A Systematic Derivation of the STG Machine Verified in Coq

Maciej Piróg
Institute of Computer Science
University of Wrocław
Wrocław, Poland
maciej.adam.pirog@gmail.com

Dariusz Biernacki
Institute of Computer Science
University of Wrocław
Wrocław, Poland
dabi@cs.uni.wroc.pl

Abstract

Shared Term Graph (STG) is a lazy functional language used as an intermediate language in the Glasgow Haskell Compiler (GHC). In this article, we present a natural operational semantics for STG and we mechanically derive a lazy abstract machine from this semantics, which turns out to coincide with Peyton-Jones and Salkild's Spineless Tagless G-machine (STG machine) used in GHC. Unlike other constructions of STG-like machines present in the literature, ours is based on a systematic and scalable derivation method (inspired by Danvy et al.'s functional correspondence between evaluators and abstract machines) and it leads to an abstract machine that differs from the original STG machine only in inessential details. In particular, it handles non-trivial update scenarios and partial applications identically as the STG machine.

The entire derivation has been formalized in the Coq proof assistant. Thus, in effect, we provide a machine checkable proof of the correctness of the STG machine with respect to the natural semantics.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory—Semantics; D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Compilers

General Terms Languages, Theory, Verification

Keywords STG, natural semantics, abstract machine, derivation, verification, Coq

1. Introduction

The Shared Term Graph (STG) language along with the Spineless Tagless G-machine (STG machine), both developed by Peyton-Jones and Salkild [11, 12], lie at the heart of the Glasgow Haskell Compiler (GHC)—the flagship Haskell compiler [7]. STG is a higher-order pure lazy functional language based on a normalized λ-calculus with multiple binders, datatype constructors and pattern matching. It is used as an intermediate language in GHC and compiled to code that mimics the execution of the STG machine. The STG abstract machine defines an operational semantics and an implementation model for STG. Since it contains a high degree of implementational detail, it is not amenable to reasoning about operational aspects of the source language. A considerably more intuitive formalism that omits inessential details of implementation is natural semantics, proposed for lazy functional languages by Launchbury [8] and later refined by Sestoft [15]. Although the results by Launchbury and by Sestoft are eminent, they do not address the STG language, but a simpler variant of a normalized λ-calculus. In turn, Encina and Peña in a series of articles proposed a natural semantics for a language that very much resembles STG [4–6], but it does not capture the evaluation model of the original STG machine in the way the heap is allocated and updated. This difference is confirmed by their abstract machines: they were obtained by an ad-hoc derivation and shown to be equivalent with the proposed natural semantics, but these machines differ from the original STG machine.

As a matter of fact, none of the existing natural semantics has been defined exactly for the original STG language which allows for multiple binders and non-trivial update scenarios directed by update flags. Also, none of the proposed natural semantics for lazy evaluation captures fully the evaluation model embodied in the original STG machine.

In order to fill this vacuum and as a first step towards a certified compiler for Haskell—a bigger project that we are working on—we present a natural operational semantics for the STG language that is an extension of the semantics given by Sestoft and from this semantics we mechanically derive the corresponding abstract machine. The derivation method we use consists of some standard steps such as argument stack introduction and environment introduction, but the critical transformation from a big-step operational semantics to the equivalent abstract machine is given by the transformation to continuation-passing style (CPS transformation) [13] followed by the defunctionalization of continuations [3, 13]. We, therefore, rely on Danvy et al.'s functional correspondence between evaluators and abstract machines [1], that has already proved useful before in the context of evaluators for a lazy lambda calculus [2], except that we transform semantic descriptions rather than interpreters. This derivation method transforms only the form of the semantics and leaves the evaluation model described by the semantics intact, so the natural semantics we propose and the abstract machine we derive are two sides of the same coin.

Interestingly and as expected, the outcome of our derivation turns out to be the STG machine, only slightly reformulated. Hence, the STG machine, though designed for efficient evaluation and—more importantly—efficient implementation on stock hardware, can be seen as a natural counterpart of our semantics for the STG language, obtained via a systematic and universal derivation method. Additionally, having a method to mechanically transform a natural operational semantics into an abstract machine, we can

Haskell'10, September 30, 2010, Baltimore, Maryland, USA.

$$
\begin{array}{lcll}
e & \to & x\ \overline{x_i} & \text{— application} \\
 & | & C\ \overline{x_i} & \text{— saturated constructor} \\
 & | & \textbf{letrec}\ \overline{x_i = lf}_i\ \textbf{in}\ e & \text{— local definition} \\
 & | & \textbf{case}\ e\ \textbf{of}\ \overline{alt_i} & \text{— case expression} \\
lf & \to & \lambda_\pi\ \overline{x_i}.e & \text{— lambda-form} \\
alt & \to & C\ \overline{x_i} \to e & \text{— case alternative} \\
\pi & \to & U \mid N & \text{— update flag}
\end{array}
$$

Figure 1. The syntax of the STG language

augment the STG language with new features and then easily obtain a consonant machine, e.g., the STG machine with unboxed arithmetics.

The entire derivation presented in this article has been formalized and proved correct in the Coq proof assistant. Thus we provide a machine checkable proof of the correctness of the STG machine with respect to the natural semantics. Additionally, we take this result as the starting point of the construction of a certified Haskell compiler in Coq.

The rest of the article is organized as follows. In Section 2, we present the syntax and the natural semantics of STG. In Section 3, we describe our enabling technology—the functional correspondence at the level of semantics. In Section 4, we transform the natural semantics into the STG machine by argument stack introduction and environment introduction followed by the transformation to defunctionalized continuation-passing style. We also argue that the resulting machine is the STG machine despite some minor discrepancies between the two and we sketch how our construction scales to some common extensions of the language. In Section 5, we briefly describe the Coq formalization. In Section 6, we compare our work with the existing derivations of lazy abstract machines. In Section 7, we conclude and put the present result in the context of building a certified compiler for Haskell.

2. STG and its semantics

We begin with the syntax and semantics for the STG language which is essentially a normalized lambda calculus with simplified algebraic datatypes.

2.1 Syntax

The syntax of the STG language is shown in Figure 1, where $\mathfrak{X} = \{x, y, z, p, q, \ldots\}$ is a set of variables and $\mathfrak{C} = \{C, C_1, C_2, \ldots\}$ is a set of names of constructors. The letters e, f, g, w will stand for expressions of the STG language.

We denote sequences by juxtaposition (e.g., $x_1 \ldots x_n$) or by a line over indexed elements (e.g., $\overline{x_i}$). If not stated otherwise, sequences may be empty. Appending sequences and inserting elements is also represented by juxtaposition. The symbol ε stands for the empty sequence.

Application We apply only single variables to tuples of variables. This limited form of application is in correspondence with the lazy evaluation: the variables are pointers to thunks representing subexpressions that will be computed only if needed (or have already been computed and updated). The tuple may be empty, so there is no need for a separate variable case in the grammar.

Constructor Constructor expressions are built using a constructor name (an element from the set $\mathfrak{C}$) and its arguments (variables). All constructors are saturated, i.e., they must be given all their arguments. Since the STG language is not typed and there are no explicit datatype definitions, we may think of constructor names as being the lowest-level identifiers, e.g., positive integers, possibly shared between datatypes. (If the STG language is used as an intermediate language in a compiler, the sharing of constructor names is not an issue, since the front-end type checking guarantees that each constructor name will be interpreted in the right datatype.)

Local definition Local definition expressions (aka **letrec** expressions) play a more significant role than in ordinary functional languages: they enclose subexpressions for lazy evaluation. Each local definition binds a lambda-form $\lambda_\pi \overline{x_i}.e$, where e is an expression and $\overline{x_i}$ is a (possibly empty) tuple of its arguments. Intuitively, we may think of it as an ordinary lambda expression. The symbol π represents an update flag (U for updatable and N for non-updatable), which indicates whether after evaluation of the lambda-form the result should overwrite the lambda-form. If a lambda-form expects some arguments, it is already in normal form (as usual, we do not reduce under lambdas), so its update flag is always N. Definitions in one **letrec** block are assumed to be mutually recursive.

Case expression Case expressions **case** e **of** $\overline{alt_i}$ perform eager evaluation (by eager we mean "up to the outermost constructor") of the subexpression e. The result is then matched against the list of alternatives $\overline{alt_i}$ which binds arguments of the constructor in the matched alternative. The body of the matched alternative is then evaluated according to the lazy evaluation strategy.

The transformation from an everyday-use functional language like Haskell to the STG language requires extraction of all non-variable subexpressions to **letrec** definitions, normalization of **case** expressions and a static analysis for the update-flag annotation.

2.2 Natural semantics

In this section, we present a natural operational semantics for the STG language. It uses a heap to store all the lambda-forms needed to evaluate an expression. Free variables serve as pointers to the elements of the heap. When the body of an updatable lambda-form (i.e., one with the U flag) is evaluated, it is overwritten with the value, so no expression sharing this lambda-form will evaluate the same node for a second time.

We split the set of variables $\mathfrak{X}$ into two disjoint, enumerably infinite sets: the set of bound variables $BOUND$ (ranged over by $x_1, x_2, \ldots$) and the set of heap pointers $POINTERS$ (ranged over by $p, q, p_1, q_1, \ldots$), so:

$$\mathfrak{X} = BOUND \cup POINTERS$$

It is needed to provide sound local freshness of names in the semantics, as will be discussed later on. We call an expression well-formed if and only if all its bound variables are in $BOUND$, and all of its free variables are in $POINTERS$. The semantics is designed for well-formed expressions only.

The semantics is given in Figure 2. It derives judgments of the form $(\Gamma : e \downarrow \Delta : f)$. The pair $(\Gamma : e)$ is called a configuration and $(\Delta : f)$ a normal form. Γ and Δ are heaps, i.e., partial functions from $\mathfrak{X}$ to LF, where LF is the set of all lambda-forms. Values in the heap are called closures.

In the following, $\Gamma\{x \mapsto lf\}$ stands for a heap Γ, explicitly indicating that $\Gamma(x) = lf$, while $\Gamma \oplus [x \mapsto lf]$ stands for a heap

$$\Gamma : C\ \overline{p_i}\ \downarrow\ \Gamma : C\ \overline{p_i} \qquad \textsc{Con}$$

$$\Gamma\{p \mapsto \lambda_N\ x_1 \ldots x_m.e\} : p\ p_1 \ldots p_n\ \downarrow\ \Gamma : p\ p_1 \ldots p_n \text{ where } n < m \qquad \textsc{App1}$$

$$\frac{\Gamma : e[x_1/p_1 \ldots x_m/p_m]\ \downarrow\ \Delta : w}{\Gamma\{p \mapsto \lambda_N\ x_1 \ldots x_m.e\} : p\ p_1 \ldots p_m\ \downarrow\ \Delta : w} \qquad \textsc{App2}$$

$$\frac{\Gamma : e[x_1/p_1 \ldots x_m/p_m]\ \downarrow\ \Delta : q\ q_1 \ldots q_k \qquad \Delta : q\ q_1 \ldots q_k\ p_{m+1} \ldots p_n\ \downarrow\ \Theta : w}{\Gamma\{p \mapsto \lambda_N\ x_1 \ldots x_m.e\} : p\ p_1 \ldots p_n\ \downarrow\ \Theta : w}\ m < n \qquad \textsc{App3}$$

$$\frac{\Gamma : e\ \downarrow\ \Delta : C\ \overline{q_i}}{\Gamma\{p \mapsto \lambda_U.e\} : p\ \downarrow\ \Delta \oplus [p \mapsto \lambda_N.C\ \overline{q_i}] : C\ \overline{q_i}} \qquad \textsc{App4}$$

$$\frac{\begin{array}{c}\Gamma : e\ \downarrow\ \Delta\{q \mapsto \lambda_N\ x_1 \ldots x_k\ x_{k+1} \ldots x_n.f\} : q\ q_1 \ldots q_k \\ \Delta \oplus [p \mapsto \lambda_N\ x_{k+1} \ldots x_n.f[x_1/q_1 \ldots x_k/q_k]] : q\ q_1 \ldots q_k\ p_1 \ldots p_m\ \downarrow\ \Theta : w\end{array}}{\Gamma\{p \mapsto \lambda_U.e\} : p\ p_1 \ldots p_m\ \downarrow\ \Theta : w} \qquad \textsc{App5}$$

$$\frac{\Gamma \oplus \overline{[p_i \mapsto lf_i[\overline{x_i/p_i}]]} : e[\overline{x_i/p_i}]\ \downarrow\ \Delta : w}{\Gamma : \textbf{letrec}\ \overline{x_i = lf_i}\ \textbf{in}\ e\ \downarrow\ \Delta : w}\ \overline{p_i} \in POINTERS \setminus Dom(\Gamma) \qquad \textsc{Letrec}$$

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

Figure 2. The natural semantics of the STG language

Γ extended or overwritten at x with lf. The operation $e[\overline{x_i/p_i}]$ simultaneously substitutes each free occurence of x_i in e with p_i.

Normal forms in this semantics are constructors and partial applications, as stated in the CON and APP1 rules. (An application is partial only in the context of a heap, which encodes the whole graph of an expression.)

There are two more rules for applications of variables representing non-updatable closures: APP2, when there are just enough arguments, and APP3, when there are too many arguments. Intuitively, we evaluate the body of the lambda-form, substituting actual arguments for formal arguments. If there are too many arguments, we use only the prefix of the argument list of the appropriate length. If the closure evaluates to a partial application, we append the remaining suffix of the argument list and continue with evaluation. If it evaluates to a constructor, the whole expression does not have a normal form, since it would be an application of the constructor to the suffix of the argument list, which is already saturated; such expression would be ill-typed in any strongly typed language.

The rules for applications of variables representing updatable closures are similar. For a variable p representing an updatable (thus argument-free) closure, if the closure evaluates to a constructor (APP4), it is the value of the expression. But we also need to update the heap with the constructor, so that if any other expression in some lambda-form in the heap uses the pointer p, the closure will not have to be evaluated again. If the closure evaluates to a partial application $q\ q_1 \ldots q_k$ (APP5), we update the closure under the pointer p with the lambda-form under the pointer q, but with first k arguments already fed with $q_1 \ldots q_k$. (In the APP5 rule $n > k$, since $q\ q_1 \ldots q_k$ is a partial application.)

Variables, addresses, and fresh pointers A variable is fresh if and only if it does not interfere with any other variable in the derivation tree by an undesired variable capture. The freshness check (sometimes called a generation of a fresh variable) is local iff it can be done using only the context of a single rule, and does not refer to the whole derivation tree or any kind of external "fresh names generator." Locality is a desirable property when one wants to reason in low-level details necessary for an implementation or formalization in proof systems like Coq.

Our solution with a bipartite set of variables solves the problem: generating fresh addresses in the LETREC rule is local (we need freshness only with respect to the heap) and it fits the design pattern of nameless bound variables representation in Coq, where bound variables are represented as de Bruijn indices, and free variables as atoms.

In order to formalize the above intuitions, we use the following definitions:

DEFINITION 1. *Let e be an expression or a lambda-form, and Γ be a heap. Then:*

1. *e is* well-formed *iff its bound variables are in $BOUND$ and its free variables are in $POINTERS$.*
2. *e is* closed by *a heap Γ iff all its free variables are in $Dom(\Gamma)$.*
3. *Γ is* well-formed *iff $Dom(\Gamma) \subseteq POINTERS$ and each closure in Γ is well-formed and closed by Γ.*
4. *The configuration $(\Gamma : e)$ is* well-formed *iff Γ and e are well-formed and e is closed by Γ.*

We do not mind that ill-formed programs and configurations may evaluate to nonsense values. For example the configuration $(\emptyset : \textbf{letrec}\ x = \lambda_N.C\ \textbf{in}\ p)$ may evaluate to C if the lambda-form in the **letrec** expression is allocated under the address p.

The following theorem ensures that if the root configuration is well-formed, configurations are well-formed throughout the derivation tree and no variables are captured:

PROPOSITION 2. *For a well-formed configuration $(\Gamma : e)$, if $(\Gamma : e \downarrow \Delta : w)$, then all configurations and normal forms in the derivation of $(\Gamma : e \downarrow \Delta : w)$ (including $\Delta : w$) are well-formed and all substitutions replace pointers for bound variables.*

Comparison with Sestoft's semantics Our semantics is inspired by the semantics proposed by Sestoft [15] as a refinement for Launchbury's semantics for a normalized λ-calculus [8]. The rules for constructors, **letrec** and **case** expressions are virtually the same. The difference is in lambda-forms, which in STG are tied to **letrec** definitions and bind multiple variables, while the Launchbury's calculus contains the usual first-class λ-abstractions binding a single

$$e \quad \rightarrow \quad \mathbf{n} \mid \mathbf{abs}\ e \mid e \odot e \quad \text{where } \mathbf{n} \in \mathbb{Z} \text{ and } \odot \in \Sigma \quad \text{— empty}$$

$$\mathbf{n} \Downarrow \mathbf{n} \qquad \frac{e \Downarrow \mathbf{n}}{\mathbf{abs}\ e \Downarrow |\mathbf{n}|} \qquad \frac{e_1 \Downarrow \mathbf{n_1} \qquad e_2 \Downarrow \mathbf{n_2}}{e_1 \odot e_2 \ \Downarrow\ \mathbf{n_1}[\odot]\mathbf{n_2}}$$

Figure 3. Arithmetic expressions—the syntax and natural semantics

variable. The restricted shape of lambda-forms in STG makes "entering" a closure in the heap always identified with application (note that since we have multiple binders, an application to zero arguments is still an application), while they are different concepts in Sestoft's semantics, represented by two different rules, VAR and APP.

In contrast to Launchbury's calculus, the STG language is more complex in that it allows for multiple binders and update flags. On the other hand, our semantics does not cater for the concept of black holes, which, as advocated by Peyton-Jones [11], is superfluous as far as only sequential computation is concerned. It is fairly easy to embed Launchbury's calculus into STG, and Sestoft's natural semantics into ours in a provably correct way.

3. Functional correspondence

In this section we describe a method that facilitates a mechanical derivation of abstract machines from natural semantics. It is inspired by functional correspondence that consists in first, transforming an evaluator in direct style that implements a natural semantics into continuation-passing style (CPS) and second, defunctionalizing the continuations of the CPS evaluator, which leads to an evaluator implementing an abstract machine [1]. We illustrate the functional correspondence with the example of evaluating arithmetic expressions.

Let $\Sigma = \{+, *, -, \ldots\}$ be a set of binary symbols and $[\cdot] \colon \Sigma \to \mathbb{Z}^{\mathbb{Z}\times\mathbb{Z}}$ be a natural interpretation of symbols in Σ. For any $\mathbf{n} \in \mathbb{Z}$ let $|\mathbf{n}|$ denote its absolute value. The syntax and semantics of arithmetic expressions are shown in Figure 3.

It is straightforward to implement an evaluator for this semantics in a functional meta-language, i.e., to write a function *eval* such that $eval\ e = \mathbf{n}$ iff the judgment $(e \Downarrow \mathbf{n})$ is provable. For each semantic rule, the function is recursively called and the final result is obtained by applying the corresponding operation to the results of the recursive calls. It could be encoded in Haskell as follows:

```
data Expr = Const Integer
          | Abs Expr
          | Op Expr String Expr

interp :: String → Integer → Integer → Integer
interp "+"   = (+)
interp "*"   = (*)
interp "-"   = (-)
interp "mod" = mod

eval :: Expr → Integer
eval (Const n)      = n
eval (Abs e)        = abs n
  where n = eval e
eval (Op e1 op e2) = interp op n1 n2
  where n1 = eval e1
        n2 = eval e2
```

In the next phase we transform the evaluator into CPS. Now, the evaluator has one more argument—a continuation. The evaluator no longer returns a value, instead it tail-calls itself or applies the continuation to a value. To compute the value of an expression, one supplies the evaluator with the identity continuation (kId):

```
evalCps :: Expr → (Integer → a) → a
evalCps (Const n) k = k n
evalCps (Abs e) k   = evalCps e (λn → k (abs n))
evalCps (Op e1 op e2) k = evalCps e1
  (λn1 → evalCps e2 (λn2 → k (interp op n1 n2)))

kId :: Integer → Integer
kId = id
```

The next step is the defunctionalization of continuations. Each construction of a continuation (either by a named value, like kId, or anonymously, like $\lambda n \to k\ (abs\ n)$) is replaced by an explicit closure, which stores all the free variables of the continuation. Each application of a continuation is replaced by an application of the function *apply* which takes the closure as an argument and evaluates accordingly:

```
data Cont a = Id
            | K1 (Cont a)
            | K2 Expr String (Cont a)
            | K3 Integer String (Cont a)

apply :: Cont Integer → Integer → Integer
apply Id n          = n
apply (K1 k) n      = apply k (abs n)
apply (K2 e2 op k) n1 = evalDcps e2 (K3 n1 op k)
apply (K3 n1 op k) n2 = apply k (interp op n1 n2)

evalDcps :: Expr → Cont Integer → Integer
evalDcps (Const n) k    = apply k n
evalDcps (Abs e) k      = evalDcps e (K1 k)
evalDcps (Op e1 op e2) k = evalDcps e1 (K2 e2 op k)
```

Note that the *Cont* datatype behaves like a stack, with *Id* corresponding to the empty stack, and *K1*, *K2* and *K3* to three kinds of its elements.

The mutually recursive functions *evalDcps* and *apply* may be thought of as evaluators of two semantics defined in terms of each other: $\mathcal{E}$, proving judgments of the form $\mathcal{E}\langle e, \overline{K_i}\rangle \nwarrow \mathbf{n}$, and $\mathcal{A}$, proving judgments of the form $\mathcal{A}\langle \mathbf{m}, \overline{K_i}\rangle \nwarrow \mathbf{n}$, where $\overline{K_i}$ is a continuation stack. We call it the *Defunctionalized CPS (D-CPS) semantics* (Figure 4). The equivalence of the two semantics may be defined as follows: $(e \Downarrow \mathbf{n})$ iff $\mathcal{E}\langle e, \varepsilon\rangle \nwarrow \mathbf{n}$, and is easy to show by simple induction. We call the transformation from the natural semantics to the D-CPS semantics the *D-CPS transformation*.

Note that the D-CPS semantics has a particular form: each rule has at most one premise, and the right-hand sides of the $\nwarrow$ symbol are identical for the premise and the conclusion. Thus, it is easy to transform the semantics into an abstract machine (Figure 5), where the states are left-hand sides of the $\nwarrow$ symbol, each semantic rule with a premise is transformed into a transition rule for the machine (from the left-hand side of the conclusion to the left-hand side of the premise) and the rule with no premises becomes a halting state:

$$\frac{\mathcal{A}\langle \mathbf{n}, \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{E}\langle \mathbf{n}, \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \frac{\mathcal{E}\langle e, \mathbf{K1} : \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{E}\langle \mathbf{abs}\ e, \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \frac{\mathcal{E}\langle e_1, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{E}\langle e_1 \odot e_2, \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \mathcal{A}\langle \mathbf{m}, \varepsilon\rangle \nwarrow \mathbf{m}$$

$$\frac{\mathcal{A}\langle |\mathbf{n}|, \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{A}\langle \mathbf{n}, \mathbf{K1} : \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \frac{\mathcal{E}\langle e_2, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{A}\langle \mathbf{n_1}, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \frac{\mathcal{A}\langle \mathbf{n_1}\ [\odot]\ \mathbf{n_2}, \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{A}\langle \mathbf{n_2}, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}}$$

Figure 4. A D-CPS semantics for arithmetic expressions

$$\begin{array}{lcl}
\mathcal{E}\langle \mathbf{n}, \overline{K_i}\rangle & \Rightarrow & \mathcal{A}\langle \mathbf{n}, \overline{K_i}\rangle \\
\mathcal{E}\langle \mathbf{abs}\ e, \overline{K_i}\rangle & \Rightarrow & \mathcal{E}\langle e, \mathbf{K1} : \overline{K_i}\rangle \\
\mathcal{E}\langle e_1 \odot e_2, \overline{K_i}\rangle & \Rightarrow & \mathcal{E}\langle e_1, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle \\
\mathcal{A}\langle \mathbf{m}, \varepsilon\rangle & \Rightarrow & \mathbf{m} \\
\mathcal{A}\langle \mathbf{n}, \mathbf{K1} : \overline{K_i}\rangle & \Rightarrow & \mathcal{A}\langle |\mathbf{n}|, \overline{K_i}\rangle \\
\mathcal{A}\langle \mathbf{n_1}, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle & \Rightarrow & \mathcal{E}\langle e_2, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle \\
\mathcal{A}\langle \mathbf{n_2}, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle & \Rightarrow & \mathcal{A}\langle \mathbf{n_1}\ [\odot]\ \mathbf{n_2}, \overline{K_i}\rangle
\end{array}$$

Figure 5. An abstract machine for arithmetic expressions

the equivalence of the D-CPS semantics and the abstract machine can be formulated as follows: $\mathcal{E}\langle e, \varepsilon\rangle \nwarrow \mathbf{n}$ iff $\mathcal{E}\langle e, \varepsilon\rangle \Rightarrow^* \mathbf{n}$, where $\Rightarrow^*$ is the reflexive and transitive closure of the relation $\Rightarrow$.

Though the transformation from the D-CPS semantics into the abstract machine-based semantics is trivial, the change is conceptually significant. The former is a big-step semantics, i.e., one that proves judgments on a relation between expressions, stacks and the final value. The latter is a small-step semantics, which describes separate steps of computation.

Our next objective will be to enhance the STG semantics to strengthen its computational properties and then to transform it into an abstract machine using the presented method.

4. From the natural semantics to the STG machine

In this section we present two additional semantics for the STG language, the first introducing the stack of arguments for applications, then refined by the introduction of environments instead of substitutions. Then we use the method described in the previous section to derive an abstract machine, which needs only a little make-up to become the Spineless Tagless G-machine.

4.1 Argument stack introduction

An essential flaw of the STG language natural semantics is its treatment of applications with too many arguments. Whenever an application lacks some arguments (the APP1 rule is used) somewhere in the derivation of the first premise of APP3, there may be more arguments "waiting" in the second premise. Consider the following program (for arbitrary e and p):

$$\begin{array}{rl} \textbf{letrec} & f = \lambda_N\ x.e \\ & g = \lambda_N.f \\ \textbf{in} & g\ p \end{array}$$

First, in APP3 g is evaluated in the first premise. g does not take any arguments, and then f is evaluated to itself by APP1, because there are not enough arguments to proceed (the argument p is temporarily "forgotten" during the computation of the "argument bottleneck" g). Only then, in the second premise of the APP3 rule, the expression $f\ p$ is created and evaluated.

To solve this problem, we introduce another entity to our judgments, which we call the *argument accumulator*. The judgments now take the form $\langle \Gamma, e, \overline{p_i}\rangle \Downarrow \langle \Delta, w, \overline{q_i}\rangle$, where $\overline{p_i}$ and $\overline{q_i}$ are the accumulators—stacks containing variables (intuitively, pointers). The intuition is that whenever we see an application, we put the arguments in the accumulator, and take them out when they are needed for entering a closure. The argument-accumulating semantics is given in Figure 6. The A-ACCUM rule is introduced. It deals with applications by putting arguments in the accumulator. All the other rules dealing with application are limited to applications to the empty tuple of arguments, while the "real" arguments are now stored in the accumulator. The application rules APP2 and APP3 from the previous semantics are now merged to form the A-APP2.5 rule. It is possible because the spare arguments do not need to be held back in the second premise of APP3, but they travel up the derivation tree in the accumulator and can be accessed when needed. Note that only constructors and applications to empty tuple of arguments are now normal forms.

The argument-accumulating semantics is sound and complete with respect to the STG language natural semantics.

PROPOSITION 3 (soundness and completeness). *If e is a closed expression, then:*

1. $(\emptyset : e \downarrow \Delta : C\ \overline{p_i})$ *iff* $\langle \emptyset, e, \varepsilon\rangle \Downarrow \langle \Delta, C\ \overline{p_i}, \varepsilon\rangle$,
2. $(\emptyset : e \downarrow \Delta : p\ \overline{p_i})$ *iff* $\langle \emptyset, e, \varepsilon\rangle \Downarrow \langle \Delta, p, \overline{p_i}\rangle$.

4.2 Replacing substitution with environment

The next step toward an abstract machine is introduction of environments. This step is made simpler by the fact that in the argument-accumulating semantics for well-formed configurations we substitute only pointers for bound variables. Thus for each expression there will be an associated environment, which will bind addresses in the heap (pointers) with the free variables of the expression.

The *explicit environment semantics* is shown in Figure 7. It proves judgments of the form $\langle \Gamma, e, \sigma, \overline{q_i}\rangle \overline{\Downarrow} \langle \Delta, w, \tau, \overline{r_i}\rangle$, where σ and τ are environments, i.e., partial functions from variables to variables.

We will denote the set of all environments by ENV. Γ, Δ and Θ are heaps of a new kind: their domain are variables from the set $\mathfrak{X}$ and their codomain are closures, i.e., elements of $lf \times ENV$. $FV(l)$ stands for the set of all free variables of the lambda form l. The environment $\sigma\restriction X$ is a subset of an environment σ with its domain trimmed to the set of variables X, $\sigma[\overline{x_i/p_i}]$ is an extension of σ by

$$\langle \Gamma, C\ \overline{p_i}, \varepsilon\rangle \ \Downarrow\!\Downarrow\ \langle \Gamma, C\ \overline{p_i}, \varepsilon\rangle \qquad \text{A-Con}$$

$$\frac{\langle \Gamma, p, (p_1 \dots p_m\ q_1 \dots q_n)\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}{\langle \Gamma, (p\ p_1 \dots p_m), q_1 \dots q_n\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}\ m > 0 \qquad \text{A-Accum}$$

$$\langle \Gamma\{p \mapsto \lambda_N\ x_1 \dots x_m.e\}, p, (p_1 \dots p_n)\rangle \ \Downarrow\!\Downarrow\ \langle \Gamma, p, (p_1 \dots p_n)\rangle \ \text{ where } n < m \qquad \text{A-App1}$$

$$\frac{\langle \Gamma, e[x_1/p_1 \dots x_m/p_m], p_{m+1} \dots p_n\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}{\langle \Gamma\{p \mapsto \lambda_N\ x_1 \dots x_m.e\}, p, (p_1 \dots p_n)\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}\ m \leqslant n \qquad \text{A-App2.5}$$

$$\frac{\langle \Gamma, e, \varepsilon\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, C\ \overline{q_i}, \varepsilon\rangle}{\langle \Gamma\{p \mapsto \lambda_U.e\}, p, \varepsilon\rangle \ \Downarrow\!\Downarrow\ \langle \Delta \oplus [p \mapsto \lambda_N.C\ \overline{q_i}], C\ \overline{q_i}, \varepsilon\rangle} \qquad \text{A-App4}$$

$$\frac{\begin{array}{c}\langle \Gamma, e, \varepsilon\rangle \ \Downarrow\!\Downarrow\ \langle \Delta\{q \mapsto \lambda_N\ x_1 \dots x_k\ x_{k+1} \dots x_n.f\}, q, (q_1 \dots q_k)\rangle \\ \langle \Delta \oplus [p \mapsto \lambda_N\ x_{k+1} \dots x_n.f[x_1/q_1 \dots x_k/q_k]], q, (q_1 \dots q_k\ p_1 \dots p_m)\rangle \ \Downarrow\!\Downarrow\ \langle \Theta, w, \overline{r_i}\rangle\end{array}}{\langle \Gamma\{p \mapsto \lambda_U.e\}, p, (p_1 \dots p_m)\rangle \ \Downarrow\!\Downarrow\ \langle \Theta, w, \overline{r_i}\rangle} \qquad \text{A-App5}$$

$$\frac{\langle \Gamma \oplus \overline{[p_i \mapsto lf_i[\overline{x_i/p_i}]]}, e[\overline{x_i/p_i}], \overline{r_i}\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, w, \overline{s_i}\rangle}{\langle \Gamma, \textbf{letrec}\ \overline{x_i = lf_i}\ \textbf{in}\ e, \overline{r_i}\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, w, \overline{s_i}\rangle}\ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma) \qquad \text{A-Letrec}$$

$$\frac{\langle \Gamma, e, \varepsilon\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, C_k\ \overline{p_j}, \varepsilon\rangle \qquad \langle \Delta, e_k[\overline{x_{kj}/p_j}], \overline{q_i}\rangle \ \Downarrow\!\Downarrow\ \langle \Theta, w, \overline{r_i}\rangle}{\langle \Gamma, \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \overline{q_i}\rangle \ \Downarrow\!\Downarrow\ \langle \Theta, w, \overline{r_i}\rangle} \qquad \text{A-Case}$$

Figure 6. The argument-accumulating semantics

$\overline{[x_i/p_i]}$. We also write $e[\sigma]$ when we use the environment σ as a substitution. Intuitively, the argument accumulator stores pointers.

The trimming of environments is not essential for the soundness and completeness of the explicit environment semantics. We decided to leave the trimming in the E-Letrec rule and in the rules performing updates to indicate that closures in the heap are an abstraction of real-life closures in a real-life heap (where the closures contain only values for variables that are actually free in the function).

To avoid confusion, we will now denote heaps used in the argument-accumulating semantics by A-heap and heaps used in the explicit environment semantics by E-heap.

Definition 4. *1. An expression is* env-well-formed *iff the set of all its variables (both bound and free) is a subset of* BOUND.
2. An environment σ *is* env-well-formed *iff it is a function from* BOUND *to* $\mathit{POINTERS}$.
3. An expression e *is* closed by *an environment* σ *iff* $\mathit{FV}(e) \subseteq \mathit{Dom}(\sigma)$.
4. A E-heap Γ *with* $\mathit{Dom}(\Gamma) \subseteq \mathit{POINTERS}$ *is* env-well-formed *iff for each closure* (e, σ) *in* Γ *both* e *and* σ *are env-well-formed and* e *is closed by* σ.

The correspondence between an A-heap and a E-heap is defined as follows:

Definition 5. *An A-heap* Γ *and a E-heap* $\Gamma^\bullet$ *are* similar *iff:*

1. $\mathit{Dom}(\Gamma) = \mathit{Dom}(\Gamma^\bullet)$,
2. $\Gamma^\bullet$ *is env-well-formed,*
3. *for any* $p \in \mathit{POINTERS}$, *if* $\Gamma(p) = (\lambda_\nu \overline{y_i}.\widetilde{e})$ *and* $\Gamma^\bullet(p) = (\lambda_\mu \overline{x_i}.e, \tau)$ *then* $\overline{y_i} = \overline{x_i}$, $\widetilde{e} = e[\tau]$ *and* $\nu = \mu$.

By $\Gamma^\bullet$ we will denote a E-heap that is similar to an A-heap Γ.

Proposition 6 (soundness and completeness). *If* e *is a closed expression, then:*

1. *If* $\langle \emptyset, e, \varepsilon\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, \widetilde{w}, \overline{q_i}\rangle$ *then there exist* $\Delta^\bullet, w, \tau$ *s.t.* $\langle \emptyset, e, \emptyset, \varepsilon\rangle \ \overline{\Downarrow}\ \langle \Delta^\bullet, w, \tau, \overline{q_i}\rangle$ *and* $\widetilde{w} = w[\tau]$.
2. *If* $\langle \emptyset, e, \emptyset, \varepsilon\rangle \ \overline{\Downarrow}\ \langle \Delta^\bullet, w, \tau, \overline{q_i}\rangle$ *then there exists* Δ *s.t.* $\langle \emptyset, e, \varepsilon\rangle \ \Downarrow\!\Downarrow\ \langle \Delta, w[\tau], \overline{q_i}\rangle$.

4.3 Transformation to Defunctionalized CPS

We are now ready to perform the D-CPS transformation. It may be done in exactly the same manner as described in Section 3, and its result is shown in Figure 8. We call the resulting semantics the D-CPS semantics.

The rules E-App4 and E-App5 give rise to two continuations, but—since the rules for them are the same—we may merge them into a single continuation $\mathbf{UPD}$ (for "update"). The continuation for the E-Case is named $\mathbf{ALT}$ (for "alternatives").

Proposition 7 (soundness and completeness). *For any* Γ, e, σ *and* $\overline{p_i}$, *the following holds:*
$\langle \Gamma, e, \sigma, \overline{p_i}\rangle \ \overline{\Downarrow}\ \langle \Delta, f, \gamma, \overline{q_i}\rangle$ *iff* $\mathcal{E}\langle \Gamma, e, \sigma, \overline{p_i}, \varepsilon\rangle \nwarrow \langle \Delta, f, \gamma, \overline{q_i}\rangle$.

4.4 From the D-CPS semantics to the abstract machine

The extraction of an abstract machine from the D-CPS semantics may be done exactly as described in Section 3. The resulting D-CPS machine is shown in Figure 9. The soundness and completeness is trivial and may be formulated as follows:

Proposition 8 (soundness and completeness). *For any* Γ, e, σ *and* $\overline{p_i}$, $\mathcal{E}\langle \Gamma, e, \sigma, \overline{p_i}, \varepsilon\rangle \nwarrow \langle \Delta, f, \gamma, \overline{q_i}\rangle$ *iff* $\mathcal{E}\langle \Gamma, e, \sigma, \overline{p_i}, \varepsilon\rangle \stackrel{dcps}{\Longrightarrow}{}^* \langle \Delta, f, \gamma, \overline{q_i}\rangle$.

4.5 The STG machine

In this section we show that the D-CPS machine is in fact the Spineless Tagless G-machine in disguise and we compare the resulting machine to the original formulation by Peyton Jones and Salkild.

4.5.1 Merging and splitting of rules

First, we notice that Q$\mathcal{A}$-App4 is of the form

$$\dots \stackrel{dcps}{\Longrightarrow} \mathcal{E}\langle \dots C\ \overline{x_i} \dots \rangle$$

$$\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon\rangle \mathrel{\overline{\Downarrow}} \langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon\rangle \qquad \text{E-CON}$$

$$\frac{\langle \Gamma, x, \sigma, (\sigma x_1 \dots \sigma x_m\ q_1 \dots q_n)\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\langle \Gamma, (x\ x_1 \dots x_m), \sigma, q_1 \dots q_n\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m > 0 \qquad \text{E-ACCUM}$$

$$\langle \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n\rangle \mathrel{\overline{\Downarrow}} \langle \Gamma, x, \sigma, p_1 \dots p_n\rangle \quad \text{where } n < m \qquad \text{E-APP1}$$

$$\frac{\langle \Gamma, e, \tau[x_1/p_1 \dots x_m/p_m], p_{m+1} \dots p_n\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\langle \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m \leqslant n \qquad \text{E-APP2.5}$$

$$\frac{\langle \Gamma, e, \tau, \varepsilon\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon\rangle}{\langle \Gamma\{\sigma x \mapsto (\lambda_U.e, \tau)\}, x, \sigma, \varepsilon\rangle \mathrel{\overline{\Downarrow}} \langle \Delta \oplus [\sigma x \mapsto (\lambda_N.C\ \overline{x_i}, \gamma\restriction\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon\rangle} \qquad \text{E-APP4}$$

$$\frac{\begin{array}{c}\langle \Gamma, e, \tau, \varepsilon\rangle \mathrel{\overline{\Downarrow}} \langle \Delta\{\gamma y \mapsto (\lambda_N\ x_1 \dots x_k\ x_{k+1} \dots x_n.f, \mu)\}, y, \gamma, q_1 \dots q_k\rangle \\ \langle \Delta \oplus [\sigma x \mapsto (\lambda_N\ x_{k+1} \dots x_n.f, \mu[x_1/q_1 \dots x_k/q_k])], y, \gamma, q_1 \dots q_k\ p_1 \dots p_m\rangle \mathrel{\overline{\Downarrow}} \langle \Theta, w, \xi, \overline{r_i}\rangle\end{array}}{\langle \Gamma\{\sigma x \mapsto (\lambda_U.e, \tau)\}, x, \sigma, p_1 \dots p_m\rangle \mathrel{\overline{\Downarrow}} \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{E-APP5}$$

$$\frac{\langle \Gamma \oplus \overline{[p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}]\restriction \mathrm{FV}(lf_i))]}, e, \sigma[\overline{x_i/p_i}], \overline{r_i}\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, w, \gamma, \overline{s_i}\rangle}{\langle \Gamma, \textbf{letrec}\ \overline{x_i = lf_i}\ \textbf{in}\ e, \sigma, \overline{r_i}\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, w, \gamma, \overline{s_i}\rangle}\ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma) \qquad \text{E-LETREC}$$

$$\frac{\langle \Gamma, e, \sigma, \varepsilon\rangle \mathrel{\overline{\Downarrow}} \langle \Delta, C_k\ \overline{y_j}, \gamma, \varepsilon\rangle \qquad \langle \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}\rangle \mathrel{\overline{\Downarrow}} \langle \Theta, w, \xi, \overline{r_i}\rangle}{\langle \Gamma, \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}\rangle \mathrel{\overline{\Downarrow}} \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{E-CASE}$$

Figure 7. The explicit environment semantics

and Q-CON is the only rule of the form

$$\mathcal{E}\langle \dots C\ \overline{x_i} \dots\rangle \overset{dcps}{\Longrightarrow} \dots$$

Therefore we can replace Q$\mathcal{A}$-APP4 with the following:

$$\begin{array}{c}\mathcal{A}\langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i}\rangle \\ \overset{dcps}{\Longrightarrow} \mathcal{A}\langle \Delta \oplus [p \mapsto \lambda_N.C\ \overline{x_i}\ (\gamma\restriction\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i}\rangle\end{array}$$

We can also split the HALT rule into two rules, one for each kind of normal forms:

$$\mathcal{A}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \varepsilon\rangle \overset{dcps}{\Longrightarrow} \langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon\rangle \qquad \text{Q-HALT-CON}$$

$$\mathcal{A}\langle \Gamma, x, \sigma, \overline{p_i}, \varepsilon\rangle \overset{dcps}{\Longrightarrow} \langle \Gamma, x, \sigma, \overline{p_i}\rangle \qquad \text{Q-HALT-APP}$$

The expression on the left-hand side of the rule HALT-APP is an application with zero arguments (x), since the only rule of the form $\dots \overset{dcps}{\Longrightarrow} \mathcal{A}\langle \dots w \dots\rangle$ where w is an application is Q-APP1, in which w has no arguments.

We will now merge the Q-APP1 rule with Q-HALT-APP and, separately, with Q-APP5. We replace these three rules with the following two:

$$\begin{array}{c}\mathcal{E}\langle \Gamma\{\sigma x \mapsto \lambda_N\ x_1 \dots x_m.e\ \tau\}, x, \sigma, p_1 \dots p_n, \varepsilon\rangle \\ \overset{dcps}{\Longrightarrow} \langle \Gamma, x, \sigma, p_1 \dots p_n\rangle \quad \text{where } n < m,\end{array}$$

$$\begin{array}{c}\mathcal{E}\langle \Delta\{\gamma y \mapsto \lambda_N\ x_1 \dots x_k\ x_{k+1} \dots x_n.f\ \mu\}, y, \gamma, q_1 \dots q_k \\ \mathbf{UPD}(p, p_1 \dots p_n) : \overline{S_i}\rangle \quad \text{where } k < n \\ \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Delta \oplus [p \mapsto \lambda_N\ x_{k+1} \dots x_n.f\ \mu[x_1/q_1 \dots x_k/q_k]], \\ y, \gamma, q_1 \dots q_k\ p_1 \dots p_n, \overline{S_i}\rangle.\end{array}$$

4.5.2 Introduction of the STG instructions

So far we have used two kinds of "instructions:" *eval* ($\mathcal{E}$) and *apply* ($\mathcal{A}$), where $\mathcal{E}$ intuitively means that we are currently evaluating an expression, and $\mathcal{A}$ means that we have just finished evaluating an expression and we need an element from the stack of continuations to go on.

After the merging of rules, we notice that the $\mathcal{A}$ instruction applies now only to the rules for constructors. We will dub such rules **return**. We also split the $\mathcal{E}$ instruction into two: one for application to zero arguments (we will dub such rules **enter**), and for any other kind of expression (dubbed **eval**). We also notice that now there is no rule for configurations of the form $\langle \mathbf{eval}, \Gamma, x, \dots\rangle$, where x is a single variable, therefore we abandon the side condition $m > 0$ in the Q-ACCUM rule, so that evaluating an application to zero arguments means entering the closure it represents.

The changes in the machine are summarized in Figure 10. We claim that this machine is the STG machine up to some minor details described in the following subsection. As evidence, in Figure 10 we put numbers next to names of the rules; these numbers are the numbers of the transition rules in Peyton Jones and Salkild's original STG machine [11] (not all numbers are present since our machine lacks primitive arithmetics and default alternatives in **case** expressions, and the HALT rules are not featured in the original STG machine).

4.5.3 Soundness and completeness

We can combine all the local soundness and completeness theorems to formulate our main proposition. Recall that by Δ and $\Delta^\bullet$ we denote a pair of similar heaps (Definition 5).

PROPOSITION 9 (completeness). *For a closed expression e, the following hold:*

1. *If $(\emptyset : e \downarrow \Delta : p\ \overline{p_i})$, there exist $\Delta^\bullet$, x and σ such that $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon\rangle \overset{stg}{\Longrightarrow}^* \langle \Delta^\bullet, x, \sigma, \overline{p_i}\rangle$ and $\sigma x = p$.*
2. *If $(\emptyset : e \downarrow \Delta : C\ \overline{p_i})$, there exist $\Delta^\bullet$, $\overline{x_i}$ and σ such that: $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon\rangle \overset{stg}{\Longrightarrow}^* \langle \Delta^\bullet, C\ \overline{x_i}, \sigma, \varepsilon\rangle$ and $\sigma\overline{x_i} = \overline{p_i}$.*

PROPOSITION 10 (soundness). *For a closed expression e, the following hold:*

1. *If $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon\rangle \overset{stg}{\Longrightarrow}^* \langle \Delta^\bullet, x, \sigma, \overline{p_i}\rangle$ then there exists Δ such that $(\emptyset : e \downarrow \Delta : (\sigma x)\ \overline{p_i})$.*

$$\mathcal{A}\langle \Gamma, w, \sigma, \overline{p_i}, \varepsilon\rangle \;\nwarrow\; \langle \Gamma, w, \sigma, \overline{p_i}\rangle \qquad \text{D-HALT}$$

$$\frac{\mathcal{A}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle} \qquad \text{D-CON}$$

$$\frac{\mathcal{E}\langle \Gamma, x, \sigma, (\sigma x_1 \ldots \sigma x_m\ q_1 \ldots q_n), \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma, (x\ x_1 \ldots x_m), \sigma, q_1 \ldots q_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m > 0 \qquad \text{D-ACCUM}$$

$$\frac{\mathcal{A}\langle \Gamma, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ n < m \qquad \text{D-APP1}$$

$$\frac{\mathcal{E}\langle \Gamma, e, \tau[x_1/p_1 \ldots x_m/p_m], p_{m+1} \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m \leqslant n \qquad \text{D-APP2.5}$$

$$\frac{\mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_N.C\ \overline{x_i}, \gamma|\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{s_i}\rangle}{\mathcal{A}\langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{s_i}\rangle} \qquad \text{D}\mathcal{A}\text{-APP4}$$

$$\frac{\mathcal{E}\langle \Gamma, e, \tau, \varepsilon, \mathbf{UPD}(\sigma x, \overline{r_i}) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{q_i}\rangle}{\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_U.e, \tau)\}, x, \sigma, \overline{r_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{q_i}\rangle} \qquad \text{D}\mathcal{E}\text{-APP4.5}$$

$$\frac{\mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_N\ x_{k+1} \ldots x_n.f, \mu[x_1/q_1 \ldots x_k/q_k])], y, \gamma, q_1 \ldots q_k\ p_1 \ldots p_m, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle}{\mathcal{A}\langle \Delta\{\gamma y \mapsto (\lambda_N\ x_1 \ldots x_k\ x_{k+1} \ldots x_n.f, \mu)\}, y, \gamma, q_1 \ldots q_k, \mathbf{UPD}(p, p_1 \ldots p_m) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{D}\mathcal{A}\text{-APP5}$$

$$\frac{\mathcal{E}\langle \Gamma \oplus \overline{[p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}]\restriction \mathrm{FV}(lf_i))]}, e, \sigma[\overline{x_i/p_i}], \overline{r_i}\ \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{s_i}\rangle}{\mathcal{E}\langle \Gamma, \mathbf{letrec}\ \overline{x_i = lf_i}\ \mathbf{in}\ e, \sigma, \overline{r_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{s_i}\rangle}\ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma) \qquad \text{D-LETREC}$$

$$\frac{\mathcal{E}\langle \Gamma, e, \sigma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{D}\mathcal{E}\text{-CASE}$$

$$\frac{\mathcal{E}\langle \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle}{\mathcal{A}\langle \Delta, C_k\ \overline{y_j}\ \gamma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{D}\mathcal{A}\text{-CASE}$$

Figure 8. The D-CPS semantics

2. *If* $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon\rangle \overset{stg}{\Longrightarrow}^{*} \langle \Delta^{\bullet}, C\ \overline{p_i}, \sigma, \varepsilon\rangle$ *then there exists* Δ *such that* $(\emptyset : e \downarrow \Delta : C\ \overline{p_i})$.

4.5.4 Design differences

The original STG machine was designed, while ours was derived. Still, the design choices we have made when introducing successive semantics have a great impact on the final machine. In this section we compare the STG machine from Figure 10 with the machine described by Peyton Jones.

Stacks Our formulation of the STG machine has two stacks (argument accumulator and continuation stack), while the original STG machine has three stacks (argument stack, return stack, and update stack). The argument accumulator works exactly like the argument stack of the original STG machine, while the continuation stack covers the return stack and the update stack. Our two-stack machine can be simulated by a single-stack machine in exactly the same way as the original STG machine [11], since the harmonics of the stack operations in both machines are identical. However, we find the formulation with two stacks particularly clean (as opposed to single stack), since the return-update stack may be seen as a continuation (in particular, when we *finish* evaluation, we *continue* with an update), while the argument stack may not (we do not use arguments with computed normal forms, instead we constantly shuffle the argument stack during evaluation).

Instructions and environments The **enter** and **return** instructions are formulated slightly differently: here, **enter** takes a variable x and an environment σ, and then enters the closure under the address σx, while the original **enter** rule takes the address. Similarly, **return** expects a constructor expression (a constructor name and a tuple of variables) and an environment, while in the original formulation it needs a constructor name and a tuple of addresses. The equivalence of both approaches is almost trivial.

Problems with ill-typed expressions As pointed out by Encina and Peña [4], the original STG machine might not behave as expected for some ill-formed programs. For example, consider the following program (which would be ill-typed in any reasonable strongly-typed language):

$$\begin{array}{rl} \mathbf{letrec} & id = \lambda_N\, x.x \\ & c = \lambda_U.C \\ & f = \lambda_U.\mathbf{case}\ id\ \mathbf{of}\ C \to D \\ \mathbf{in} & f\ c \end{array}$$

The original machine allocates the declarations, pushes the pointer to c on the argument stack and continues with evaluation of f. The **case** expression in f first computes id, which evaluates to C (we have an argument for it on the argument stack!). The whole expression finally evaluates to D.

Even though the STG language is not typed, the intuition is that the evaluation should be broken in the **case** expression, since id should not get its argument. Indeed, it is the case when the three stacks of the original STG machine are simulated by a single stack, where the argument is "guarded" by an element containing **case** alternatives. This is a minor flaw, since for well-typed programs

$$\mathcal{A}\langle \Gamma, w, \sigma, \overline{p_i}, \varepsilon\rangle \overset{dcps}{\Longrightarrow} \langle \Gamma, w, \sigma, \overline{p_i}\rangle \qquad \text{Q-HALT}$$

$$\mathcal{E}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{A}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \qquad \text{Q-CON}$$

$$\mathcal{E}\langle \Gamma, (x\ x_1 \dots x_m), \sigma, q_1 \dots q_n, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, x, \sigma, (\sigma x_1 \dots \sigma x_m\ q_1 \dots q_n), \overline{S_i}\rangle \quad \text{where } m > 0 \qquad \text{Q-ACCUM}$$

$$\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{A}\langle \Gamma, x, \sigma, p_1 \dots p_n, \overline{S_i}\rangle \quad n < m \qquad \text{Q-APP1}$$

$$\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, e, \tau[x_1/p_1 \dots x_m/p_m], p_{m+1} \dots p_n, \overline{S_i}\rangle \quad m \leqslant n \qquad \text{Q-APP2.5}$$

$$\mathcal{A}\langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_N.C\ \overline{x_i}, \gamma\restriction\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i}\rangle \qquad \text{Q}\mathcal{A}\text{-APP4}$$

$$\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_U.e, \tau)\}, x, \sigma, \overline{r_i}, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, e, \tau, \varepsilon, \mathbf{UPD}(\sigma x, \overline{r_i}) : \overline{S_i}\rangle \qquad \text{Q}\mathcal{E}\text{-APP4.5}$$

$$\mathcal{A}\langle \Delta\{\gamma y \mapsto (\lambda_N\ x_1 \dots x_k\ x_{k+1} \dots x_n.f, \mu)\}, y, \gamma, q_1 \dots q_k, \mathbf{UPD}(p, p_1 \dots p_m) : \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_N\ x_{k+1} \dots x_n.f, \mu[x_1/q_1 \dots x_k/q_k])], y, \gamma, (q_1 \dots q_k\ p_1 \dots p_m), \overline{S_i}\rangle \qquad \text{Q}\mathcal{A}\text{-APP5}$$

$$\overset{dcps}{\Longrightarrow} \frac{\mathcal{E}\langle \Gamma, \mathbf{letrec}\ \overline{x_i = lf_i}\ \mathbf{in}\ e, \sigma, \overline{r_i}, \overline{S_i}\rangle}{\mathcal{E}\langle \Gamma \oplus [p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}]\restriction \mathrm{FV}(lf_i))], e, \sigma[\overline{x_i/p_i}], \overline{r_i}, \overline{S_i}\rangle} \quad \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma) \qquad \text{Q-LETREC}$$

$$\mathcal{E}\langle \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, e, \sigma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \qquad \text{Q}\mathcal{E}\text{-CASE}$$

$$\mathcal{A}\langle \Delta, C_k\ \overline{y_j}, \gamma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}, \overline{S_i}\rangle \qquad \text{Q}\mathcal{A}\text{-CASE}$$

Figure 9. The D-CPS abstract machine

the machine with one stack behaves exactly like the machine with three stacks.

Our formulation avoids this problem by storing the argument stack in the continuation when evaluating the inner expression in the D$\mathcal{E}$-CASE rule. The original STG machine behavior would be obtained if we leave the argument stack for the first premise in the argument-accumulating semantics in the A-Case rule:

$$\frac{\langle \Gamma, e, \overline{q_i}\rangle \Downarrow \langle \Delta, C_k\ \overline{p_j}, \overline{r_i}\rangle \quad \langle \Delta, e_k[\overline{x_{kj}/p_j}], \overline{r_i}\rangle \Downarrow \langle \Theta, w, \overline{s_i}\rangle}{\langle \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \overline{q_i}\rangle \Downarrow \langle \Theta, w, \overline{s_i}\rangle}$$

An alternative update rule The APP5 rule from our natural semantics (and its successors: A-APP5, E-APP5 and S-APP1APP5) are problematic to implement in a compiler since we cannot create new expressions on the fly. Moreover, updating the heap as follows:

$$\Delta \oplus [p \mapsto (\lambda_N\ x_{k+1} \dots x_n.f, \mu[x_1/q_1 \dots x_k/q_k])]$$

would in practice mean modifying the expressions because environments are precomputed, i.e., each variable in an expression is statically bound to the n-th element on the stack, the n-th slot of the current closure, or a register. One solution is to update the heap not with the partially applied lambda-form, but with the computed normal form:

$$\frac{\Gamma : e \downarrow \Delta\{q \mapsto \lambda_N\ x_1 \dots x_k\ x_{k+1} \dots x_n.f\} : q\ q_1 \dots q_k \quad \Delta \oplus [p \mapsto \lambda_N.\ q\ q_1 \dots q_k] : q\ q_1 \dots q_k\ p_1 \dots p_n \downarrow \Theta : w}{\Gamma\{p \mapsto \lambda_U.e\} : p\ p_1 \dots p_n \downarrow \Theta : w}$$

Now it is sufficient to create code for partial applications for all possible number of arguments. This approach is used in the original formulation of the STG machine as an alternative update rule. [1]

[1] The original STG rule to which we would come via all our transformations is called 17a.

4.6 Extensions

An advantage of using a constructive derivation method for obtaining an abstract machine from the underlying natural semantics is its scalability. Any changes in the latter smoothly ensue in the machine.

A useful example of such an operation is adding primitive (unboxed) arithmetics. We need to include numerical literals and operators as base constructions in the language, and to extend the STG natural semantics with a few intuitive rules, with no need to alter any of the original rules. If careful when introducing environments, we can augment the machine with a primitive arithmetics similar to the one presented in the original papers [11, 12].

In the same way we can add other concepts that are easy to express in the natural semantics, but may not be that trivial in the machine, like (monadic) input/output, or the Haskell *seq* operator.

5. Formalization in Coq

One of our main contributions is a formalization of the derivation of the STG machine in the Coq proof assistant. The complete source code with documentation can be found at: `http://www.ii.uni.wroc.pl/~dabi/Publications/Haskell10/stg-in-coq/`. The development is about 7500 lines of code long (ca. 100 definitions and 230 theorems).

The STG language is formalized as a Coq datatype. Variables are represented either by an abstract type `atom`, which models the free variables of an expression, or as de Bruijn indices for bound variables (they correspond to the *POINTERS* and *BOUND* sets, respectively). We use de Bruijn notation to handle all three kinds of binders: arguments in lambda-forms, parameters of constructors in alternatives, and names of lambda-forms in **letrec** definitions. The Coq definition reads as follows:

$$\langle \mathbf{return}, \Gamma, C\ \overline{x_i}, \sigma, \overline{p_i}, \varepsilon\rangle \overset{stg}{\Longrightarrow} \langle \Gamma, C\ \overline{x_i}, \sigma, \overline{p_i}\rangle \qquad \text{S-HALT-CON}$$

$$\langle \mathbf{enter}, \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n, \varepsilon\rangle \overset{stg}{\Longrightarrow} \langle \Gamma, x, \sigma, p_1 \dots p_n\rangle \quad n < m \qquad \text{S-APP1HALT}$$

$$\langle \mathbf{eval}, \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{return}, \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \qquad \text{S-CON (5)}$$

$$\langle \mathbf{eval}, \Gamma, (x\ x_1 \dots x_m), \sigma, q_1 \dots q_n, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{enter}, \Gamma, x, \sigma, (\sigma x_1 \dots \sigma x_m\ q_1 \dots q_n), \overline{S_i}\rangle \qquad \text{S-ACCUM (1)}$$

$$\begin{array}{l}\langle \mathbf{enter}, \Delta\{\gamma y \mapsto (\lambda_N\ x_1 \dots x_k\ x_{k+1} \dots x_n.f, \mu)\}, y, \gamma, q_1 \dots q_k, \mathbf{UPD}(p, p_1 \dots p_m) : \overline{S_i}\rangle \\ \overset{stg}{\Longrightarrow} \langle \mathbf{enter}, \Delta \oplus [p \mapsto (\lambda_N\ x_{k+1} \dots x_n.f, \mu[x_1/q_1 \dots x_k/q_k])], y, \gamma, (q_1 \dots q_k\ p_1 \dots p_m), \overline{S_i}\rangle \end{array} \quad k < n \qquad \text{S-APP1APP5 (17)}$$

$$\begin{array}{l}\langle \mathbf{enter}, \Gamma\{\sigma x \mapsto (\lambda_N\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n, \overline{S_i}\rangle \\ \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma, e, \tau[x_1/p_1 \dots x_m/p_m], p_{m+1} \dots p_n, \overline{S_i}\rangle \end{array} \quad m \leqslant n \qquad \text{S-APP2.5 (2)}$$

$$\langle \mathbf{return}, \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{return}, \Delta \oplus [p \mapsto (\lambda_N.C\ \overline{x_i}, \gamma\restriction\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i}\rangle \qquad \text{S}\mathcal{A}\text{-APP4CON (16)}$$

$$\langle \mathbf{enter}, \Gamma\{\sigma x \mapsto (\lambda_U.e, \tau)\}, x, \sigma, \overline{r_i}, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma, e, \tau, \varepsilon, \mathbf{UPD}(\sigma x, \overline{r_i}) : \overline{S_i}\rangle \qquad \text{S}\mathcal{E}\text{-APP4.5 (15)}$$

$$\begin{array}{l}\langle \mathbf{eval}, \Gamma, \mathbf{letrec}\ \overline{x_i = lf_i}\ \mathbf{in}\ e, \sigma, \overline{r_i}, \overline{S_i}\rangle \\ \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma \oplus [p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}]\restriction \mathrm{FV}(lf_i))], e, \sigma[\overline{x_i/p_i}], \overline{r_i}, \overline{S_i}\rangle \end{array} \quad \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma) \qquad \text{S-LETREC (3)}$$

$$\langle \mathbf{eval}, \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma, e, \sigma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \qquad \text{S}\mathcal{E}\text{-CASE (4)}$$

$$\langle \mathbf{return}, \Delta, C_k\ \overline{y_j}, \gamma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}, \overline{S_i}\rangle \qquad \text{S}\mathcal{A}\text{-CASE (6)}$$

Figure 10. The STG machine

```
Parameter atom : Set.
Inductive var : Set :=
| Index : nat -> var
| Atom : atom -> var.

Inductive expr : Set :=
| App : var -> list var -> expr
| Constr : constructor -> list var -> expr
| Letrec : list lambda_form -> expr -> expr
| Case : expr -> list alt -> expr
with lambda_form : Set :=
| Lf : upd_flag -> nat -> expr -> lambda_form
with alt : Set :=
| Alt : constructor -> nat -> expr -> alt.
```

The `nat` arguments of constructors `Lf` and `Alt` determine how many arguments a lambda-form or alternative binds. The definitions in `Letrec` are enumerated top-down. For example, consider the following expression:

$$\begin{array}{rl}\mathbf{letrec} & f = \lambda_N x\ y.\ f\ g\ x\ y \\ & g = \lambda_N x.\ x \\ \mathbf{in} & f\ g\end{array}$$

The corresponding Coq term is:

```
Letrec
 [Lf Dont_update 2
     (App (Index 2) [Index 3, Index 1, Index, 0]),
  Lf Dont_update 1
     (App (Index 0) nil)]
 (App (Index 0) [Index 1])
```

well-formed terms we consider terms that are locally closed, i.e., in which none of the variables is an index exceeding the number of surrounding binders (which corresponds to Definition 1). In the semantics with environments we allow a variable to be an exceeding index if it is in the domain of the associated environment (as in Definition 4). Hence, our approach is not "locally nameless", as we work on free de Bruijn indices.

The natural semantics and abstract machines are defined as inductive predicates. Heaps are represented by partial functions, for example the type of A-heaps is:

```
var -> option lambda_form.
```

The type of the predicate representing the natural semantics for the STG language is:

```
heapA -> expr -> heapA -> expr -> Prop,
```

while the type of transitions of the STG machine is:

```
configuration -> configuration -> Prop,
```

where `configuration` is equal to:

```
instruction * heapB * expr * env * vars * stack.
```

The soundness and completeness theorems for each semantics are generalized to obtain stronger induction hypothesis, and proven by the standard Coq structural induction, or—if necessary—by a well-founded induction on the height of derivations of judgments.

6. Related work

The idea of deriving lazy machines from natural semantics was first proposed by Sestoft [15]. He used an informal method to change rules for constructing derivations in natural semantics into rules for constructing a sequence of machine states. Then Mountjoy suggested that the same method for an extended semantics may lead to a machine that is closer to STG, and gave a proof of equivalence of some more elaborate abstract machines (but still far from the

STG machine) [10]. The work of Mountjoy was continued by Encina and Peña [4–6]. They used similar methods to invent STG-like machines and gave detailed proofs of their equivalence with an initial natural semantics.[2]

Though our approach may at first seem similar to the Encina and Peña's, it is based on different principles. To underscore the differences, we will examine the four main concepts: languages, semantics, derivations and abstract machines.

Languages In their articles, Encina and Peña introduce two new languages, both bearing the same name Fun. While neither of them is very different from STG, they were designed to fit the sole purpose of proving equivalence of a semantics and a machine.

Our approach, in turn, is to take the well-known STG language exactly as introduced by Peyton Jones, and give it a natural semantics, which is an interesting challenge even outside the context of deriving abstract machines. Nevertheless, starting with the natural semantics for *the* STG language was the key to obtaining *the* STG machine.

Semantics In our work, semantics for **letrec** and **case** expressions are similar to Encina and Peña's. They follow the approach of Launchbury and Sestoft. The key difference is in the treatment of multiple λ-binders and partial applications.

The two semantics for both Fun languages consequently evaluate partial applications by allocation in the heap. They allocate either a primitive heap element *pap*, or the lambda-form with the actual arguments substituted by the corresponding prefix of formal arguments. Though in the machine this allocation may be fused with an update, we do not find such solution elegant when concerning natural semantics.

Encina and Peña admit that their semantics, just as their languages, are tailored for the transformation into a particular machine. Our ambition, on the other hand, is to propose a more general and intuitive natural semantics, ready for any other formal reasoning, like preservation of semantics by program transformations in optimizing compilers.

We are also the first to address update flags in the semantics, which, if assigned correctly by a static program analysis, lead to a boost of performance.

Derivations Encina and Peña present their machines, but they do not explain how they invented them. They only refer to Sestoft's approach, who used his intuition of flattening derivation trees into sequences of machine transitions. This is hardly a derivation understood as a transformation from one entity (in this case a semantics) into another (an abstract machine) using a well-defined method. Moreover, their machines do not implement exactly the same evaluation model as their semantics (for example, S3 from [6] allocates more closures then the corresponding machine).

Our machine is a result of a method strongly inspired by a well-known transformation of programs, which preserves most important properties, including the evaluation model.

Abstract machines Both Fun languages are different than STG, thus their STG-like machines differ from the original STG machine. The most striking difference is the lack of `enter`, `eval` and `return` instructions which are STG-tuned incarnations of the eval ($\mathcal{E}$) and apply ($\mathcal{A}$) instructions arising naturally from the D-CPS transformation.

[2] In [6] Encina and Peña present two machines: push/enter and eval/apply. We are interested only in the former, since it resembles the original STG machine presented in [11].

7. Conclusion and future work

We have presented the natural semantics underlying the Spineless Tagless G-machine as evidenced by Danvy et al.'s functional correspondence between evaluators and abstract machines. Thus, we have shown that the functional correspondence, when lifted to the level of operational semantics is still effective and furnish provably correct transformations of non-trivial natural semantics into non-trivial abstract machines, without the need to pull the latter out of thin air. In particular, we have shown that the STG machine, though originally obtained by refining simpler machines (the G-machine and the Spineless G-machine) is just another incarnation of the natural semantics we have introduced.

Our main result, i.e., the equivalence between the natural semantics and the STG machine, has two facets. First, it provides a proof of correctness of the STG machine with respect to the natural semantics that, in fact, is a generalization of the commonly accepted and well-understood semantics by Sestoft. From the compiler's perspective, this result can be seen as a formal justification of the compilation process of the language Haskell: an abstract functional language is given provably correct low-level semantics that facilitates imperative code generation [11].

Symmetrically, we provide a proof of correctness of our natural semantics with respect to the well-known operational semantics of Haskell given by the STG machine, which ensures that one can safely reason about the operational aspects of Haskell code using the natural semantics instead of the abstract machine.

Our ultimate goal is a Coq-certified compiler for a subset of Haskell and the present article is a first step towards it. Having an abstract machine, STG expressions can be easily compiled into a set of imperative instructions, which change the global state to mimic the execution of the STG machine. Formalized and verified in Coq, this process can be automatically transformed into a working compiler by the Coq program extraction mechanism (in fact, it has been done as the first author's MSc thesis). Combined with the result from this article, it yields a compiler to a virtual machine with respect to the natural semantics.

Acknowledgments

We would like to thank Małgorzata Biernacka and Filip Sieczkowski for numerous discussions and useful comments on this work as well as Jeremy Gibbons and the anonymous reviewers for helping us improve the presentation.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Miller [9], pages 8–19.

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004.

[3] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.

[4] Alberto de la Encina and Ricardo Peña. Formally deriving an STG machine. In Miller [9], pages 102–112.

[5] Alberto de la Encina and Ricardo Peña. Proving the correctness of the STG machine. In Ricardo Pena and Thomas Arts, editors, *IFL*, volume 2670 of *Lecture Notes in Computer Science*, pages 88–104. Springer, 2003.

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

[7] Haskell homepage: `http://www.haskell.org`.

[8] John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.

[9] Dale Miller, editor. *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, Uppsala, Sweden, August 2003. ACM Press.

[10] Jon Mountjoy. The spineless tagless G-machine, naturally. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 1, pages 163–173, Baltimore, Maryland, September 1998. ACM Press.

[11] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[12] Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 184–201, London, England, September 1989. ACM Press.

[13] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [14].

[14] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[15] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.