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.
At each refinement, a formal proof of the soundness and completeness of the lower level with respect to the upper one is carried out. 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, even though 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 $\lambda_{-}$ denotes a vector $A_1, \ldots, A_n$ of subscripted entities. It is a normalized $\lambda$-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.

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 $\Gamma$, reduces to normal form $w$ and produces the final heap $\Delta$. When fresh pointers are created, freshness is understood w.r.t. (dom $\Gamma$) $\cup$ $A$, where $A$ contains the addresses of the closures under evaluation

$$\begin{align*}
\Gamma : \lambda x.e & \Downarrow_A \Gamma : \lambda x.e & \text{Lam} \\
\Gamma : C \bar{p} & \Downarrow_A \Gamma : C \bar{p} & \text{Cons} \\
\Gamma : e \Downarrow_A \Delta : \lambda x.e' & \Delta : e'[p/x] \Downarrow_A \Theta : w & \text{App} \\
\Gamma : e \Downarrow_A \Theta : w & \text{Var} \\
\Gamma \cup [p \mapsto e] : e \Downarrow_A \Delta : w & \text{Letrec} \\
\Gamma : \text{case } e \text{ of } C_i \bar{x}_i \mapsto e_i & \text{Case}
\end{align*}$$

(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 $p, p_1, \ldots$ 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 $\lambda$-abstractions, instead of just $\lambda$-abstractions. The reason for this was to forbid a $\lambda$-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, $\lambda$-abstractions were not 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 Appd makes a $\lambda$-abstraction to appear in the control expression, in contradiction with the desire of having $\lambda$-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 $\lambda$-abstractions or constructions. We will use the term lambda forms to refer to $\lambda$-abstractions or constructions alike. The motivation for this decision is to generate more efficient code as it will be seen in Section 5.1.
\[
e \rightarrow e \bar{x}_i^a
\]

\[\begin{align*}
t & \quad \longrightarrow x & \quad \text{-- variable} \\
\text{letrec } x_i = \bar{y}_j & \text{ in } e & \quad \text{-- recursive let} \\
\text{case } e \text{ of } alt_i & \quad \text{-- case expression} \\
alt & \quad \rightarrow C \bar{x} \rightarrow e & \quad \text{-- case alternative}
\end{align*}\]

\[\begin{align*}
if & \quad \rightarrow \lambda \bar{x}^a. e & \quad \text{-- lambda abstraction} \\
& \quad \rightarrow C \bar{x}^a & \quad \text{-- constructor application} \\
e & \quad \rightarrow e & \quad \text{-- expression}
\end{align*}\]

Figure 3: Language FUN

2. We relax applications to have the form \( e \bar{x}_i^a \), 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 Launchbury’s. The syntax of the language, called FUN, is shown in Figure 3. Notice that applications are done to \( n \) arguments at once, being \( e \bar{x}_i^a \) an abbreviation of \( \ldots (e \bar{x}_1 \ldots) \ldots \bar{x}_n \), and that consequently \( \lambda \)-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 names are always distinguishable from program variables. In rule CaseS3, 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 \( \Gamma \) and \( p \mapsto e \). Please notice that this notation may not coincide with other notations in which \( \Gamma \) and \( \Gamma[p \mapsto e] \) denote different heaps. Finally notice that, besides rule LetrecS3, also rule AppS3 creates closures in the heap.

Language FUN is at least as expressive as Launchbury’s \( \Lambda \)-calculus. The following normalization function transforms any Launchbury’s expression into a semantically equivalent FUN expression.

\[
\text{Definition } 1. \text{ We define the normalization function } N : \text{ Launch } \rightarrow \text{ FUN:}
\]

\[\begin{align*}
N x & \quad \overset{\text{def}}{=} x \\
N (e \, x) & \quad \overset{\text{def}}{=} (N \, e) \, x \\
N (\lambda \, x. e) & \quad \overset{\text{def}}{=} \text{letrec } y = N' (\lambda \, x. e) \text{ in } y \\
N (C \bar{x}) & \quad \overset{\text{def}}{=} \text{letrec } y = C \bar{x} \text{ in } y \\
N (\text{letrec } x_i = e_i^a \text{ in } e) & \quad \overset{\text{def}}{=} \text{letrec } x_i = N' \, e_i^a \text{ in } N \, e \\
N (\text{case } e \text{ of } C_i \bar{y}_j \rightarrow e_i) & \quad \overset{\text{def}}{=} \text{case } N \, e \text{ of } C_i \bar{y}_j \rightarrow N \, e_i
\end{align*}\]

\( N', N'' : \text{ Launch } \rightarrow \text{ FUN:} \)

\[\begin{align*}
N' (C \bar{x}^a) & \quad \overset{\text{def}}{=} C \bar{x}^a \\
N' e & \quad \overset{\text{def}}{=} N' \, e & \quad , e \neq C \bar{x}^a \\
N'' (\lambda \, x. e) & \quad \overset{\text{def}}{=} \lambda \, x. N'' \, e \\
N'' e & \quad \overset{\text{def}}{=} N \, e & \quad , e \neq \lambda \, x. e'
\end{align*}\]

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

**Proposition 1.**

1. Let \( e \in \text{ Launch } \) then \( N \, e \in \text{ FUN } \)
2. Let \( e \in \text{ if } \) then \( N' \, e \in \text{ if } \)
3. Let \( e \neq \lambda \, x. e' \) and \( e \neq C \bar{x}^a \) then, \( N \, e = N' \, e \)
4. \( (N \, e)[p/x_i] = N \, (e[p/x_i]) \)
5. \( (N' \, e)[p/x_i] = N' \, (e[p/x_i]) \)

**Proof.**

1. By structural induction on \( e \).
2. Trivial.
3. Trivial.
4. By definition of \( N \) and of substitutions.
5. By definition of \( N' \) and of substitutions.

\[\square\]

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: \( \alpha \) 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 \( \alpha \). 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.** \((\text{Sestoft } \Rightarrow \text{ Sestoft}^*)\) For all \( e \in \text{ Launch } \) we have:

\[\{ e \} : N \, e \triangleleft \Delta : w \Rightarrow \exists \alpha. \; \{ e \} : N' \, (\alpha \, w) \triangleleft \Delta \]

**Proof.** By induction on the number of reductions of Launchbury expressions. \[\square\]

**Proposition 3.** \((\text{Sestoft}^* \Rightarrow \text{ Sestoft})\) For all \( e \in \text{ FUN } \) we have:

\[\{ e \} : N' \, e \triangleleft \Delta : w \Rightarrow \forall e' \in \text{ Launch } . \; \{ e' \} : N' \, (\alpha \, w') \triangleleft \Delta \]

\[\exists \alpha. \; N' \, (\alpha \, w') = w \]
\[
\begin{align*}
\Gamma[p \mapsto \lambda x^n.e] : p &\vdash \Gamma : p & \text{Lam}_{SS} \\
\Gamma[p \mapsto C \overline{p}] : p &\vdash \Gamma : p & \text{Cons}_{SS} \\
\Gamma : e \Downarrow \Delta[p \mapsto \lambda x^n.e'] : p &\vdash \text{fresh} m, n > 0, q \text{ fresh} & \text{App}_{SS} \\
\Gamma : e \Downarrow \Delta[p \mapsto \lambda x^n.e] : p &\vdash \Delta : e'[p_i/x]^{n+1} \ldots p_n \Downarrow \Theta[q \mapsto w] : q & n \geq m & \text{Var}_{SS} \\
\Gamma \cup \Delta[p \mapsto e] : p \Downarrow \Delta \cup \Theta[q \mapsto w] : q & & \text{Letrec}_{SS} \\
\Gamma : e \Downarrow \Delta : p &\vdash \text{fresh} & \text{Case}_{SS} \\
\end{align*}
\]

Figure 4: Semantics S3

**Proof.** By induction on the number of reductions of Launchbury expressions. \(\square\)

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 \(\Rightarrow\) S3, completeness of S3) For all \(e \in \text{FUN}\) we have:

\[
\{ \} : e \Downarrow \Delta : w \Rightarrow \left\{ \{ \} : e \Downarrow \Delta : w' \right\} \exists \alpha. \alpha w = w' \Delta \subseteq \alpha \Delta'
\]

**Proof.** By induction on the number of reductions of FUN expressions. \(\square\)

**Proposition 5.** (S3 \(\Rightarrow\) Sestoft, soundness of S3) For all \(e \in \text{FUN}\) we have:

\[
\{ \} : e \Downarrow \Delta[p \mapsto w] : p \Rightarrow \left\{ \{ \} : e \Downarrow \Delta' : w' \right\} \exists \alpha. \alpha w' = w \Delta' \subseteq \alpha \Delta 
\]

**Proof.** By induction on the number of reductions of FUN expressions. \(\square\)

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 \((\Gamma, e, S)\) where \(\Gamma\) 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 \(\text{alts}\) 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 \text{letrec} in Figure 5 is a literal transcription of the \text{Letrec}_{SS} rule of Figure 4. The semantic rules for \text{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 \(\Gamma, e, S\) and \(\Delta\), then \(\Gamma : e \Downarrow_S \Delta[p \mapsto w] : p \iff (\Gamma, e, S) \rightarrow^* (\Delta', p', S')\), where

1. \(\Delta' \subseteq \Delta\)
2. \(\Delta[p \mapsto C \overline{p}] \text{ then } S = S', p = p' \text{ and } \Delta'[p \mapsto C \overline{p}]\)
3. \(\Delta[p \mapsto \lambda x^n.e] \text{ then there exists } m \geq 0 \text{ s.t. } \Delta'[p \mapsto \lambda x^n.e'] \text{ and } S' = \pi^m : S \text{ and } e' = e'[q_i/y]^{m}\)
We will use the following notation:

\[ \Gamma \vdash \{ x_i = \underline{f}_i \} \quad \text{in} \quad e \quad \text{and} \quad S \]

\( \vdash \text{letrec (1)} \)

\( \Delta \vdash \underline{f}_i \quad \text{and} \quad m > n \geq 0 \)

\( \text{if and only if} \quad \Gamma \vdash \{ \{ \}, e_0, \{ \}, \} \quad \text{STG-2} \quad (\Gamma, x, E[x \leftarrow q], \underline{p}_m) \quad \text{and} \quad \text{either:} \)

\( \Delta \vdash \underline{f}_i \quad \text{and} \quad n = 0 \)

\( \text{or} \quad \Gamma \vdash (\{ \}, e, \{ \}, [\underline{f}_i]) \quad \text{and} \quad m > n \geq 0 \quad \text{and} \quad e = E' e' \).

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

The proposition shows that the semantic rule App32 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 \( \Delta \vdash \underline{f}_i \) when a lambda is reached (see rule var2 in Figure 5).

A final expression pointing to a lambda with \( n > m \) 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 \( (\Gamma, e, E, S) \) where \( E \) is the environment of \( e \), the alternatives are pairs \( (alts, E) \), and a closure is a pair \( (\underline{f}, E) \). Now expressions and lambda forms keep their original variables and the associated environment maps them to pointers in the heap. The notation \( E[\cdot] \) 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 \( e \).

\( \{ \}, e_0, \{ \}, \} \quad \text{STG-1} \quad (\Delta, q, \underline{p}_m) \) where either:

- \( \Delta[q \leftarrow C \underline{p}_m] \land n = 0 \)

- \( \text{or} \quad \Delta[q \leftarrow \underline{f}_i] \land m > n \geq 0 \)

**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.

The 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 \( (is, S, node, \Gamma, cs) \), 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 \( cs \) belongs to), \( \Gamma \) 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 \( \Gamma \), \( as \) and \( ws \) for lists of such pointers, and \( p \) for
In the expression whose translation is environment, having a heap pointer for every free variable updated. A closure is a pair \((\boxed{\lambda}^m \downarrow \Gamma)\) corresponding to an alternative of a translation sequence is. In Figure 8, 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 translation to C of the 9 instructions of the ISTG should appear straightforward for the reader. For instance, BUILDENV and BUILDCLS can be implemented by a sequence of assignments, copying values to/from the stack, or 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 contain 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.

Figure 6: The enriched semantics

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

\[
\frac{\Gamma : e \prod_{n} \Delta[p \mapsto \lambda x^m \cdot \lambda y^n \cdot e'] : p}{\Gamma : e \prod_{n} \downarrow \Gamma \cup [q \mapsto \lambda y^n \cdot e'[p/x]_\Gamma] : q} \quad m, n > 0, q \text{ fresh} \quad \text{App}_{SS} \]

\[
\frac{\Gamma : e \prod_{n} \Delta[p \mapsto \lambda x^m \cdot e'] : p \quad \Delta : e'[p/x]^{m+1}_\Gamma \quad p_{n+1} \cdots p_n \downarrow \theta[q \mapsto w] : q}{\Gamma : e \prod_{n} \downarrow \Gamma \theta : q} \quad n \geq m \quad \text{App}'}_{SS} \]

\[
\frac{\Gamma : e \prod_{p \in S} \Delta[q \mapsto w] : q}{\Gamma \cup [p \mapsto e] : p \downarrow S \cup [p \mapsto w] : q} \quad \text{Var}_{SS} \]

\[
\frac{\Gamma : e \prod_{\text{alts} \in S} \Delta[p \mapsto C_k \overline{p}] : p \quad \Delta : e_k[p/x]^{n+1}_\Gamma \quad p_{n+1} \cdots p_n \downarrow \theta[q \mapsto w] : q}{\Gamma : \text{case } e \text{ of } \text{alts} \downarrow \theta : q} \quad \text{Case}_{SS} \]
variable is located by maintaining two compile-time environments $\rho$ and $\eta$. The first one $\rho$ corresponds to the environment kept in the stack, while the second one $\eta$ 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 $\rho$ 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 $\rho$ is a list $[(\delta_1, m_1, n_1), \ldots, (\delta_k, m_k, n_k)]$ of blocks. It describes the variables in the stack starting from the top. In a block $(\delta, m, n)$, $\delta$ is an environment mapping exactly $m - |n|$ program variables to disjoint numbers in the range $1..m-|n|$. The empty environment, denoted $\rho_0$ is the list $[(\emptyset), 0, 0)]$.

A block $(\delta, m, n)$ 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 $\eta$ 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 $\rho$ from the top of the stack, denoted $\rho x$, is given by

$$\rho x \triangleq (\sum_{i=1}^{k} m_i) - \delta_i x, \text{ being } x \in \text{dom } \delta_i$$

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

$$(\rho, \eta) x \triangleq (\text{STACK}, \rho x) \quad \text{if } x \in \text{dom } \rho$$

$$(\rho, \eta) x \triangleq (\text{NODE}, \eta x) \quad \text{if } x \in \text{dom } \eta$$

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. $((\delta, m, 0) : \rho) + n \triangleq ((\emptyset), 0, 0) : (\delta, m + n, n) : \rho$
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 \( \text{cs}[p \mapsto \ldots] \) means that the corresponding translation scheme has a side effect which consists of creating a code sequence in the code store \( \text{cs} \) and pointing it by the code pointer \( p \).

**Figure 8: The ISTG machine**

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

2. All free variables of \( e \) are defined either in \( \rho \) or in \( \eta \). Moreover, \( \text{dom } \rho \cap \text{dom } \eta = \emptyset \).

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. \( \square \)

In order to prove the correctness of the translation, we only need to consider ISTG machine configurations of the form \((p, S, i, \text{node}, \Gamma, \text{cs})\) in which \( p \) is generated by a call to \( trE \) for some expression \( e \) and environments \( \rho, \eta \). We call these stable configurations. We enrich these configurations with three additional components: the environments...
Definition 6. A STG-2 environment $E$ is equivalent to an ISTG environment defined by $\rho, \eta, S, \Gamma_1$ and node, denoted $E \equiv (\rho, S, \eta, \Gamma_1, \text{node})$ if $\text{dom}(E) \subseteq \text{dom}(\rho) \cup \text{dom}(\eta)$ and

\[
\forall x \in \text{dom}(E) \quad E \ x = S_l \ (\rho \ x) \quad \text{if } x \in \text{dom}(\rho) \\
E \ x = \text{node}_{\Gamma_l} \ (\eta \ x) \quad \text{if } x \in \text{dom}(\eta)
\]

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

1. Whenever $\rho = (\delta, m, 0) : \rho'$, then $S_l = \pi^n : S'_l$ and $S \equiv (\rho', S'_l, S_{env})$
2. Whenever $\rho = (\delta, m, n) : \rho'$, $n > 0$, then $S = \pi^n : S'$, $S_l = \pi^n : S'_l$, and $S' \equiv (\rho', S'_l, S_{env})$
3. Whenever $\rho = (\delta, m, -1) : \rho'$, then $S = (\text{alts}, E) : S'$, $S_l = \pi^n : S'_l$, and $S' \equiv (\rho_{\text{alts}}, S'_l, S_{env})$
4. Whenever $S = \#a : S'$ and $S_l = \#a : S'_l$, then $S' \equiv (\rho, S'_l, S_{env})$
5. Additionally, $[] \equiv ([], [], [])$
Definition 8. A STG-2 heap $\Gamma$ is equivalent to an ISTG pair $(\Gamma_I, cs)$, denoted $\Gamma \equiv (\Gamma_I, cs)$, if for all $p$ we have $\Gamma[p \mapsto (\Gamma^0, E)]$ if and only if $\Gamma[p \mapsto (q, ws)]$, $cs[q \mapsto is] = trB (\Gamma^0)$ and $ws = E x^n$.

Definition 9. A STG-2 configuration is equivalent to an ISTG enriched stable configuration, denoted $(\Gamma, e, E, S) \equiv (is, \rho, \eta, S_{env}, node, \Gamma_I, cs)$ 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 $(\emptyset, e_0, \emptyset, \emptyset, \emptyset)$ and $(trE e_0 \rho_0 \emptyset \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$, where $cs$ is the code stored 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 close 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 $e x^n$ instead of $x x^n$ 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


