Formally Deriving an STG Machine[*]

Alberto de la Encina
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
Avda. Complutense s/n, 28040 Madrid
albertoe@sip.ucm.es

Ricardo Peña
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
Avda. Complutense s/n, 28040 Madrid
ricardo@sip.ucm.es

ABSTRACT

Starting from P. Sestoft semantics for lazy evaluation, we define a new semantics in which normal forms consist of variables pointing to lambdas or constructions. This is in accordance with the more recent changes in the *Spineless Tagless G-machine* (STG) machine, where constructions only appear in closures (lambdas only appeared in closures already in previous versions). We prove the equivalence between the new semantics and Sestoft's. Then, a sequence of STG machines are derived, formally proving the correctness of each derivation. The last machine consists of a few imperative instructions and its distance to a conventional language is minimal.

The paper also discusses the differences between the final machine and the actual STG machine implemented in the Glasgow Haskell Compiler.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics, syntax*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; D.3.4 [**Programming Languages**]: Processors—*code generation, compilers*; F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages—*operational semantics*

General Terms

Theory, Languages, Verification

Keywords

Functional programming, abstract machines, operational semantics, compiler verification

1. INTRODUCTION

The *Spineless Tagless G-machine* (STG) [6] is at the heart of the *Glasgow Haskell Compiler* (GHC) [7] which is perhaps the Haskell compiler generating the most efficient code. For a description of Haskell language see [8]. Part of the secret for that is the set of analysis and transformations carried out at the intermediate representation level. Another part of the explanation is the efficient design and implementation of the STG machine.

A high level description of the STG can be found in [6]. If the reader is interested in a more detailed view, then the only available information is the Haskell code of GHC (about 80.000 lines, 12.000 of which are devoted to the implementation of the STG machine) and the C code of its different runtime systems (more than 40.000 lines)[1].

In this paper we provide a step-by-step derivation of the STG machine, starting from a description higher-level than that of [6] and arriving at a description lower-level than that.

Our starting point is a commonly accepted operational semantics for lazy evaluation provided by Peter Sestoft in [10] as an improvement of John Launchbury's well-known definition in [4]. Then, we present the following refinements:

1. A new operational semantics, which we call semantics $S3$ —acknowledging that semantics 1 and 2 were defined by Mountjoy in a previous attempt [5]—, where normal forms may appear only in bindings.

2. A first machine, called STG-1, derived from $S3$ in which explicit replacement of pointers for variables is done in expressions.

3. A second machine STG-2 introducing environments in closures, **case** alternatives, and in the control expression.

4. A third machine, called ISTG (I stands for *imperative*) with a very small set of elementary instructions, each one very easy to be implemented in a conventional language such as C.

5. A translation from the language of STG-2 to the language of ISTG in which the data structures of STG-2 are represented (or implemented) by the ISTG data structures.

[*] Work partially supported by the Spanish project TIC 2000-0738.

PPDP'03, August 27–29, 2003, Uppsala, Sweden.

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

Figure 1: Launchbury's normalized λ-calculus

At each refinement, a formal proof of the soundness and completeness of the lower level with respect to the upper one is carried out [1] . In the end, the final implementation is shown correct with respect to Sestoft's operational semantics.

The main contribution of the work is showing that an efficient machine such as STG can be presented, understood, and formally reasoned about at different levels of abstraction. Also, there are some differences between the machine we arrive at and the actual STG machine implemented in the Glasgow Haskell Compiler. We argue that some design decisions in the actual STG machine are not properly justified.

The plan of the paper is as follows: after this introduction, in Section 2, a new language called FUN is introduced and the semantics $S3$ for this language is defined. Two theorems relating Launchbury's original language and semantics to the new ones are presented. Section 3 defines the two machines STG-1 and STG-2. Some propositions show the consistency between both machines and the correctness and completeness of STG-1 with respect to $S3$, eventhough the latter creates more closures in the heap and produces different (but equivalent) normal forms. Section 4 defines machine ISTG and Section 5 defines the translation from STG-2 expressions to ISTG instructions. Two invariants are proved which show the correctness of the translation. Section 6 discusses the differences between our translation and the actual implementation done by GHC. Finally, Section 7 concludes.

2. A NEW SEMANTICS FOR LAZY EVALUATION

We begin by reviewing the language and semantics given by Sestoft as an improvement to Launchbury's semantics. Both share the language given in Figure 1 where $\overline{A_i}$ denotes a vector $A_1, \ldots, A_n$ of subscripted entities. It is a normalized λ-calculus, extended with recursive **let**, constructor applications and **case** expressions. Sestoft's normalization process forces constructor applications to be saturated and all applications to only have variables as arguments. Weak head normal forms are either lambda abstractions or constructions. Throughout this section, w will denote (weak head) normal forms.

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

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

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

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

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

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

Figure 2: Sestoft's natural semantics

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 fresh pointers are created, freshness is understood w.r.t. $(\mathit{dom}\ \Gamma) \cup A$, where A contains the addresses of the closures under evaluation (see rule *Var*). The notation $\hat{e}$ in rule *Letrec* means the replacement of the variables x_i by the fresh pointers p_i. This is the only rule where new closures are created and added to the heap. We use the term *pointers* to refer to dynamically created free variables, bounded to expressions in the heap, and the term *variables* to refer to (lambda-bound, let-bound or case-bound) program variables. We consistently use $p, p_i, \ldots$ to denote free variables and $x, y, \ldots$ to denote program variables.

J. Mountjoy's [5] had the idea of changing Launchbury-Sestoft's language and 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 that normal forms were either constructions (as they were in Sestoft's semantics) or variables pointing to closures containing λ-abstractions, instead of just λ-abstractions. The reason for this was to forbid a λ-abstraction in the control expression as it happens in the STG machine. Another change was to force applications to have the form $x\ x_1$, i.e. consisting of a variable in the functional part. This is also what the STG language requires. These changes forced Mountjoy to modify the source language and to define a normalization from Launchbury's language to the new one. Mountjoy proved that the normalization did not change the normal forms arrived at by both semantics.

The second semantics, which we call semantics $S2$, forced applications to be done at once to n arguments instead of doing it one by one. Correspondingly, λ-abstractions were allowed to have several arguments. This is exactly what the STG machine requires. Semantics $S2$ was informally derived and contained some mistakes. In particular, (cf. [5, pag. 171]) rule App_M makes a λ-abstraction to appear in the control expression, in contradiction with the desire of having λ-abstractions only in the heap.

Completing and correcting Mountjoy's work we have defined a new semantics $S3$ in which the main changes in the source language w.r.t. Mountjoy's are the following:

1. We force constructor applications to appear only in bindings, i.e. in heap closures. Correspondingly, normal forms are variables pointing to either λ-abstractions or constructions. We will use the term *lambda forms* to refer to λ-abstractions or constructions alike. The motivation for this decision is to generate more efficient code as it will be seen in Section 5.1.

[1] The details of the proofs can be found in a technical report at one of the author's page http://dalila.sip.ucm.es/~albertoe.

$$
\begin{array}{lcll}
e & \rightarrow & e\ \overline{x_i}^n & \text{-- } n > 0\text{, application} \\
  & \mid & x & \text{-- variable} \\
  & \mid & \textbf{letrec}\ \overline{x_i = lf_i}\ \textbf{in}\ e & \text{-- recursive let} \\
  & \mid & \textbf{case}\ e\ \textbf{of}\ \overline{alt_i} & \text{-- case expression} \\
alt & \rightarrow & C\ \overline{x_j} \rightarrow e & \text{-- case alternative} \\
 & & & \\
lf & \rightarrow & \lambda\ \overline{x_i}^n.e & \text{-- } n > 0\text{, lambda abstraction} \\
  & \mid & C\ \overline{x_i}^n & \text{-- constructor application} \\
  & \mid & e & \text{-- expression}
\end{array}
$$

Figure 3: Language FUN

2. We relax applications to have the form $e\ \overline{x_i}^n$, where e is an arbitrary expression (excluding, of course, lambda forms). The initial motivation for this was not to introduce unjustified restrictions. In the conclusions we discuss that the generated code is also more efficient than the one produced by restricting applications to be of the form $x\ \overline{x_i}^n$.

Additionally, our starting point is Sestoft's semantics instead of Launchbury's. The main difference is that Sestoft substitutes fresh pointers for program variables in rule *Letrec* while Launchbury substitutes fresh variables for all bound variables in rule *Var* instead.

The syntax of the language, called FUN, is shown in Figure 3. Notice that applications are done to n arguments at once, being $e\ \overline{x_i}^n$ an abbreviation of $(\ldots(e\ x_1)\ldots)\ x_n$, and that consequently λ-abstractions may have several arguments. Its operational semantics, called $S3$ is given in Figure 4. For simplicity, we have eliminated the set A of pending updates appearing in Sestoft's semantics. This set is not strictly needed provided that the fresh name generator for pointers does not repeat previously generated names and provided that pointer's names are always distinguishable from program variables. In rule $Case_{S3}$, expression e_k is the righthand side expression of the alternative alt_k. The notation $\Gamma[p \mapsto e]$ highlights the fact that $(p \mapsto e) \in \Gamma$, and $\Gamma \cup [p \mapsto e]$ means the disjoint union of Γ and $(p \mapsto e)$. Please notice that this notation may not coincide with other notations in which Γ and $\Gamma[p \mapsto e]$ denote different heaps. Finally notice that, besides rule $Letrec_{S3}$, also rule App_{S3} creates closures in the heap.

Language FUN is at least as expressive as Launchbury's λ-calculus. The following normalization function transforms any Launchbury's expression into a semantically equivalent FUN expression.

Definition 1. We define the normalization function $\mathbb{N}$: Launch $\rightarrow$ FUN:

$$
\begin{array}{ll}
\mathbb{N}\ x & \stackrel{\text{def}}{=} x \\
\mathbb{N}\ (e\ x) & \stackrel{\text{def}}{=} (\mathbb{N}\ e)\ x \\
\mathbb{N}\ (\lambda\ x.e) & \stackrel{\text{def}}{=} \textbf{letrec}\ y\ =\ \mathbb{N}'\ (\lambda\ x.e)\ \textbf{in}\ y \\
 & \qquad\quad ,\ y\ \text{fresh} \\
\mathbb{N}\ (C\ \overline{x_i}) & \stackrel{\text{def}}{=} \textbf{letrec}\ y\ =\ C\ \overline{x_i}\ \textbf{in}\ y \\
 & \qquad\quad ,\ y\ \text{fresh} \\
\mathbb{N}\ (\textbf{letrec}\ \overline{x_i\ =\ e_i}^n\ \textbf{in}\ e) & \stackrel{\text{def}}{=} \textbf{letrec}\ \overline{x_i\ =\ \mathbb{N}'\ e_i}^n\ \textbf{in}\ \mathbb{N}\ e \\
\mathbb{N}\ (\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \rightarrow e_i}) & \stackrel{\text{def}}{=} \textbf{case}\ \mathbb{N}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \rightarrow \mathbb{N}\ e_i}
\end{array}
$$

$\mathbb{N}',\ \mathbb{N}''$: Launch $\rightarrow$ FUN:

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

The following proposition prove that the normalization functions are well defined.

PROPOSITION 1.

1. *Let* $e \in$ *Launch then* $\mathbb{N}\ e \in$ *FUN*
2. *Let* $e \in lf$ *then* $\mathbb{N}'\ e \in lf$
3. *Let* $e \neq \lambda x.e'$ *and* $e \neq C\ \overline{x_i}$ *then,* $\mathbb{N}\ e = \mathbb{N}'\ e$
4. $(\mathbb{N}\ e)[\overline{p_i/x_i}] = \mathbb{N}\ (e[\overline{p_i/x_i}])$
5. $(\mathbb{N}'\ e)[\overline{p_i/x_i}] = \mathbb{N}'\ (e[\overline{p_i/x_i}])$

PROOF.

1. By structural induction on e.
2. Trivial.
3. Trivial.
4. By definition of $\mathbb{N}$ and of substitutions.
5. By definition of $\mathbb{N}'$ and of substitutions.

□

To see that both semantics reduce an expression to equivalent normal forms, first we prove that the normalization does not change the meaning of an expression within Sestoft's semantics. Then, we prove that both semantics reduce any FUN expression to equivalent normal forms, provided that such normal forms exist.

2.1 Soundness and completeness

The following two propositions prove that the normalization does not change the meaning of an expression. We use the following notation: α denotes a one-to-one renaming of pointers, and $\Gamma \subseteq_\alpha \Gamma'$ means that $\alpha\ \Gamma \subseteq \Gamma'$. This is needed to express the equivalence between the heaps of both semantics up to some renaming α. As $S3$ generates more closures than Sestoft's, it is not possible to guarantee that the fresh pointers are exactly the same in the two heaps.

PROPOSITION 2. *(Sestoft* $\Rightarrow$ *Sestoft*) For all* $e \in$ Launch *we have:*

$$
\{\,\} : e \Downarrow \Delta : w \Rightarrow \left\{ \begin{array}{ll} \{\,\} : \mathbb{N}\ e \Downarrow \Delta^* : w^* & \\ \exists\alpha. & \begin{array}{l} w^* = \mathbb{N}'\ (\alpha\ w) \\ \mathbb{N}'\ \Delta \subseteq_\alpha \Delta^* \end{array} \end{array} \right.
$$

PROOF. By induction on the number of reductions of Launchbury expressions. □

PROPOSITION 3. *(Sestoft** $\Rightarrow$ *Sestoft) For all* $e \in$ FUN *we have:*

$$
\{\,\} : e \Downarrow \Delta : w \Rightarrow
$$

$$
\forall e' \in \text{Launch}. \left(\mathbb{N}\ e' = e \Rightarrow \left\{ \begin{array}{ll} \{\,\} : e' \Downarrow \Delta' : w' & \\ \exists\alpha. & \begin{array}{l} \mathbb{N}'\ (\alpha\ w') = w \\ \mathbb{N}'\ \Delta' \subseteq_\alpha \Delta \end{array} \end{array} \right. \right)
$$

$$\Gamma[p \mapsto \lambda \overline{x_i}^n.e] : p \Downarrow \Gamma : p \qquad \mathit{Lam}_{S3}$$

$$\Gamma[p \mapsto C\,\overline{p_i}] : p \Downarrow \Gamma : p \qquad \mathit{Cons}_{S3}$$

$$\frac{\Gamma : e \Downarrow \Delta[p \mapsto \lambda \overline{x_i}^n.\lambda \overline{y_i}^m.e'] : p}{\Gamma : e\ \overline{p_i}^n \Downarrow \Delta \cup [q \mapsto \lambda \overline{y_i}^m.e'[\overline{p_i/x_i}^n]] : q}\ m, n > 0,\ \text{q fresh} \qquad \mathit{App}_{S3}$$

$$\frac{\Gamma : e \Downarrow \Delta[p \mapsto \lambda \overline{x_i}^m.e'] : p \quad \Delta : e'[\overline{p_i/x_i}^m]\ p_{m+1} \ldots p_n \Downarrow \Theta[q \mapsto w] : q}{\Gamma : e\ \overline{p_i}^n \Downarrow \Theta : q}\ n \geq m \qquad \mathit{App}'_{S3}$$

$$\frac{\Gamma : e \Downarrow \Delta[q \mapsto w] : q}{\Gamma \cup [p \mapsto e] : p \Downarrow \Delta \cup [p \mapsto w] : q} \qquad \mathit{Var}_{S3}$$

$$\frac{\Gamma \cup [\overline{p_i \mapsto l\hat{f}_i}] : \hat{e} \Downarrow \Delta[p \mapsto w] : p}{\Gamma : \textbf{letrec}\ \overline{x_i = lf_i}\ \textbf{in}\ e \Downarrow \Delta : p}\ \overline{p_i}\ \text{fresh} \qquad \mathit{Letrec}_{S3}$$

$$\frac{\Gamma : e\ \Downarrow \Delta[p \mapsto C_k\ \overline{p_j}] : p \qquad \Delta : e_k[\overline{p_j/y_{kj}}] \Downarrow \Theta[q \mapsto w] : q}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to e_i} \Downarrow \Theta : q} \qquad \mathit{Case}_{S3}$$

Figure 4: Semantics $S3$

Proof. By induction on the number of reductions of Launchbury expressions. □

Now we prove the equivalence between the two semantics. We consider only FUN expressions because it has been proved that the normalization does not change the meaning of an expression.

Proposition 4. *(Sestoft ⇒ $S3$, completeness of $S3$) For all $e \in$ FUN we have:*

$$\{\,\} : e \Downarrow \Delta : w \Rightarrow \begin{cases} \{\,\} : e \downarrow \Delta'[p \mapsto w'] : p \\ \exists \alpha. \quad \begin{array}{l} \alpha\ w = w' \\ \Delta \subseteq_\alpha \Delta' \end{array} \end{cases}$$

Proof. By induction on the number of reductions of FUN expressions. □

Proposition 5. *($S3$ ⇒ Sestoft, soundness of $S3$) For all $e \in$ FUN we have:*

$$\{\,\} : e \downarrow \Delta[p \mapsto w] : p \Rightarrow \begin{cases} \{\,\} : e \Downarrow \Delta' : w' \\ \exists \alpha. \quad \begin{array}{l} \alpha\ w' = w \\ \Delta' \subseteq_\alpha \Delta \end{array} \end{cases}$$

Proof. By induction on the number of reductions of FUN expressions. □

Once adapted the source language to the STG language, we are ready to derive an STG-like machine from semantics $S3$.

3. A VERY SIMPLE STG MACHINE

Following a similar approach to Sestoft MARK-1 machine [10], we first introduce a very simple STG machine, which we will call STG-1, in which explicit variable substitutions are done. A configuration in this machine is a triple (Γ, e, S) where Γ represents the heap, e is the control expression and S is the stack. The heap binds pointers to lambda forms which, in turn, may reference other pointers. The stack stores three kinds of objects: arguments p_i of pending applications, case alternatives *alts* of pending pattern matchings, and marks $\#p$ of pending updates.

In Figure 5, the transitions of the machine are shown. They look very close to the lazy semantics $S3$ presented in Section 2. For instance, the single rule for **letrec** in Figure 5 is a literal transcription of the Letrec_{S3} rule of Figure 4. The semantic rules for **case** and applications are split each one into two rules in the machine. The semantic rule for variable is also split into two in order to take care of updating the closure. So, in principle, an execution of the STG-1 machine could be regarded as the linearization of the semantic derivation tree by introducing an auxiliary stack.

But sometimes appearances are misleading. The theorem below shows that in fact STG-1 builds less closures in the heap than the semantics and it may arrive to different (but semantically equivalent) normal forms. In order to prove the soundness and completeness of STG-1, we first enrich the semantics with a stack parameter S in the rules. The new rules for $\Downarrow_S$ (only those which modify S) are shown in Figure 6. It is trivial to show that the rules are equivalent to the ones in Figure 4 as the stack is just an observation of the derivations. It may not influence them. The following theorem establishes the correspondence between the (enriched) semantics and the machine.

Proposition 6. *Given Γ, e and S, then $\Gamma : e \Downarrow_S \Delta[p \mapsto w] : p$ iff $(\Gamma, e, S) \longrightarrow^* (\Delta', p', S')$, where*

1. *$\Delta' \subseteq \Delta$*
2. *if $\Delta[p \mapsto C\ \overline{p_i}^n]$ then $S = S', p = p'$ and $\Delta'[p' \mapsto C\ \overline{p_i}^n]$*
3. *if $\Delta[p \mapsto \lambda \overline{x_i}^n.e']$ then there exists $m \geq 0$ s.t. $\Delta'[p' \mapsto \lambda \overline{y_i}^m.\lambda \overline{x_i}^n.e'']$ and $S' = \overline{q_i}^m : S$ and $e' = e''[\overline{q_i/y_i}^m]$*

	Heap	Control	Stack	rule
	Γ	$\mathbf{letrec}\ \{\overline{x_i = \mathit{lf}_i}\}\ \mathbf{in}\ e$	S	letrec ([1])
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto \mathit{lf}_i[\overline{p_j/x_j}]}]$	$e[\overline{p_i/x_i}]$	S	
	Γ	$\mathbf{case}\ e\ \mathbf{of}\ \mathit{alts}$	S	case1
$\Longrightarrow$	Γ	e	$\mathit{alts} : S$	
	$\Gamma[p \mapsto C_k\ \overline{p_i}]$	p	$\overline{C_j\ \overline{y_{ji}} \rightarrow e_j} : S$	case2
$\Longrightarrow$	Γ	$e_k[\overline{p_i/y_{ki}}]$	S	
	Γ	$e\ \overline{p_i}^n$	S	app1
$\Longrightarrow$	Γ	e	$\overline{p_i}^n : S$	
	$\Gamma[p \mapsto \lambda \overline{x_i}^n.e]$	p	$\overline{p_i}^n : S$	app2
$\Longrightarrow$	Γ	$e[\overline{p_i/x_i}^n]$	S	
	$\Gamma \cup [p \mapsto e']$	p	S	var1
$\Longrightarrow$	Γ	e'	$\#p : S$	
	$\Gamma[p \mapsto \lambda \overline{x_i}^k.\lambda \overline{y_i}^m.e]$	p	$\overline{p_i}^k : \#q : S$	var2
$\Longrightarrow$	$\Gamma \cup [q \mapsto p\ \overline{p_i}^k]$	p	$\overline{p_i}^k : S$	
	$\Gamma[p \mapsto C_k\ \overline{p_i}]$	p	$\#q : S$	var3
$\Longrightarrow$	$\Gamma \cup [q \mapsto C_k\ \overline{p_i}]$	p	S	

([1]) $\overline{p_i}$ are distinct and fresh w.r.t. Γ, $\mathbf{letrec}\ \{\overline{x_i = \mathit{lf}_i}\}\ \mathbf{in}\ e$, and S

Figure 5: The STG-1 Machine

Proof. By induction on the number of reductions of FUN expressions. □

The proposition shows that the semantic rule *App$_{S3}$* of Figure 4 is not literally transcribed in the machine. The machine does not create intermediate lambdas in the heap unless an update is needed. Rule *app2* in Figure 5 applies a lambda always to *all* its arguments provided that they are in the stack. For this reason, a lambda with more parameters than that of the semantics may be arrived at as normal form of a functional expression. Also for this reason, an update mark may be interspersed with arguments in the stack when a lambda is reached (see rule *var2* in Figure 5). A final implication is that the machine may stop with m pending arguments in the stack and a variable in the control expression pointing to a lambda with $n > m$ parameters. The semantics always ends a derivation with a variable as normal form and an empty stack.

Again, following Sestoft and his MARK-2 machine, once we have proved the soundness and completeness of STG-1, we introduce STG-2 having environments instead of explicit variable substitutions. Also, we add trimmers to this machine so that environments kept in closures and in **case** alternatives only reference the free variables of the expression instead of *all* variables in scope. A configuration of STG-2 is a quadruple (Γ, e, E, S) where E is the environment of e, the alternatives are pairs (alts, E), and a closure is a pair (lf, E). Now expressions and lambda forms keep their original variables and the associated environment maps them to pointers in the heap. The notation $E\mid^t$ means the trimming of environment E to the trimmer t. A trimmer is just a collection of variable names. The resulting machine is shown in Figure 7.

Proposition 7. *Given a closed expression e_0. $(\{\,\}, e_0, [\,]) \xrightarrow{STG\text{-}1} (\Delta, q, \overline{p_i}^n)$ where either:*

- $\Delta[q \mapsto C\ \overline{q_i}^m] \ \wedge\ n = 0$
- *or* $\Delta[q \mapsto \lambda \overline{x_i}^m.e] \ \wedge\ m > n \geq 0$

if and only if $(\{\,\}, e_0, \{\,\}, [\,]) \xrightarrow{STG\text{-}2} (\Gamma, x, E[x \mapsto q], \overline{p_i}^n))$ *and either:*

- $\Gamma[q \mapsto (C\ \overline{x_i}^m, \{\overline{x_i \mapsto q_i}^m\})] \ \wedge\ n = 0$
- *or* $\Gamma[q \mapsto (\lambda \overline{x_i}^m.e', E')] \ \wedge\ m > n \geq 0 \ \wedge\ e = E'\ e'$.

Proof. By induction on the number of reductions. □

4. AN IMPERATIVE STG MACHINE

In this Section we 'invent' an imperative STG machine, called ISTG, by defining a set of machine instructions and an operational semantics for them in terms of the state transition that each instruction produces. In fact, this machine tries 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 [6] we find only a 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 gap to be bridged is too high. Moreover, it is not possible to reason about the correctness of the implementation when so many details are introduced at once. The ISTG architecture has been inspired by the actual implementation of the STG machine done by GHC, and the ISTG instructions have been derived from the STG-2 machine by analyzing the elementary steps involved in every machine transition.

An ISTG machine configuration consists of a 5-tuple $(\mathit{is}, S, \mathit{node}, \Gamma, \mathit{cs})$, where *is* is a machine instruction sequence ended with the instruction ENTER or RETURNCON, S is the stack, *node* is a heap pointer pointing to the closure under execution (the one to which *is* belongs to), Γ is the heap and *cs* is a *code store* where the instruction sequences resulting from compiling the program expressions are kept.

We will use the following notation: a for pointers to closures in Γ, *as* and *ws* for lists of such pointers, and p for

$$\frac{\Gamma : e \downarrow_{\overline{p_i}^n : S} \Delta[p \mapsto \lambda \overline{x_i}^n . \lambda \overline{y_i}^m . e'] : p}{\Gamma : e \ \overline{p_i}^n \downarrow_S \Delta \cup [q \mapsto \lambda \overline{y_i}^m . e'[\overline{p_i / x_i}^n]] : q} \ m, n > 0, \text{ q fresh} \qquad App_{S3}$$

$$\frac{\Gamma : e \downarrow_{\overline{p_i}^n : S} \Delta[p \mapsto \lambda \overline{x_i}^m . e'] : p \quad \Delta : e'[\overline{p_i / x_i}^m] \ p_{m+1} \dots p_n \downarrow_S \Theta[q \mapsto w] : q}{\Gamma : e \ \overline{p_i}^n \downarrow_S \Theta : q} \ n \geq m \qquad App'_{S3}$$

$$\frac{\Gamma : e \downarrow_{\# p : S} \Delta[q \mapsto w] : q}{\Gamma \cup [p \mapsto e] : p \downarrow_S \Delta \cup [p \mapsto w] : q} \qquad Var_{S3}$$

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

Figure 6: The enriched semantics

pointers to code fragments in cs. By $cs[p \mapsto is]$ we denote that the code store cs maps pointer p to the instruction sequence is and, by $cs[p \mapsto \overline{is_i}^n]$, that cs maps p to a vectored set of instruction sequences $is_1, \dots, is_n$, each one corresponding to an alternative of a **case** expression with n constructors $C_1, \dots, C_n$. 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.

Stack S may store pointers a to closures in Γ, pointers p to code sequences and code alternatives in cs, and update marks $\#a$ indicating that closure pointed to by a must be updated. A closure is 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.

These representation decisions are very near to the GHC implementation. In its runtime system all these elements (stack, heap, node register and code) are present [9]. Our closures are also a small simplification of theirs.

In Figure 8, the ISTG machine instructions and its operational semantics are shown. The machine instructions BUILDENV, PUSHALTS and UPDTMARK roughly correspond to the three possible pushing actions of machine STG-2. The SLIDE instruction has no clear correspondence in the STG-2. As we will see in Section 5, it will be used to change the current environment when a new closure is entered. Instructions ALLOC and BUILDCLS will 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 ENTER and RETURNCON are typical of the actual STG machine as described in [6]. It is interesting to note that it has been possible to describe our previous STG machines without any reference to them. In our view, they belong to ISTG, i.e. to a lower level of abstraction. Finally, instruction ARGCHECK, which implements updates with lambda normal forms, is here at the same level of abstraction as RETURNCON, which implements updates with constructions normal forms. Predefined code is stored in cs for updating with a partial application and for blackholing a closure under evaluation. The corresponding code pointers are respectively called p_{pap}^{n+1} and p_{bh} in Figure 8. The associated code is the following:

$$\begin{array}{lcl} p_{bh} & = & [\,] \\ p_{pap}^{n+1} & = & [\mathit{BUILDENV} \ [(\mathit{NODE}, 1), \dots, (\mathit{NODE}, n+1)\ ], \\ & & \ \mathit{ENTER}\ ] \end{array}$$

The code of a blackhole just blocks the machine as there is no instruction to execute. There is predefined code for partial applications with different values for n. The code just copies the closure into the stack and jumps to the first pointer that is assumed to be pointing to a λ-abstraction closure.

The translation to C of the 9 instructions of the ISTG should appear straightforward for the reader. For instance, BUILDCLS and BUILDENV can be implemented by a sequence of assignments, copying values to/from the stack an the heap; PUSHALTS, UPDTMARK and ENTER do straightforward stack manipulation; SLIDE is more involved but can be easily translated to a sequence of loops moving information within the stack to collapse a number of stack fragments. The more complex ones are RETURNCON and ARGCHECK. Both contains a loop which updates the heap with normal forms (respectively, constructions and partial applications) as long as they encounter update marks in the stack. Finally, the installation of a new instruction sequence in the control made by ENTER and RETURNCON are implemented by a simple jump.

5. FORMAL TRANSLATION FROM STG-2 TO ISTG

In this Section, we provide first the translation schemes for the FUN expressions and lambda forms and then prove that this translation correctly implements the STG-2 machine on top of the ISTG machine. Before embarking into the details, we give some hints to intuitively understand the translation:

- The ISTG stack will represent not only the STG-2 stack, but also (part of) the current environment E and all the environments associated to pending **case** alternatives. So, care must be taken to distinguish between environments and other objects in the stack.

- The rest of the current environment E is kept in the current closure. The translation knows where each free

	Heap	Control	Environment	Stack	rule
	Γ	**letrec** $\{\overline{x_i = \mathit{lf}_i \vert^{t_i}}\}$ **in** e	E	S	letrec (^{1})
$\Rightarrow$	$\Gamma \cup [\overline{p_i \mapsto (\mathit{lf}_i, E' \vert^{t_i})}]$	e	E'	S	
	Γ	**case** e **of** $\mathit{alts} \vert^t$	E	S	case1
$\Rightarrow$	Γ	e	E	$(\mathit{alts}, E \vert^t) : S$	
	$\Gamma[p \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto p\}$	$(\mathit{alts}, E') : S$	case2 (^{2})
$\Rightarrow$	Γ	e_k	$E' \cup \{\overline{y_{ki} \mapsto p_i}\}$	S	
	Γ	$e\ \overline{x_i}^n$	$E\{\overline{x_i \mapsto p_i}^n\}$	S	app1
$\Rightarrow$	Γ	e	E	$\overline{p_i}^n : S$	
	$\Gamma[p \mapsto (\lambda \overline{x_i}^n.e, E')]$	x	$E\{x \mapsto p\}$	$\overline{p_i}^n : S$	app2
$\Rightarrow$	Γ	e	$E' \cup \{\overline{x_i \mapsto p_i}^n\}$	S	
	$\Gamma \cup [p \mapsto (e, E')]$	x	$E\{x \mapsto p\}$	S	var1
$\Rightarrow$	Γ	e	E'	$\#p : S$	
	$\Gamma[p \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E')]$	x	$E\{x \mapsto p\}$	$\overline{p_i}^k : \#q : S$	var2 (^{3})
$\Rightarrow$	$\Gamma \cup [q \mapsto (x\ \overline{x_i}^k, E''])$	x	E	$\overline{p_i}^k : S$	
	$\Gamma[p \mapsto (C_k\ \overline{x_i}, E')]$	x	$E\{x \mapsto p\}$	$\#q : S$	var3
$\Rightarrow$	$\Gamma \cup [q \mapsto (C_k\ \overline{x_i}, E')]$	x	E	S	

(^{1}) $\overline{p_i}$ are distinct and fresh w.r.t. Γ, **letrec** $\{\overline{x_i = \mathit{lf}_i}\}$ **in** e, and S. $E' = E \cup \{\overline{x_i \mapsto p_i}\}$
(^{2}) Expression e_k corresponds to alternative $C_k\ \overline{y_{ki}} \to e_k$ in *alts*
(^{3}) $E'' = \{x \mapsto p, \overline{x_i \mapsto p_i}^k\}$

Figure 7: The STG-2 machine

variable is located by maintaining two compile-time environments ρ and η. The first one ρ corresponds to the environment kept in the stack, while the second one η corresponds to the free variables accessed through the *node* pointer.

- The stack can be considered as divided into big blocks separated by code pointers p pointing to case alternatives. Each big block topped with such a pointer corresponds to the environment of the associated alternatives.
- In turn, each big block can be considered as divided into small blocks, each one topped with a set of arguments of pending applications. The compile-time environment ρ is likewise divided into big and small blocks, so reflecting the stack structure.
- When a variable is reached in the current instruction sequence, an ENTER instruction is executed. This will finish the current sequence and start a new one. The upper big block of the stack must be deleted (corresponding to changing the current environment) but arguments of pending applications must be kept. This stack restructuring is accomplished by a SLIDE operation with an appropriate argument.

Definition 2. A stack environment ρ is a list $[(\delta_k, m_k, n_k), \ldots, (\delta_1, m_1, n_1)]$ of blocks. It describes the variables in the stack starting from the top. In a block (δ, m, n), δ is an environment mapping exactly $m- \mid n \mid$ program variables to disjoint numbers in the range $1..m- \mid n \mid$. The empty environment, denoted $\rho_\emptyset$ is the list $[(\{\}, 0, 0)]$.

A block (δ, m, n) corresponds to a small block in the above explanation. Blocks with $n = -1$, are topped with a code pointer pointing to alternatives. So, they provide a separation between big blocks. The upper big block consists of all the small blocks up to (and excluding) the first small block with $n = -1$. Blocks with $n > 0$ have $m - n$ free variables and are topped with n arguments of pending applications. The upper block is the only one with $n = 0$ meaning that it is not still closed and that it can be extended.

Definition 3. A closure environment η with n variables is a mapping from these variables to disjoint numbers in the range $1..n$.

Definition 4. The offset of a variable x in ρ from the top of the stack, denoted $\rho\ x$, is given by

$$\rho\ x \overset{\text{def}}{=} (\sum_{i=l}^{k} m_i) - \delta_l\ x, \quad \text{being} \quad x \in \mathit{dom}\ \delta_l$$

If the initial closed expression to be translated has different names for bound variables, then the compile time environments ρ and η will never have duplicate names. It will be proved below that every free variable of an expression being compiled will necessarily be either in ρ or in η, and never in both. This allows us to introduce the notation $(\rho, \eta)\ x$ to mean

$$(\rho, \eta)\ x \overset{\text{def}}{=} \begin{array}{ll} (\mathit{STACK}, \rho\ x) & \text{if } x \in \mathit{dom}\ \rho \\ (\mathit{NODE}, \eta\ x) & \text{if } x \in \mathit{dom}\ \eta \end{array}$$

The stack environment may suffer a number of operations: closing the current small block with a set of arguments, enlarging the current small block with new bindings, and closing the current big block with a pointer to **case** alternatives. These are formally defined as follows.

Definition 5. The following operations with stack environments are defined:

1. $((\delta, m, 0) : \rho) + n \overset{\text{def}}{=} (\{\}, 0, 0) : (\delta, m + n, n) : \rho$

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

$(^1)$ a_m is a pointer to a new closure with space for m free variables, and Γ' is the resulting heap after the allocation

$(^2)$ $a_i = \begin{array}{ll} S!i & \text{if } z_i = (\mathit{STACK}, i) \\ \mathit{node}_\Gamma!i & \text{if } z_i = (\mathit{NODE}, i) \end{array}$

Figure 8: The ISTG machine

2. $((\delta, m, 0) : \rho) + (\{\overline{x_i \mapsto j_i}^n\}, n) \overset{\text{def}}{=} (\delta \cup \{\overline{x_i \mapsto m + j_i}^n\}, m + n, 0) : \rho$

3. $((\delta, m, 0) : \rho)^{+} \overset{\text{def}}{=} (\{\}, 0, 0) : (\delta, m + 1, -1) : \rho$

5.1 Translation schemes

Functions *trE* and *trA* respectively translate a FUN expression and a **case** alternative to a sequence of ISTG machine instructions; function *trAs* translates a set of alternatives to a pointer to a vectored set of machine instruction sequences in the code store; and function *trB* translates a lambda form to a pointer to a machine instruction sequence in the code store. The translation schemes are shown in Figure 9.

The notation $\ldots$ & $cs[p \mapsto \ldots]$ means that the corresponding translation scheme has a side effect which consists of creating a code sequence in the code store cs and pointing it by the code pointer p.

Proposition 8. *(static invariant) Given a closed expression e_0 with different bound variables and an initial call trE e_0 $\rho_\emptyset$ $\{\}$, in all internal calls of the form trE e ρ η:*

1. *The stack environment has the form $\rho = (\delta, m, 0) : \rho'$. Moreover, there is no other block (δ', m', n) in ρ' with $n = 0$. Consequently, all environment operations in the above translation are well defined.*

2. *All free variables of e are defined either in ρ or in η. Moreover, dom $\rho \cap$ dom $\eta = \emptyset$.*

3. *The last instruction generated for e is ENTER. Consequently, the main instruction sequence and all sequences corresponding to* **case** *alternatives and to non-constructor closures, end in an ENTER.*

Proof. (1) and (2) are proved by induction on the tree structure of calls to *trE*; (3) is proved by structural induction on FUN expressions. □

In order to prove the correctness of the translation, we only need to consider ISTG machine configurations of the form $(is, S_I, node, \Gamma_I, cs)$ in which is is generated by a call to *trE* for some expression e and environments ρ, η. We call these *stable* configurations. We enrich then these configurations with three additional components: the environments

$$
\begin{array}{lcl}
\mathit{trE}\ (e\ \overline{x_i}^n)\ \rho\ \eta & = & [\mathit{BUILDENV}\ \overline{(\rho,\eta)\ x_i}^n] \mathbin{+\!\!+} \\
 & & \mathit{trE}\ e\ (\rho + n)\ \eta
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{trE}\ (\textbf{case}\ e\ \textbf{of}\ \mathit{alts}\ |^{\overline{x_i}^n})\ \rho\ \eta & = & [\mathit{BUILDENV}\ \mathit{zs}, \mathit{PUSHALTS}\ p] \mathbin{+\!\!+} \mathit{trE}\ e\ \rho'^{+\!\!+}\ (\eta - \mathit{xs})
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & p & = & \mathit{trAs}\ \mathit{alts}\ \rho' \\
 & \rho' & = & \rho + (\{\overline{\mathit{xs}_j \mapsto m - j + 1}^m\}, m) \\
 & \mathit{xs} & = & [x \mid x \leftarrow \overline{x_i}^n \wedge x \in \mathit{dom}\ \eta] \\
 & \mathit{zs} & = & [(\mathit{node}, \eta\ x) \mid x \leftarrow \mathit{xs}] \\
 & m & = & |\,\mathit{xs}\,|
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{trE}\ (\textbf{letrec}\ \overline{x_i = \mathit{lf}_i\ |^{\overline{y_{ij}}^{m_i}}}^n\ \textbf{in}\ e)\ \rho\ \eta & = & [\mathit{ALLOC}\ m_n, \ldots, \mathit{ALLOC}\ m_1] \mathbin{+\!\!+} \\
 & & [\overline{\mathit{BUILDCLS}\ (i-1)\ p_i\ \mathit{zs}_i}^n] \mathbin{+\!\!+} \\
 & & \mathit{trE}\ e\ \rho'\ \eta
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & \rho' & = & \rho + (\{\overline{x_i \mapsto n - i + 1}^n\}, n) \\
 & p_i & = & \mathit{trB}\ (\mathit{lf}_i\ |^{\overline{y_{ij}}^{m_i}}), \qquad i \in \{1..n\} \\
 & \mathit{zs}_i & = & \overline{(\rho', \eta)\ y_{ij}}^{m_i}, \qquad i \in \{1..n\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{trE}\ x\ \rho\ \eta & = & [\mathit{BUILDENV}\ [(\rho,\eta)\ x], \\
 & & \ \mathit{SLIDE}\ ((1,0) : \mathit{ms}), \\
 & & \ \mathit{ENTER}]
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & \mathit{ms} & = & \mathit{map}\ (\backslash(\_, m, n) \rightarrow (n, m - n))\ (\mathit{takeWhile}\ \mathit{nn}\ \rho) \\
 & \mathit{nn}\ (\_, m, -1) & = & \mathit{False} \\
 & \mathit{nn}\ \_ & = & \mathit{True}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{trAs}\ (\overline{\mathit{alt}_i}^n)\ \rho & = p\ \ \&\ \ \mathit{cs}[p \mapsto \overline{\mathit{trA}\ \mathit{alt}_i\ \rho}^n] \\
 & \\
\mathit{trA}\ (C\ \overline{x_i}^n \rightarrow e)\ \rho & = \mathit{trE}\ e\ \rho\ \{\overline{x_i \mapsto i}^n\} \\
 & \\
\mathit{trB}\ (C_k^n\ \overline{x_i}^n\ |^{\overline{x_i}^n}) & = p\ \ \&\ \ \mathit{cs}[p \mapsto [\mathit{RETURNCON}\ C_k^n]] \\
 & \\
\mathit{trB}\ (\lambda \overline{x_i}^l.e\ |^{\overline{y_j}^n}) & = p\ \ \&\ \ \mathit{cs}[p \mapsto [\mathit{ARGCHECK}\ l] \mathbin{+\!\!+} \mathit{trE}\ e\ \rho\ \eta]
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & \rho & = & [(\{\overline{x_i \mapsto l - i + 1}^l\}, l, 0)] \\
 & \eta & = & \{\overline{y_j \mapsto j}^n\}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{trB}\ (e\ |^{\overline{y_j}^n}) & = p\ \ \&\ \ \mathit{cs}[p \mapsto [\mathit{UPDTMARK}] \mathbin{+\!\!+} \mathit{trE}\ e\ \rho_\emptyset\ \eta]
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & \eta & = & \{\overline{y_j \mapsto j}^n\}
\end{array}
$$

Figure 9: Translation schemes from STG-2 to ISTG

ρ and η used to generate *is*, and an environment stack S_{env} containing a sequence of stack environments. The environments in S_{env} are in one to one correspondence with **case** pointers stored in S_I. Initially S_{env} is empty. Each time an instruction PUSHALTS is executed (see *trE* definition for **case**), the environment ρ' the corresponding alternatives are compiled with, is pushed onto stack S_{env}. Each time a RETURNCON pops a **case** pointer, stack S_{env} is also pop-ed. So, enriched ISTG configurations have the form $(is, \rho, \eta, S_I, S_{env}, node, \Gamma_I, cs)$.

Definition 6. A STG-2 environment E is equivalent to an ISTG environment defined by $\rho, \eta, S_I, \Gamma_I$ and *node*, denoted $E \equiv (\rho, S_I, \eta, \Gamma_I, node)$ if $dom\ E \subseteq dom\ \rho \cup dom\ \eta$ and

$$
\forall x \in dom\ E \quad
\begin{array}{ll}
E\ x = S_I\ !\ (\rho\ x) & \text{if } x \in dom\ \rho \\
E\ x = node_{\Gamma_I}\ !\ (\eta\ x) & \text{if } x \in dom\ \eta
\end{array}
$$

Definition 7. A STG-2 stack S is equivalent to a triple (ρ, S_I, S_{env}) of an ISTG enriched configuration, denoted $S \equiv (\rho, S_I, S_{env})$, if

1. Whenever $\rho = (\delta, m, 0) : \rho'$, then $S_I = \overline{a_i}^m : S'_I$ and $S \equiv (\rho', S'_I, S_{env})$
2. Whenever $\rho = (\delta, m, n) : \rho', n > 0$, then $S = \overline{a_i}^n : S'$, $S_I = \overline{a_i}^n : \overline{b_j}^{m-n} : S'_I$ and $S' \equiv (\rho', S'_I, S_{env})$
3. Whenever $\rho = (\delta, m, -1) : \rho'$, then $S = (alts, E) : S'$, $S_I = p_{alts} : S'_I$, $S_{env} = \rho_{alts} : S'_{env}$, $p_{alts} = trAs\ alts\ \rho_{alts}$, $E \equiv (\rho_{alts}, S'_I, \_, \_, \_)$ and $S' \equiv (\rho_{alts}, S'_I, S'_{env})$
4. Whenever $S = \#a : S'$ and $S_I = \#a : S'_I$, then $S' \equiv (\rho, S'_I, S_{env})$
5. Additionally, $[\,] \equiv (\{\}, [\,], [\,])$

Definition 8. A STG-2 heap Γ is equivalent to an ISTG pair (Γ_I, cs), denoted $\Gamma \equiv (\Gamma_I, cs)$, if for all p we have $\Gamma[p \mapsto (\mathit{lf}\ |^{\overline{x_i}^n}, E)]$ if and only if $\Gamma_I[p \mapsto (q, ws)]$, $cs[q \mapsto is]$, $is = trB\ (\mathit{lf}\ |^{\overline{x_i}^n})$ and $ws = \overline{E\ x_i}^n$.

Definition 9. A STG-2 configuration is equivalent to an ISTG enriched stable configuration, denoted $(\Gamma, e, E, S) \equiv (is, \rho, \eta, S_I, S_{env}, node, \Gamma_I, cs)$ if

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

PROPOSITION 9. *(dynamic invariant) Given a closed expression* e_0 *with different bound variables and initial STG-2 and ISTG configurations, respectively* $(\{\}, e_0, \{\}, [\,])$ *and* $(trE\ e_0\ \rho_\emptyset\ \{\}, \rho_\emptyset, \{\}, [\,], [\,], \bot, \{\}, cs)$*, where* cs *is the code store generated by the whole translation of* e_0*, then both machines evolve through equivalent configurations.*

PROOF. By induction on the number of transitions of both machines. Only transitions between ISTG stable configurations are considered. □

COROLLARY 10. *The translation given in Section 5.1 is correct.*

6. DIFFERENCES WITH THE ACTUAL STG MACHINE

There are some differences between the machine translation presented in Section 5 and the actual code generated by GHC. Some are just omissions, other are non-substantial differences and some other are deeper ones.

In the first group it is the treatment of basic values, very elaborated in GHC (see for example [3]) and completely ignored here. We have preferred to concentrate our study in the functional kernel of the machine but, of course, a formal reasoning about this aspect is a clear continuation of our work.

In the second group it is the optimization of update implementation. In GHC, updates can be done either by indirection or by closure creation, depending on whether there is enough space or not in the old closure to do update in place. This implies to keep closure size information somewhere. GHC keeps it in the so called *info table*, a static part shared by all closures created from the same bind. This table forces an additional indirection to access the closure code. Our model has simplified these aspects. We understand also that stack restructuring, as the one performed by our SLIDE instruction, is not implemented in this way by GHC. Apparently, stubbing of non used stack positions is done instead. An efficiency study could show which implementation is better. The cost of our SLIDE instruction is in $O(n)$, being n the number of arguments to be preserved in the stack when the current environment is discarded.

Perhaps the deeper difference between our derived machine and the actual STG is our insistence in that FUN applications should have the form $e\ \overline{x_i}^n$ instead of $x\ \overline{x_i}^n$ as it is the case in the STG language. This decision is not justified in the GHC papers and perhaps could have a noticeable negative impact in performance. In a lazy language, the functional part of an application should be eagerly evaluated, but GHC does it lazily. This implies constructing a number of closures that will be immediately entered (and perhaps updated afterwards), with a corresponding additional cost both in space and time. Our translation avoids creating and entering these closures. If the counter-argument were having the possibility of sharing functional expressions, this is always available in FUN since a variable is a particular case of an expression. What we claim is that the normalization process in the Core-to-STG translation should not introduce unneeded sharing.

7. CONCLUSIONS

We have presented a stepwise derivation of a (well known) abstract machine starting from Sestoft's operational semantics, going through several intermediate machines and arriving at an imperative machine very close to a conventional imperative language. This strategy of adding a small amount of detail in each step has allowed us both to provide insight on fundamental decisions underlying the STG design and, perhaps more importantly, to be able to show the correctness of each refinement with respect to the previous one and, consequently, the correctness of the whole derivation. To our knowledge, this is the first time that formal translation schemes and a formal proof of correctness of the STG to C translation has been done.

Our previous work [2] followed a different path: it showed the soundness and completeness of a STG-like machine called STG-1S (laying somewhere between machines STG-2 and ISTG of this paper) with respect to Sestoft's semantics. The technique used was also different: a bisimulation between the STG-1S machine and Sestoft's MARK-2 machine was proved. We got the inspiration for the strategy followed here from Mountjoy [5] and in Section 2 we have explained the differences between his and our work. The previous machines of all these works, including STG-1 and STG-2 of this paper, are very abstract in the sense that they deal directly with functional expressions. The new machine ISTG introduced here is a really low level machine dealing with raw imperative instructions and pointers. Two contributions of this paper have been to bridge this big gap by means of the translations schemes and the proof of correctness of this translation.

Our experience is that formal reasoning about even well known products always reveals new details, give new insight, makes good decisions more solid and provides trust in the behavior of our programs.

8. REFERENCES

[1] A. at URL: **http://www.haskell.org/ghc/**.

[2] A. Encina and R. Peña. Proving the Correctness of the STG Machine. In *Implementation of Functional Languages, IFL'01. Selected Papers. LNCS 2312*, pages 88–104. Springer-Verlag, 2002.

[3] S. P. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. *Conference on Functional Programming Languages and Computer Architecture FPCA'91, LNCS 523*, September 1991.

[4] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. Conference on Principles of Programming Languages, POPL'93*. ACM, 1993.

[5] J. Mountjoy. The Spineless Tagless G-machine, Naturally. In *Third International Conference on Functional Programming, ICFP'98, Baltimore*. ACM Press, 1998.

[6] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine, Version 2.5. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[7] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.

[8] S. L. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. URL http://www.haskell.org, February 1999.

[9] S. L. Peyton Jones, S. Marlow, and A. Reid. The STG Runtime System (revised). http://www.haskell.org/ghc/docs, 1999.

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