From Push/Enter to Eval/Apply by Program Transformation

Maciej Piróg Jeremy Gibbons
Department of Computer Science
University of Oxford
maciej.pirog@cs.ox.ac.uk jeremy.gibbons@cs.ox.ak

Push/enter and eval/apply are two calling conventions used in implementations of functional languages. In this paper, we explore the following observation: when considering functions with multiple arguments, the stack under the push/enter and eval/apply conventions behaves similarly to two particular implementations of the list datatype: the regular cons-list and a form of lists with lazy concatenation respectively. Along the lines of Danvy *et al.*'s functional correspondence between definitional interpreters and abstract machines, we use this observation to transform an abstract machine that implements push/enter into an abstract machine that implements eval/apply. We show that our method is flexible enough to transform the push/enter Spineless Tagless G-machine (which is the semantic core of the GHC Haskell compiler) into its eval/apply variant.

1 Introduction

There are two standard calling conventions used to efficiently compile curried multi-argument functions in higher-order languages: *push/enter* (PE) and *eval/apply* (EA). With the PE convention, the caller pushes the arguments on the stack, and jumps to the function body. It is the responsibility of the function to find its arguments, when they are needed, on the stack. With the EA convention, the caller first evaluates the function to a normal form, from which it can read the number and kinds of arguments the function expects, and then it calls the function body with the right arguments. The difference between the two calling conventions is thoroughly discussed by Marlow and Peyton Jones [12].

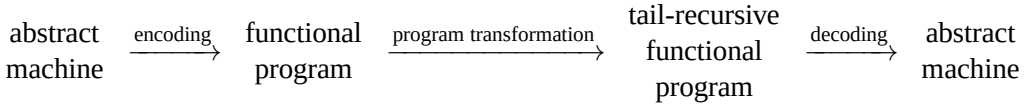
The terms PE and EA are also used in the literature to describe abstract machines, which deal directly with program expressions, and thus do not distinguish between 'code' of a function and a caller. Roughly speaking, a PE machine keeps on its stack some kind of *resources*, which can be accessed whenever needed. In case of curried functions, they are arguments, which can be freely used by a function body. An EA machine keeps on its stack *continuations*, which are very often synonymous with evaluation contexts [4, 7]. Such a machine has two kinds of configurations: 'eval' configurations evaluate (sub)expressions independently of the context, while 'apply' configurations construct a new expression from the obtained value and the next continuation.

Encouraged by the fact that some machines (such as the Spineless Tagless G-machine (STG) [12] used in the Glasgow Haskell Compiler) come in two versions realising the two conventions, one may suspect that the calling convention of a machine is orthogonal to the other aspects of its computational model. If so, is there a generic method of relating two such incarnations of a single machine? In this article, we give a partial answer to this question: we present a semi-mechanical method of derivation of an EA machine from its PE counterpart.

Our method is based on the correspondence between interpreters written in a functional language and abstract machines, studied extensively by Danvy *et al.* [1]. The key observation is that tail-recursive functions are similar in structure to state-transition systems, and thus tail-recursive interpreters of programming calculi correspond to abstract machines. Moreover, a flat tail-recursive structure can always

Olivier Danvy & Ugo de'Liguoro (Eds.): WoC'15 Postproceedings
EPTCS 212, 2016, pp. 53–62, doi:10.4204/EPTCS.212.4

be obtained via CPS transformation followed by defunctionalisation of the resulting continuations. This method can be used to transform between different abstract machines, as long as we can encode the initial abstract machine as a functional program:



Our method proposes a new tool for the 'encoding' part, as there are multiple choices of how one represents an abstract machine as a functional program. In particular, one can use different data structures to represent the type of the stack of the original machine. In detail, our derivation consists of the following steps:

- We start with a PE machine, which is a slight generalisation of a machine proposed by Krivine [10]. It normalises terms to weak-head normal form using the call-by-name evaluation strategy. Then, we give a (big-step [6]) encoding of this machine in Haskell. (We do not use any Haskell-specific features, and the choice of Haskell over any other functional language is in this case arbitrary. The 'lazy list concatenation' that we use in this paper is unrelated to the lazy semantics of Haskell.)
- We then perform semantics-preserving program transformations on this encoding. They are purely syntactic, that is they are not directed by any semantic understanding of the machine. In detail, we implement the stack using a list with lazy concatenation, and then apply the CPS transformation to reify the recursive calls of the operations that work on the stack as transitions of the machine.
- Finally, we decode the resulting EA machine from the program obtained by these transformations.

There are some strong similarities between the resulting machines and the application-related rules of the two versions of the STG machine [12] used in the Haskell GHC compiler. Thus our method proves to be useful even when dealing with real-life implementations.

2 The Krivine machine for a language with multi-argument binders

We first define λ^{MULT}, a version of the lambda calculus in which a λ-abstraction can bind more than one variable at a time, and in which an expression can be applied to a tuple of arguments. Its syntax is given by the following grammar, where $e, e_0, \ldots$ stand for expressions, $x, x_1, \ldots$ denote variables, and $\langle a_1 \ldots a_n \rangle$ is a non-empty tuple containing elements $a_1, \ldots, a_n$:

$$e ::= x \mid e_0 \langle e_1 \ldots e_n \rangle \mid \lambda \langle x_1 \ldots x_n \rangle . e$$

Intuitively, an expression $e_0 \langle e_1 \ldots e_n \rangle$ roughly corresponds to $e_0 e_1 \cdots e_n$ in the standard lambda calculus, while $\lambda \langle x_1 \ldots x_n \rangle . e$ corresponds to $\lambda x_1 . \ldots \lambda x_n . e$. Note that the λ^{MULT} calculus is different from the one obtained by simply adding finite products to the standard lambda calculus and treating λ-abstractions as uncurried functions: for example, the term $(\lambda \langle x_1 \ldots x_4 \rangle . e) \langle x_1 x_2 \rangle \langle x_3 x_4 \rangle$ has an 'arity' mismatch in the latter calculus.

We give an operational semantics to λ^{MULT} in terms of a PE abstract machine, which normalises a given expression to weak-head normal form using the call-by-name evaluation strategy. The machine is a natural extension of an abstract machine proposed by Krivine [10] (see also Biernacka and Danvy [2]), which was originally defined for a language with multi-argument binders, but with applications limited to a single argument (and not a tuple of arguments). Each machine configuration is a pair (e, s) consisting

of an expression and a stack of expressions. We write ':' for the stack constructor (the 'push' operation), and ε for the empty stack. We write $e[e_1/x_1,\ldots,e_n/x_n]$ to denote an expression e in which expressions $e_1,\ldots,e_n$ are substituted for variables $x_1,\ldots,x_n$ respectively. The transition rules are as follows:

$$(e_0\,\langle e_1\ldots e_n\rangle, s) \Rightarrow (e_0, e_1{:}\cdots{:}e_n{:}s) \qquad \text{(K-APP)}$$

$$(\lambda\langle x_1\ldots x_n\rangle.e, e_1{:}\cdots{:}e_n{:}s) \Rightarrow (e[e_1/x_1,\ldots,e_n/x_n], s) \qquad \text{(K-FUN)}$$

The initial configuration of the machine for an expression e is (e,ε). The transition K-APP evaluates applications: it pushes the arguments on the stack and continues with the head of the application. The transition K-FUN deals with abstractions: if the abstraction needs n arguments and there are at least n expressions on the stack, the machine continues with the body of the abstraction, with the appropriate variables substituted. The machine halts when there are no transitions that match the left-hand sides of K-APP or K-FUN, that is, when trying to evaluate an application for which there are not enough arguments on the stack or when trying to evaluate a free variable.

3 Haskell implementation

Now, we present a deep embedding of λ^{MULT} and the machine in Haskell. The definition of terms is as follows:

```
type Identifier = ...
data Term = Var Identifier
          | App Term [Term]
          | Fun [Identifier] Term

subst :: [(Term, Identifier)] -> Term -> Term
subst = ...
```

Neither the choice of the type of identifiers nor any concrete implementation of `subst` affect the derivation in any way. The Haskell encoding of the machine uses an abstract datatype representing the stack of the machine. It has two operations:

```
type Stack a = ...
push :: [a] -> Stack a -> Stack a
pop  :: Int -> Stack a -> Maybe ([a], Stack a)
```

In the above, the function `push` places a tuple of values on top of the stack. The function `pop n s` attempts to extract the first `n` values from the top of the stack `s`. If the stack contains too few elements, `pop` returns `Nothing`. The most obvious implementation of this interface uses the list datatype:

```
type Stack a = [a]

push :: [a] -> Stack a -> Stack a
push = (++)

pop  :: Int -> Stack a -> Maybe ([a], Stack a)
pop n xs | length xs >= n = Just (take n xs, drop n xs)
         | otherwise      = Nothing
```

The encoding of the transition rules is straightforward. We include the cases in which the machine halts:

```
eval :: Term -> Stack Term -> (Term, Stack Term)
eval (App t ts) s = eval t (push ts s)                          -- K-APP
eval (Fun xs t) s =
  case pop (length xs) s of
    Just (ts1, ts2) -> eval (subst (zip ts1 xs) t) ts2          -- K-FUN
    Nothing         -> (Fun xs t, s)                            -- halt
eval (Var i)    s = (Var i, s)                                  -- halt
```

4 Deriving a PE machine

Now, we transform the program described in the previous section. We proceed in three stages. First, we choose a different implementation of the stack datatype and the two associated operations. Then we perform CPS transformation, which reifies the recursive calls in `pop` as steps of execution of the abstract machine. Finally, to eliminate the higher-order functions that arise from CPS transformation, we perform inlining.

4.1 Lazy concatenation

Another possible implementation of the `Stack` datatype is a list with lazy concatenation. It is defined as a list of lists, so that the `push` operation can be defined simply as consing the first argument to the front of the structure:

```
type Stack a = [[a]]

push :: [a] -> Stack a -> Stack a
push xs xss = xs : xss

pop :: Int -> Stack a -> Maybe ([a], Stack a)
pop n ys = pop' [] n ys

pop' :: [a] -> Int -> Stack a -> Maybe ([a], Stack a)
pop' acc n ys
  | m == n                = Just (acc, ys)
  | m > n                 = Just (take n acc, drop n acc : ys)
  | m < n && length ys > 0 = pop' (acc ++ head ys) n (tail ys)
  | otherwise             = Nothing
  where m = length acc
```

The definition of `pop` uses an auxiliary function `pop'`, which is tail-recursive. We maintain the invariant that there are no empty inner lists. That is why we include the separate case `m == n` to make sure that `pop'` does not leave an empty list at the front of the structure.

We use these definitions in our encoding of the machine. Note that we do not do that for performance reasons, because we do not intend to ever execute these programs. We use Haskell as a metalanguage to derive one abstract machine from another abstract machine. The equivalence of the programs guarantees

the equivalence of the machines. We still use regular cons-lists to represent tuples, hence standard list functions `length` and `zip`.

4.2 CPS translation

Since `pop'` is tail-recursive, we can see its definition as an encoding of yet another abstract machine. Our intention is to fuse the two machines together. Thus, `eval` cannot treat `pop` as an atomic operation any longer. We need an explicit transfer of control: `eval` calls `pop`, which calls `eval` back instead of just returning a value. We can easily achieve this behaviour with the call-by-value CPS translation of `pop` and `pop'`. We obtain functions `popCPS` and `pop'CPS`, with the property `popCPS n ys k = k (pop n ys)` and `pop'CPS acc n ys k = k (pop' acc n ys)` for any continuation `k`, that is, a function of the type `k :: (Maybe ([a], Stack a) -> o)`. From this, we easily obtain that `evalCPS` given below is extensionally equal to `eval`.

```
popCPS :: Int -> Stack a -> (Maybe ([a], Stack a) -> o) -> o
popCPS n ys k = pop'CPS [] n ys k

pop'CPS :: [a] -> Int -> Stack a -> (Maybe ([a], Stack a) -> o) -> o
pop'CPS acc n ys k
  | m == n                 = k (Just (acc, ys))
  | m > n                  = k (Just (take n acc, drop n acc : ys))
  | m < n && length ys > 0 = pop'CPS (acc ++ head ys) n (tail ys) k
  | otherwise              = k Nothing
  where m = length acc

evalCPS :: Term -> Stack Term -> (Term, Stack Term)
evalCPS (App t ts) s = evalCPS t (push ts s)
evalCPS (Fun xs t) s = popCPS (length xs) s aux
 where
  aux (Just (ts1, ts2)) = evalCPS (subst (zip ts1 xs) t) ts2
  aux Nothing           = (Fun xs t, s)
evalCPS (Var i)    s = (Var i, s)
```

4.3 Inlining

We observe that in the entire program the function `popCPS` is called in one place only, and its continuation `aux` is not modified throughout the recursive calls of `pop'CPS`. So, we know what the continuation inside `popCPS` and `pop'CPS` is, up to its free variables. We can inline the body of the continuation, and supply values for its free variables as arguments of the `popCPS` operation. In detail, we first lambda-lift `aux`. As a result, it becomes a separate function:

```
evalCPS2 :: Term -> Stack Term -> (Term, Stack Term)
evalCPS2 (App t ts) s = evalCPS2 t (push ts s)
evalCPS2 (Fun xs t) s = popCPS (length xs) s (aux2 xs t s)
evalCPS2 (Var i)    s = (Var i, s)
```

```
aux2 :: [Identifier] -> Term -> Stack Term -> Maybe ([Term], Stack Term)
                                                    -> (Term, Stack Term)
aux2 xs t s (Just (ts1, ts2)) = evalCPS2 (subst (zip ts1 xs) t) ts2
aux2 xs t s Nothing           = (Fun xs t, s)
```

Second, we inline the definitions of `popCPS` and `push` in `evalCPS2`. Additionally, we define a function `pop'In` with the property that `pop'CPS vs n s (aux xs t s) = pop'In vs n t xs s`. Applying this equality in the definition of `evalCPS2`, we obtain the following:

```
pop'In :: [Term] -> Int -> Term -> [Identifier] -> Stack Term
                                                  -> (Term, Stack Term)
pop'In acc n t xs ys
  | m == n = evalIn (subst (zip acc xs) t) ys
  | m > n  = evalIn (subst (zip (take n acc) xs) t) (drop n acc : ys)
  | m < n && length ys > 0 = pop'In (acc ++ head ys) n t xs (tail ys)
  | otherwise              = (Fun xs t, [])
  where m = length acc

evalIn :: Term -> Stack Term -> (Term, Stack Term)
evalIn (App t ts) s = evalIn t (ts : s)
evalIn (Fun xs t) s = pop'In [] (length xs) t xs s
evalIn (Var i)    s = (Var i, s)
```

Now, we are almost ready to decode the final machine. The very last observation is that the argument `n` is redundant in `pop'In`: it is always equal to the length of `xs`, since they are equal in the call in `evalIn`, and they are not changed throughout the recursive calls. Therefore, we can eliminate it in the decoding, and compare the length of the accumulator to the length of `xs`. So, the Haskell encoding becomes as follows, in which we have `pop'In2 acc t xs ys = pop'In acc (length xs) t xs ys`.

```
pop'In2 :: [Term] -> Term -> [Identifier] -> Stack Term
                                                -> (Term, Stack Term)
pop'In2 acc t xs ys
  | m == n = evalIn2 (subst (zip acc xs) t) ys
  | m > n  = evalIn2 (subst (zip (take n acc) xs) t) (drop n acc : ys)
  | m < n && length ys > 0 = pop'In2 (acc ++ head ys) t xs (tail ys)
  | otherwise              = (Fun xs t, [])
  where m = length acc
        n = length xs

evalIn2 :: Term -> Stack Term -> (Term, Stack Term)
evalIn2 (App t ts) s = evalIn2 t (ts : s)
evalIn2 (Fun xs t) s = pop'In2 [] t xs s
evalIn2 (Var i)    s = (Var i, s)
```

We are left with two tail- and mutually-recursive functions. Seen as transition rules, they represent two types of configuration of an abstract machine: `evalIn2` represents ‘eval’ configurations with an expression and a stack, while `pop'In2` represents ‘apply’ configurations, which we denote **pap** (short

$$
\begin{array}{lll}
 & (e_0\,\langle e_1 \ldots e_k\rangle, s) & \text{(E-APP)} \\
\Rightarrow & (e_0, \langle e_1 \ldots e_k\rangle{:}s) & \\
 & (\lambda\langle x_1 \ldots x_n\rangle.e, s) & \text{(E-FUN)} \\
\Rightarrow & (\mathbf{pap}(\varepsilon, e, x_1 \ldots x_n), s) & \\
 & (\mathbf{pap}(e_1{:}\cdots{:}e_n{:}\varepsilon, e, x_1 \ldots x_n), s) & \text{(A-EQ)} \\
\Rightarrow & (e[e_1/x_1, \ldots, e_n/x_n], s) & \\
 & (\mathbf{pap}(e_1{:}\cdots{:}e_k{:}\varepsilon, e, x_1 \ldots x_n), s) & \text{(A-GT)} \\
\Rightarrow & (e[e_1/x_1, \ldots, e_n/x_n], \langle e_{n+1} \ldots e_k\rangle{:}s), \text{ where } k > n & \\
 & (\mathbf{pap}(e_1{:}\cdots{:}e_k{:}\varepsilon, e, x_1 \ldots x_n), \langle f_1 \ldots f_m\rangle{:}s) & \text{(A-LT)} \\
\Rightarrow & (\mathbf{pap}(e_1{:}\cdots{:}e_k{:}f_1{:}\cdots{:}f_m{:}\varepsilon, e, x_1 \ldots x_n), s), \text{ where } k < n &
\end{array}
$$

Figure 1: The eval/apply abstract machine

for 'partial application'). The latter consists of a list of accumulated actual arguments (`acc`), the body of the abstraction (`t`), a tuple of formal arguments (`xs`), and the stack (`ys`). The transitions of the machine are shown in Figure 1.

Intuitively, a **pap** configuration stores an abstraction and a small stack of accumulated actual arguments. When there are enough arguments, the application is performed (A-EQ and A-GT). When there are too few arguments, the small stack is extended with the arguments from the top frame of the main stack (A-LT).

5 The applicative fragment of the STG machine

The machine that we arrived at is the applicative fragment of the EA STG machine (as introduced by Marlow and Peyton Jones [12]) in disguise. We only need to slightly rearrange the rules, as shown in Figure 2. First, if e_0 in E-APP is an abstraction, the transition is always followed by E-FUN, which can be followed only by A-LT (since n in E-FUN is greater than 0), and then one of the rules A-EQ, A-GT, or A-LT. We fuse these three possible execution paths into three rules: STG-TCALL, STG-EXACT, and STG-CALLK. We can reuse them to deal with abstractions when there are some arguments on the stack, by constructing an application (STG-PAP2), so that a path E-FUN⇒A-LT⇒(A-EQ, A-GT, or A-LT) becomes STG-PAP2⇒(STG-TCALL, STG-EXACT, or STG-CALLK). We proceed similarly with the **pap** configurations (STG-RETFUN), so that transitions A-EQ, A-GT, and A-LT become STG-RETFUN⇒STG-TCALL, STG-RETFUN⇒STG-EXACT, and STG-RETFUN⇒STG-CALLK respectively. Note that the rearranging of the rules can be done also on the level of Haskell programs by means of inlining and expanding of the definitions.

Marlow and Peyton Jones [12] introduced two versions of the STG machine: push/enter and eval/apply, which differ only in the part that deals with abstractions and applications (they share transitions rules for algebraic data types and call-by-need updates). The rules of the machine shown in Figure 2 correspond to the application-related rules of the EA STG machine: a rule called STG-XXX here is called XXX in

$$
\begin{array}{ll}
& (e_0\,\langle e_1 \ldots e_k\rangle,\, s) \qquad \text{(STG-TCALL)} \\
\Rightarrow & (e_0,\, \langle e_1 \ldots e_k\rangle{:}s), \text{ where } e_0 \text{ is not an abstraction} \\
& ((\lambda\langle x_1 \ldots x_n\rangle.e)\,\langle e_1 \ldots e_n\rangle,\, s) \qquad \text{(STG-EXACT)} \\
\Rightarrow & (e[e_1/x_1, \ldots, e_n/x_n],\, s) \\
& ((\lambda\langle x_1 \ldots x_n\rangle.e)\,\langle e_1 \ldots e_k\rangle,\, s) \qquad \text{(STG-CALLK)} \\
\Rightarrow & (e[e_1/x_1, \ldots, e_n/x_n],\, \langle e_{n+1} \ldots e_k\rangle{:}s), \text{ where } k > n \\
& ((\lambda\langle x_1 \ldots x_n\rangle.e)\,\langle e_1 \ldots e_k\rangle,\, s) \qquad \text{(STG-PAP2)} \\
\Rightarrow & (\mathbf{pap}(e_1{:}\cdots{:}e_k{:}\varepsilon, e, x_1 \ldots x_n),\, s), \text{ where } k < n \\
& (\lambda\langle x_1 \ldots x_n\rangle.e,\, \langle e_1 \ldots e_k\rangle{:}s) \qquad \text{(STG-RETFUN)} \\
\Rightarrow & ((\lambda\langle x_1 \ldots x_n\rangle.e)\,\langle e_1 \ldots e_k\rangle,\, s) \\
& (\mathbf{pap}(e_1{:}\cdots{:}e_n{:}\varepsilon, e, x_1 \ldots x_k),\, \langle f_1 \ldots f_m\rangle{:}s) \qquad \text{(STG-PCALL)} \\
\Rightarrow & ((\lambda\langle x_1 \ldots x_k\rangle.e)\,\langle e_1 \ldots e_n\, f_1 \ldots f_m\rangle,\, s)
\end{array}
$$

Figure 2: The STG-like abstract machine

the formulation by Marlow and Peyton Jones [12]. The difference is that partial applications in the EA STG machine are allocated in the heap. Therefore, there is no need for a separate **pap** configuration in STG, but there is a **pap** type of heap objects.

6 Discussion

Eval/apply machines are also known as 'eval/continue' machines (see Danvy [5] for a discussion), since they have a strong connection with continuations and evaluation contexts [3, 4, 7, 14], which makes them modular and more amenable for reasoning. That is why they are attractive underlying evaluation models for modular and formally verified compilation techniques. The direction of our transformation (from PE to EA) also reflects the pragmatic choice of EA over PE in the Glasgow Haskell Compiler [12]. Our derivation can be easily applied to different known PE machines, such as Leroy's ZINC [11].

Our method is inspired by Danvy *et al.*'s functional correspondence between evaluators and abstract machines [1]. It is important to notice that Danvy *et al.* use CPS translation as a tool to flatten the structure of evaluators, while here, since the `pop'` operation is already in a flat, tail-recursive form, we use it to reify the recursive calls of `pop'` as transitions of the machine, and no new stack of continuations emerges.

The PE STG machine is in reality a hybrid machine, which is apparent in its original, three-stack formulation [9]: the machine is PE in the argument stack, but it is EA in the return and update stacks. The equivalence of two simpler variants of the EA and PE STG machines has been shown by Encina and Peña [8] by deriving both machines from a single natural semantics. Piróg and Biernacki [13] used a technique based on Danvy's functional correspondence to derive the PE STG machine from a big-step operational semantics.

Acknowledgements

We are grateful to Filip Sieczkowski, Dariusz Biernacki, and the anonymous reviewers for their valuable comments.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy & Jan Midtgaard (2003): *A functional correspondence between evaluators and abstract machines*. In: *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, ACM, pp. 8–19, doi:10.1145/888251.888254.

[2] Malgorzata Biernacka & Olivier Danvy (2007): *A concrete framework for environment machines*. *ACM Transactions on Computational Logic* 9(1), pp. 1–30, doi:10.1145/1297658.1297664.

[3] Malgorzata Biernacka & Olivier Danvy (2007): *A syntactic correspondence between context-sensitive calculi and abstract machines*. *Theoretical Computer Science* 375(1-3), pp. 76–108, doi:10.1016/j.tcs.2006.12.028.

[4] Olivier Danvy (2004): *On Evaluation Contexts, Continuations, and the Rest of the Computation*. In Hayo Thielecke, editor: *ACM-SIGPLAN Continuations Workshop (CW'04)*, Technical report CSR-04-1, School of Computer Science, University of Birmingham, United Kingdom, pp. 13–23. Available at http://cs.au.dk/~danvy/DSc/29_danvy_cw-2004.pdf.

[5] Olivier Danvy (2009): *Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part I: Denotational Semantics, Natural Semantics, and Abstract Machines*. In Jens Palsberg, editor: *Semantics and Algebraic Specification, Lecture Notes in Computer Science* 5700, Springer Berlin Heidelberg, pp. 162–185, doi:10.1007/978-3-642-04164-8_9.

[6] Olivier Danvy & Kevin Millikin (2008): *On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion*. *Information Processing Letters* 106(3), pp. 100–109, doi:10.1016/j.ipl.2007.10.010.

[7] Olivier Danvy & Lasse R. Nielsen (2004): *Refocusing in Reduction Semantics*. Technical Report RS-04-26, Basic Research in Computer Science (BRICS), University of Aarhus, Denmark. Available at http://www.brics.dk/RS/04/26/BRICS-RS-04-26.pdf.

[8] Alberto de la Encina & Ricardo Peña-Marí (2009): *From natural semantics to C: A formal derivation of two STG machines*. *Journal of Functional Programming* 19(1), pp. 47–94, doi:10.1017/S0956796808006746.

[9] Simon L. Peyton Jones (1992): *Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine*. *Journal of Functional Programming* 2(2), pp. 127–202, doi:10.1017/S0956796800000319.

[10] Jean-Louis Krivine (2007): *A call-by-name lambda-calculus machine*. *Higher-Order and Symbolic Computation* 20(3), pp. 199–207, doi:10.1007/s10990-007-9018-9.

[11] Xavier Leroy (1990): *The ZINC experiment: an economical implementation of the ML language*. Technical report 117, INRIA. Available at http://gallium.inria.fr/~xleroy/publi/ZINC.pdf.

[12] Simon Marlow & Simon L. Peyton Jones (2006): *Making a fast curry: push/enter vs. eval/apply for higher-order languages*. *Journal of Functional Programming* 16(4-5), pp. 415–449, doi:10.1017/S0956796806005995.

[13] Maciej Piróg & Dariusz Biernacki (2010): *A systematic derivation of the STG machine verified in Coq*. In Jeremy Gibbons, editor: *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, ACM, pp. 25–36, doi:10.1145/1863523.1863528.

[14] Filip Sieczkowski, Malgorzata Biernacka & Dariusz Biernacki (2010): *Automating Derivations of Abstract Machines from Reduction Semantics: A Generic Formalization of Refocusing in Coq*. In Jurriaan Hage &

Marco T. Morazán, editors: *Implementation and Application of Functional Languages—22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers, Lecture Notes in Computer Science* 6647, Springer, pp. 72–88, doi:10.1007/978-3-642-24276-2_5.