gamma, e, S' +\!\!+ S) \rightarrow^*_S (\Gamma[p \mapsto w], p, p^* : S)$, as a derivation $(\Gamma, e, S' +\!\!+ S) \rightarrow^* (\Gamma[p \mapsto w], p, p^* : S)$ ending in a normal form with respect to S, in which the stack of all the intermediate configurations contains S as a suffix.

The following theorem establishes the soundness and correctness of STG$_{PE}$-1 with respect to $S3$.

Theorem 20
Let be $\Gamma' : e' \downarrow_{S'}$ an $S3$ configuration and $(\Gamma, e, \overline{z_i}^l : S)$ an STG$_{PE}$-1 configuration such that $\Gamma' : e' \downarrow_{S'} \equiv (\Gamma, e, \overline{z_i}^l : S)$. Then:

$$\Gamma' : e' \downarrow_{S'} \Delta'[p' \mapsto w'] : p' \text{ iff } (\Gamma, e, \overline{z_i}^l : S) \rightarrow_S^* (\Delta[p \mapsto w], p, t^* : S)$$
$$\text{and } \Delta'[p' \mapsto w'] : p' \downarrow_{S'} \equiv (\Delta[p \mapsto w], p, t^* : S)$$

Proof
$\Rightarrow$ By induction on the depth of the $\downarrow_{S'}$ derivation.
$\Leftarrow$ By induction on the number of steps of STG$_{PE}$-1.
As an example of the proof technique, we will show the proof of the case where the initial configuration has an application in the control expression and the functional variable is pointing to a partial application binding.

($S3 \Rightarrow$STG$_{PE}$-1) These are the hypothesis:

H1 $\Gamma'[p' \mapsto \text{pap}(q'\ \overline{q_i'}^m)] : p'\ \overline{p_i'}^n \downarrow_{S'} \equiv (\Gamma, e, \overline{z_i}^l : S)$
H2 $\Gamma'[p' \mapsto \text{pap}(q'\ \overline{q_i'}^m)] : p'\ \overline{p_i'}^n \downarrow_{S'} \Delta'[t' \mapsto w'] : t'$

The proof proceeds through the following steps:

P1 By H1 and by the $S3$ rule $AppPap_{S3}$ we know that:

$$\Gamma' : q'\ \overline{q_i'}^m\ \overline{p_i'}^n \downarrow_{S'} \Delta'[t' \mapsto w'] : t'$$

P2 By definition of $\equiv$, there exists two possibilities: (a) e is a variable, and $l = n$; and (b) e is an equivalent partial application via the α-renaming used in $\equiv$, and $l = 0$. We will consider the second case as the other one appears as a subset of it. In this case, we know that $\exists \beta.\ \Gamma : e \overset{\beta}{\equiv}_{fv\ S} \Gamma'[p' \mapsto \text{pap}(q'\ \overline{q_i'}^m)] : p'\ \overline{p_i'}^n$. Then:

$$\Gamma : e = \Gamma[p \mapsto \text{pap}(q\ \overline{q_i}^m)] : p\ \overline{p_i}^n, \quad p\ \overline{p_i}^n = \beta\ (p'\ \overline{p_i'}^n) \text{ and } q\ \overline{q_i}^m = \beta\ (q'\ \overline{q_i'}^m)$$

So, we can build the equivalent configurations $\Gamma : q\ \overline{q_i}^m\ \overline{p_i}^n \overset{\beta}{\equiv}_{fv\ S} \Gamma' : q'\ \overline{q_i'}^m\ \overline{p_i'}^n$

P3 By making evolve the STG$_{PE}$-1 machine, we have:

$$(\Gamma, q\ \overline{q_i}^m\ \overline{p_i}^n, S) \overset{\text{appPE1}}{\longrightarrow} (\Gamma, q, \overline{q_i}^m : \overline{p_i}^n : S)$$

Notice also that:

$$(\Gamma, p\ \overline{p_i}^n, S) \overset{\text{appPE1}}{\longrightarrow} (\Gamma[p \mapsto \text{pap}(q\ \overline{q_i}^m)], p, \overline{p_i}^n : S) \overset{\text{appPE3}}{\longrightarrow} (\Gamma, q, \overline{q_i}^m : \overline{p_i}^n : S)$$

P4 By P3 and by induction hypothesis on H1, we have:

$$(\Gamma, q, \overline{q_i}^m : \overline{p_i}^n : S) \rightarrow_S^* (\Delta, t, t^* : S \text{ and } \Delta'[t' \mapsto w'] : t' \downarrow_{S'} \equiv (\Delta, t, t^* : S)$$

P5 Finally, by P3 and P4, we get: $(\Gamma, p\ \overline{p_i}^n, S) \rightarrow_S^* (\Delta, t, t^* : S)$ $\square$

(STG$_{PE}$-1$\Rightarrow S3$) These are the hypothesis:

H1 $\Gamma' : e' \downarrow_{S'} \equiv (\Gamma[p \mapsto \text{pap}(q\ \overline{q_i}^m)], p\ \overline{p_i}^n, S)$

$$
\begin{array}{lll}
\textbf{H2} & (\Gamma, p\ \overline{p_i}^n, S) & \{appPE1\} \\
\rightarrow & (\Gamma[p \mapsto \mathrm{pap}(q\ \overline{q_i}^m)], p, \overline{p_i}^n : S) & \{appPE3\} \\
\rightarrow & (\Gamma, q, \overline{q_i}^m : \overline{p_i}^n : S) & \\
\rightarrow_S^* & (\Delta[t \mapsto w], t, t^* : S) &
\end{array}
$$

The proof proceeds through the following steps:

P1 By H1, $\exists\beta.\ \Gamma[p \mapsto \mathrm{pap}(q\ \overline{q_i}^m)] : p\ \overline{p_i}^n \overset{\beta}{\equiv}_{fv\ S} \Gamma' : e'$. Then:

$$\Gamma' : e' = \Gamma'[p' \mapsto \mathrm{pap}(q'\ \overline{q_i'}^m)] : p'\ \overline{p_i'}^n, \quad p\ \overline{p_i}^n = \beta\ (p'\ \overline{p_i'}^n) \text{ and } q\ \overline{q_i}^m = \beta\ (q'\ \overline{q_i'}^m)$$

So, we can build the equivalent configurations: $\Gamma : q\ \overline{q_i}^m\ \overline{p_i}^n \overset{\beta}{\equiv}_{fv\ S} \Gamma' : q'\ \overline{q_i'}^m\ \overline{p_i'}^n$. We notice that:

$$(\Gamma, q\ \overline{q_i}^m\ \overline{p_i}^n, S) \xrightarrow{\text{appPE1}} (\Gamma, q, \overline{q_i}^m : \overline{p_i}^n : S)$$

P2 Now, by induction hypothesis on H2, we have $\Gamma' : q'\ \overline{q_i'}^m\ \overline{p_i'}^n \downarrow_{S'} \Delta'[t' \mapsto w'] : t'$ and $\Delta'[t' \mapsto w'] : t' \equiv (\Delta[t \mapsto w], t, t^* : S)$.

P3 By applying rule $AppPap_{S3}$, we finally get: $\Gamma' : p'\ \overline{p_i'}^n \downarrow_{S'} \Delta'[t' \mapsto w'] : t'$ □

As a consequence of the theorem, we have the following corollary:

Corollary 21
Given a closed *Fun* expression e,

$$\{\} : e \downarrow_{[]} \Delta'[p' \mapsto w'] : p' \text{ iff } (\{\}, e, [\,]) \rightarrow^* (\Delta[p \mapsto w], p, [\,]) \text{ and } \Delta'[p' \mapsto w'] : p' \downarrow_{[]} \equiv (\Delta[p \mapsto w], p, [\,])$$

Let us observe that both the machine (rule appPE4 for partial applications) and the semantics end up with an empty stack.

4.3.2 The eval/apply machine

In this case, the equivalence between $S3$ and the machine is simpler. Both the machine STG_{EA}-1 and the semantics build the same number of heap bindings and they are of the same type. So, we do not even need to restrict ourselves to the live parts of the heaps: they should be identical, modulo an α-renaming.

Definition 22
Let be $\Gamma' : e' \downarrow_{S'}$ an $S3$ configuration, and (Γ, e, S) an STG_{EA}-1 configuration. We say that they are *equivalent*, denoted $\Gamma' : e' \downarrow_{S'} \equiv (\Gamma, e, S)$, if there exists an α-renaming β such that $\Gamma' : e' = \beta\ \Gamma : \beta\ e$ and $S' = \beta\ S$.

The definitions of normal form and normal form derivation with respect to a stack are simpler than Definitions 18 and 19 we used for the push/enter machine. The soundness and correctness theorem is very similar.

Definition 23
Given a fixed stack S, we say that an STG_{EA}-1 configuration of the form $(\Gamma[p \mapsto w], p, S)$ is in *normal form with respect to* S.

Definition 24
We define a *normal form derivation of STG$_{EA}$-1 with respect to S*, denoted $(\Gamma, e, S' +\!\!+ S) \rightarrow_S^* (\Gamma[p \mapsto w], p, S)$, as a derivation $(\Gamma, e, S' +\!\!+ S) \rightarrow^* (\Gamma[p \mapsto w], p, S)$ ending in a normal form with respect to S, in which the stack of all the intermediate configurations contains S as a suffix.

Theorem 25
Let be $\Gamma' : e' \downarrow_{S'}$ an $S3$ configuration and (Γ, e, S) an STG$_{EA}$-1 configuration such that $\Gamma' : e' \downarrow_{S'} \equiv (\Gamma, e, S)$. Then,

$$\Gamma' : e' \downarrow_{S'} \Delta'[p' \mapsto w'] : p' \text{ iff } (\Gamma, e, S) \rightarrow_S^* (\Delta[p \mapsto w], p, S)$$
$$\text{and } \Delta'[p' \mapsto w'] : p' \downarrow_{S'} \equiv (\Delta[p \mapsto w], p, S)$$

Proof
$\Rightarrow$ Trivial, by induction on the depth of the $\downarrow_{S'}$ derivation.
$\Leftarrow$ Trivial, by induction on the number of steps of STG$_{EA}$-1.
□

As a consequence, we have the corollary:

Corollary 26
Given a closed *Fun* expression e,

$$\{\} : e \downarrow_{[]} \Delta'[p' \mapsto w'] : p' \text{ iff } (\{\}, e, [\,]) \rightarrow^* (\Delta[p \mapsto w], p, [\,])$$
$$\text{and } \Delta'[p' \mapsto w'] : p' \downarrow_{[]} \equiv (\Delta[p \mapsto w], p, [\,])$$

Also in this case, the machine and the semantics end up with an empty stack.

4.4 Adding environments to the machines

Following Sestoft's derivation of his MARK-2 machine, in this section we introduce two STG-2 machines, respectively called STG$_{PE}$-2 and STG$_{EA}$-2, having runtime environments instead of doing explicit substitution of pointers for variables. Now, expressions and normal forms keep their original *Fun* variables, and the associated environments E maps them to pointers p pointing to heap bindings. Environments are associated to the control expression, to pending alternatives living in the stack, and to expressions stored in heap bindings, which from now on will be called *closures*.

In principle, environments E are finite functions mapping all variables in scope in an expression to their corresponding heap pointers. However, most of variables in scope are not frequently free in the expression. In order to minimise the amount of information stored in the heap and in the stack, and also to save copying time, the environments living there will be *trimmed* to the corresponding set of free variables. A trimmer t is just a collection of variable names. The notation $E\,|^t$ expresses the trimming of environment E to the trimmer t. We include trimmers in our STG-2 machines because they are used in the actual STG implementation and also because they will have some impact in the imperative translation provided in Section 5.4.

A configuration of an STG-2 machine is a quadruple (Γ, e, E, S), where E is the environment of e. Now the alternatives in the stack are pairs $(alts, E)$, and the heap bindings are of the form $[p \mapsto (e, E)]$ or $[p \mapsto (w, E)]$. A pair (e, E) is a thunk

	Heap	Control	Environment	Stack	rule
	Γ	**letrec** $\overline{x_i = be_i}$ **in** e	E	S	letrec ([1])
$\rightarrow$	$\Gamma \uplus [\overline{p_i \mapsto (be_i, E' \mid^{t_i})}]$	e	E'	S	
	Γ	**case** e **of** $\overline{C_i\ \overline{y_{ij}} \rightarrow e_i}$	E	S	case1 ([2])
$\rightarrow$	Γ	e	E	$(\overline{C_i\ \overline{y_{ij}} \rightarrow e_i}, E \mid^t) : S$	
	$\Gamma[p \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto p\}$	$(\overline{C_i\ \overline{y_{ij}} \rightarrow e_i}, E') : S$	case2
$\rightarrow$	Γ	e_k	$E' \uplus \{\overline{y_{ki} \mapsto p_i}\}$	S	
	$\Gamma \uplus [p \mapsto (e, E')]$	x	$E\{x \mapsto p\}$	S	var1 ([3])
$\rightarrow$	Γ	e	E'	$\#p : S$	
	$\Gamma[q \mapsto (w, E')]$	x	$E\{x \mapsto q\}$	$\#p : S$	var2
$\rightarrow$	$\Gamma \uplus [p \mapsto (w, E')]$	x	E	S	

([1]) $t_i = fv\ be_i$, $E' = E \uplus \{\overline{x_i \mapsto p_i}\}$, and $\overline{p_i}$ fresh w.r.t. Γ, **letrec** $\overline{x_i = be_i}$ **in** e, and S
([2]) $t = fv\ (\overline{C_i\ \overline{y_{ij}} \rightarrow e_i})$
([3]) $e \neq w$

Fig. 9. The STG-2 common machine with environments.

closure and a pair (w, E) is a normal form closure. The common part of our two STG-2 machines is shown in Figure 9. By $E\{x \mapsto p\}$ we just highlight that the mapping $x \mapsto p$ belongs to E. Notice that trimming is used in the evaluation of **case** expressions in order to reduce the environment stored in the stack, and in **letrec** expressions for reducing the environment stored in the heap closures.

4.4.1 The push/enter machine with environments

The specific rules for the push/enter machine STG_{PE}-2 are shown in Figure 10. Since it will have an important impact in how the translation to imperative code will be done in Section 5.4, we highlight how is the environments 'life' in machine STG_{PE}-2:

1. A new environment is installed in the control expression whenever a closure is entered. This happens in rules *case2*, *var1*, *appPE2*, *appPE3*, and *appPE4*.
2. In these rules, the previous control environment dies.
3. Environments are duplicated in rule *case1*, as the current control environment must be (trimmed and) copied to the stack.
4. Environments are extended in rules *letrec*, and *case2*.
5. Control environments remain unmodified in rules *case1*, *var2*, and *appPE1*.

4.4.2 The eval/apply machine with environments

The specific rules of the eval/apply STG_{EA}-2 machine are shown in Figure 11. Similarly to machine STG_{PE}-2, new environments are installed in rules *appEA2*, *appEA3*, and *appEA4*, and the environment is preserved in rule *appEA1*. In rule *appEA5*, the environment E is extended with new bindings for the arguments of the application.

	Heap	Control	Environment	Stack	rule
	Γ	$x\ \overline{x_i}$	$E\{x \mapsto p, \overline{x_i \mapsto p_i}\}$	S	appPE1
$\rightarrow$	Γ	x	E	$\overline{p_i} : S$	
	$\Gamma[p \mapsto (\lambda \overline{x_i}^n.e, E')]$	x	$E\{x \mapsto p\}$	$\overline{p_i}^n : S$	appPE2
$\rightarrow$	Γ	e	$E' \uplus \{\overline{x_i \mapsto p_i}^n\}$	S	
	$\Gamma[p \mapsto (\text{pap}(y\ \overline{y_i}), \{y \mapsto q, \overline{y_i \mapsto q_i}\})]$	x	$E\{x \mapsto p\}$	$\overline{p_i} : S$	appPE3(^{1})
$\rightarrow$	Γ	z	$\{z \mapsto q\}$	$\overline{q_i} : \overline{p_i} : S$	
	$\Gamma[p(\mapsto \lambda \overline{x_i}^n.\lambda \overline{y_i}.e, E')]$	x	$E\{x \mapsto p\}$	$\overline{p_i}^n : S$	appPE4 (^{2})
$\rightarrow$	$\Gamma \uplus [q \mapsto (\text{pap}(y\ \overline{y_i}^n), \{y \mapsto p, \overline{y_i \mapsto p_i}^n\})]$	z	$\{z \mapsto q\}$	S	

(^{1}) $n > 0$ and z fresh w.r.t. Γ, x, and $\overline{p_i}^n : S$
(^{2}) $\overline{p_i}^n$ is the whole set of pointers on top of stack S and z, q fresh w.r.t. Γ, x, and $\overline{p_i}^n : S$

Fig. 10. The specific part of machine STG_{PE}-2 (push/enter) with environments.

	Heap	Control	Environment	Stack	rule
	$\Gamma[p \mapsto (e, E')]$	$x\ \overline{x_i}^n$	$E\{x \mapsto p, \overline{x_i \mapsto p_i}^n\}$	S	appEA1
$\rightarrow$	Γ	x	E	$\bullet\overline{p_i}^n : S$	
	$\Gamma[p \mapsto (\lambda \overline{x_i}^n.e, E')]$	$x\ \overline{x_i}^n\ \overline{y_i}^l$	$E\left\{\begin{matrix} x \mapsto p, \overline{x_i \mapsto p_i}^n \\ , \overline{y_i \mapsto q_i}^l \end{matrix}\right\}$	S	appEA2
$\rightarrow$	Γ	e	$E' \uplus \{\overline{x_i \mapsto p_i}^n\}$	$\bullet\overline{q_i}^l : S$	
	$\Gamma[p \mapsto (\text{pap}(y\ \overline{y_i}^m), E')]$	$x\ \overline{x_i}^n$	$E\{x \mapsto p, \overline{x_i \mapsto p_i}^n\}$	S	appEA3(^{1})
$\rightarrow$	Γ	$y\ \overline{y_i}^m\ \overline{x_i}^n$	$E' \uplus \{\overline{x_i \mapsto p_i}^n\}$	S	
	$\Gamma[p \mapsto (\lambda \overline{x_i}^n.\lambda \overline{y_i}.e, E')]$	$x\ \overline{x_i}^n$	$E\{x \mapsto p, \overline{x_i \mapsto p_i}^n\}$	S	appEA4(^{2})
$\rightarrow$	$\Gamma \uplus [q \mapsto (\text{pap}(x\ \overline{x_i}^n), E'')]$	y	$\{y \mapsto q\}$	S	
	$\Gamma[p \mapsto (w, E')]$	x	$E\{x \mapsto p\}$	$\bullet\overline{p_i}^n : S$	appEA5(^{3})
$\rightarrow$	Γ	$x\ \overline{x_i}^n$	$E \uplus \{\overline{x_i \mapsto p_i}^n\}$	S	

(^{1}) $E' = \{y \mapsto p, \overline{y_i \mapsto p_i}^n\}$
(^{2}) $E'' = \{x \mapsto p, \overline{x_i \mapsto p_i}^n\}$, y, q fresh w.r.t. Γ, *dom* E, and S
(^{3}) $\overline{x_i}^n$ fresh w.r.t. Γ, *dom* E, and $\bullet\overline{p_i}^n : S$

Fig. 11. The STG-2 specific machine with environments (eval/apply).

4.4.3 Soundness and completeness of the STG-2 machines

Adding runtime environments to the STG-1 machines is a transformation aiming to improve their efficiency, and at the same time to preserve their correctness. In fact, environments can be just seen as delayed substitutions of pointers for variables. So, we will be more informal in this section, just providing the essential ideas of machine equivalence and the formulation of the main correctness theorem.

The difference between an STG-1 configuration (Γ, e, S) and an STG-2 equivalent one (Γ', e', E, S') is that the second includes environments associated to the control expression e', to the heap expressions in Γ' and to the alternatives in stack S'. It can be expected that, if we replaced all expressions e in Γ', e' and S' by explicitly

substituted expressions $E\ e$, being E the environment associated to e, we would arrive to equivalent configurations, modulo and appropriate α-renaming β. This idea can be easily formalised and we can arrive to the definition of an equivalence relation between an STG-1 configuration and an STG-2 one. We will denote this equivalence by $(\Gamma, e, S) \equiv (\Gamma', e', E, S')$. Then, the soundness and correctness theorem for each pair of corresponding STG-1 and STG-2 machines looks as follows:

Theorem 27
Let (Γ, e, S) be an STG-1 configuration and (Γ', e', E, S') be an STG-2 configuration such that $(\Gamma, e, S) \equiv (\Gamma', e', E, S')$. Then,

$$(\Gamma, e, S) \longrightarrow (\Gamma_1, e_1, S_1) \text{ iff } (\Gamma', e', E, S') \longrightarrow (\Gamma'_1, e'_1, E_1, S'_1) \text{ and } (\Gamma_1, e_1, S_1) \equiv (\Gamma'_1, e'_1, E_1, S'_1)$$

Proof
By cases on the respective machine rules. □

5 Deriving two imperative STG machines

In this Section we introduce two imperative, STG-like, virtual machines, one for each evaluation model, respectively called ISTG_{PE} for push/enter, and ISTG_{EA} for eval/apply. We define the machines using a small step operational semantics, by formally describing for each machine instruction the state transition produced by it.

These machines try to provide an intermediate level of reasoning between the STG-2 machine and the final C implementation. In the actual GHC implementation, 'below' the operational description of Peyton Jones (1992), and Marlow & Peyton Jones (2004) the only formal description we find is the compiler code producing a direct translation to C. By looking at the compiler and at the runtime system listings, one can grasp some details, but many others are lost. We think that the distance to be saved is too high. Moreover, it is not possible to reason about the correctness of the implementation when so many details are introduced at once.

An ISTG machine configuration consists of a 5-tuple $(is, S, node, \Gamma, cs)$, where the syntax and meaning of each component is as follows:

- *is* is a machine instruction sequence implementing the control expression of the STG-2 machines. It can be thought of as a list of instructions from one of the sets shown in figures 12 and 13. By $i : is$ we highlight that i is the first instruction to be executed and *is* is the rest. As we will see, there are no branch instructions in a sequence *is*, except at the end.
- S is the stack, which is a list of the following types of objects :

a	pointer to a heap closure
p	pointer to a vector $\overline{is_i}^n$ of n code sequences stored in cs
$\#a$	update mark with a pointer to the closure a to be updated
$\bullet\overline{a_i}^m$	packet of arguments (ISTG_{EA} machine only)

- *node* is a heap pointer pointing to the closure under execution, which is the one that *is* belongs to.
- Γ is the machine heap, binding pointers a to closures. We explain closure syntax below.

	Instructions	Stack	Node	Heap	Code	
heap						
	$ALLOC\ l : is$	S	$node$	Γ	cs	([1])
$\Longrightarrow$	is	$a_l : S$	$node$	Γ'	cs	
	$BUILDCLS\ i\ p\ \overline{z_i}^l : is$	S	$node$	Γ	$cs[p \mapsto is]$	([2])
$\Longrightarrow$	is	S	$node$	$\Gamma[S!i \mapsto (p, \overline{a_i}^l)]$	cs	
stack						
	$BUILDENV\ \overline{z_i}^n : is$	S	$node$	Γ	cs	([2])
$\Longrightarrow$	is	$\overline{a_i}^n : S$	$node$	Γ	cs	
	$PUSHALTS\ p : is$	S	$node$	Γ	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow$	is	$p : S$	$node$	Γ	cs	
	$UPDTMARK : is$	S	$node$	$\Gamma[node \mapsto (p, ws)]$	cs	
$\Longrightarrow$	is	$\#node : S$	$node$	$\Gamma[node \mapsto (p_{bh}, ws)]$	cs	
	$SLIDE\ n\ m : is$	$\overline{a_i}^n : \overline{b_j}^m : S$	$node$	Γ	cs	
$\Longrightarrow$	is	$\overline{a_i}^n : S$	$node$	Γ	cs	
control						
	$[RETURNCON\ C_k^l]$	$p : S$	$node$	Γ	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow$	is_k	S	$node$	Γ	cs	
	$[RETURNCON\ C_k^l]$	$\#a : S$	$node$	$\Gamma[a \mapsto (p_{bh}, as),$ $node \mapsto (p, ws)]$	cs	
$\Longrightarrow$	$[RETURNCON\ C_k^l]$	S	$node$	$\Gamma[a \mapsto (p, ws)]$	cs	
	$[ENTER]$	$a : S$	$node$	$\Gamma[a \mapsto (p, ws)]$	$cs[p \mapsto is]$	
$\Longrightarrow$	is	S	a	Γ	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^m : S$	$node$	Γ	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	$node$	Γ	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : [\,]$	$node$	Γ	cs	$n < m$
$\Longrightarrow$	$ARGCHECK\ m : is$	$[\,]$	a	$\Gamma \uplus [a \mapsto (p_{pap}^{n+1}, node : \overline{a_i}^n)]$	cs	$fresh\ a$
	$ARGCHECK\ m : is$	$\#b : S$	$node$	$\Gamma[b \mapsto (p_{bh}, as),$ $node \mapsto (p, ws)]$	cs	
$\Longrightarrow$	$ARGCHECK\ m : is$	S	$node$	$\Gamma[b \mapsto (p, ws)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#b : S$	$node$	$\Gamma[b \mapsto (p_{bh}, ws)]$	cs	$n < m$
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	$node$	$\Gamma[b \mapsto (p_{pap}^{n+1}, node : \overline{a_i}^n)]$ $\uplus [a \mapsto (p_{pap}^{n+1}, node : \overline{a_i}^n)]$	cs	$fresh\ a$

([1]) a_l is a pointer to a new closure with space for l free variables, and Γ' is the resulting heap after the allocation

([2]) $a_i = \begin{cases} S!j & \text{if } z_i = (stack, j) \\ node!j & \text{if } z_i = (node, j) \end{cases}$

Fig. 12. The push/enter ISTG machine.

- cs is the code store. It can be thought of as a finite mapping from code pointers p, either to instruction sequences such as is, or to vectors $\overline{is_i}^n$ of code sequences. The code store is the result of compiling *Fun* programs to machine code. This translation is presented is Section 5.4.

Instructions	Stack	Node	Heap	Code	
heap					
$ALLOC\ l : is$	S	$node$	Γ	cs	(1)
$\Longrightarrow is$	$a_l : S$	$node$	Γ'	cs	
$BUILDCLS\ t\ i\ p\ \overline{z_i}^l : is$	S	$node$	Γ	$cs[p \mapsto is]$	(2)
$\Longrightarrow is$	S	$node$	$\Gamma[S\,!\,i \mapsto t : (p, \overline{a_i}^l)]$	cs	
stack					
$BUILDENV\ \overline{z_i}^n : is$	S	$node$	Γ	cs	(2)
$\Longrightarrow is$	$\overline{a_i}^n : S$	$node$	Γ	cs	
$PUSHALTS\ p : is$	S	$node$	Γ	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow is$	$p : S$	$node$	Γ	cs	
$UPDTMARK : is$	S	$node$	$\Gamma[node \mapsto \texttt{THUNK}\ (p, ws)]$	cs	
$\Longrightarrow is$	$\#node : S$	$node$	$\Gamma[node \mapsto \texttt{THUNK}\ (p_{bh}, ws)]$	cs	
$SLIDE\ n\ m : is$	$\overline{a_i}^n : \overline{b_j}^m : S$	$node$	Γ	cs	
$\Longrightarrow is$	$\overline{a_i}^n : S$	$node$	Γ	cs	
control					
$[RETURNCON\ C_k^l]$	$p : S$	$node$	Γ	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow is_k$	S	$node$	Γ	cs	
$[RETURNCON\ C_k^l]$	$\#a : S$	$node$	$\Gamma[a \mapsto \texttt{THUNK}\ (p_{bh}, as),$ $node \mapsto \texttt{CONS}\ (p, ws)]$	cs	
$\Longrightarrow [RETURNCON\ C_k^l]$	S	$node$	$\Gamma[a \mapsto \texttt{CONS}\ (p, ws)]$	cs	
$[EVAL\ m]$	$a : \overline{a_i}^m : S$	$node$	$\Gamma[a \mapsto \texttt{FUN}\ (m, p, ws)]$	$cs[p \mapsto is]$	
$\Longrightarrow is$	$\overline{a_i}^m : S$	a	Γ	cs	
$[EVAL\ m]$	$a : \overline{a_i}^m : S$	$node$	$\Gamma[a \mapsto \texttt{FUN}\ (n, p, ws)]$	$cs[p \mapsto is]$	$m > n$
$\Longrightarrow is$	$\overline{a_i}^n : \bullet\{a_{n+1} \ldots a_{mj}\} : S$	a	Γ	cs	
$[EVAL\ m]$	$b : \overline{a_i}^m : S$	$node$	$\Gamma[b \mapsto \texttt{THUNK}\ (p, ws)]$	$cs[p \mapsto is]$	
$\Longrightarrow is$	$\bullet\overline{a_i}^m : S$	b	Γ	cs	
$[EVAL\ m]$	$a : \overline{a_i}^m : S$	$node$	$\Gamma[a \mapsto \texttt{FUN}\ (n, p, ws)]$	cs	$m < n$
$\Longrightarrow [EVAL\ 0]$	$b : S$	$node$	$\Gamma[b \mapsto \texttt{PAP}\ (p_{pap}^{m+1}, a : \overline{a_i}^m)]$	cs	fresh b
$[EVAL\ m]$	$b : \overline{a_i}^m : S$	$node$	$\Gamma[b \mapsto \texttt{PAP}\ (a\ \overline{b_i}^n)]$	cs	
$\Longrightarrow [EVAL\ (m+n)]$	$b : \overline{b_i}^n : \overline{a_i}^m : S$	$node$	Γ	cs	
$[EVAL\ 0]$	$b : \bullet\,\overline{a_i}^m : S$	$node$	$\Gamma[b \mapsto \texttt{FUN}\ /\texttt{PAP}\ ]$	cs	
$\Longrightarrow [EVAL\ m]$	$a : \overline{a_i}^m : S$	$node$	Γ	cs	
$[EVAL\ 0]$	$b : \#a : S$	$node$	$\Gamma[b \mapsto \texttt{FUN}\ /\texttt{PAP}\ ]$	cs	
$\Longrightarrow [EVAL\ 0]$	$b : S$	$node$	$\Gamma[a \mapsto \texttt{FUN}\ /\texttt{PAP}\ ]$	cs	
$[EVAL\ 0]$	$b : S$	$node$	$\Gamma[b \mapsto \texttt{THUNK}\ (p, ws)]$	$cs[p \mapsto is]$	
$\Longrightarrow is$	S	b	Γ	cs	
$[EVAL\ 0]$	$b : S$	$node$	$\Gamma[b \mapsto \texttt{CONS}\ (p, ws)]$	$cs[p \mapsto is]$	
$\Longrightarrow is$	S	b	Γ	cs	

(1) a_l is a pointer to a new closure with space for l free variables, and Γ' is the resulting heap after the allocation

(2) $a_i = \begin{cases} S\,!\,j & \text{if } z_i = (stack, j) \\ node\,!\,j & \text{if } z_i = (node, j) \end{cases}$

Fig. 13. The eval/apply ISTG machine.

We use the following notation: a, b for pointers to closures in Γ, as, bs and ws for lists of such pointers, and p for pointers to code fragments in cs. By $cs[p \mapsto is]$ we denote that cs maps p to the instruction sequence is and, by $cs[p \mapsto \overline{is_i}^n]$, that cs maps p to a vector of instruction sequences $is_1, \ldots, is_n$, each one corresponding to an alternative of a **case** expression with n constructors $C_1, \ldots, C_n$. We assume that each **case** has a complete set of alternatives and that the constructors appear in the order of their declaration in the algebraic datatype definition. By C_k^l we denote a data constructor C with arity l which appears the k-th in its **data** declaration.

Also, $S\,!\,i$ will denote the i-th element of the stack S, counting from the top and starting at 0. Likewise, $node_\Gamma\,!i$ will denote the i-th free variable of the closure pointed to by $node$ in Γ, this time starting at 1. A closure is at least a pair (p, ws) where p is a pointer to an instruction sequence is in cs, and ws is the closure environment, having a heap pointer for every free variable in the expression whose translation is is. In the eval/apply model, closures may contain additional information. We explain the specific details of each model below.

5.1 The push/enter imperative machine ISTG$_{PE}$

In this machine, closures need not be tagged. In the actual GHC implementation they can be considered to be tagged because the compiler adds to each one the so-called 'info-table' in which a lot of information about the type and layout of the closure is kept. This is needed for garbage collection and other side activities, but not for evaluation. So, we consider closures in this machine as having the general form $(p, \overline{a_i}^l)$ where p is a code pointer to the closure code, and $\overline{a_i}^l, l \geqslant 0$, is its runtime environment.

In Figure 12, the syntactic and operational description of the ISTG$_{PE}$ machine instructions its shown. The **heap** and **stack** groups, and the *RETURNCON* instruction in the **control** group are in common with the instruction set of the ISTG$_{EA}$ machine, except for the fact that closures are slightly different in this latter machine. We will explain their meaning only in this section.

Instructions *ALLOC* and *BUILDCLS* implement heap closure creation in the *letrec* rule of STG-2. Both *BUILDENV* and *BUILDCLS* make use of a list of pairs, each pair indicating whether the source variable is located in the stack or in the current closure. Of course, it is not intended this test to be done at runtime. An efficient translation of these 'machine' instructions to an imperative language will generate the appropriate copy statement for each pair.

Instructions *PUSHALTS* and *UPDTMARK* roughly correspond to the two possible pushing actions of the common machine STG-2. Instruction *BUILDENV* is used in the ISTG$_{PE}$ machine with two purposes: on the one hand, to push environment fragments to the stack; this use is in common with the ISTG$_{EA}$ machine; on the other hand, to push arguments of pending applications to the stack; this use is specific of the ISTG$_{PE}$ machine. The *SLIDE* instruction has no clear correspondence in the STG-2 machine. As we will see in Section 5.4, it will be used to delete fragments of the current environment when a new closure is entered.

Instruction *RETURNCON* is common to both ISTG machines. It implements pattern matching and updates with constructions normal forms.

Instructions *ENTER* and *ARGCHECK* are specific of the ISTG$_{PE}$ machine. The first one is typical of the old implementation of the STG machine as described in Peyton Jones (1992), and implements a jump to a new closure. Instruction *ARGCHECK*, which implements updates with partial applications, is here at the same level of abstraction as the rest of instructions. It does not appear in Peyton Jones (1992) but this action can be recognised when looking at the compiler code. The last rule of *ARGCHECK* implements the sequence of rules *appPE4, var2* and *appPE3* of machine STG$_{PE}$-2, avoiding to pop and push again arguments in the stack. It creates two partial application closures, as STG$_{PE}$-2 does. In reality, only the second one is created, as updates are always done by indirection in GHC.

It is interesting to note that it has been possible to describe the STG-2 machine without any reference to either *RETURNCON* or *ENTER*, while these instructions were rather central to the STG machine, as described in Peyton Jones (1992). In our view, these instructions belong to ISTG, i.e. to a lower level of abstraction than STG-2. Also, we make note that the field *node* is only modified by *ENTER*, when jumping into a new closure, but it never comes back to a previous value. This indicates that ISTG is a 'jumping' machine which does not follow the call/return scheme of virtual machines for imperative languages. It can also be regarded as a continuation passing style machine where the next action to be performed is always in the stack.

We are assuming in both ISTG machines that there is always enough space to update closures in place. As we have said, in the actual implementation updates are always made by indirections and all update-able closures have enough space for storing an indirection.

Predefined code is stored in *cs* for partial applications with *m* arguments, for a finite set of values of *m*, and for a black-hole. The latter is a transient closure used to temporarily update a closure under evaluation. If a black-hole is entered, this means infinite recursion and the machine should stop. The corresponding code pointers are respectively called p_{pap}^{m+1} and p_{bh} in Figures 12 and 13. This trivial code is the following:

$$cs[p_{bh} \mapsto [\,]]$$
$$cs[p_{pap}^{m+1} \mapsto [BUILDENV\ \overline{(node, i)}^{m+1}, ENTER]].$$

5.2 The eval/apply imperative machine ISTG$_{EA}$

In this machine closures are tagged. As we have explained, actual closures in GHC are also tagged, but tags are not needed for evaluation in the push/enter model. Here, they are needed. Closures syntax is as follows:

$$\begin{array}{lcll} closure & \rightarrow & \texttt{FUN}\,(n, p, ws) & \text{-- } \lambda\text{-abstraction closure} \\ & & \texttt{CONS}\,(p, ws) & \text{-- construction closure} \\ & & \texttt{PAP}\,(p_{pap}, ws) & \text{-- partial application closure} \\ & & \texttt{THUNK}\,(p, ws) & \text{-- unevaluated expression closure} \end{array}$$

In closure `FUN`, the field *n* represents the number of formal arguments of the λ-abstraction. In all closure types, *p* or p_{pap} represent the code pointer and *ws* the runtime environment.

In Figure 13, the syntactic and operational description of the ISTG_{EA} machine instructions its shown. As we said, the rules in the groups **heap** and **stack**, and the *RETURNCON* instruction in the group **control** are almost identical to the corresponding ones in the ISTG_{PE} machine. The only differences are, on the one hand that we make explicit in the rules the closure tag, and on the other the addition of an argument t to instruction *BUILDCLS* for indicating the tag of the closure being created. Also, the predefined code for a partial application closure with m arguments is now:

$$cs[p_{pap}^{m+1} \mapsto [BUILDENV\ \overline{(node, i)}^{m+1}, EVAL\ m]].$$

The *ENTER* instruction of ISTG_{PE} is not needed here. That represented an unconditional jump to the closure code, because that code was responsible for whatever action was necessary. Now, the situation is different: a functional closure must not be entered, unless the machine can ensure that there are enough arguments in the stack. For the same reason, the ISTG_{PE} *ARGCHECK* instruction is not useful now. To replace both, the ISTG_{EA} machine provides the *EVAL* instruction. It has a natural number m as argument whose meaning is the number of unpacked arguments which are waiting in the stack. The *EVAL* instruction can be seen as a kind of loop that takes all the actions needed in order to ensure that the closure code will be able to successfully process the information in the stack. That loop exits in the following three cases:

- When a `FUN` closure with n formal arguments must be entered and there are exactly n unpacked arguments waiting in the stack.
- When a `THUNK` closure must be entered.
- When a `CONS` closure must be entered.

The *EVAL* instruction is used to implement all the specific rules of the STG_{EA}-2 machine depicted in Figure 11. These rules correspond to the first six cases of *EVAL* shown in Figure 13. More precisely, cases 1st and 2nd implement the *appEA2* rule; case 3rd, the *appEA1* rule; case 4th, *appEA4*; case 5th, *appEA3*; and case 6th, *appEA5*. The last three cases respectively correspond to the rules *var2, var1*, and *case2* of the common STG-2 machine of Figure 9. Notice also the packing and unpacking of arguments in the stack. The property we want to preserve is the following: when arguments are unpacked, they are part of the runtime environment of the closure being entered; when they are packed, they constitute a stack frame that will be processed as a whole by a future closure still not created.

5.3 Translation to C

After having presented the two imperative machines ISTG_{PE} and ISTG_{EA}, we would like to add a word about the implementation of these machines in a conventional imperative language such as C. Now, we believe that the distance to be saved between the machine instructions and the implementation language is small enough so that no further proofs of correctness would be needed. Rather than providing a procedure for each machine instruction, the idea would be to implement them as macros which would be expanded into small fragments of C code. If the stack and the heap were represented as C arrays, the expected code would be as follows:

- *ALLOC* would just be a simple manipulation of the heap pointer.

- *BUILDENV* and *BUILDCLS* will essentially produce a sequence of C assignments, each one copying a word from the stack—if the source pointer belongs to the stack environment—, or from the heap—if it belongs to the current closure environment—, to the stack in the case of *BUILDENV*, or to the heap in the case of *BUILDCLS*.
- *PUSHALTS* and *UPDTMARK* would be simple C assignments copying pieces of information to the stack.
- *SLIDE* would be a small C loop moving information from one part of the stack to another.
- *RETURNCON*, *ARGCHECK*, and *EVAL* are more complex instructions as they take care of updates, pattern matching, and of the generation of partial application closures. Even though, each one could be implemented by a single C loop.
- Finally, *ENTER* would essentially be a jump within the code area.

The translation to C offers some opportunities for code optimisations. For instance, an instruction *SLIDE n* 0, or a *BUILDENV* instruction with an empty list should generate no code at all.

5.4 Formal translation from STG-2 to ISTG

In this Section, we provide the translation schemes from *Fun* programs to ISTG machine code. As there are many details which are similar in both models, we introduce first some common design decisions and the common translation schemes, and then the specific translation schemes for each model.

5.4.1 Design decisions

The main decisions influencing the translation are the following three:

1. The ISTG stack will contain not only the STG-2 stack, but also part of the environment E of the control expression.
2. The rest of the environment E will be kept in the closure under evaluation (the one pointed to by the *node* register). The translation knows where each free variable is located by maintaining two compile-time environments ρ and η. The first one ρ corresponds to the part of the environment kept in the stack, while the second one η corresponds to the free variables accessed through the *node* register.
3. The duplication of environments in rule *case1* of the STG-2 machine (see Figure 9) will be implemented by making the alternatives *to reuse* the control expression environment. So, that part of the environment must not be removed from the stack when leaving the current closure.

By observing *Fun* expressions evaluation in both machines and their effect in the current environment, as explained in Sections 4.4.1 and 4.4.2, we see that:

1. Expressions **letrec** and **case** either extend or preserve the current environment and proceed with the evaluation of a subexpression.

2. Expressions $x\ \overline{x_i}^n$ and x remove the current environment and install a new one. At the same time they jump to a different expression, the one in the closure being entered.

This leads us to conjecture the following invariant that will be formally proved below: The stack can be considered as divided into two big blocks; the upper block, which is described by ρ and contains (part of) the environment of the current expression, and the lower part, which contains other things such as update frames, pending arguments and **case** alternatives with their own environment. The upper part is divided into a possibly empty top environment and a sequence of **case** alternatives, each one with its own environment. The following regular expression syntax formalises this invariant for any stack S:

$$
\begin{array}{lcll}
S & \rightarrow & upper\ lower & \\
upper & \rightarrow & E\ alt^* & \\
lower & \rightarrow & (update \mid args \mid alt)^* & \\
alt & \rightarrow & p\ E & \\
args & \rightarrow & \overline{a_i} & \text{-- push/enter only} \\
 & \mid & \bullet\overline{a_i} & \text{-- eval/apply only} \\
update & \rightarrow & \#a & \\
E & \rightarrow & \overline{a_i} &
\end{array}
$$

Notice than an environment E and a set $\overline{a_i}$ of arguments in the push/enter machine are indistinguishable. Both consist of a sequence of heap pointers. In fact, we will exploit this equivalence in the translation by making that a set of arguments pushed to the stack become part of the environment of the λ-abstraction being entered. This is the most efficient way of implementing rule *appPE2* of Figure 10.

When an expression $x\ \overline{x_i}^n$, with $n \geqslant 0$, is evaluated, the *ENTER* instruction—in the ISTG_{PE} machine—or the *EVAL* instruction —in ISTG_{EA}—are executed. Then the top environment E of the upper block of the stack must be removed, but the environments associated to **case** alternatives must be left alone. This stack restructuring is accomplished by a *SLIDE* operation with appropriate arguments.

As we have said, the compile time environment ρ describes the part of the control expression environment which resides in the stack, i.e. the upper block of the stack. We will assume that both a code pointer p and a heap pointer a use one word of memory. Then ρ can be thought as a sequence of small blocks (δ_i, m_i) each one describing the length m_i in words of that part of the stack, and a proper environment δ_i mapping program variables to stack offsets.

Definition 28

A stack environment ρ is a list $[(\delta_k, m_k), \ldots, (\delta_1, m_1)]$ of environments, being (δ_k, m_k) the topmost one. It maps program variables to positions in the stack counting from the top, i.e. if $\rho\ x = j$, then the runtime value of x is $S!j$. In an environment (δ, m), δ maps $m-1$ program variables to disjoint numbers in the range $1 \ldots m-1$, except for the topmost environment (δ_k, m_k) which maps exactly m_k program variables to the range $1 \ldots m_k$. The empty environment, denoted $\rho_\emptyset$, is the list $[(\{\}, 0)]$.

An environment (δ, m) describes a small block of the stack upper block. All small blocks except the topmost one correspond to **case** alternatives. They are topped

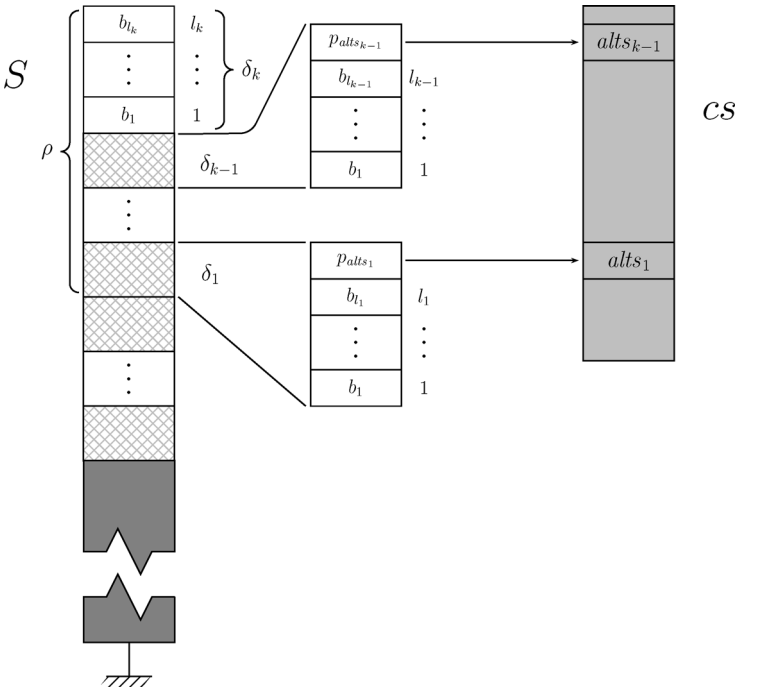

Fig. 14. Correspondence between the compile time environment and the stack.

with a code pointer p pointing to the alternatives. Because the topmost environment belongs to the expression under evaluation, it can be extended (this would be the case if it happens to be a **letrec** expression). The rest of small blocks cannot be extended. Should the upper block be extended, then the offset of its free variables will change. For this reason, numbers are assigned to the variables from the bottom of the environment. The offset of a given variable can always be computed by knowing the length of its block, and those of the blocks above it. For a better understanding, see Figure 14.

With these conventions, the offset of a variable x in ρ from the top of the stack, denoted $\rho\ x$, is formally given by:

$$\rho\ x \stackrel{\text{def}}{=} \left(\sum_{i=j}^{k} m_i\right) - \delta_j\ x, \quad \text{being} \quad x \in dom\ \delta_j, \qquad 1 \leqslant j \leqslant k$$

The following operations with stack environments are defined. The first one corresponds to extending the topmost environment, and it will be used when compiling **letrec** expressions. The second one corresponds to closing the topmost environment with a code pointer, and it will be used when compiling **case** expressions.

Definition 29

1. $((\delta, l) : \rho) + (\{\overline{x_i \mapsto j_i}^{n}\}, n) \stackrel{\text{def}}{=} (\delta \cup \{\overline{x_i \mapsto l + j_i}^{n}\}, l + n) : \rho$
2. $((\delta, l) : \rho)^{\dashv\vdash} \stackrel{\text{def}}{=} (\{\}, 0) : (\delta, l + 1) : \rho$

The rest of the control expression environment is located in the current closure. The *node* register is pointing to it while the instruction sequence *is* of the closure

expression is being executed. The compile-time environment η knows the position of each free variable in the closure.

Definition 30
A closure environment η with $n \geqslant 0$ variables is a mapping from these variables to disjoint numbers in the range $1 \ldots n$.

If the initial closed *Fun* expression to be translated uses different names for all bound variables, then the compile time environments ρ and η will always be disjoint. We will prove below that every free variable of the expression being compiled will necessarily be either in $dom\ \rho$ or in $dom\ \eta$, and never in both. This allows us to introduce the notation $(\rho, \eta)\ x$, whose meaning is:

$$(\rho, \eta)\ x \stackrel{\text{def}}{=} \begin{cases} (stack, \rho\ x) & \text{if } x \in dom\ \rho \\ (node, \eta\ x) & \text{if } x \in dom\ \eta \end{cases}$$

We will use in our examples a special **error** variable which can be free in any expression. This variable is supposed to be used by the compiler to complete missing alternatives in **case** expressions written by the programmer. Its semantics is to abort the program, but for the purposes of translation this free variable is located neither in the stack nor in the closure environments. In order to preserve our invariants, we will consider that the runtime value of **error** is located in the heap and its address e in known statically. We write then (ρ, η) **error** $\stackrel{\text{def}}{=} (static, e)$. This extension to the convention should be taken into account when translating the argument list of the instructions *BUILDCLS* and *BUILDENV* to C.

In GHC there exists a third kind of environment which describes the static addresses of the top-level bindings. The program is considered to be a set of static bindings instead of a top-level **letrec** expression. The access to these variables is different from that of other free variables: they are accessed directly by using a pointer to the static memory. Our **error** variable could provide an example of this kind of access.

5.4.2 Common translation schemes

In Figure 15 we present the translation schemes from *Fun* to machine code which are common to both machines ISTG_{PE} and ISTG_{EA}.

Function *trE* translates a *Fun* expression into a sequence of machine instructions. Function *trAs* translates a set of alternatives into a pointer to a vector of machine instruction sequences in the code store. The code store *cs* can be considered as a global variable of the translation. The notation & $cs[p \mapsto is]$ expresses that, as a side effect of the translation, the instruction sequence *is* is added to the code store *cs* and is pointed to by the fresh pointer *p*.

The translation presented for **letrec** is the one corresponding to the ISTG_{EA} machine, because it is more general and closer to what GHC does: it includes tags for the closures being built. Notice that function *trB* returns a pair whose first component is the closure tag. The second component is the code pointer of the instruction sequence to which the binding expression has been translated. Our push/enter machine will simply ignore these tags.

$$
\begin{array}{lll}
trE\ (\textbf{letrec}\ \overline{x_i = be_i}^{\,n}\ \textbf{in}\ e)\ \rho\ \eta & = & [ALLOC\ l_n, \ldots, ALLOC\ l_1] +\!\!+ \\
 & & [\overline{BUILDCLS\ tag_i\ (i-1)\ p_i\ zs_i}^{\,n}] +\!\!+ \\
 & & trE\ e\ \rho'\ \eta \\
\quad \text{where}\ \ \rho' & = & \rho + (\{\overline{x_i \mapsto n-i+1}^{\,n}\}, n) \\
\quad (tag_i, p_i) & = & trB\ be_i, \qquad i \in \{1 \ldots n\} \\
\quad zs_i & = & \overline{(\rho', \eta)\ y_{ij}}^{\,l_i}, \qquad i \in \{1 \ldots n\} \\
\quad \overline{y_{ij}}^{\,l_i} & = & fv\ be_i, \qquad i \in \{1 \ldots n\}
\end{array}
$$

$$
\begin{array}{lll}
trE\ (\textbf{case}\ e\ \textbf{of}\ alts)\ \rho\ \eta & = & [BUILDENV\ zs, \\
 & & PUSHALTS\ p] +\!\!+ \\
 & & trE\ e\ \rho'^{+\!\!+}\ (\eta - xs) \\
\quad \text{where}\ \ p & = & trAs\ alts\ \rho' \\
\quad \rho' & = & \rho + (\{\overline{xs_j \mapsto l' - j + 1}^{\,l'}\}, l') \\
\quad zs & = & [(node, \eta\ x) \mid x \leftarrow xs] \\
\quad l' & = & |xs| \\
\quad xs & = & [x \mid x \leftarrow \overline{x_i}^{\,l} \wedge x \in dom\ \eta] \\
\quad \overline{x_i}^{\,l} & = & fv\ alts
\end{array}
$$

$$
trAs\ (\overline{alt_i}^{\,n})\ \rho = p \qquad \& \ cs[p \mapsto \overline{trA\ alt_i\ \rho}^{\,n}]
$$

$$
trA\ (C\ \overline{x_i}^{\,n} \rightarrow e)\ \rho = trE\ e\ \rho\ \{\overline{x_i \mapsto i}^{\,n}\}
$$

$$
trB\ (C_k^n\ \overline{x_i}^{\,n}) = (\texttt{CONS}, p) \qquad \& \ cs[p \mapsto [RETURNCON\ C_k^n]]
$$

$$
\begin{array}{lll}
trB\ (e) & = (\texttt{THUNK}, p) & \& \ cs[p \mapsto [UPDTMARK] +\!\!+ trE\ e\ \rho_\emptyset\ \eta] \\
\quad \text{where}\ \ \eta & = \{\overline{y_j \mapsto j}^{\,n}\} & \\
\quad \overline{y_j}^{\,n} & = fv\ e &
\end{array}
$$

Fig. 15. Common code generation for *Fun* expressions.

The environments of the closures being created in the heap are trimmed to the free variables $\overline{y_{ij}}^{\,l_i}$ of the corresponding binding expression be_i, as it was suggested in rule *letrec* of Figure 9. The environment ρ' used to translate the main expression e is the original environment ρ extended with the locations of the new free variables $\overline{x_i}^{\,n}$ introduced by the **letrec**.

The translation of **case** closely follows the rule *case1* of Figure 9. Notice that the first *BUILDENV* instruction saves into the stack those variables of the environment belonging to the current closure. Otherwise the alternatives would not be able to find these variables when they were activated, as the current closure at that time would be a different one. Notice also that the stack environment of the alternatives is not trimmed, contrarily to what it was suggested in Figure 9. It could be, but this would be in contradiction with reusing that part of the control expression environment. The translation of the alternatives uses the environment ρ', an extension of ρ, for the variables located in the stack, and an environment $\eta = \{\overline{x_i \mapsto i}^{\,n}\}$ to access the pattern variables $\overline{x_i}$ because, when the alternative expression e is executed, the current closure will be a construction $C\ \overline{x_i}^{\,n}$. Finally, the environment $\rho'^{+\!\!+}$ used to translate the discriminant expression takes into account that a code pointer has been pushed to the stack.

The translation of a binding is straightforward. The *RETURNCON* instruction used for constructions ensures that the pending updates will be performed before

$$
\begin{array}{lll}
trE\ (x\ \overline{x_i}^n)\ \rho\ \eta & = & [BUILDENV\ ((\rho,\eta)\ x : \overline{(\rho,\eta)\ x_i}^n), \\
 & & \ SLIDE\ (n+1)\ m, \\
 & & \ ENTER] \\
\quad \text{where}\ (\_, m) : \_ & = & \rho \\
 & & \\
trE\ x\ \rho\ \eta & = & [BUILDENV\ [(\rho,\eta)\ x], \\
 & & \ SLIDE\ 1\ m, \\
 & & \ ENTER] \\
\quad \text{where}\ (\_, m) : \_ & = & \rho \\
 & & \\
trB\ (\lambda\overline{x_i}^n.e) & = & (\texttt{FUN}, p) \quad \& \ cs[p \mapsto [ARGCHECK\ n] +\!\!+ trE\ e\ \rho\ \eta] \\
\quad \text{where}\ \rho & = & [(\{\overline{x_i \mapsto n-i+1}^n\}, n)] \\
\quad \eta & = & \{\overline{y_j \mapsto j}^l\} \\
\quad \overline{y_j}^l & = & fv\ (\lambda\overline{x_i}^n.e)
\end{array}
$$

Fig. 16. Specific code generation for the ISTG$_{PE}$ machine.

jumping into a **case** alternative. The *UPDTMARK* instruction at the beginning of a non-normal-form expression translation ensures that an update mark will be pushed to the stack, as the rule *var1* of the STG-2 machine prescribes. The stack environment $\rho_\emptyset$ used to compile a closure expression is the empty one, and the closure environment η is the same built by *BUILDCLS* when it created the closure. Notice that the translation of a λ-abstraction binding is missing in this common part as it would be done in a different way for every ISTG machine.

5.4.3 The push/enter specific translation schemes

The specific translation schemes for the ISTG$_{PE}$ machine are presented in Figure 16.

The translation of an application starts by pushing to the stack the actual arguments and the functional closure pointer, followed by a *SLIDE* instruction in order to remove the topmost part of the current environment. Finally, an *ENTER* instruction is generated in order to jump to the functional closure.

The translation of a single variable is very similar. The only difference is that there are no arguments to push to the stack. If the closure being entered is a thunk or a construction, the translation given in the common part for these binding types is enough to perform the required actions: respectively, pushing an update mark and performing a pattern matching after possibly performing some update actions.

The translation of a λ-abstraction binding starts in this machine by the *ARGCHECK* instruction which ensures that there are enough arguments on top of the stack, possibly performing pending updates before entering the λ-abstraction body. The translation proceeds with the body expression by using a stack environment ρ which expects the arguments in the stack (x_1 in the top, x_2 below it, and so on), and an environment η expecting the free variables in the closure environment.

In order to illustrate the translation, in Figure 17 we show to the left a small *Fun* program, and to the right its translation to ISTG$_{PE}$ machine code.

```
letrec
  head = \xs.case xs of
           Nil       -> error
           Cons y ys -> y
  list = Cons x1 x2
  x1   = One
  x2   = Nil
in
  head list
```

```
[ ALLOC 0, ALLOC 0, ALLOC 2, ALLOC 0,
  BUILDCLS 0 p1 [],
  BUILDCLS 1 p2 [(stack,2),(stack,3)],
  BUILDCLS 2 p3 [],
  BUILDCLS 3 p4 [],
  BUILDENV [(stack,0)(stack,1)],
  SLIDE 2 4,
  ENTER
]
cs[p1]=[ARGCHECK 1, BUILDENV [], PUSHALTS pa,
        BUILDENV [(stack,1)], SLIDE 1 0, ENTER]
cs[p2]=[RETURNCONS Cons]
cs[p3]=[RETURNCONS One]
cs[p4]=[RETURNCONS Nil]
cs[pa]=[[BUILDENV [(static,e)], SLIDE 1 1, ENTER],
        [BUILDENV [(node,1)], SLIDE 1 1, ENTER]]
```

Fig. 17. A code generation example for the ISTG$_{PE}$ machine.

$$
\begin{array}{lll}
trE\ (x\ \overline{x_i}^n)\ \rho\ \eta & = & [BUILDENV\ (((\rho,\eta)\ x) : \overline{(\rho,\eta)\ x_i}^n), \\
 & & \ SLIDE\ (n+1)\ m, \\
 & & \ EVAL\ n] \\
\quad \text{where}\ \ (\_, m) : \_ = \rho & & \\
 & & \\
trE\ x\ \rho\ \eta & = & [BUILDENV\ [(\rho,\eta)\ x], \\
 & & \ SLIDE\ 1\ m, \\
 & & \ EVAL\ 0] \\
\quad \text{where}\ \ (\_, m) : \_ = \rho & &
\end{array}
$$

$$
\begin{array}{lll}
trB\ (\lambda\overline{x_i}^n.e) & = (\texttt{FUN}, p) \quad \& \ cs[p \mapsto trE\ e\ \rho\ \eta] & \\
\quad \text{where}\ \ \rho & = [(\{\overline{x_i \mapsto n-i+1}^n\}, n, 0)] & \\
\qquad \eta & = \{\overline{y_j \mapsto j}^l\} & \\
\qquad \overline{y_j}^l & = fv\ (\lambda\overline{x_i}^n.e) &
\end{array}
$$

Fig. 18. Specific code generation for the ISTG$_{EA}$ machine.

5.4.4 The eval/apply specific translation schemes

The specific translation schemes from *Fun* to ISTG$_{EA}$ are presented in Figure 18.

The difference now when translating applications or variables is that the *ENTER* instruction of the ISTG$_{PE}$ machine has been replaced by an *EVAL n* instruction, being n the number of actual arguments. As we have seen, this instruction is responsible for ensuring the correct number of arguments on top of the stack for the corresponding lambda, and for the rest of the specific rules for applications of the ISTG$_{EA}$ machine. The difference in the translation of a λ-abstraction binding is that the ISTG$_{PE}$ *ARGCHECK* instruction is now missing. The rest of the translation is identical to that of the ISTG$_{PE}$ machine.

In order to also illustrate the schemes, in Figure 19 we show the translation to ISTG$_{EA}$ machine code of the *Fun* program of Figure 17.

```
[ ALLOC 0, ALLOC 0, ALLOC 2, ALLOC 0,
  BUILDCLS FUN 0 p1 [],
  BUILDCLS CONS 1 p2 [(stack,2),(stack,3)],
  BUILDCLS CONS 2 p3 [],
  BUILDCLS CONS 3 p4 [],
  BUILDENV [(stack,0)(stack,1)],
  SLIDE 2 4,
  EVAL 1
]
cs[p1]=[BUILDENV [], PUSHALTS pa,
        BUILDENV [(stack,1)], SLIDE 1 0, EVAL 0]
cs[p2]=[RETURNCONS Cons]
cs[p3]=[RETURNCONS One]
cs[p4]=[RETURNCONS Nil]
cs[pa]=[[BUILDENV [(static,e)], SLIDE 1 1, EVAL 0],
        [BUILDENV [(node,1)], SLIDE 1 1, EVAL 0] ]
```

Fig. 19. A code generation example for the ISTG_{EA} machine.

5.5 Soundness and completeness of the ISTG machines

In this Section, we will prove that the previous translations correctly implements the STG_{PE}-2 and the STG_{EA}-2 machines on top of the corresponding ISTG machines. As there are not many differences between the ISTG_{PE} and the ISTG_{EA} machines, we will proceed with both at the same time, interleaving the differences as required.

We will need some properties of the translation itself, given by the following:

Proposition 31 (*Static invariant*)
Given a closed expression e_0 with different names for all bound variables and an initial call $trE\ e_0\ \rho_\emptyset\ \{\}$, we have:

1. It is an invariant precondition of any call to $trE\ e\ \rho\ \eta$ that $dom\ \rho \cap dom\ \eta = \emptyset$, and $\forall x \in fv\ e\ .\ x \neq \textbf{error} \rightarrow x \in dom\ \rho \vee x \in dom\ \eta$.
2. It is an invariant postcondition of $trE\ e\ \rho\ \eta$ that the last instruction generated is *ENTER* (in the push/enter machine), or *EVAL* (in the eval/apply machine), and this instruction may not appear in any other place of the sequence.

Proof

1. By induction on the tree structure of calls to *trE*
2. By structural induction on *Fun* expressions. □

To prove the equivalence between a functional and an imperative machine is not an easy task, since the translation, the functional semantics, and the imperative semantics must be handled at the same time. The main idea is to define an equivalence relation between STG-2 and ISTG configurations and then to show that both machines evolve through equivalent configurations.

The machines cannot be compared at any point in their respective evolutions as one STG-2 step will correspond in general to several ISTG steps. So, we first define some states of the STG-2 machine in which the comparison with ISTG makes sense. We call these configurations *stable*. In essence, they correspond to ISTG machine configurations $(is, S_I, node, \Gamma_I, cs)$ in which *is* is the translation of some expression *e*, so the control expression *e* in the STG-2 configuration should be one of the

expressions written by the programmer in the original *Fun* program. The following definition makes this idea precise.

Definition 32
Given a closed expression e_0:

1. The initial configuration $(\{\}, e_0, \{\}, [])$ of the STG-2 machine is *stable*.
2. If $(\Gamma, e, E, S) \rightarrow (\Gamma', e', E', S')$ then (Γ', e', E', S') is stable if the rule applied in the step is *letrec, case1, case2, var1, appPE2* (push/enter), or *appEA2* (eval/apply).

It is not necessary to independently define the stable configurations in ISTG. Instead, we define an equivalence relation between STG-2 stable configurations and ISTG configurations. Then the ISTG stable configurations would be those equivalent to the STG-2 stable ones.

First, we define the equivalence between STG-2 and ISTG runtime environments. Essentially, an STG-2 environment E corresponds to the imperative environments ρ and η. As the environment ρ refers to positions in the stack and the environment η refers to positions in the heap closure pointed to by *node*, we need these components in the definition.

Definition 33
An STG-2 environment E is equivalent to an ISTG environment defined by ρ, η, S_I, Γ_I and *node*, via an α-renaming β, and denoted $E \stackrel{\beta}{\equiv} (\rho, S_I, \eta, \Gamma_I, node)$, if $dom\ E \subseteq (dom\ \rho \cup dom\ \eta)$ and

$$\forall x \in dom\ E\ .\ \begin{cases} E\ x = \beta\ (S_I\ !\ (\rho\ x)) & \text{if } x \in dom\ \rho \\ E\ x = \beta\ (node_{\Gamma_I}\ !\ (\eta\ x)) & \text{if } x \in dom\ \eta \end{cases}$$

Notice that, if $\eta = \{\}$, then the values Γ_I and *node* do not matter. The following definition establishes the equivalence between STG-2 and ISTG stacks. Basically, it says that the environment on top of the ISTG stack should be ignored and that the rest of stack frames in equivalent positions of both stacks should be equivalent. The code store of the ISTG machine is needed in the definition.

Definition 34
An STG-2 stack S is equivalent to an ISTG stack defined by a triple (ρ, S_I, cs), via an α-renaming β, and denoted $S \stackrel{\beta}{\equiv} (\rho, S_I, cs)$, if $\rho = (\delta, m) : \rho'$, $S_I = \overline{a_i}^m : S_I'$, and $S \stackrel{\beta}{\equiv}_{aux} (S_I', cs)$, where $S \stackrel{\beta}{\equiv}_{aux} (S_I', cs)$ is defined as follows:

1. If $S = (\overline{alt_i}^n, E) : S'$, then $S_I = p_{alts} : S_I'$ and there exists ρ_{alts} such that $cs(p_{alts}) = \overline{trA\ alt_i\ \rho_{alts}}^n$. Also, $E \stackrel{\beta}{\equiv} (\rho_{alts}, S_I', \{\}, \_, \_)$, and $S' \stackrel{\beta}{\equiv} (\rho_{alts}, S_I', cs)$
2. If $S = \#p : S'$, then $S_I = \#a : S_I'$, $p = \beta\ a$, and $S' \stackrel{\beta}{\equiv}_{aux} (S_I', cs)$
3. (eval/apply) If $S = \bullet\overline{a_i}^n : S'$, then $S_I = \bullet\overline{\beta\ a_i}^n : S_I'$, and $S' \stackrel{\beta}{\equiv}_{aux} (S_I', cs)$
4. (push/enter) If $S = \overline{a_i}^n : S'$, then $S_I = \overline{\beta\ a_i}^n : S_I'$, and $S' \stackrel{\beta}{\equiv}_{aux} (S_I', cs)$
5. Additionally, $[] \stackrel{\beta}{\equiv}_{aux} ([], \_)$

Leaving apart environments and α-renaming, there is a non-essential difference between the STG-2 and the ISTG heaps: closures under evaluation in STG-2 are removed from the heap, while they are 'black-holed' in ISTG. So, we give the following equivalence definition:

Definition 35
An STG-2 heap Γ is equivalent to the ISTG pair (Γ_I, cs), via an α-renaming β, and denoted $\Gamma \overset{\beta}{\equiv} (\Gamma_I, cs)$, if for all $p \in dom\ \Gamma$ we have:

1. $\Gamma_I[a \mapsto (q, ws)]$ iff $\Gamma[\beta\ a \mapsto (be, E)]$, $(\_, q) = trB\ be$, $\overline{x_i}^n = fv\ be$, $\beta\ ws = \overline{E\ x_i}^n$, $q \neq p_{bh}$ and $q \in dom\ cs$
2. $\Gamma_I[b \mapsto (p_{pap}^{n+1}, a : \overline{a_i}^n)]$ iff $\Gamma[\beta\ b \mapsto \texttt{PAP}\ (y\ \overline{y_i}^n, E)]$, $\beta\ (a : \overline{a_i}^n) = E\ y : \overline{E\ y_i}^n$

Additionally, for every closure $[a \mapsto (q_{bh}, ws)] \in \Gamma_I$, we have $\beta\ a \notin dom\ \Gamma$.

Finally, we define the following equivalence relation between an STG-2 stable configuration and an ISTG configuration.

Definition 36
An STG-2 stable configuration is equivalent to an ISTG configuration, denoted $(\Gamma, e, E, S) \equiv (is, S_I, node, \Gamma_I, cs)$, if there exists an α-renaming β and two compile-time environments ρ and η such that:

1. $\Gamma \overset{\beta}{\equiv} (\Gamma_I, cs)$
2. $is = trE\ e\ \rho\ \eta$
3. $E \overset{\beta}{\equiv} (\rho, S_I, \eta, \Gamma_I, node)$
4. $S \overset{\beta}{\equiv} (\rho, S_I, cs)$

Stable configurations do not cover normal forms of both machines, so we extend the previous definition to normal forms.

Definition 37
A normal form STG-2 configuration is equivalent to a normal form ISTG configuration, denoted $(\Gamma[p \mapsto w], x, E[x \mapsto p], [\,]) \equiv (is, [\,], node, \Gamma_I, cs)$, if there exists an α-renaming β such that:

1. $\Gamma \overset{\beta}{\equiv} (\Gamma_I, cs)$
2. $\beta\ node = p$

In the following theorem, by $conf \overset{\text{STG-2}^+}{\longrightarrow} conf'$ we denote the minimum non-empty possible evolution of the STG-2 machine in order to reach a stable configuration $conf'$ from a stable configuration $conf$, if there are more than one stable configuration after $conf$, or to reach a normal form $conf'$ if there is only one stable configuration after $conf$. This apparently awkward definition is needed because the STG-2 machine goes through a last stable configuration before stopping in a normal form, and this stable configuration has no counterpart in the ISTG machine, which directly stops in a normal form configuration.

Theorem 38 (*Dynamic invariant*)
Let (Γ, e, E, S) be an STG-2 stable configuration, and $(is, S_I, node, \Gamma_I, cs)$ be an ISTG configuration such that $(\Gamma, e, E, S) \equiv (is, S_I, node, \Gamma_I, cs)$. Then:

1. If $(\Gamma, e, E, S) \overset{\text{STG-2}^+}{\longrightarrow} (\Gamma', e', E', S')$, then there exists $(is, S_I, node, \Gamma_I, cs) \Rightarrow^+ (is', S_I', node', \Gamma_I', cs)$ such that $(\Gamma', e', E', S') \equiv (is', S_I', node', \Gamma_I', cs)$

2. If (Γ, e, E, S) cannot evolve either to a stable point or to a normal form, then $(is, S_I, node, \Gamma_I, cs)$ cannot evolve either to a stable point or to a normal form.

Proof
By cases on stable configurations of the STG-2 machine. As there many cases and subcases, we sketch the general proof and then present in detail two of them. In the second one we do the proof separately for STG_{PE}-2 and STG_{EA}-2.

In a stable STG-2 configuration, the control expression may be any *Fun* expression. If it is a **letrec** or a **case**, after a single transition (respectively, a *Letrec* or a *Case*1 one), the machine reaches another stable configuration. If it is an application or a variable, there may be many transitions before STG-2 reaches a new stable configuration. The idea of the proof is to make evolve the STG-2 configuration until a new stable state is reached. Then we make evolve the translated code in the ISTG machine until an equivalent configuration is reached. During both evolutions, we keep track of the fresh variables created in the machines in order to define the appropriate α-renaming. We present below the proof for **letrec** and applications.

letrec We know by hypothesis:

H1 $conf_{STG2} = (\Gamma, \textbf{letrec}\ \overline{x_i = be_i}^n\ \textbf{in}\ e, E, S) \equiv (is, S_I, node, \Gamma_I, cs) = conf_{ISTG}$. Let us call β to the α-renaming establishing this equivalence.

H2 $conf_{STG2} \rightarrow (\overbrace{\Gamma \cup [\overline{p_i \mapsto (be_i, E' | \overline{y_{ij}}^{m_i})}^n]}^{\Gamma'}, e, E', S)$ where $E' = E \cup [\overline{x_i \mapsto p_i}^n]$, $\overline{y_{ij}}^{m_i} = fv\ be_i$, and $\overline{p_i}$ fresh, by applying the STG-2 *Letrec* rule.

Then, by H1, we have:

$$conf_{ISTG} = ([ALLOC\ m_n, \ldots, ALLOC\ m_1, \overline{BUILDCLS\ (i-1)\ q_i\ zs_i}^n] +\!\!+ trE\ e\ \rho'\ \eta, \ldots)$$

and, by applying the ISTG transition rules for *ALLOC* and *BUILDCLS*, we get:

$$conf_{ISTG} \Rightarrow^+ (trE\ e\ \rho'\ \eta, \overline{a_i}^n : S_I, node, \underbrace{\Gamma_I \cup [\overline{a_i \mapsto (q_i, \overline{ws_{ij}}^{m_i})}^n]}_{\Gamma'_I}, cs)$$

where $\rho' = \rho + (\{\overline{x_i \mapsto n - i + 1}^n\}, n)$, $\overline{a_i}$ fresh, $zs_i = \overline{(\rho', \eta)\ y_{ij}}^{m_i}$, $(\_, q_i) = trB\ be_i$, and

$$ws_{ij} = \begin{cases} (\overline{a_i}^n : S_I)!k & \text{if } (zs_i)_j = (stack, k), \\ node_{\Gamma'_I}!k & \text{if } (zs_i)_j = (node, k). \end{cases}$$

Let us define β' as:

$$\beta'\ a = \begin{cases} p_i & \text{if } a = a_i,\ 1 \leqslant i \leqslant n \\ \beta\ a & \text{otherwise.} \end{cases}$$

By using this β', ρ', and η we can easily prove the desired result:

$$(\Gamma', e, E', S) \equiv (trE\ e\ \rho'\ \eta, \overline{a_i}^n : S_I, node, \Gamma'_I, cs) \qquad \surd$$

$x\ \overline{x_i}^n$ *(push/enter)* The STG_{PE}-2 machine first pushes the arguments $\overline{x_i}^n$ into the stack and then enters the closure pointed to by x (rule *AppPE1*). Depending on

the closure type and of the stack contents, the machine may perform one of the following traces before reaching a new stable state:

*Var*1 This is the case in which the closure is a thunk.

AppPE2 The closure expression is a λ-abstraction with less than, or equal to, or more than n formal arguments. In the last case, the extra arguments were on top of the stack before pushing $\overline{x_i}^n$.

$(AppPE4\ \ Var2^+\ \ AppPE3)^+\ \ AppPE2$ The closure expression is a λ-abstraction, with more than n formal arguments. In the stack there are one or more update marks interleaved with arguments. The update marks are removed, some partial application closures are created, and eventually there are enough arguments in the stack so that the original λ-abstraction is applied.

$AppPE3\ \ (AppPE4\ \ Var2^+\ \ AppPE3)^*\ \ AppPE2$ The closure expression is a partial application. As above, in the stack there are one or more update marks interleaved with arguments, the update marks are removed, some partial application closures are created, and eventually the original λ-abstraction is applied.

All the paths above can be proved correct. The proof is tedious but not difficult. As an example we show the proof for the trace *AppPE1 AppPE2* where the λ-abstraction have less than or equal to n formal arguments, the other case being similar.

H1 By hypothesis we know: $conf_{STG2} = (\Gamma, x\ \overline{x_i}^n, E\{x \mapsto p, \overline{x_i \mapsto p_i}^n\}, S) \equiv (is, S_I, node, \Gamma_I, cs) = conf_{ISTG}$. Let us assume $\Gamma[p \mapsto (\lambda\ \overline{y_i}^{n'}.e, E')]$, and $n' \leqslant n$. Let us call β to the α-renaming establishing this equivalence.

H2 By the trace *AppPE1 AppPE2*, we have:

$$\begin{array}{lll} conf_{STG2} & \rightarrow & (\Gamma, x, E\{x \mapsto p\}, \overline{p_i}^n : S) \\ & \rightarrow & (\Gamma, e, E' \uplus \{\overline{y_i \mapsto p_i}^{n'}\}, \overline{p_j}^{n-n'} : S) \end{array}$$

P1 By H1 and ISTG_{PE} rules, we have:

$$\begin{array}{lll} conf_{ISTG} & = & ([BUILDENV\ ((\rho, \eta)\ (x : \overline{x_i}^n)), SLIDE\ (n+1)\ m, ENTER], \ldots) \\ conf_{ISTG} & \Rightarrow & ([SLIDE\ (n+1)\ m, ENTER], a : \overline{a_i}^n : S_I, node, \Gamma_I, cs) \\ & \Rightarrow & ([ENTER], a : \overline{a_i}^n : S'_I, node, \Gamma_I, cs) \end{array}$$

being $S_I = \overline{b_i}^m : S'_I$, $\overline{p_i}^n = \beta\ \overline{a_i}^n$, $p = \beta\ a$, and $S \overset{\beta}{\equiv}_{aux} (S'_I, cs)$.

P2 Again by H1, we have $\Gamma \overset{\beta}{\equiv} (\Gamma_I, cs)$, $\Gamma_I[a \mapsto (p_{be}, ws)]$, $(\_, p_{be}) = trB\ (\lambda\ \overline{y_i}^{n'}.\ e)$, $fv\ (\lambda\ \overline{y_i}^{n'}.e) = \overline{z_j}^l$, and $\overline{E'\ z_j}^l = \beta\ ws$

P3 Proceeding with the execution of ISTG_{PE}, we get:

$$\begin{array}{lll} & & ([ENTER], a : \overline{a_i}^n : S'_I, node, \Gamma_I, cs) \\ & \Rightarrow & (ARGCHECK\ n' : trE\ e\ \rho'\ \eta', \overline{a_i}^n : S'_I, a, \Gamma_I, cs) \\ & \Rightarrow & (trE\ e\ \rho'\ \eta', \overline{a_i}^n : S'_I, a, \Gamma_I, cs) \end{array}$$

where, by H1, $\rho' = [(\{\overline{y_i \mapsto n' - i + 1}^{n'}\}, n')]$ and $\eta' = \{\overline{z_j \mapsto j}^l\}$.

P4 By P1, P2, and P3, we have: $E' \uplus \{\overline{y_i \mapsto p_i}^{n'}\} \overset{\beta}{\equiv} (\rho', \overline{a_i}^n : S'_I, \eta', \Gamma_I, a)$

P5 By P1 and P3, we have: $\overline{p_j}^{n-n'} : S \overset{\beta}{\equiv} (\rho', \overline{a_i}^n : S'_I, cs)$

P6 Finally, by P2, P3, P4 and P5, we have:

$$(\Gamma, e, E' \uplus \{\overline{y_i \mapsto p_i}^{n'}\}, \overline{p_j}^{n-n'} : S) \equiv (trE\ e\ \rho'\ \eta', \overline{a_i}^n : S'_I, a, \Gamma_I, cs) \qquad \surd$$

$\boxed{x\ \overline{x_i}^n\ (eval/apply)}$ Depending on the closure type pointed to by x and of the stack contents, the machine may perform one of the following traces before reaching a new stable state:

*AppEA1 Var*1 This is the case in which the closure is a thunk.

AppEA2 The closure expression is a λ-abstraction with less than, or equal to, n formal arguments.

$(AppEA4\ Var2^*\ AppEA5\ AppEA3)^+\ AppEA2$ The closure expression is a λ-abstraction with more than n formal arguments, then a partial application closure is created. After zero or more more update marks, there should be a packet of arguments in the stack. Then *AppEA5* followed by *AppEA3* are executed and the control expression would be again of the form $x\ \overline{x_i}^n$ with x pointing to a λ-abstraction. The trace may iterate several times and eventually there would be enough arguments in the control expression so that the λ-abstraction would be applied (rule *AppEA2*).

$AppEA3\ (AppEA4\ Var2^*\ AppEA5\ AppEA3)^*\ AppEA2$ The closure expression is a partial application. It is copied to the control expression and then the functional variable would point to a λ-abstraction. The rest of the trace is covered by the second or by the third case of this description.

As an example we show the proof for the trace *AppEA1 Var1*.

H1 By hypothesis we know: $conf_{STG2} = (\Gamma, x\ \overline{x_i}^n, E\{x \mapsto p, \overline{x_i \mapsto p_i}^n\}, S) \equiv (is, S_I, node, \Gamma_I, cs) = conf_{ISTG}$. Let us assume $\Gamma = \Gamma' \uplus [p \mapsto (e, E')]$. Let us call β to the α-renaming establishing this equivalence.

H2 By the trace *AppEA1 Var1*, we have:

$$\begin{array}{rcl} conf_{STG2} & \rightarrow & (\Gamma, x, E\{x \mapsto p\}, \bullet\overline{p_i}^n : S) \\ & \rightarrow & (\Gamma', e, E', \#p : \bullet\overline{p_i}^n : S) \end{array}$$

P1 By H1 and ISTG_{EA} rules, we have:

$$\begin{array}{rcl} conf_{ISTG} & = & ([BUILDENV\ ((\rho, \eta)\ (x : \overline{x_i}^n)), SLIDE\ (n+1)\ m, ENTER], \ldots) \\ conf_{ISTG} & \Rightarrow & ([SLIDE\ (n+1)\ m, EVAL\ n], a : \overline{a_i}^n : S_I, node, \Gamma_I, cs) \\ & \Rightarrow & ([EVAL\ n], a : \overline{a_i}^n : S'_I, node, \Gamma_I, cs) \end{array}$$

being $S_I = \overline{b_i}^m : S'_I$, $\overline{p_i}^n = \beta\ \overline{a_i}^n$, $p = \beta\ a$, and $S \overset{\beta}{\equiv}_{aux} (S'_I, cs)$.

P2 Again by H1, we have $\Gamma \overset{\beta}{\equiv} (\Gamma_I, cs)$, $\Gamma_I = \Gamma'_I \uplus [a \mapsto (p_{be}, ws)]$, $(\_, p_{be}) = trB\ e$, $fv\ e = \overline{z_j}^l$, and $\overline{E'\ z_j}^l = \beta\ ws$

P3 Proceeding with the execution of ISTG_{EA}, we get:

$$\begin{array}{rcl} & & ([EVAL\ n], a : \overline{a_i}^n : S'_I, node, \Gamma'_I \uplus [a \mapsto (p_{be}, ws)], cs) \\ \Rightarrow & & (UPDTMARK\ : trE\ e\ \rho_\emptyset\ \eta', \bullet\overline{a_i}^n : S'_I, a, \Gamma'_I \uplus [a \mapsto (p_{be}, ws)], cs) \\ \Rightarrow & & (trE\ e\ \rho_\emptyset\ \eta', \#a : \bullet\overline{a_i}^n : S'_I, a, \Gamma'_I \uplus [a \mapsto (p_{bh}, ws)], cs) \end{array}$$

where, by H1, $\eta' = \{\overline{z_j \mapsto j}^l\}$.

P4 By P1, P2, and P3, we have: $E' \overset{\beta}{\equiv} (\rho_\emptyset, \#a : \bullet\overline{a_i}^n : S'_I, \eta', \Gamma'_I \uplus [a \mapsto (p_{bh}, ws)], a)$

P5 By P1 we have: $\#p : \bullet\overline{p_i}^n : S \overset{\beta}{\equiv} (\rho_\emptyset, \#a : \bullet\overline{a_i}^n : S'_I, cs)$

P6 By P2 we have: $\Gamma' \overset{\beta}{\equiv} (\Gamma'_I \uplus [a \mapsto (p_{bh}, ws)], cs)$

P7 Finally, by P3, P4, P5, and P6, we have:

$$(\Gamma', e, E', \#p : \bullet\overline{p_i}^n : S) \equiv (trE\ e\ \rho_\emptyset\ \eta', \#a : \bullet\overline{a_i}^n : S'_I, a, \Gamma'_I \uplus [a \mapsto (p_{bh}, ws)], cs) \quad \surd$$

Abnormal termination An informal reasoning of why both machines should stop abnormally at the same time follows. Both machines start from equivalent stable configurations. By hypothesis, STG-2 can evolve neither to a new stable configuration nor to a normal form. We should prove that ISTG neither can. For STG-2 not to evolve, one of the following situations must happen:

- There is a λ-abstraction about to be applied, the stack does not contain enough arguments, and there is something in the stack different from an update mark. As the stack is not empty, it should be a **case** continuation. As stacks are equivalent, then the ISTG stack should have also a **case** continuation and could not evolve.
- A construction has been arrived at, and there is something in the stack different from **case** continuations or update marks. As the stack is not empty, they should be arguments. As stacks are equivalent, then the ISTG machine could not evolve either.
- The control expression is a pointer to a non-existent closure (i.e. to a closure under evaluation). Then the closure code pointer in ISTG would be p_{bh} and the machine would also stop.

□

Corollary 39
Given a closed expression e_0 with different bound variables, the initial STG-2 configuration $(\{\}, e_0, \{\}, [\,])$, and the initial ISTG configuration $(trE\ e_0\ \rho_\emptyset\ \{\}, [\,], \_, \{\}, cs)$, then:

$$(\{\}, e_0, \{\}, [\,]) \xrightarrow{\text{STG-2}^*} (\Delta[p \mapsto w], x, E\{x \mapsto p\}, [\,]) \quad \text{iff}$$
$$(trE\ e_0\ \rho_\emptyset\ \{\}, [\,], \_, \{\}, cs) \Rightarrow^* (is, [\,], node, \Delta_I, cs)$$
$$\text{and } (\Delta[p \mapsto w], x, E\{x \mapsto p\}, [\,]) \equiv (is, [\,], node, \Delta_I, cs)$$

where cs is the code store generated by the whole translation of e_0.

Corollary 40
The translation given in Figures 15, 16, and 18 is correct.

6 Related work and conclusions

We have already mentioned in Section 2 the works by Sestoft and Mountjoy and their influence in our own work.

Mountjoy (Mountjoy 1998) had the idea of restricting the language *Basic* and its semantics in order to get closer to the STG language and then to derive the STG machine from the new semantics. He developed two different semantics: in the first one, which we call semantics $S1$, the main change was the requirement for normal forms to be either constructions (as they were in Sestoft's semantics) or variables pointing to heap bindings containing λ-abstractions, instead of just λ-abstractions. The reason for this was to forbid the presence of λ-abstractions in the control expression. Another change was to force applications to have the form $x\ x_1$, i.e.

with a variable in the functional part. These changes forced Mountjoy to modify the source language and to define a normalisation transformation from *Basic* to the new language. Mountjoy proved that the normalisation process did not change the normal forms arrived at by both semantics.

The second semantics, which we call semantics $S2$, forced applications of n arguments to be done simultaneously instead of one by one. Correspondingly, λ-abstractions were allowed to have several arguments. Semantics $S2$ was informally derived and contained some mistakes. In particular, the rule App_M made a λ-abstraction to appear in the control expression, in contradiction with the desire of having λ-abstractions only in the heap. This makes rule App_M incorrect and consequently rules to correctly cope with partial applications are missing. The derivation of an STG-like machine from the semantics $S2$ was done informally and no proof of correctness was provided.

The differences with our semantics $S3$ are: (1) we correctly cope with partial applications; (2) we also consider as normal forms variables pointing to constructions or to partial applications; and (3) we provide proofs of correctness for the relation between the semantics $S3$ and the STG-1 machines. Also, at the time of Mountjoy's work the eval/apply model had not been developed, so no derivation of this machine was done.

There have been other successful derivations of abstract machines starting from high level descriptions of the semantics. For instance, in Hannan & Miller (1992) and Ager *et al.* (2003) a number of such derivations are presented. Well-known abstract machines for the λ-calculus such as SECD, Krivine's, CLS, and CAM are derived and proved correct. These papers propose general schemes for achieving this kind of derivations. The differences with the present work are the following:

- They concentrate on the pure λ-calculus and they consider neither sharing nor heaps. Algebraic types, **case** and **letrec** expressions are not considered either.
- In the second paper, the starting point is the denotational meaning of the source language.
- In order to refine their machines they use predefined correct transformations such as closure conversion, transformation into continuation passing style, defunctionalization, and in-lining.
- They ignore the compilation issues from the source language to machine instructions.

In Kluge (2005) a broad survey of abstract and virtual machines for the λ-calculus and for practical functional languages is presented, which contains the details of many well-known (and some not so well-known) abstract machines. Most of them are full-normalising and some are weakly normalising, as is usually the case when implementing real-life functional languages. When the machines execute compiled code, the translation schemes are also provided. The aim of the book is to serve as a text for a graduate course and no attempt is made at providing proofs of correctness either for the machines or for the compilation schemes.

Regarding the STG machine and the GHC compiler, there are some differences between the machine translations we present in Section 5.4 and the actual code generated by GHC. Some are just omissions, some are non-substantial differences and yet some others are deeper ones.

In the first group is included the treatment of basic values, very elaborated in GHC—see for example Peyton Jones & Launchbury (1991)—and completely ignored here. For the sake of simplicity, we have preferred to concentrate our study on the functional kernel of the machine, but of course a formal analysis of this aspect could also be undertaken along similar lines.

The second group contains, for instance, the optimisation of the implementation of updates, which in GHC are always done by indirection. Also, GHC maintains the so called *info-table*, a static table shared by all closures created from the same binding, which contains detailed information about the size and layout of the closures. Our model has simplified these aspects. Finally, we understand that stack restructuring, as the one performed by our *SLIDE* instruction, is not implemented in this way by GHC. Apparently, they maintain up to four compile-time environments for free variables: one based on stack offsets, another one based on the current closure, a third one containing the addresses of the static bindings, and a fourth one where numbers relative to the heap pointer are assigned to variables. When they push a **case** continuation framework, they save in the stack the free variables needed by the continuation expression. Thus, they do not reuse the control expression environment as we do in our translation, and instead they trim the environment to just the set of variables free in the alternatives.

A deeper difference between our derived STG machines and the actual implementation made by GHC is the way the latter processes the stack frames. Our model is based on the current machine instruction somehow inspecting the kind of stack frame on top of the stack, and taking the appropriate action based on that. That is, stack frames are implicitly assumed to be tagged. In contrast, GHC's implementation is based on having a *return address* at the beginning of each stack frame. The machine just jumps to this return address instead of inspecting the stack, and then falls into code that knows the exact size and layout of the stack frame, and how to process it. This (replacing tags by jumps) can be considered as an optimisation, but it also makes the translation different. For instance, our *RETURNCON* instruction cannot be recognised as such in the code. Its behavior is split up into the return address code associated to update stack frames and the return address code associated to **case** continuation stack frames. Likewise, our *EVAL* instruction behavior is split up into the return address code for argument and update stack frames, and the code generated for each call site. GHC's programmers have even developed for this site code several optimised versions of a generic function `stgApply` specialised in the number and type of arguments being applied. The translation of a variable of a functional type generates code which inspects the closure pointed to by the variable and jumps to the closure code if it happens to be a thunk. Otherwise, it jumps to the top stack frame return address.

We find it difficult to introduce these optimisations in our model and at the same time to keep the correctness argument simple. We have been able to show that our environment conventions, choice of machine instructions, and formal translation schemes make this reasoning feasible with a reasonable amount of effort. It might be possible to get closer to the actual implementation by introducing more refinements about primitive values, runtime environments, and stack frames, but we decided to stop there. We believe that we have gone most of the way from the original Sestoft's

semantics to a reasonably efficient imperative implementation, and provided a solid basis for anyone willing to continue this kind of formal work.

To sum up, we believe that the derivation method we have followed to arrive at the final imperative code could constitute an attractive approach to design abstract machines and imperative translations of functional languages. We can briefly describe the steps as follows:

1. Start from a well-known, accepted, natural semantics of the language.
2. Manipulate the semantics and the language in order to avoid constructions which may be costly to implement (in our case, the back and forth thrashing of values between the heap and the control expression). Prove that the new semantics is equivalent to the original one.
3. Introduce a stack to traverse the semantic derivation tree in a sequential way.
4. Replace explicit substitutions by runtime environments.
5. Observe in which transitions new environments are installed, saved, duplicated, extended, or discarded.
6. Decide how to represent environments. A number of choices are: as frames in the stack, as part of the current closure, as static addresses, as offsets from the heap pointer, etc. Establish appropriate compile-time and run-time invariants for the environments.
7. Concurrently design the instructions of the imperative machine and the translation schemes of the source language, in order to implement every transition rule of the reference abstract machine.
8. Prove that every refinement preserves the semantics.

The second part of our contribution has consisted of clarifying how and why the two STG machines work and in providing some help for those wishing to understand the detailed explanations contained in Marlow & Peyton Jones (2006) and in the actual code of the GHC compiler.

Acknowledgments

We want to thank the anonymous referees for the comments provided to previous versions of this work. They have helped us very much to improve and extend the ideas presented here. We also thank our sponsors, the *Spanish Ministry of Education and Science* and the *Madrid Region Research Department*, for their generous funding. Finally, we are grateful to our colleague Miguel Palomino for helping us to polish the english.

References

Ager, M. S., Biernacki, D., Danvy, O., & Midtgaard, J. (2003). A functional correspondence between evaluators and abstract machines. In *Proceedings of Principles and practice of Declarative Programming, PPDP'03*, Uppsala, Sweden, ACM Press, pp. 8–19.

Encina, A. de la, & Peña, R. (2002). Proving the correctness of the STG machine. In *Selected Papers of Implementation of Functional Languages, IFL'01, Stockholm, Sweden*. LNCS 2312. Springer. pp. 88–104.

Encina, A. de la, & Peña, R. (2003): Formally deriving a STG machine. In *Principles and Practice of Declarative Programming, PPDP'03*. Uppsala, Sweden: ACM Press, pp. 102–112.

Hannan, J., & Miller, D. (1992). From operational semantics to abstract machines. *Math. Struct. Comput. Sci.* **2**(4), 415–459.

Kluge, W. (2005). *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer Texts in Theoretical Computer Science.

Launchbury, J. (1993). A Natural semantics for lazy evaluation. In *Proceedings Conference on Principles of Programming Languages, POPL'93*, Charleston, South Carolina, ACM Press, pp. 144–154.

Marlow, S., & Peyton Jones, S. L. (2004). Making a fast curry: Push/enter vs. eval/apply for higher-order languages. *Sigplan Not.* **39**(9), 4–15.

Marlow, S., & Peyton Jones, S. L. (2006). Making a fast curry: Push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.* **16**(4-5), 415–449.

Morrisett, G., Felleisen, M., & Harper, R. (1995). Abstract models of memory management. In *International Conference on Functional Programming Languages and Computer Architecture, FPCA'95*. La Jolla, California: ACM Press, pp. 66–77.

Mountjoy, J. (1998). The spineless tagless G-machine, naturally. In *Third International Conference on Functional Programming, ICFP'98*. Baltimore: ACM Press, pp. 163–173.

Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: The spineless tagless G-machine, version 2.5. *J. Funct. Program.* **2**(2), 127–202.

Peyton Jones, S. L., & Hughes, J. (eds). (1999). *Report on the Programming Language Haskell 98*. URL http://www.haskell.org.

Peyton Jones, S. L., & Marlow, S. (2002). Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program.* **12**(4), 393–434.

Peyton Jones, S. L., & Salkild, J. (1989). The spineless tagless G-machine. In *Proceedings of Functional Programming Languages and Computer Arquitecture Conference. FPCA'89*, London. ACM, pp. 184–201.

Peyton Jones, S. L., & Santos, A. (1998). A transformation-based optimiser for haskell. *Sci. Computer Program.* **32**(1-3), 3–47.

Peyton Jones, S. L., Hall, C. V., Hammond, K., Partain, W. D., & Wadler, P. L. (1993). The Glasgow Haskell Compiler: A technical overview. In *Joint Framework for Information Technology*. Keele, UK.

Peyton Jones, S. L. (1996). Compiling Haskell by Program Transformation: A report from the trenches. In *6th European Symposium on Programming ESOP'96*, Sweden. LNCS 1058. Springer. pp. 18–44.

Peyton Jones, S. L., & Launchbury, J. (1991). Unboxed values as first class citizens in a non-strict functional language. *Conference on Functional Programming Languages and Computer Architecture FPCA'91,* Cambridge, MA. LNCS 523. Springer. pp. 636–666.

Pitts, A. (2005). Alpha-structural recursion and induction. In *Theorem Proving in Higher Order Logics, TPHOL'05*. Oxford, UK. LNCS 3603. Springer. pp. 17–34.

Sestoft, P. (1997). Deriving a Lazy Abstract Machine. *J. Funct. Program.* **7**(3), 231–264.

Urban, C., Pitts, A. M., & Gabbay, M. J. (2004). Nominal Unification. *Theor. Comput. Sci.* **323**, 473–497.