Electronic Notes in Theoretical Computer Science 78 (2016)
URL: http://www.elsevier.nl/locate/entcs/volume78.html 8 pages

Implementation of Generalized Refocusing

Klara Zielińska Małgorzata Biernacka

University of Wrocław, Poland

Abstract

na na
bla bla
bla
bla
bla

Keywords: reduction semantics, abstract machines, formal verification, Coq

1 Introduction

Refocusing is a procedure that may be used to generate an efficient evaluator or an abstract machine from a properly stated reduction semantics.

Both evaluators and abstract machines that realize calculi based on rewriting systems — like a lambda calculus — are composed from a part that looks for a redex and a part that rewrites a found redex. Generally, refocusing says how to regenerate an efficient former part from a grammar of evaluation contexts. This, in fact, gives almost an entire evaluator/abstract machine as the latter part is just an application of the rewriting relation, which in most cases may be copied with little or no changes from the reduction semantics.

The concept was first described by Danvy and Nielsen in [1] and revisited in some later papers like [3,5] or [4] (the last one was a starting point for this work).

The issue that we aim here is that the current statements of refocusing does not support reduction semantics where grammars of contexts have multiple context non-terminal symbols. Specifically, the original statement formulated in [1] is ill-formed and interpreting it for such reduction semantics may lead to incorrect evaluators[1], whereas, other authors rather refer to the original, then restate the term. Thus, up to our knowledge, there is no solution for the problem.

Despite the lack of interest, the issue is significant. It is because there is at least one important class of languages, which require more then one non-terminal context symbol to be defined as a reduction semantics. Namely, the fully normalized lambda calculus and languages that are based on it (e.g., the core language of Coq).

X

To show that the machines generated by our refocusing precisely realize reduction semantics given as the input we developed a definition of *following*. It appeared that this definition is an equivalent of the relation called "bisimulations" in Hardin's and Maranget's paper [2], when we reason only about deterministic systems (machines, semantics), still, we decided to go on with our definition as it is more general. Particularly, it applies to any abstract rewriting systems including non-deterministic ones.

Due to lack of space, we assume in this paper some general knowledge and standard notations for contexts, terms and formal grammars.

2 Contexts

Evaluation contexts (also known as reduction contexts) are ubiquitous in the work on operational semantics. As demonstrated by Danvy et al. they can be mechanically obtained by defunctionalizing continuations of an interpreter in the CPS form – one thus obtains the popular "inside-out" representation of contexts [?].

Intuitively, the "inside-out" contexts are ordinary contexts where all edges on the path from the root to the hole – in the tree-representation of a context – have been reversed. If we do not reason about the grammars of contexts, then the choice of representation is just an implementational issue of picking an appropriate data structure to store contexts, and as such it does not affect the mathematical definition of contexts. In case when we reason about the grammars things change, as the grammars are influenced by this choice. Particularly, switching between these approaches closely corresponds to reversing regular grammars on words.

The deterministic "inside-out" grammars are useful for many applications (such as evaluation) where they allow to immediately determine the local neighbourhood of the hole and the kind of its context.

[1] The reader may check the problem by referring to [1], taking a grammar of contexts such that $C ::= []$, $D ::= c\,([])$ and C is the starting symbol, and then running the refocusing procedure on $(c\,(r)\,, [])$ for some redex r. Still, we have no space to put it properly.

$$\begin{aligned} \underline{E} &::= [\,]_E \mid \lambda x.E \mid F\,t \mid a\,E \\ F &::= [\,]_F \mid F\,t \mid a\,E \end{aligned}$$

Figure 1. A grammar of contexts for lambda calculus with the normal-order strategy accepted by our refocusing; where t ranges over lambda terms, a over neutral terms and E is the starting symbol

$$\begin{array}{|l|l} \hline E,\ \ \lambda x.\,[\,] & \\ \hline E,\ [\,]\,(\lambda y.y) & \\ \hline F,\ \ \ x\,[\,] & \leftarrow \text{top} \\ \hline \end{array}$$

Figure 2. Our representation of the context $\lambda x.\,(x\,[\,]_E)\,(\lambda y.y)$ w.r.t. the grammar from Fig. 1 is the pair of the hole kind, E, and the above stack.

This property, however, does not scale to non-deterministic grammars. Even if we determinize such a grammar, we may still need to scan a whole context up to the root in order to determine the kind of context local at the hole. (Consider, e.g., the following grammar: $E ::= E\,[a\,[\,]] \mid b\,[\,]$, $F ::= F\,[a\,[\,]] \mid [\,]$, where E, F are both start symbols.) Because reduction semantics may depend on these kinds, such a representation is not good as a general solution.

The problem can be solved by using ordinary grammars and representing contexts by their derivations – in the form of stacks of elementary contexts (also called context frames), where the head element corresponds to the nearest surrounding of the hole, as in "inside-out" contexts. We find such a representation optimal for our purposes: it is natural, it withstands determinization, it requires only a small space overhead, which generally does not change asymptotic size of contexts (if a grammar has not too many context non-terminal symbols), the eventual overhead may be eliminated by an optimization of an abstract machine utilizing such context representation, and the time complexity of operations on contexts is proper for the purpose of evaluation.

In general, when we give a reduction semantics, we may introduce multiple kinds of reductions – to be used depending on the kind of the current evaluation context. This naturally leads to grammars of contexts with multiple starting symbols, where each of these symbols generate one kind of contexts. However, the same behavior may also be achieved in grammars with a single starting symbol by introducing multiple kinds of holes. So, instead of saying that "we can use a reduction α in a context generated by a grammar $\mathcal{A}$", we say that "we can use α in a context that has the hole $[\,]_\alpha$". It can be proved that both these approaches are interchangeable. (Note that in Fig. 1 we do not need more then one kind of reduction, but the holes are labeled for the purpose of demonstration and to correspond to our refocusing method, which we describe later.)

In our refocusing method we make use of the two assumptions stated in this section, i. e., we use ordinary grammars of context with (possibly) multiple kinds of holes and we represent contexts as stacks of context frames.

3 Refocusing

Our refocusing implementation translates a subset of reduction semantics to deterministic abstract machines. The subset is determined by additional properties that need to be satisfied. Probably, it is the set of all deterministic reduction semantics, but we have not checked it, yet.

3.1 Preliminaries

An *abstract rewriting system* is a pair of a set $\mathcal{T}$ of arbitrary entities and a binary relation $\rightarrow$ over them.

We assume that k, k', l range over non-terminal context symbols in grammars of contexts. The set of these symbols in a grammar $\mathcal{C}$ is denoted by $\mathcal{C}_{\mathcal{K}}$ and the start non-terminal by $\mathcal{C}_{init}$.

Without loss of generality we assume that all *grammars of contexts are normalized*, that is, for every production $k \rightarrow P$, we have $P = []_k$ or P does not contain a hole.

We write $\mathcal{C}_k$ for the grammar $\mathcal{C}$ where all productions containing holes, except $k \rightarrow []_k$, have been removed; if $C \in \mathcal{C}_k$, then we say, C has a hole of the kind k.

3.2 Reduction semantics

A *reduction semantics* represented by our implementation is an abstract rewriting system with an additional grammar of contexts, $\mathcal{C}$, and a set $\{\rightharpoonup_k | k \in \mathcal{C}_{\mathcal{K}}\}$ of partial functions (cf. * in Sec. 5) on $\mathcal{T}$, where $\mathcal{T}$ is a set of terms and $t_1 \rightarrow t_2 \iff t_1 = C\,[t_3] \wedge t_2 = C\,[t_4] \wedge t_3 \rightharpoonup_k t_4$ for some k, t_3, t_4 and $C \in \mathcal{L}\,(\mathcal{C}_k)$. In other words, the harpoon-arrows directly state the rewriting rules — by some authors they are called *contraction* — and the full arrow states the *reduction* relation. Beside of this rather standard definition our reduction semantics specifies also a *set of values*, $\mathcal{V}_k$, per each $k \in \mathcal{C}_{\mathcal{K}}$. Generally, values are irreducible terms that can occur in the holes with corresponding labels. Irreducible terms that are not values represent *error terms* — e.g., `2/0`.

In order to apply our refocusing to such semantics the semantics must be deterministic and the reduction must be computable. To satisfies these properties one needs to provide: $\rightharpoonup_k$ as Coq functions, a *computable DFS evaluation strategy* and proofs that guarantee that the strategy finds all redexes and always comes to an end. The depth-first search means here that a strategy turns back only if it has checked all possible subterms and it has found that the current term is a value. Strategies that satisfy the mentioned properties will be called *exhaustive*.

$$
\begin{array}{rlrl}
[t_1\, t_2]_k & \Downarrow [t_1]_F\ t_2 & [[a]_F\ t]_k & \Uparrow a\ [t]_E \\
[x]_k & \Downarrow \mathit{value} & [a\ [v]_E]_k & \Uparrow \mathit{value} \\
[\lambda x.t]_E & \Downarrow \lambda x.\, [t]_E & [\lambda x.\, [v]_E]_k & \Uparrow \mathit{value} \\
[\lambda x.t]_F & \Downarrow \mathit{value} & &
\end{array}
$$

Figure 3. Strategy for the grammar in Fig. 1

In Fig. 3 we show a simplified example of two partial functions that determinate a strategy for the grammar from Fig. 1, which is appropriate for our refocusing. The indexes represent the last context non-terminal symbol of the current context derivation.

3.3 Abstract machines

Our *abstract machines* are abstract rewriting systems, where $\mathcal{T}$ is a set of configurations, $\rightarrow$ is a computable (possibly) non-deterministic function (cf. Sec. 4), and initial and final states are explicitly specified.

We require $\rightarrow$ to be a computable non-deterministic function to ensure that machines are anyhow realistic, that is, they can be build in the real world.

Each machine that is a result of our refocusing has the same following form. Each configuration is a triple $\langle t, \mathtt{C}, k\rangle$ or $\langle \mathtt{C}, k, v\rangle$, where t is a term, k is the last context non-terminal symbol in the derivation of a current evaluation context, $\mathtt{C}$ is the stack representing the rest of the derivation (cf. Fig. 2) and v is a value from $\mathcal{V}_k$. The initial configurations are $\langle t, \varepsilon, \mathcal{C}_{init}\rangle$; the final configurations are $\langle \varepsilon, \mathcal{C}_{init}, v\rangle$; and the transitions are: $\langle t, \mathtt{C}, k\rangle \rightarrow \langle \mathtt{C}, k, t\rangle$ if $[t]_k \Downarrow \mathit{value}$, $\langle t, \mathtt{C}, k\rangle \rightarrow \langle t', \mathtt{C}, k\rangle$ if $[t]_k \Downarrow \mathit{undef.} \wedge t \rightharpoonup_k t'$, $\langle t, \mathtt{C}, k\rangle \rightarrow \langle t', \mathtt{C}\,(k, C_0)\,, k'\rangle$ if $[t]_k \Downarrow C_0\,[t']_{k'}$, $\langle \mathtt{C}\,(l, C_0)\,, k, v\rangle \rightarrow \langle \mathtt{C}, l, C_0\,[v]\rangle$ if $[C_0\,[v]_k]_l \Uparrow \mathit{value}$, $\langle \mathtt{C}\,(l, C_0)\,, k, v\rangle \rightarrow \langle t, l, \mathtt{C}\rangle$ if $[C_0\,[v]_k]_l \Uparrow \mathit{undef.} \wedge C_0\,[v] \rightharpoonup_l t$, $\langle \mathtt{C}\,(l, C_0)\,, k, v\rangle \rightarrow \langle t, \mathtt{C}\,(l, C_1)\,, k'\rangle$ if $[C_0\,[v]_k]_l \Uparrow C_1\,[t]_{k'}$ (we are sorry for the compressed presentation). In fact, k in $\langle \mathtt{C}, k, v\rangle$ configurations has no computational meaning in our implementation, as the corresponding argument of $\Uparrow$ is always determined by others, and so it can be removed from configurations.

3.4 An example of a generated machine

Let us generate a machine with our refocusing implementation for lambda calculus with normal-order strategy. For simplicity, let us assume that we can perform a substitution in a single contraction step.

The input reductions semantics is defined as follows: $\mathcal{T}$ is the set of lambda terms, $\mathcal{C}$ is the grammar of contexts defined in Fig. 1, contractions are defined as $(\lambda x.t_1)\, t_2 \rightharpoonup_k t_1\, [x/t_2]$ for $k = E, F$, the values $\mathcal{V}_E$ are β-normal forms and

$$\begin{array}{rlrl}
\langle x, \mathtt{C}, k\rangle & \to \langle \mathtt{C}, k, x\rangle & \langle \mathtt{C}\,(l, []\ t_2)\,, F, \lambda x.t_1\rangle & \to \langle t_1\,[x/t_2]\,, \mathtt{C}, l\rangle \\
\langle t_1\, t_2, \mathtt{C}, k\rangle & \to \langle t_1, \mathtt{C}\,(k, []\ t_2)\,, F\rangle & \langle \mathtt{C}\,(l, []\ t_2)\,, F, a\rangle & \to \langle t_2, \mathtt{C}\,(l, a\ [])\,, E\rangle \\
\langle \lambda x.t, \mathtt{C}, E\rangle & \to \langle t, \mathtt{C}\,(E, \lambda x.\, [])\,, E\rangle & \langle \mathtt{C}\,(l, a\ [])\,, E, v\rangle & \to \langle \mathtt{C}, l, a\ v\rangle \\
\langle \lambda x.t, \mathtt{C}, F\rangle & \to \langle \mathtt{C}, F, \lambda x.t\rangle & \langle \mathtt{C}\,(E, \lambda x.\, [])\,, E, v\rangle & \to \langle \mathtt{C}, E, \lambda x.v\rangle
\end{array}$$

Figure 4. Transitions of the machine generated for lambda calculus with normal-order

$\mathcal{V}_F$ are neutral terms plus all lambda abstractions. The strategy is defined by the functions in Fig. 3. Besides, we need to provide few additional definitions and prove around 20 small lemmas that guarantee that our representation is well-formed and that the strategy is exhaustive. Despite the number of the lemmas they are, in deed, very small and mostly very simple to prove — i.e., only one proof even requires induction (it has two occurrences in the code, but they are exactly the same).

The result machine is defined as described in the previous section, specifically, with $\mathtt{C} ::= \varepsilon \mid (E, \lambda x.\, [])\,\mathtt{C} \mid (E, a\ \ [])\,\mathtt{C} \mid (E, []\ \ t)\,\mathtt{D}$, $\mathtt{D} ::= \varepsilon \mid (F, a\ \ [])\,\mathtt{C} \mid (F, []\ \ t)\,\mathtt{D}$ and the transition relation shown in Fig. 4. Note that the behavior of the machine, in deed, depends on non-terminals introduced in configurations by two bottom transitions on the left and the top transition on the right. This should give a taste that in general case we do need these non-terminals.

Still, in this case we can easily optimize the contexts, as we have a nice grammar and every small (elementary) context in a derivation determines the next non-terminal symbol. Hopefully, this somehow verifies one of our claims from Section **??**.

3.5 Following

For every machine generated by our refocusing our implementation provides a proof that the input reduction semantics is followed by this machine.

Definition 3.1 *An abstract rewriting system $\langle \mathcal{T}_1, \to\rangle$ **follows** another system $\langle \mathcal{T}_2, \Rightarrow\rangle$ if there exists a surjection $[\![]\!] : \mathcal{T}_1 \to \mathcal{T}_2$ such that*

(i) *if $t_1 \to t_2$, then $[\![t_1]\!] = [\![t_2]\!] \vee t_1 \Rightarrow t_2$,*

(ii) *if $s_1 \Rightarrow s_2$, then for each t_0 such that $[\![t_0]\!] = s_1$ there exists a sequence $t_1 \to \cdots \to t_{n+1}$, where $[\![t_0]\!] = \cdots = [\![t_n]\!]$ and $[\![t_{n+1}]\!] = s_2$, and*

(iii) *there is no infinite sequences $t_0 \to t_1 \to \ldots$ (silent loops), where $[\![t_n]\!] \not\Rightarrow [\![t_{n+1}]\!]$ for all n.*

As we mentioned, our following is a more general version of the, so-called, "bisimulation" from Hardin's and Maranget's paper [2]. Precisely, it is very easy to prove that they are equivalent for deterministic system; that they are not equivalent for non-deterministic ones; and to show that the "bisimulation" acts badly for non-deterministic systems.

4 Implementation

Our work is implemented as a library in Coq 8.5. We extensively use dependent types to represent context derivations — what, however, does not introduce much complication. We also use dependent types to describe paths in rewriting systems, and subsets — these are harder to work with, but the core of the library can be used without theses features.

The library contains some type classes, but we are going to reconsider this feature, as we have encountered a serious problem related to them. Generally, we have discovered that a user may easily and unwittingly break the algorithm that search for class instances, by adding an instance that generates an infinite breach in the search tree (cf. the source-code file `Lib/Subset.v`). We deeply believe that the search algorithm should use BFS, not DFS like it does now, to prevent such situations, and we generally discourage usage of type classes until introducing a solution for this issue.

A little, unusual idea in our implementation is that we introduce computable (possibly) non-deterministic functions. We represent these by Coq functions that take an extra argument, called entropy. Intuitively, it represents a random prefix of the tape of a non-deterministic Turing machine.

One another thing that we found slightly interesting is that we was able to exploit computable predicates — predicates that satisfies: `forall x, {P x} + {~P x}` — to define types that properly represent subsets of other types (cf. the source-code file `Lib/Subset.v`). By “properly” we mean that they contain at most one representation of each value of the original type.

5 Some conclusions and observations

Our implementation is powerful enough to generate many efficient abstract machines, like CEK or Krivine machine — not a new result — and machines for full β-normalization — a new result. However, it does not support reduction semantics with advanced control flow, like call/cc, and adding it is a future work.

The decision of not-using “inside-out” grammars has an impact on the size of states of generated machines — which stands in favor of the “inside-out” grammars. Nevertheless, the overhead may be eliminated by an easy optimization of the result machines, so we do not consider this a problem.

(*) The contractions, $\rightharpoonup_k$, in our reduction semantics should be relations, not partial functions. The current state is inherited form an earlier version.

X

References

[1] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics, 2004.

[2] Thérèse Hardin and Luc Maranget. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.*, 8(2):131–176, 1998.

[3] J. Johannsen. *On Computational Small Steps and Big Steps: Refocusing for Outermost Reduction*. Department Office Computer Science, Aarhus University, 2015.

[4] Filip Sieczkowski, Malgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: - A generic formalization of refocusing in coq. In *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*, pages 72–88, 2010.

[5] Wouter Swierstra. From mathematics to abstract machine: A formal derivation of an executable krivine machine. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP 2012, Tallinn, Estonia, 25 March 2012.*, pages 163–177, 2012.