Formally verified derivation of an executable and terminating CEK machine from call-by-value $\lambda\hat{p}$-calculus

WOJCIECH ROZOWSKI*, ECS, University of Southampton, UK

1 BACKGROUND AND MOTIVATION

Abstract machines describe the semantics of programming languages as first-order transition systems. They offer a realistic, yet abstract model of executing a program. Well known examples include STG [10], CEK [8] or Krivine machine [11]. Biernacka & Danvy [3] have shown that abstract machines are not invented, but rather can be derived mathematically from calculi with explicit substitutions. The authors introduced a formalism called $\lambda\hat{p}$, and demonstrate how several abstract machines can be derived from well understood program transformations [4].

It is worth considering formalising such derivation in dependently-typed programming languages, such as Coq or Agda [6] to obtain machines formally verified to be correct. Sieczkowski [13] and Biernacka [2] provided a Coq formalisation of Biernacka & Danvy's framework and showed the correctness of the transforms leading to abstract machines. Although proven to be correct, those formalisations are not executable, as their machines are inductively defined relations between the states. Swierstra [14] provided an Agda formalisation of $\lambda\hat{p}$ under call-by-name and derived an executable Krivine machine, following Biernacka & Danvy's approach. An interesting contribution of this study is proof of the termination of the executable Krivine machine, which relies on the strong normalisation property for simply typed call-by-name $\lambda\hat{p}$ proved using the Bove-Capretta method [5] and a Tait-style logical relation [15].

This paper sums up our efforts on extending Swierstra's research to handle other abstract machines, beyond the Krivine machine.

2 CONTRIBUTIONS

We successfully extend the Swierstra formalisation of $\lambda\hat{p}$ to a call-by-value case, including the properties of head reduction. We provide proof of a Strong Normalisation property for simply typed call-by-value $\lambda\hat{p}$-calculus using a Tait-style logical relation in a similar way to Altenkirch & Chapman [1]. From such terminating evaluator, we obtain an abstract machine corresponding to Felleisen's CEK [8]. We show the correctness and termination of the obtained machine by showing trace equivalence with the call-by-value $\lambda\hat{p}$ evaluator. All those theorems and properties have been fully implemented and checked by Agda. The corresponding code is freely available online. [1]. In this paper we try to omit Agda code and instead provide a high-level overview of our formalisation, sketching the main proofs and presenting the main data types in the form of sequent-style rules.

3 SINGLE STEP EVALUATOR FOR CALL-BY-VALUE $\lambda\hat{p}$

The formalisation of $\lambda\hat{p}$ is defined on top of simply-typed λ-calculus with intrinsic typing and De Bruijn indices in a fashion similar to Wadler et al [16]. The terms of $\lambda\hat{p}$, called closed terms, are either closures (that is λ-terms with corresponding substitution environment for each of the free variables) or an application of two closed terms. The only values in that language are closures of λ-abstractions. We treat the semantics as reduction semantics in the style of Felleisen [7]. Figure 1 shows the definitions of terms of call-by-value $\lambda\hat{p}$, its redexes and evaluation contexts. Similarly to

*Research advisor: Julian Rathke; ACM student member number: 3641070; Category: undergraduate

[1] https://github.com/wkrozowski/CEK-from-lambda-p-hat

Author's address: Wojciech Rozowski, ECS, University of Southampton, Southampton, UK, wojciech.rozowski@gmail.com.

Swierstra [14], we parametrise the evaluation contexts by their types, where the first type is the current type of the hole and the second is the original type of expression under evaluation. To define

$$\boxed{\texttt{Closed u}}\ \text{Closed term of type u}$$

$$\frac{\texttt{t} : \Gamma \vdash u \quad \texttt{e} : \texttt{Env}\ \Gamma}{\texttt{Closure t e} : \texttt{Closed u}} \quad (1)$$

$$\frac{\texttt{f} : \texttt{Closed (u} \Rightarrow \texttt{v)} \quad \texttt{x} : \texttt{Closed u}}{\texttt{Clapp f x} : \texttt{Closed v}} \quad (2)$$

$$\boxed{\texttt{Env}\ \Gamma}\ \text{Substitution environment for type context } \Gamma$$

$$\frac{\texttt{c} : \texttt{Closed u} \quad \texttt{e} : \texttt{Env}\ \Gamma}{c \cdot e : \texttt{Env}\ (u :: \Gamma)} \quad (3)$$

$$\frac{}{\texttt{Nil} : \texttt{Env []}} \quad (4)$$

$$\boxed{\texttt{Redex}\ \sigma}\ \text{Reducible expression of type } \sigma$$

$$\frac{\texttt{i} : \Gamma \ni \sigma \quad \texttt{e} : \texttt{Env}\ \Gamma}{\texttt{Lookup i env} : \texttt{Redex}\ \sigma} \quad (5)$$

$$\frac{\texttt{f} : \Gamma \vdash (\sigma \Rightarrow \tau) \quad \texttt{x} : \Gamma \vdash \sigma \quad \texttt{e} : \texttt{Env}\ \Gamma}{\texttt{App f x e} : \texttt{Redex}\ \tau} \quad (6)$$

$$\frac{\texttt{f} : (\sigma :: \Gamma) \vdash \tau \quad \texttt{e} : \texttt{Env}\ \Gamma \quad \texttt{v} : \texttt{Value}\ \sigma}{\texttt{Beta f e v} : \texttt{Redex}\ \tau} \quad (7)$$

$$\boxed{\texttt{EvalContext u v}}\ \text{Evaluation context for types u and v}$$

$$\frac{}{\texttt{MT} : \texttt{EvalContext u u}} \quad (8)$$

$$\frac{\texttt{c} : \texttt{Closed u} \quad \texttt{ctx} : \texttt{EvalContext v w}}{\texttt{ARG c ctx} : \texttt{EvalContext (u}\Rightarrow\texttt{v) w}} \quad (9)$$

$$\frac{\texttt{v} : \texttt{Value (a}\Rightarrow\texttt{b)} \quad \texttt{ctx} : \texttt{EvalContext b c}}{\texttt{FN v ctx} : \texttt{EvalContext a c}} \quad (10)$$

Fig. 1. Closed terms and substitution environments

single-step reduction, we follow Biernacka & Danvy and define three functions: `decompose` (which takes a term and returns a redex in corresponding evaluation context), `contract` (which performs single-step reduction of a redex), and `plug` which takes a closed term (redex upon reduction), and recreates the term basing on accumulated context. The `decompose` function is defined in terms of an auxiliary function, `decompose'`, a state transition function operating on configurations that are pairs of closed terms and corresponding evaluation contexts, by calling it with an empty frame. Composing `decompose`, `contract` and `plug` yields `headReduce` - a small-step reduction function for call-by-value $\lambda\hat{p}$.

4 REFOCUSING

For a single-step evaluator defined in the way shown by Biernacka & Danvy, there is a simplifying observation to be made. When performing multiple head reduction steps, reduced redexes are plugged to recreate the original expression, only to be decomposed a step later. Danvy [7] observed that `decompose` composed with `plug` is equivalent to `decompose'`, so instead of rebuilding the term, we can continue traversal from current configuration. Such a transform is referred to in literature as a refocusing transform [3].

Theorem 4.1 (Refocusing theorem). *For any* `u` *and* `v` *let* `c` *denote closed term of type* `u` *and let* `ctx` *denote an* `EvalContext` *parametrised by types* `u` *and* `v`. *Then,* `decompose (plug ctx c) ≡ decompose' ctx c`

The above property trivially follows from the definition. After introducing `refocus` function equal to `decompose'`, we can simplify the three-step evaluator, to two-step process consisting of composition of `refocus` and `contract`. An evaluator in this form is often referred to as pre-abstract machine [3].

5 STRONG NORMALISATION PROPERTY FOR CALL-BY-VALUE $\lambda\hat{p}$

Recursive calling of the small-step evaluation function is not structurally recursive, therefore it does not pass the termination checker in Agda. To prove its termination we use the Bove-Capretta method [5], which relies on providing an execution trace, which carries no computational value, but assists the termination checker. We rely on Swierstra's [14] Tait-style logical relation for reducible closed terms and reducible environments. To populate the relation we prove the following three properties:

Lemma 5.1 (Preservation lemma). *If $e \rightsquigarrow e'$, then e is reducible $\iff$ e' is reducible*

Lemma 5.2 (Reducibility lemma). *Closure of a well-typed term with a reducible environment is always reducible*

Lemma 5.3 (Right hand side reducibility lemma). *If x is reducible, then for any* `body` *and* `env` `headReduce (Clapp (Closure (λ body) env) x)` *is also reducible*

Lemma 5.2 allows us to obtain a Bove-Capretta trace for any well-typed term with no free variables, as an empty environment is trivially reducible. The proof of this property relies on the observation that we can easily show the reducibility of current closure after one-step reduction, and then appeal to Lemma 5.1 to show the desired property. In a λ-abstraction case, we need to additionally show that passing a reducible argument to it, yields a reducible result [9]. In the cases when performing β-reduction it is quite easy to show with Lemma 5.2, but in call-by-value, such a situation would only happen if the right-hand side of the application is a value. To cope with that, we introduce Lemma 5.3, which shows that with a finite amount of reductions right-hand side eventually becomes a value, leading to the β-reduction case.

6 CEK MACHINE

We can further simplify a pre-abstract machine, which we have obtained through refocusing transform. If we get rid of the `contract` function, and simply call `refocus` with already contracted redex, we obtain a state transition function of an abstract machine. Also, upon making an observation that in all of the steps we always store closures, rather than applications of closed terms, we can refine the obtained transition rules to operate on triples of λ-terms, environments and evaluation contexts. It can be noticed, that such presentation of the rules corresponds to Felleisen's CEK machine [8], however operating on De Bruijn variables, and having optimized closure-making step. We introduce a new Bove-Capretta data type, where each of the constructors corresponds to one of the possible state transitions. We can easily show termination and correctness, by obtaining such a trace by rewriting the trace obtained through Lemma 5.2.

7 CONCLUSIONS AND FUTURE WORK

We successfully extended Swierstra's approach to a broader, call-by-value context. It would be beneficial to take it even further and consider formalising executable machines for context-sensitive languages with first-class control operators, such as Parigiot's $\lambda\mu$-calculus [12]. Another interesting area could include non-normalising languages, and attempts to obtain executable machines for real-life functional programming languages.

ACKNOWLEDGMENTS

I would like to thank Julian Rathke for supervising this project and to Wouter Swierstra, Thorsten Altenkirch and Filip Sieczkowski for the valuable discussions and feedback.

REFERENCES

[1] Thorsten Altenkirch and James Chapman. 2009. Big-step normalisation. *Journal of Functional Programming* 19, 3-4 (2009), 311–333. https://doi.org/10.1017/S0956796809007278

[2] Biernacka. 2016. Generalized Refocusing: a Formalization in Coq.

[3] Małgorzata Biernacka and Olivier Danvy. 2007. A Concrete Framework for Environment Machines. *ACM Trans. Comput. Logic* 9, 1 (Dec. 2007), 6–es. https://doi.org/10.1145/1297658.1297664

[4] Malgorzata Biernacka and Olivier Danvy. 2007. A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines. *Theor. Comput. Sci.* 375 (05 2007), 76–108. https://doi.org/10.1016/j.tcs.2006.12.028

[5] Ana Bove. 2003. General Recursion in Type Theory. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–58.

[6] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–78.

[7] Olivier Danvy and Lasse R. Nielsen. 2004. Refocusing in Reduction Semantics.

[8] M. Felleisen and D. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts*.

[9] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press, USA.

[10] Peyton Jones, Simon L, and Simon Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* 2 (July 1992), 127–202.

[11] Jean-Louis Krivine. 2007. A Call-by-Name Lambda-Calculus Machine. *Higher Order Symbol. Comput.* 20, 3 (Sept. 2007), 199–207. https://doi.org/10.1007/s10990-007-9018-9

[12] Michel Parigot. 1992. $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning*, Andrei Voronkov (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–201.

[13] Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. 2011. Automating Derivations of Abstract Machines from Reduction Semantics:. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–88.

[14] Wouter Swierstra. 2012. From Mathematics to Abstract Machine: A formal derivation of an executable Krivine machine. In Proceedings Fourth Workshop on *Mathematically Structured Functional Programming*, Tallinn, Estonia, 25 March 2012 *(Electronic Proceedings in Theoretical Computer Science, Vol. 76)*, James Chapman and Paul Blain Levy (Eds.). Open Publishing Association, 163–177. https://doi.org/10.4204/EPTCS.76.10

[15] W. W. Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

[16] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda*. http://plfa.inf.ed.ac.uk/20.07/