Proceedings of the APPSEM-II Workshop on the Krivine and ZINC Abstract Machines (KAZAM)

Olivier Danvy and Hayo Thielecke

May 17, 2005 — University of Birmingham

Acknowledgements

We are grateful to Achim Jung and Mike Mislove for associating KAZAM with MFPS XXI, and to Jean-Louis Krivine and Xavier Leroy, to accept our invitation to speak at the workshop.

A concrete framework for environment machines

Małgorzata Biernacka*
BRICS†
Department of Computer Science
University of Aarhus‡

Abstract

We materialize the common belief that calculi with explicit substitutions provide an intermediate step between an abstract specification of substitution in the λ-calculus and its actual implementations. To this end, we show a systematic derivation leading from a slight extension of Curien's calculus of closures, capable of expressing one-step reduction strategies, to the environment-based Krivine's abstract machine for call-by-name evaluation in the λ-calculus. The derivation consists of two phases: the first one employs Danvy and Nielsen's refocusing method to construct an abstract machine for the calculus of closures; the second performs an unfolding of closures to make the environment part explicit in the resulting abstract machine.

1 Introduction

Krivine's machine is probably the simplest example of an abstract machine implementing an evaluation function of the λ-calculus [10]. As many other abstract machines for languages with binding constructs, it is environment-based, i.e., roughly, one component of a machine configuration stores terms to be substituted for free variables during the process of evaluation. The transitions of the machine provide a precise way of handling substitution, contrasting with the usual abstract specification of substitution in the λ-calculus, where one expresses the β-rule using a meta-level notion of (implicit) substitution.

Actual implementations, however, do not use implicit substitutions. Instead, they keep an explicit representation of what should have been substituted and leave the term untouched.

*Joint work with Olivier Danvy [2]

†Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

‡IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: {`mbiernac,danvy`}`@brics.dk`

To bridge the two worlds of implicit and explicit substitutions, various calculi of explicit substitutions have been proposed [1, 4, 11]. The idea of explicit substitutions is to incorporate the notion of substitution into the syntax of the language and to specify suitable rewrite rules that realize the substitution.

Our goal in this work is to present a completely systematic way of deriving an abstract machine with environment from a specification of one-step reduction strategy in a calculus of closures, by employing Danvy and Nielsen's refocusing technique [6] followed by an unfolding of the data type of closures.

We first turn our attention to Curien's original calculus of closures $\lambda\rho$ [3], mediating between the standard λ-calculus and its implementations via abstract machines. We observe that one-step reductions cannot be expressed in this calculus, and therefore we propose a minimal extension to $\lambda\rho$, capable of expressing such computations ($\lambda\widehat{\rho}$-calculus). We then show a derivation of Krivine's machine [10] from the specification of the call-by-name one-step strategy in $\lambda\widehat{\rho}$.

2 Curien's calculus of closures

The language of $\lambda\rho$ has three syntactic categories: terms, closures and substitutions. Terms are defined as in the λ-calculus, using de Bruijn indices for variables:

$$\begin{array}{ll} \text{(terms)} & t ::= i \mid t\ t \mid \lambda t \\ \text{(closures)} & c ::= t[s] \\ \text{(substitutions)} & s ::= \bullet \mid c \cdot s \end{array}$$

A closure is a pair consisting of a term and a substitution, which itself is a finite list of closures to be substituted for free variables in the term.

The weak reduction relation $\xrightarrow{\rho}$ is specified by the following rules:

$$\text{(Eval)} \qquad \frac{t_0[s] \xrightarrow{\rho}{}^* (\lambda t_0')[s']}{(t_0\ t_1)[s] \xrightarrow{\rho} t_0'[(t_1[s]) \cdot s']}$$

$$\text{(Var)} \qquad i[c_1 \cdots c_m] \xrightarrow{\rho} c_n \quad \text{if } 1 \le n \le m$$

$$\text{(Sub)} \qquad \frac{c_1 \xrightarrow{\rho}{}^* c_1' \quad \ldots \quad c_m \xrightarrow{\rho}{}^* c_m'}{t[c_1 \cdots c_m] \xrightarrow{\rho} t[c_1' \cdots c_m']}$$

where $\xrightarrow{\rho}{}^*$ is the reflexive, transitive closure of $\xrightarrow{\rho}$. All reductions are performed on closures, and not on individual terms. However minimalist, this calculus is powerful enough to compute weak head normal forms.

The rules of the calculus are nondeterministic and can be restricted in order to define a specific deterministic evaluation strategy. For instance, the call-by-name strategy is obtained by removing the rule (Sub).

3 A minimal extension to Curien's calculus of closures

It turns out that the the $\lambda\rho$-calculus is not expressive enough for specifying one-step computations. This is due to the fact that the specification of one-step reduction requires a way of "composing" intermediate results of computation – here, closures – to form a new closure, which can be reduced further. In $\lambda\rho$, there is no such possibility. A simple solution to this problem is to extend the syntax of closures with a construct denoting closure composition. We denote it simply by juxtaposition. The modified grammar of closures is then as follows:

$$\text{(closures)} \quad c ::= t[s] \mid c\, c$$

With the extended syntax we are now in a position to define the one-step reduction relation on the language of closures.

$$(\beta_{\mathrm{c}}) \qquad ((\lambda t)[s])\, c \xrightarrow{\widehat{\rho}} t[c \cdot s]$$

$$\text{(Var)} \qquad i[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}} c_n \quad \text{if } 1 \leq n \leq m$$

$$\text{(App)} \qquad (t_0\, t_1)[s] \xrightarrow{\widehat{\rho}} (t_0[s])\, (t_1[s])$$

$$\text{(Sub)} \qquad \frac{c_1 \xrightarrow{\widehat{\rho}} c_1' \ \ldots\ c_m \xrightarrow{\widehat{\rho}} c_m'}{t[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}} t[c_1' \cdots c_m']}$$

$$(\nu) \qquad \frac{c_1 \xrightarrow{\widehat{\rho}} c_1'}{c_1\, c_2 \xrightarrow{\widehat{\rho}} c_1'\, c_2}$$

$$(\mu) \qquad \frac{c_2 \xrightarrow{\widehat{\rho}} c_2'}{c_1\, c_2 \xrightarrow{\widehat{\rho}} c_1\, c_2'}$$

The call-by-name evaluation strategy can be obtained from the full system of $\lambda\widehat{\rho}$ by restricting it to the following rules:

$$(\beta_{\mathrm{c}}) \qquad ((\lambda t)[s])\, c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} t[c \cdot s]$$

$$\text{(Var)} \qquad i[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c_n, \quad \text{if } 1 \leq i \leq m$$

$$\text{(App)} \qquad (t_0\, t_1)[s] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t_0[s])\, (t_1[s])$$

$$(\nu) \qquad \frac{c_1 \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c_1'}{c_1\, c_2 \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c_1'\, c_2}$$

4 From call-by-name reduction to environment machine

The above specification of the call-by-name strategy as a deterministic relation can be rewritten as a function in the obvious way:

$$\begin{aligned}
\mathsf{reduce} &: \mathsf{Closure} \to \mathsf{Closure} \\
\mathsf{reduce}\,(((\lambda t)[s])\;c) &= t[c \cdot s] \\
\mathsf{reduce}\,(n[c_1 \cdots c_m]) &= c_n \\
\mathsf{reduce}\,((t_0\;t_1)[s]) &= (t_0[s])\;(t_1[s]) \\
\mathsf{reduce}\,(c_1\;c_2) &= (\mathsf{reduce}\,c_1)\;c_2
\end{aligned}$$

As proposed by Felleisen [7, 8, 9], this reduction function can be equivalently expressed using evaluation contexts. As observed by Danvy and Nielsen [5, 6], these evaluation contexts can be mechanically obtained by (1) transforming reduce into continuation-passing style, and (2) defunctionalizing the resulting continuations. The resulting grammar of evaluation contexts, plug function, and CPS-transformed reduction function read as follows:

$$\text{(evaluation contexts)} \qquad C ::= [\,] \mid \mathsf{ARG}(C,\, c)$$

$$\begin{aligned}
\mathsf{plug} &: \mathsf{Context} \times \mathsf{Closure} \to \mathsf{Closure} \\
\mathsf{plug}\,([\,], c) &= c \\
\mathsf{plug}\,(\mathsf{ARG}(C,\, c'), c) &= (\mathsf{plug}\,(C, c))\;c'
\end{aligned}$$

$$\begin{aligned}
\mathsf{reduce}' &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Closure} \\
\mathsf{reduce}'((\lambda t)[s],\, \mathsf{ARG}(C,\, c)) &= \mathsf{plug}\,(C, t[c \cdot s]) \\
\mathsf{reduce}'(n[c_1 \cdots c_m],\, C) &= \mathsf{plug}\,(C, c_n) \\
\mathsf{reduce}'((t_0\;t_1)[s],\, C) &= \mathsf{plug}\,(C, t_0[s]\;t_1[s]) \\
\mathsf{reduce}'(c_1\;c_2,\, C) &= \mathsf{reduce}'(c_1,\, \mathsf{ARG}(C,\, c_2))
\end{aligned}$$

As traditional, evaluation is defined as the reflexive and transitive closure of one-step reduction. The following function evaluate therefore computes the value of a term:

$$\begin{aligned}
\mathsf{evaluate} &: \mathsf{Value} + \mathsf{Computation} \to \mathsf{Value} \\
\mathsf{evaluate}\;v &= v \\
\mathsf{evaluate}\;c &= \mathsf{evaluate}\,(\mathsf{reduce}\,c), \\
&\qquad \text{where } \mathsf{reduce}\,c = \mathsf{reduce}'(c,\, [\,])
\end{aligned}$$

Next we mechanically optimize the evaluation function into an abstract machine using Danvy and Nielsen's refocusing technique [6], which yields the following abstract machine for the $\lambda\rho$-calculus (and for the $\lambda\widehat{\rho}$-calculus with the grammar of closures restricted to that of $\lambda\rho$):

$$\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Value} \\
\mathsf{refocus}\,((\lambda t)[s],\, \mathsf{ARG}(C,\, c)) &= \mathsf{refocus}\,(t[c \cdot s],\, C) \\
\mathsf{refocus}\,(n[c_1 \cdots c_m],\, C) &= \mathsf{refocus}\,(c_n,\, C) \\
\mathsf{refocus}\,((t_0\;t_1)[s],\, C) &= \mathsf{refocus}\,(t_0[s],\, \mathsf{ARG}(C,\, t_1[s])) \\
\mathsf{refocus}\,((\lambda t)[s],\, [\,]) &= (\lambda t)[s]
\end{aligned}$$

To obtain the standard definition of Krivine's machine, we perform the unfolding of the data type of closures. We consider the language of the $\lambda\rho$-calculus. If we read each syntactic category as a type, then the type of substitutions is an inductive type representing a list of closures:

$$\mathsf{Substitution} \stackrel{\text{def}}{=} \mathsf{Closure\ list},$$

and the type of closures is also an inductive type, satisfying the following equation:

$$\mathsf{Closure} = \mathsf{Term} \times \mathsf{Closure\ list}.$$

Hence $\mathsf{Closure} = \mu X.\mathsf{Term} \times X\mathsf{list}$, and one unfolding of this type yields

$$\begin{aligned} \mathsf{Closure} &\stackrel{\text{def}}{=} \mu X.\mathsf{Term} \times X\mathsf{list} \\ &= \mathsf{Term} \times (\mu X.\mathsf{Term} \times X\mathsf{list})\mathsf{list} \\ &= \mathsf{Term} \times \mathsf{Closure\ list} \\ &= \mathsf{Term} \times \mathsf{Substitution} \end{aligned}$$

For any closure $t[s]$ of type $\mathsf{Closure}$, its unfolding gives a pair $(t,\ s)$ of type $\mathsf{Term} \times \mathsf{Substitution}$, and we can replace each closure in the definition of $\mathsf{refocus}$ by its unfolding. The final result reads as follows:

$$\begin{aligned} \mathsf{refocus} &: \mathsf{Term} \times \mathsf{Substitution} \times \mathsf{Context} \to \mathsf{Value} \\ \mathsf{refocus}\,(\lambda t,\ s,\ \mathsf{ARG}(C,\ c)) &= \mathsf{refocus}\,(t,\ c \cdot s,\ C) \\ \mathsf{refocus}\,(n,\ c_1 \cdots c_m,\ C) &= \mathsf{refocus}\,(t,\ s,\ C),\ \text{where } c_n = t[s] \\ \mathsf{refocus}\,(t_0\ t_1,\ s,\ C) &= \mathsf{refocus}\,(t_0,\ s,\ \mathsf{ARG}(C,\ (t_1[s]))) \\ \mathsf{refocus}\,(\lambda t,\ s,\ [\,]) &= (\lambda t)[s] \end{aligned}$$

It coincides with the definition of Krivine's machine.

4.1 Correctness

We state the correctness of the Krivine's machine with respect to evaluation in the $\lambda\widehat{\rho}$-calculus, for the restricted grammar of closures. The following theorem is a corollary of the full correctness of refocusing [6].

Theorem 1. *Let $\stackrel{\widehat{\rho}}{\to}_{\mathrm{n}}{}^*$ be the reflexive, transitive closure of $\stackrel{\widehat{\rho}}{\to}_{\mathrm{n}}$. For any closure $t[s]$ in $\lambda\widehat{\rho}$,*

$$t[s] \stackrel{\widehat{\rho}}{\to}_{\mathrm{n}}{}^* (\lambda t')[s'] \quad \textit{if and only if} \quad \mathsf{refocus}\,(t,\ s,\ [\,]) = (\lambda t')[s'].$$

The theorem states that Krivine's machine is correct in the sense that it computes closed weak head normal forms, and it realizes the exact call-by-name strategy in the $\lambda\widehat{\rho}$-calculus. As a corollary, we also obtain the correctness of this machine with respect to evaluation in Curien's $\lambda\rho$-calculus.

However, Krivine's machine is better known as an abstract machine with environment, computing weak head normal forms of λ-terms. The following theorem states the correspondence between the evaluation via Krivine's machine and in the λ-calculus with de Bruijn indices (called λ).

Theorem 2 (Correspondence). *For any term t, the call-by-name evaluation of t in λ terminates with a value $\lambda t'$ if and only if*

$$\mathsf{refocus}\ (t, \bullet, [\,]) = (\lambda t'')[s],$$

and $\sigma((\lambda t'')[s], 1) = \lambda t'$, where σ is a function "forcing" all the delayed substitutions in a $\lambda\rho$-closure (definition omitted).

5 Conclusion

We have presented a systematic derivation of Krivine's machine from the call-by-name strategy expressed in the extended calculus of closures $\lambda\widehat{\rho}$. The final step of the derivation (closure unfolding) provides the exact characterization of what has now become folklore: that languages with explicit substitutions mediate between abstract machines using the meta-level operation of substitution (implicit substitution) and those using an environment. The derivation also shows that indeed Curien's calculus is the minimal language of explicit substitutions that corresponds to Krivine's machine, in the sense of Theorem 2. A similar development can be carried over to the call-by-value case, yielding another well known machine—the CEK machine [9].

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In Paul Hudak, editor, *Proceedings of the S teenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, California, January 1990. ACM Press.

[2] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. Research Report BRICS RS-05-15, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005.

[3] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[4] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[5] 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. Extended version available as the technical report BRICS RS-01-23.

[6] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001.

[7] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[8] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.

[9] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[10] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available online at `http://www.pps.jussieu.fr/~krivine/`, 1985.

[11] Pierre Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 60–69, Portland, Oregon, January 1994. ACM Press.

From Continuation Semantics To Abstract Machines

Th. STREICHER and and B. REUS

[1] Fachbereich 4 Mathematik, TH Darmstadt, Schlossgartenstr. 7, 64289 Darmstadt,
streiche@mathematik.th-darmstadt.de
[2] Department of Informatics, University of Sussex, Brighton BN1 9QH
bernhard@sussex.ac.uk

This talk presents a summary of results published in [22]. The objective at the time was to provide an approach to continuation semantics that appeals to domain theorists. Consequently, by contrast to the operational approach (see [19, 18, 3–5, 8, 7, 9, 20, 15]) based on CPS-transformations, we employ a domain theoretic semantics. From the semantic equations of the λ-calculus under consideration it is possible to derive the operational rules for an abstract machine. For the call-by-name λ-calculus with control we derive Krivine's Machine. The same treatment can be successfully applied to the call-by-value λ-calculus with control features yielding the CEK machine, and Parigot's $\lambda\mu$ calculus yielding a machine found independently by De Groote.

It is a well-known fact that classical logic can be translated into constructive logic via so-called $\neg\neg$-translations [24]. As constructive logic has a *proof semantics* corresponding to a (model of a) simple functional language, such translations give rise to a *proof semantics for classical logic*.

The basic idea of our work is based on the $\neg\neg$-translation introduced by Krivine and Girard, see [12, 13], where classical propositions are mapped to negated intuitionistic propositions which are closed under intuitionistic implication and contain the proposition $\bot$ (*falsity*). Classical logic can therefore be considered a subsystem of constructive logic, namely its negative fragment.

These considerations are reflected in the category $\mathcal{N}_R$ of *Negated Domains* which is defined as the full subcategory of the category of domains and continuous functions on objects of the form R^A, where A is a predomain and R is some fixed domain of *responses*. Interpreting the λ-calculus in $\mathcal{N}_R$ instead of ordinary domains gives rise to a *continuation semantics*. The interpretation of a term is an object of R^A mapping continuations in A to responses in R.

Due to the isomorphism $(R^B)^{(R^A)} \cong R^{R^A \times B}$, the domain of continuations for the exponential $(R^B)^{(R^A)}$ is $R^A \times B$. Accordingly, a continuation for a function f from R^A to R^B is a pair $\langle\, d, k \,\rangle$ where $d \in R^A$ is an argument for f and $k \in B$ is a continuation for $f(d)$. The canonical map from $R^{R^{R^A}}$ to R^A sending Φ to $\lambda a{:}A.\, \Phi(\lambda f{:}R^A.\, f(a)) \in R^A$ provides an interpretation of the classical proof principle $\neg\neg P \supset P$. It is (a variant of) this interpretation of *reductio ad absurdum* which serves as interpretation of the control operator $\mathcal{C}$ originally introduced by Felleisen [6]. The idea to understand the control operator $\mathcal{C}$ as a proof of *reductio ad absurdum* via the principle of propositions-as-types was first introduced by Griffin in [11].

In order to interpret untyped λ-calculus in $\mathcal{N}_R$ one has to exhibit a so-called *reflexive object* in $\mathcal{N}_R$ ie a C with $R^C \cong R^{R^C \times C}$. For this purpose it suffices to provide a domain C with $C \cong R^C \times C$. Objects in C are *continuations* and objects in $D = R^C$ are *denotations*. Reflexive objects in $\mathcal{N}_R$ of this form are called *continuation models* of untyped λ-calculus. It turns out that these – up to isomorphism – coincide with Scott's D_∞ with $D = R$.

We use these continuation models of untyped λ-calculus for interpreting the $\lambda\mathcal{C}$-calculus, a λ-calculus extended by Felleisen's control operator $\mathcal{C}$, and (an extension of) M. Parigot's $\lambda\mu$-calculus, see eg. [16]. But continuation semantics have more to offer than just a denotational explanation of control features.

The semantic equations for untyped (call-by-name) λ-calculus can be viewed as transition rules of an environment machine for computing weak head normal forms of λ-terms. We obtain a well-known abstract machine: Krivine's Machine. The correspondence is given by identifying expressions of the form $[\![M]\!]\, e\, k$, ie the meaning of term M in environment e applied to continuation k, with the states of Krivine's Machine, ie expressions of the form $\langle\, [M, env],\, S\,\rangle$ where env is an environment assigning closures to variables and S is a stack of closures.

For extensions of the machines where reduction under λ- and μ-abstractions, resp., is allowed, we prove *computational adequacy*. As a corollary, those machines compute a head normal form of a term t if, and only if, the denotation of t is different from $\bot$. To accomplish this, one uses Andy Pitts' technique for computational adequacy proofs [17].

An analogous treatment is possible for call-by-value languages but in this case one has to employ the opposite of $\mathcal{N}_R$ which is isomorphic to the Kleisli category for the continuation monad $R^{R^{(-)}}$. The relationship (or duality) between call-by-name and call-by-value for $\lambda\mu$ has been further studied by Selinger [21] and Levy [14] (see also [23, 10]). More recent work by Ager, Danvy et al. [1] shows that virtual machines and compilers can be derived from interpreters or normalisation functions. This technique works even for call-by-need λ-calculus [2].

References

1. M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19, New York, NY, USA, 2003. ACM Press.
2. M.S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90(5):223–232, 2004.
3. A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
4. Ph. de Groote. A CPS-translation of the $\lambda\mu$-calculus. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 85–99, Berlin, 1994. Springer.
5. Ph. de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In F. Pfenning, editor, *Proc. 5th International Conference*

on Logic and Automated Reasoning, LPAR '94, volume 822 of *Lecture Notes in Artifical Intelligence*, pages 31–43, Berlin, 1994. Springer.
6. M. Felleisen. *The Calculi of λ_v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher Order Programming Languages.* PhD thesis, Indiana University, 1986.
7. M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.
8. M. Felleisen and D.P. Friedman. Control operators, the SECD machine, and the λ-calculus. In M. Wirsing, editor, *Formal Descriptions of Programming Concepts III*, pages 193–217. North Holland, 1986.
9. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.
10. C. Fürmann and H. Thielecke. On the call-by-value CPS transform and its semantics. *Inf. Comput.*, 188(2):241–283, 2004.
11. T.H. Griffin. A formula-as-types notion of control. In *Conference Record of the Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 47–58. ACM Press, 1990.
12. Y. Lafont. Negation versus implication. Draft, 1991.
13. Y. Lafont, B. Reus, and T. Streicher. Continuation semantics or expressing implication by negation. Technical Report 93-21, University of Munich, 1993.
14. P.B. Levy. *Call-By-Push-Value – A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.
15. C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. *LISP and Symbolic Computation*, 7:57–81, 1994.
16. M. Parigot. $\lambda\mu$-calculus : an algorithmic interpretation of classical natural deduction. In *Proc. International Conference on Logic Programming and Automated Deduction, St. Petersburg*, volume 624 of *LNCS*, pages 190–201, Berlin, 1992. Springer.
17. A.M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *9th MFPS Conference*, volume 802 of *SLNCS*, pages 72–82, Berlin, 1994. Springer.
18. G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
19. J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. of the ACM Annual Conference*, pages 717–740, 1972.
20. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proc. of ACM Conference on Lisp and Symolic Computation*, 1992. Also Technical Report 92-180, Rice University.
21. P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.
22. Th. Streicher and B. Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.
23. H. Thielecke. *Categorical Structure of Continuation Passing Style.* PhD thesis, University of Edinburgh, 1997.
24. A. Troelstra and D. van Dalen. *Constructivism in Mathematics.* North Holland, 1988.

A Typed Calculus Supporting Shallow Embeddings of Abstract Machines

Aaron Bohannon
Zena M. Ariola
Amr Sabry

May 3, 2005

1 Overview

The goal of this work is to draw a formal connection between steps taken by abstract machines and reductions in a system of proof terms for a version of the sequent calculus. We believe that by doing so we shed light on some essential characteristics of abstract machines, proofs in sequent calculus systems, and weak normalization of λ-terms. The machines that we consider are the (call-by-name) Krivine machine and a call-by-value machine that may be called a "right-to-left CEK machine" but with some modifications can be seen as a proto-ZINC machine.

The formal connection we exhibit is, in fact, an embedding of the machines into the term calculus. We embed run-time data structures, such as the control stack and environment, in such a way that the operational semantics of the machine corresponds to reduction steps in the calculus. The abstract machine state, including the code that it executes, is captured as a term; the abstract machine transitions are captured as term reductions.

This is in contrast to specifying the operational semantics on top of the calculus. In other words, our goal is to provide a *shallow embedding* of an abstract machine in a calculus/logic, as opposed to a *deep embedding*. This allows reasoning about the machine *inside* the logic itself instead of on *top* of it. The logical formulae that are assigned to proof terms provide a type system for the term language via the Curry-Howard isomorphism, and because of the method of embedding the machines into the terms, this type system can be directly lifted to the machine code and machine states, thereby allowing an elegant and simple formulation of safety, based on the subject reduction theorem of the calculus.

2 Tail-Recursive Evaluators

Plotkin [9] showed how abstract machines could be seen as an implementation of an evaluation function for a functional programming language. It can be noted that a basic evaluation function, whether implementing call-by-name or call-by-value semantics, is not tail-recursive. This corresponds to the fact that the small-step operational semantics has a recursive definition, relying on congruence rules. An implementation of these rules naturally involves a process of searching for the next redex, which may be arbitrarily deep in a term. This search must be managed with care when computing a sequence of reductions, since the cost of computing a single reduction step is linear in the size of the term [4].

An implementation of an abstract machine, on the other hand, can be directly written down as a tail-recursive function. One view of abstract machines is that they are simply tail-recursive evaluators. The process of constructing such an evaluator, as presented by Reynolds [11] and more recently explored by Ager *et al.* [2], consists of defunctionalizing the continuations in a CPS interpreter. A defunctionalized continuation is actually just a data structure representing an evaluation context. As shown by Herbelin [7], proof terms for sequent calculi have a computational interpretation as evaluation contexts. Thus, it is very natural to believe that sequent calculi would have a relationship with abstract machines. A simple connection between the $\overline{\lambda}\mu\tilde{\mu}$-calculus and the Krivine machine was observed by Curien and Herbelin [3].

3 Calculi for Machines

If a machine correctly implements evaluation of λ-terms, then it will certainly be possible to prove a correspondence between the machine and the λ-calculus. However, there are various calculi that may be much closer to abstract machines, *i.e.* have a more direct statement and proof of correspondence. The closest correspondence would occur when we could define a translation from machine states to terms (or vice-versa) in a purely compositional manner, such that the transitions of the machine could be matched up with the steps of a particular reduction strategy on the term so that neither one would ever take more than a statically fixed number of steps for a given step in the other system.

In order to achieve a correspondence at this level, the term calculus must possess certain features. First, it must be able to simulate weak β-reduction without performing arbitrarily deep searches for the next redex. Otherwise, one step in the calculus would correspond to an arbitrary number of steps of the machine. The $\overline{\lambda}\mu\tilde{\mu}$-calculus [3] described by Curien and Herbelin has this property. They define translations from λ-terms to $\overline{\lambda}\mu\tilde{\mu}$-terms, such that the computational reduction rules of the $\overline{\lambda}\mu\tilde{\mu}$-calculus can be used without any congruence rules to simulate weak β-reduction of the λ-terms. Moreover, by making a simple choice of which way to resolve a critical pair, the same computational rules can be used to simulate either call-by-name or call-by-value reduction on λ-terms.

Abstract machines are also designed to break down the process of substitution into small steps, and they generally carry out these substitutions in a lazy manner. Thus, the other important feature of a good calculus for our simulations is an explicit notion of substitution. Calculi with explicit substitutions were investigated by Abadi *et al.* [1] and certain variants have been used to prove the correctness of abstract machines, *e.g.* the λ_{env}, which was used by Leroy [8] when the ZINC machine was introduced. Hardin, Maranget, and Pagano [6] proposed the $\lambda\sigma_w$-calculus as a "calculus of closures" for proving the correctness of abstract machines and representing the output of compilers. They succeeded in providing an elegant calculus specialized for weak β-reduction, but the calculus was still based upon the structures of natural deduction and therefore cannot satisfy the requirement of the last paragraph. On the other hand, the $\overline{\lambda}\mu\tilde{\mu}$-calculus does not provide any notion of explicit substitution, so it is not immediately satisfactory, either.

A version of the $\overline{\lambda}\mu\tilde{\mu}$-calculus with explicit substitutions has been studied and found to be well-behaved [10]. Unfortunately, the inclusion of explicit substitutions is not, in itself, enough to guarantee that a calculus has the properties that we desire. The reason is that when a term has multiple substitutions at the outermost level, the next redex must (eventually) be a propagation of the innermost substitution. The search for this redex, which may be arbitrarily deep, would not mirror the operation of an abstract machine. One way around this is to take the approach of the $\lambda\sigma_w$-calculus and use *simultaneous* explicit substitutions. However, we take another approach and represent environments within the calculus. This approach was inspired by Douence and Fradet [5], but instead of working abstractly at the level of combinators, we provide a concrete embedding of environments in the calculus, which gives us the benefit of being able to apply the type system of the calculus to the environments in a direct way.

4 Our Development

The calculus into which we embed the abstract machines is a slightly modified version of the $\overline{\lambda}\mu\tilde{\mu}$-calculus with explicit substitutions that was studied by Polonovski [10]. Our first modification is the addition of an explicit weakening construct that acts as a method of garbage collection. This is necessary for simulating the mutable machine registers that typical abstract machines have. The other modification involves focusing on a subset of terms for which we no longer care about α-equivalence. We do this by constraining both the grammar of the terms and the contexts of the typing judgments. This is not technically necessary but makes the embedding more transparent. The restricted set of terms are allowed to use only a single term variable—which is used to encode an accumulator register—and two context variables—one is used for encoding the run-time stack pointer and the other for encoding the environment pointer. We call this the $\overline{\lambda}\mu\tilde{\mu}\mathtt{r}\uparrow$-calculus.

The $\overline{\lambda}\mu\tilde{\mu}\mathtt{r}\uparrow$-calculus imposes a useful structure on terms that is closer to the

level of an abstract machine. In fact, the individual reduction steps in this system are much more fine-grained that one would see in most abstract machines. In order to show how the reduction steps of this calculus correspond to abstract machines, such as the Krivine machine, it is useful to develop a sort of toolkit of "macros" for commands and terms in the $\overline{\lambda}\mu\tilde{\mu}\mathtt{x}\uparrow$-calculus. Thereafter, we exhibit a set of reduction steps on these macro-terms that correspond to multiple reduction steps at the raw term level. These macro-reductions implement a specific strategy of small-step reductions; hence, we present one set implementing call-by-name and one set implementing call-by-value, with a concrete description of the strategies that they implement on the pure $\overline{\lambda}\mu\tilde{\mu}\mathtt{x}\uparrow$-terms.

It becomes apparent that these coarse-grained systems are, in a literal sense, abstract machines themselves built directly out of the $\overline{\lambda}\mu\tilde{\mu}\mathtt{x}\uparrow$-calculus. It is then a very small (almost trivial) step to draw the correspondence with the traditional Krivine machine and a call-by-value machine that is similar to the ZINC machine, and we see how these machines arise out of the duality of the calculus. The typing rules of the calculus are then also lifted in the obvious way to give a type system to the macro-terms and, by extension, the abstract machines themselves, thus allowing an elegant statement of safety at the level of machines.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–46, New York, 1990. ACM Press.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Research Series RS-03-14, BRICS, March 2003. 36 pp.

[3] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 233–243, New York, 2000. ACM Press.

[4] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2001.

[5] Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. *ACM Trans. Program. Lang. Syst.*, 20(2):344–387, 1998.

[6] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends within the lambda-sigma calculus. Technical Report RR-3034, INRIA, November 1996.

[7] H. Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Proc. Annual Conference of the European Association for Computer Science Logic, Kazimierz, Poland*, volume 933 of *Lecture Notes in Computer Science*, Berlin, 1994. Springer-Verlag.

[8] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, February 1990.

[9] G. D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[10] Emmanuel Polonovski. *Substitutions explicites, logique et normalisation*. PhD thesis, Université Paris 7, 2004.

[11] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717–740. ACM Press, 1972.

A call-by-name lambda-calculus machine

Jean-Louis Krivine

Université Paris VII, C.N.R.S.

2 place Jussieu 75251 Paris cedex 05

krivine@pps.jussieu.fr

Introduction

We present, in this paper, a particularly simple lazy machine which runs programs written in -calculus. It was introduced by the present writer more than twenty years ago. It has been, since, used and implemented by several authors, but remained unpublished.
In the first section, we give a rather informal, but complete, description of the machine. In the second part, definitions are formalized, which allows us to give a proof of correctness for the execution of -terms. Finally, in the third part, we build an extension for the machine, with a control instruction (a kind of call-by-name `call/cc`) and with continuations.

This machine uses *weak head reduction* to execute -calculus, which means that the active redex must be at the very beginning of the -term. Thus, computation stops if there is no redex at the head of the -term. In fact, we reduce at once a whole chain $x_1 \ldots x_n$. Therefore, execution also stops if there are not enough arguments.

The first example of a -calculus machine is P. Landin's celebrated SECD-machine [6]. The one presented here is quite different, in particular because it uses call-by-name. This needs some explanation, since functional programming languages are, most of the time, implemented through call-by-value. Here is the reason for this choice :

Starting in the sixties, a fascinating domain has been growing between logic and theoretical computer science, that we can designate as the *Curry-Howard correspondence*. Succinctly, this correspondence permits the transformation of a mathematical proof into a program, which is written :

in -calculus if the proof is intuitionistic et only uses logical axioms ;

in -calculus extended with a control instruction, if one uses the law of excluded middle [2] and the axioms of set theory [3], which is most often the case.

Other instructions are necessary if one uses additional axioms, such as the Axiom of Choice [4]. The programs obtained in this way are indeed very complex and two important problems immediately arise : how should we *execute* them and what is their *behaviour*? Naturally, these questions are not independent, so let us give a more precise formulation :

(i) How should one execute these programs so as to obtain a *meaningful* behaviour ?
(ii) Assuming an answer to question (i), what is the common behaviour (if any) of the programs obtained from different proofs of the same theorem ?

It is altogether surprising that *there be an answer* to question (i) ; it is the machine presented below. I believe that, in itself, is a strong reason for being interested in it.
Let us give a very simple but illuminating example, namely the following theorem of Euclid :

There exists infinitely many prime numbers.

Let us consider a proof D of this theorem, using the axioms of classical analysis, or those of classical set theory ; consider, further, the program P_D extracted from this proof. One would like to have the following behaviour for P_D :

wait for an integer n ;
produce then a prime number $p \quad n$.

That is exactly what happens when the program P_D is executed by the present machine. But it's not true anymore if one uses a different execution mechanism, for instance call-by-value. In this case one gets, in general, an aberrant behaviour and no meaningful output.

This machine was thus conceived to execute programs obtained from mathematical proofs. It is an essential ingredient of the classical realizability theory developed in [3, 4] to extend the Curry-Howard correspondence to analysis and set theory. Thanks to the remarkable properties of weak head reduction, one can thus, *inter alia*, search for the *specification* associated with a given mathematical theorem, meaning the shared behaviour of the programs extracted from the various proofs of the theorem under consideration : this is question (ii) stated earlier. That problem is a very interesting one, it is also quite difficult and has only been solved, up to now, in very few cases, even for tautologies (cf. [5]). A further interesting side of this theory is that it illuminates, in a new way, the problem of proving programs, so very important for applications.

1 Description of the machine

Terms of -calculus are written with the notation $(t)u$ for application of t to u. We shall also write tu if no ambiguity arise ; $(\ldots((t)u_1)u_2\ldots)u_k$ will be also denoted by $(t)u_1\ldots u_k$ or $tu_1\ldots u_k$.

We consider three areas in the memory : the *stack*, the *heap* and the *term area* where are written the -terms to be performed. We denote by $\&t$ the address of the term t.

In the heap, we have objects of the following kinds :

- environment : a finite sequence $(e, \ _1, \ldots, \ _k)$ where e is the address of an environment, and $_1, \ldots, \ _k$ are *closures*. There is also an empty environment.
- closure : An ordered pair $(\&t, e)$ built with the address of a term (in the term area) and the address of an environment.

The elements of the stack are closures.

Intuitively, closures are the values which -calculus variables take.

Execution of a term

The term t_0 to be performed is written, in ''compiled form'' in the term area. The ''compiled form'' of a term is obtained by replacing each occurrence of x with and each variable oc-

currence by an ordered pair of integers $(\ , k)$ (it is a variant of the *de Bruijn* notation [1], see the definition below). We assume that t_0 is a closed term. Thus, the term area contains only closed terms.
Nevertheless, terms may contain symbols of constant, which are performed with some predefined programs. For example :

- the constant symbol is the name of another closed term and the program consists in the execution of this term.
- there may be an input-output library.

The execution consists in constantly updating a closure (T, E) and the stack. T is the address of the current subterm : it is, therefore, an instruction pointer which runs along the term to be performed ; E is the current environment.
At the beginning, T is the address of the first term t_0 to be performed. Since it is a closed term, E is the null pointer (which points to the empty environment).
At each moment, there are three possibilities according to the term pointed by T : it may be an application $(t)u$, an abstraction $\ x\,t$ or a variable.

- Execution of $(t)u$.
  We push the closure $(\&u, E)$ on the top of the stack and we go on by performing t : thus T points now to t and E does not change.
- Execution of $\ x_1 \ldots\ x_n\ t$ where t does not begin with a $\ $; thus, T points to $\ x_1$.
  A new environment $(e, \ _1, \ldots, \ _n)$ is created : e is the address of E, $\ _1, \ldots, \ _n$ are ''popped'' : we take the n top entries off the stack. We put in E the address of this new environment and we go on by performing t : thus T points now to t.
- Execution of x (a $\ $-calculus variable).
  We fetch as follows the value of the variable x in the environment E : indeed, it is a bound occurrence of x in the initial term t_0. Thus, it was replaced by an ordered pair of integers $<\ , k>$. If $\ = 0$, the value we need is the k-th closure of the environment E. If $\ \ 1$, let E_1 be the environment which has its address in E, E_2 the one which has its address in E_1, etc. Then, the value of x is the k-th closure of E . This value is an ordered pair $(T\ , E\ )$ which we put in (T, E).

Remark.
The intuitive meaning of these rules of execution is to consider the symbols $\ x, (, x$ of $\ $-calculus as elementary instructions :

- '' $\ x$'' is : ''pop'' in x and increment the instruction pointer.
- ''('' is : ''push'' the address of the corresponding '')'' and increment the instruction pointer.
- ''x'' is : go to the address which is contained in x.

It remains to explain how we compute the integers $\ , k$ for each occurrence of a variable x, i.e. how we ''compile'' a closed term t. More generally, we compute $\ $ for an occurrence of x in an arbitrary term t, and k when it is a bound occurrence in t. This is done by induction on the length of t.

If $t = x$, we set $= 0$. If $t = uv$, the occurrence of x we consider is in u (resp. v). We compute , and possibly k, in u (resp. v).
Let now $t = x_1 \dots x_n u$ with $n > 0$, u being a term which does not start with . If the occurrence of x we consider is free in t, we compute in t by computing in u then adding 1. If this occurrence of x is bound in u, we compute and k in u. Finally, if this occurrence is free in u and bound in t, then we have $x = x_i$. We compute in u, and we set $k = i$.

2 Formal definitions and correction proof

Compiled terms or $_B$-terms (this notion is a variant of the *de Bruijn* notation) are defined as follows :

- A constant a or an ordered pair $<, k>$ ($k \quad 1$) of integers is a $_B$-term (*atomic* term).
- If t, u are $_B$-terms, then so is $(t)u$.
- If t is a $_B$-term which does not start with ^{i} and if $n \quad 1$, then $^n t$ is a $_B$-term.

Let us consider, in a $_B$-term t, an occurrence of a constant a or of $<, k>$ (ordered pair of integers). We define, in an obvious way, the *depth* of this occurrence, which is the number of ^{n} symbols above it. The definition is done by induction on the length of t :
If there is no symbol in t, the depth is 0.
If $t = (u)v$, the occurrence we consider is either in u or in v. We compute its depth in this subterm and do not change it.
If $t = {}^n u$, we compute the depth of this occurrence in the subterm u and we add 1 to it.

An occurrence of $<, k>$ in t is said to be *free* (resp. *bound*) if its depth in t is (resp. $>$). Of course, each occurrence of a constant a is free. Thus, we could write constants as ordered pairs $<, k>$.

Weak head reduction

Consider a $_B$-term of the form $(\ ^n t)u_1 \dots u_p$ with $p \quad n$. Then, we can carry out a *weak head reduction step* : we get the $_B$-term $t\, u_{n+1} \dots u_p$ (or t , if $n = p$) ; the term t is obtained by replacing, in t, each occurrence of $<, i>$ $(1 \quad i \quad n)$ the depth of which is exactly , with u_i. We write $t \quad u$ if u is obtained from t by a finite (possibly null) number of weak head reduction steps.

Alpha-equivalence

Let t be a closed -term, with constants. We define, by induction on t, its ''compiled'' form, which is a $_B$-term denoted by $\mathsf{B}(t)$:
$\mathsf{B}(a) = a$; if $t = uv$, then $\mathsf{B}(t) = \mathsf{B}(u)\mathsf{B}(v)$.
If $t = x_1 \dots x_n u$ where u does not begin with , consider the $_B$-term :
$\mathsf{B}(u[a_1/x_1, \dots, a_n/x_n])$, where $a_1, \dots, a_n$ are new constants.
We replace in it each occurrence of a_i by the ordered pair $<, i>$, where is the depth of this occurrence in $\mathsf{B}(u[a_1/x_1, \dots, a_n/x_n])$. We get in this way a $_B$-term U and we set :
$\mathsf{B}(t) = {}^n U$.

The compiled form of a -term t is a variant of the de Bruijn notation for t. Its main property, expressed by theorem 1, is that it depends only on -equivalence class of t. This property is not used in the following, but the brevity of the proof below convinced me to give it here.

It is clear that the weak head reduction of a -term t corresponds to the weak head reduction of its compiled form $\mathsf{B}(t)$.

Theorem 1. *Two closed -terms t, t are -equivalent if and only if $\mathsf{B}(t) = \mathsf{B}(t\ )$.*

We use the notation $t \quad t$ for -equivalence. The proof is done by induction on t. The result is clear if $t = a$ or $t = uv$.

Assume that $t = \ x_1 \ldots \ x_n\, u$ where u does not begin with . If $t \quad t$ or if $\mathsf{B}(t) = \mathsf{B}(t\ )$, then $t = \ x_1 \ldots \ x_n\, u$ where u does not begin with . Let $a_1, \ldots, a_n$ be new constants ; then, by definition of -equivalence, we have :

$t \quad t \quad u[a_1/x_1, \ldots, a_n/x_n] \quad u\,[a_1/x_1, \ldots, a_n/x_n]$; by induction hypothesis, this is equivalent to $\mathsf{B}(u[a_1/x_1, \ldots, a_n/x_n]) = \mathsf{B}(u\,[a_1/x_1, \ldots, a_n/x_n])$.

If $\mathsf{B}(u[a_1/x_1, \ldots, a_n/x_n]) = \mathsf{B}(u\,[a_1/x_1, \ldots, a_n/x_n])$, we obviously have $\mathsf{B}(t) = \mathsf{B}(t\ )$. But conversely, we get $\mathsf{B}(u[a_1/x_1, \ldots, a_n/x_n])$ from $\mathsf{B}(t)$, by removing the initial ^{n} and replacing $< , i>$ with a_i for every occurrence of $< , i>$ the depth of which is precisely equal to .

Therefore, we have $\mathsf{B}(u[a_1/x_1, \ldots, a_n/x_n]) = \mathsf{B}(u\,[a_1/x_1, \ldots, a_n/x_n]) \quad \mathsf{B}(t) = \mathsf{B}(t\ )$ and finally $t \quad t \quad \mathsf{B}(t) = \mathsf{B}(t\ )$.

□

Closures, environments and stacks

We now define recursively *closures* and *environments* :

is an environment (the empty environment) ; if e is an environment and $_1, \ldots, \ _n$ are closures $(n \quad 0)$, then the finite sequence $(e, \ _1, \ldots, \ _n)$ is an environment.

A closure is an ordered pair (t, e) composed with a $_B$-term t and an environment e.

A *stack* is a finite sequence $= (\ _1, \ldots, \ _n)$ of closures.

We denote by . the stack $(\ , \ _1, \ldots, \ _n)$ obtained by ''pushing'' the closure on the top of the stack .

Execution rules

A *state* of the machine is a triple $(t, e, \ )$ where t is a $_B$-term, e an environment and a stack. We now give the execution rules, by which we pass from a state $(t, e, \ )$ to the next one $(t\ , e\ , \ )$:

- If $t = (u)v$, then $t\ = u$, $e\ = e$ and $\ = (v, e)$. .
- If $t = \ ^n u$, then $e\ = (e, \ _1, \ldots, \ _n)$ and $\ = \ _1 \ldots \ _n$. .

The length of the stack must be n, otherwise the machine stops.

- If $t = < , k>$: let $e_0 = e$ and let e_{i+1} be the environment which is the first element of e_i, for $i = 0, 1, \ldots$ If $e_i =$ for an i , then the machine stops.

Otherwise, we have $e \ = (e\ _{+1}, (t_1, e_1), \ldots, (t_p, e_p))$. If $k \quad p$, we set $t\ = t_k$, $e\ = e_k$ and $\ = \ .$

If $k > p$, the machine stops.

The value of a closure

Given any closure $= (t, e)$, we define a $_B$-term which is denoted by $^-$ or $t[e]$; it is defined by induction on e as follows :
We set $t[\ ] = t$; $t[(e, \ _1, \ldots, \ _n)] = u[e]$ where u is the $_B$-term we obtain by replacing in t each occurrence of $< \ , i>$ with :
$\leq \quad 1, i>$ if $\quad$ is strictly greater than the depth of this occurrence ;
$\bar{}_i$ (resp. d) if $\quad$ is equal to the depth of this occurrence and $i \quad n$ (resp. $i > n$) ; d is a fixed constant.

Remark. We observe that $t[e]$ is a closed $_B$-term, which is obtained by replacing in t free occurrences of $< \ , i>$ with suitable $_B$-terms. These closed $_B$-terms are recursively provided by the environment e ; the constant d is used as a "wild card", when the environment e does not provide anything.

Theorem 2.
Let $(t, e, \ )$, $(t \ , e \ , \ )$ be two consecutive states of the machine, with $\quad = (\ _1, \ldots, \ _m)$ and $\quad = (\ _1, \ldots, \ _m)$. Then $t[e]\bar{}_1 \ldots \bar{}_m$ is a closed $_B$-term and $t[e]\bar{}_1 \ldots \bar{}_m \quad t\,[e\,]\bar{}_1 \ldots \bar{}_m$.

Recall that the symbol $\quad$ denotes the weak head reduction. We shall use the notation $t[e]^-$ for $t[e]\bar{}_1 \ldots \bar{}_m$ when $\quad$ is the stack $(\ _1, \ldots, \ _m)$.
There are three possible cases for t :

• $t = (u)v$: we have $t[e] = u[e]v[e]$, $t\,[e\,] = u[e]$ (since $e \ = e$) and $\quad = (v, e)$. . Therefore $t[e]^- = t\,[e\,]^-$.

• $t = \ ^n u$: then we have $n \quad m$ and $t = u$, $e = (e, \ _1, \ldots, \ _n)$, $\quad = (\ _{n+1}, \ldots, \ _m)$. We must show that $(\ ^n u)[e]\bar{}_1 \ldots \bar{}_n \quad u[(e, \ _1, \ldots, \ _n)]$.
By the definition of the value of a closure, we have $u[(e, \ _1, \ldots, \ _n)] = v[e]$, where v is obtained by substituting, in u, $\bar{}_i$ for the occurrences of $< \ , i>$ the depth of which is $\quad$ and $< \quad 1, i>$ for the ones the depth of which is $<$.
Now, if we perform a sep of weak head reduction in $(\ ^n u)[e]\bar{}_1 \ldots \bar{}_n$, we carry out exactly the substitution which is defined by e on the free occurrences of $< \ , i>$ in v ; we therefore get $v[e]$.

• $t = < \ , k>$: let $e_0 = e$ and e_{j+1} the environment which is the first element of e_j, if $e_j =$. Then, by definition of $t[e]$, we have $t[e] = \bar{}_k$ where $\ _k$ is the k-th closure of the environment $e \ = (e \ _{+1}, \ _1, \ldots, \ _p)$. Now, we have $\ _k = (t \ , e \ )$ by the reduction rules of the machine. Therefore, $t[e]^- = \ \bar{}_k{}^- = t\,[e\,]^-$, since $\quad = \quad$.

□

This theorem shows that the machine which has been described above computes correctly in the following sense : if $t \quad at_1 \ldots t_k$, where t is a closed -term and a is a constant, then the execution of $\mathsf{B}(t)$, from an empty environment and an empty stack, will end up in $a\mathsf{B}(t_1) \ldots \mathsf{B}(t_k)$. In particular, if $t \quad a$, then the execution of $\mathsf{B}(t)$ will end up in a.

3 Control instruction and continuations

We now extend this machine with a call-by-name control instruction, and with continuations. There are two advantages : first, an obvious utility for programming ; second, in the frame of realisability theory (see the introduction), this allows the typing of programs in *classical logic* and

no longer only in intuitionistic logic. Indeed, the type of the instruction `call/cc` is Peirce's law $((A \quad B) \quad A) \quad A$ (see [2]).
As we did before, we give first an informal description of the machine, then mathematical definitions.

3.1 Description of the machine

We describe only the changes. Terms are the same but there is one more constant, which is denoted by `cc`. There are still three memory areas : the *stack* and the *term area*, which are the same as before, and the *heap* which contains objects of the following kinds :

- environment : same definition.
- closure : it is, either an ordered pair $(\&t, e)$ built with the address of a term (in the term area) and the address of an environment ; or the address $\&$ of a *continuation*.
- continuation : it is a sequence $= (\ _1, \ldots, \ _n)$ of closures.

Execution of a term

The execution consists in constantly updating the *current closure* and the stack. There are now two possible forms for the current closure : $(\& \ , e)$ (where is a term) or $\&$ (where is a continuation).
Consider the first case : $= (\& \ , e)$. There are now four possibilities for the term : an application $(t)u$, an abstraction $x\,t$, a variable x or the constant `cc`. Nothing is changed during execution in the first two cases.

- Execution of x (-calculus variable).
  As before, we fetch the value of the variable x in the environment e, which gives a closure which becomes the current closure . The stack does not change.
- Execution of `cc`.
  We pop a closure which becomes the current closure . We save the stack in a continuation and we push the address of (this address is a closure).
  Therefore, the stack which was of the form $(\ , \ _1, \ldots, \ _n)$, has become $(\& \ , \ _1, \ldots, \ _n)$ with $= (\ _1, \ldots, \ _n)$.

Consider now the second case, when the current closure is of the form $\&$. Then, the execution consists in popping a closure , which becomes the current closure and in replacing the current stack with .

3.2 Formal definitions

$_B$-terms are defined as before, with a distinguished constant, which is denoted by `cc`.
We define recursively the *closures*, the *environments* and the *stacks* (which are now also called *continuations*) :
is an environnement (the empty environnement) ; if e is an environment and $\ _1, \ldots, \ _n$ are

closures ($n \quad 0$), then the finite sequence $(e, \ _1, \ldots, \ _n)$ is an environnement.
A closure is either *a stack*, or an ordered pair (t, e) composed with a $\ _B$-term t and an environment e.
A *stack* (or *continuation*) is a finite sequence $\ = (\ _1, \ldots, \ _n)$ of closures. We denote by $\ .\ $ the stack $(\ , \ _1, \ldots, \ _n)$ which is obtained by "pushing" the closure $\ $ on the top of the stack $\ $.

Execution rules

A *state* of the machine is an ordered pair $(\ , \ )$ where $\ $ is a closure and $\ $ is a stack. We give now the execution rules, by which we pass from a state $(\ , \ )$ to the next one $(\ , \ )$:

- If $\ $ is a stack, then $\ $ is the closure which is on the top of the stack $\ $ (if $\ $ is empty, the machine stops) and $\ = \ $.
- Else, we have $\ = (t, e)$ and there are four possibilities for the $\ _B$-term t :
  - If $t = (u)v$, then $\ = (u, e)$ and $\ = (v, e)$. .
  - If $t = \ ^n u$, then $\ = (u, e\ )$ with $e\ = (e, \ _1, \ldots, \ _n)$ and $\ = \ _1 \cdots \ _n$. .

The length of the stack $\ $ must be $\ n$, otherwise the machine stops.

- If $t = <\ , k>$: let $e_0 = e$ and let e_{i+1} be the environment which is the first element of e_i, for $i = 0, 1, \ldots$ If $e_i = \ $ for an $i \ $, then the machine stops.

Else, we have $e\ = (e\ _{+1}, \ _1, \ldots, \ _p)$. If $k \quad p$, we set $\ = \ _k$ and $\ = \ $.
If $\ k > p$, the machine stops.

- If $t = \texttt{cc}$, then $\ $ is the closure which is on the top of the stack $\ $ (if $\ $ is empty, the machine stops). Thus, we have $\ = \ .\ $ where $\ $ is a stack. Therefore, $\ $ is also a closure, which we denote by $\ .$ Then, we set $\ = \ . .$

References

[1] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae, 34, p. 381-392, 1972.

[2] T. Griffin. A formulæ-as-type notion of control. In Conference Record of the 17th A.C.M. Symposium on principles of Programming Languages, 1990.

[3] J.-L. Krivine. Typed $\ $-calculus in classical Zermelo-Frænkel set theory. Archiv for Mathematical Logic, 40, 3, p. 189-205, 2001.

[4] J.-L. Krivine. Dependent choice, 'quote' and the clock. Theoretical Computer Science, 308, p. 259-276, 2003.

[5] V. Danos, J.-L. Krivine. Disjunctive tautologies and synchronisation schemes. Computer Science Logic'00, Lecture Notes in Computer Science n. 1862, p. 292-301, 2000.

[6] P. J. Landin. The mechanical evaluation of expressions. The Computer Journal, vol. 6, p. 308-320, 1964.

From Krivine's machine to the Caml implementations

Xavier Leroy
INRIA
http://pauillac.inria.fr/~xleroy

Calculating an Exceptional Machine *

Graham Hutton
School of Computer Science and IT
University of Nottingham
http://www.cs.nott.ac.uk/~gmh

Abstract

In previous work we showed how to verify a compiler for a small language with exceptions. In this article we show how to calculate, as opposed to verify, an abstract machine for this language. The key step is the use of Reynold's *defunctionalization*, an old program transformation technique that has recently been rejuvenated by the work of Danvy et al.

*Joint work with Joel Wright.

Jumping Semantics For Call-By-Push-Value

Paul Blain Levy

University of Birmingham

Abstract. We give a jumping machine for a higher-order language, embodying the intuition that calling a procedure is a jump, and returning from a procedure is also a jump. The machine makes it very easy to execute a program on paper, so it is a kind of pedagogical tool. It represents a closure in a graphical way, so that a jump does not need to be accompanied by a separate change of environment (as it does in the Krivine machine).
The language used is call-by-push-value, making it easy to obtain similar jumping machines for call-by-value and call-by-name calculi (as these are fragments of call-by-push-value).

1 Introduction

1.1 Jumping Semantics

Beginning programmers learn a simple intuition for procedures and functions.

- A procedure or function call causes a *jump* from the calling code to the procedure or function
- The return of a value by a function, or the termination of a procedure, causes a *jump* to the frame on top of the stack, which is popped.

The goal of this paper is to present, informally, a *jumping machine* that embodies these two intuitions, for a higher-order language. The machine is based on a graphical view of closures.

It must be stressed from the outset what this kind of operational semantics does not achieve:

- it is not suitable as a practical implementation of programming languages, principally because there is no garbage collection
- it is not a convenient way of reasoning about programs, because—like many graphical notations—its formalization (which we omit in this paper) is rather complex.

So what is its contribution? Simply that it is a very easy way of executing a program *on paper*. It can, therefore, be seen as a kind of pedagogical tool.

A somewhat similar formalization of jumping in a higher-order setting appears in [DR99]. In that paper, after a very careful analysis of the "geometry of interaction" machine for MELL, a jumping machine is given as an optimization. This induces a jumping machine for simply typed CBN λ-calculus with a single free type identifier ι. Because pattern-matching (in particular, conditionals) is absent from this language, there are no frames on the stack.

1.2 Languages

Many analyses of abstract machines, such as [ABDM03], consider both call-by-value (CBV) and call-by-name (CBN) variants. In this paper, instead, we present our machine for *call-by-push-value* (CBPV) [Lev99]. This is a calculus that contains both CBV and CBN calculi as fragments, and consequently jumping machines for these calculi can easily be obtained from the CBPV one.

In [Lev04], a similar jumping machine is given for a CPS language, of course without a stack. From this, one can obtain a jumping machine for CBPV, by applying the appropriate CPS transform (described in [Lev04]). But that is different from the machine we give in this paper, which is *not* continuation-passing: when calling a procedure, we do not pass the stack as an additional argument. This is surely closer to the programmer's intuition that we are trying to capture.

1.3 Structure Of Paper

In Sect. 2, we review call-by-push-value, omitting the denotational aspects. We present operational semantics in two traditional styles (first-order interpreter and CK-machine) in Sect. 3. In Sect. 4, we give an *informal* account of the jumping semantics, executing an example program in detail; another example is given in Sect. 5. We discuss correctness in Sect. 6.

Finally, in Sect. 7, we compare and contrast our jumping machine to other machines in the literature.

2 Review Of Call-By-Push-Value

CBPV has two disjoint classes of terms: values and computations. It likewise has two disjoint classes of types: a value has a value type, while a computation has a computation type. For clarity, we underline computation types. The types are given by

$$\begin{array}{lll} \text{value types} & A ::= & U\underline{B} \mid \sum_{i\in I} A_i \mid 1 \mid A \times A \\ \text{computation types} & \underline{B} ::= & FA \mid \prod_{i\in I} \underline{B}_i \mid A \to \underline{B} \end{array}$$

where I can be any countable set (finite, in finitary CBPV). The meaning of F and U is as follows. A computation of type FA *returns* a value of type A. A value of type $U\underline{B}$ is a *thunk* of a computation of type $\underline{B}$. When later required, it can be *forced* i.e. executed.

Unlike in call-by-value, a function in CBPV is a computation, and hence a function type is a computation type. We will discuss this further in Sect. 3.

Like in call-by-value, an identifier in CBPV can be bound only to a value, so it must have value type. We accordingly define a *context* Γ to be a sequence

$$\mathtt{x}_0 : A_0, \ldots, \mathtt{x}_{n-1} : A_{n-1}$$

of identifiers with associated value types. We often omit the identifiers and write just $A_0, \ldots, A_{n-1}$. We write $\Gamma \vdash^{\mathsf{v}} V : A$ to mean that V is a value of type A, and we write $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ to mean that M is a computation of type $\underline{B}$.

The terms of CBPV are given in Fig. 1. We assume formally that all terms are explicitly typed, but in this paper, to reduce clutter, we omit explicit typing information. We omit the rules for 1, which follow those for $\times$.

We explain some of the less familiar constructs as follows. M `to x`. N is the sequenced computation that first executes M, and when, this returns a value V proceeds to execute N with `x` bound to V. This was written in Moggi's syntax using `let`, but we reserve `let` for mere binding. The keyword `pm` stands for "pattern-match", and the symbol ‘ represents application in reverse order. Because we think of $\prod_{i \in I}$ as the type of functions taking each $i \in I$ to a computation of type $\underline{B}_i$, we have made its syntax similar to that of $\to$.

$$\frac{}{\Gamma, \mathtt{x} : A, \Gamma' \vdash^{\mathsf{v}} \mathtt{x} : A} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \quad \Gamma, \mathtt{x} : A \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{let}\ V\ \mathtt{be\ x}.\ M : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{c}} \mathtt{return}\ V : FA} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : FA \quad \Gamma, \mathtt{x} : A \vdash^{\mathsf{c}} N : \underline{B}}{\Gamma \vdash^{\mathsf{c}} M\ \mathtt{to\ x}.\ N : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{v}} \mathtt{thunk}\ M : U\underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : U\underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{force}\ V : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A_i}{\Gamma \vdash^{\mathsf{v}} (i, V) : \sum_{i \in I} A_i}\ i \in I \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : \sum_{i \in I} A_i \quad \cdots \quad \Gamma, \mathtt{x} : A_i \vdash^{\mathsf{c}} M_i : \underline{B}\ \cdots_{i \in I}}{\Gamma \vdash^{\mathsf{c}} \mathtt{pm}\ V\ \mathtt{as}\ \{\ldots, (i, \mathtt{x}).M_i, \ldots\} : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \quad \Gamma \vdash^{\mathsf{v}} V' : A'}{\Gamma \vdash^{\mathsf{v}} (V, V') : A \times A'} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \times A' \quad \Gamma, \mathtt{x} : A, \mathtt{y} : A' \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{pm}\ V\ \mathtt{as}\ (\mathtt{x}, \mathtt{y}).M : \underline{B}}$$

$$\frac{\cdots \quad \Gamma \vdash^{\mathsf{c}} M_i : \underline{B}_i \quad \cdots_{i \in I}}{\Gamma \vdash^{\mathsf{c}} \lambda\{\ldots, i.M_i, \ldots\} : \prod_{i \in I} \underline{B}_i} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : \prod_{i \in I} \underline{B}_i}{\Gamma \vdash^{\mathsf{c}} \hat{\imath}`M : \underline{B}_i}\ \hat{\imath} \in I$$

$$\frac{\Gamma, \mathtt{x} : A \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \lambda \mathtt{x}.M : A \to \underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \quad \Gamma \vdash^{\mathsf{c}} M : A \to \underline{B}}{\Gamma \vdash^{\mathsf{c}} V`M : \underline{B}}$$

Fig. 1. Terms of Call-By-Push-Value

To avoid confusion between tags and identifiers, we adopt the convention that tags begin with #, and identifiers do not.

Computational Effects

CBPV can be extended with many different computational effects. We consider the example of printing, given by the typing rule

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{print}\ c.\ M : \underline{B}}$$

where c ranges over an alphabet $\mathcal{A}$.

3 Traditional Operational Semantics

We give operational semantics in two traditional styles, before moving on to the jumping semantics. The first is a first-order definitional interpreter [Rey72], that evaluates every closed computation to a *terminal* computation of the same type. The terminal computations are defined by

$$T ::= \quad \mathtt{return}\ V \ \mid\ \lambda\{\ldots, i.M_i, \ldots\} \ \mid\ \lambda \mathtt{x}.M$$

and the interpreter is shown in Fig. 2.

To evaluate

- $\lambda \mathtt{x}.M$, return $\lambda \mathtt{x}.M$
- $\mathtt{return}\ V$, return $\mathtt{return}\ V$
- $\lambda\{\ldots, i.M_i, \ldots\}$, return $\lambda\{\ldots, i.M_i, \ldots\}$
- $\mathtt{force}\ \mathtt{thunk}\ M$, evaluate M
- $M\ \mathtt{to}\ \mathtt{x}.\ N$, evaluate M, and if this returns $\mathtt{return}\ V$, then evaluate $N[V/\mathtt{x}]$
- V‘M, evaluate M, and if this returns $\lambda \mathtt{x}.N$, then evaluate $N[V/\mathtt{x}]$
- $\hat{\imath}$‘M, evaluate M, and if this returns $\lambda\{\ldots, i.M_i, \ldots\}$, then evaluate M_i
- $\mathtt{print}\ c.\ M$, print c and then evaluate M.

Fig. 2. First-Order Definitional Interpreter For CBPV

The other traditional style is the CK-machine [FF86], also based on [Rey72]. At any point in time, the machine has configuration M, K when M is the computation we are evaluating and K is a stack of contexts. In this stack, we abbreviate the context V‘$[\cdot]$ as V, and the context $\hat{\imath}$‘$[\cdot]$ as $\hat{\imath}$. The CK-machine is shown in Fig. 3.

The classification of $\lambda \mathtt{x}.M$ as a computation (and of function types as computation types) often surprises people familiar with call-by-value. But it makes sense when we look at the CK-machine. We see that

- V‘ can be regarded as an instruction “push V”
- $\lambda \mathtt{x}$ can be regarded as an instruction “pop $\mathtt{x}$”.

Initial Configuration

$$M \qquad \texttt{nil}$$

Transitions

$$\begin{array}{ll}
\texttt{let}\ V\ \texttt{be}\ \texttt{x}.\ M & K \\
\rightsquigarrow M[V/\texttt{x}] & K \\
\\
M\ \texttt{to}\ \texttt{x}.\ N & K \\
\rightsquigarrow M & [\cdot]\ \texttt{to}\ \texttt{x}.\ N :: K \\
\\
\texttt{return}\ V & [\cdot]\ \texttt{to}\ \texttt{x}.\ N :: K \\
\rightsquigarrow N[V/\texttt{x}] & K \\
\\
\texttt{force}\ \texttt{thunk}\ M & K \\
\rightsquigarrow M & K \\
\\
\texttt{pm}\ (\hat{\imath}, V)\ \texttt{as}\ \{\ldots, (i, \texttt{x}).M_i, \ldots\} & K \\
\rightsquigarrow M_i[V/\texttt{x}] & K \\
\\
\texttt{pm}\ (V, V')\ \texttt{as}\ (\texttt{x}, \texttt{y}).M & K \\
\rightsquigarrow M[V/\texttt{x}, V'/\texttt{y}] & K \\
\\
\hat{\imath}\text{`}M & K \\
\rightsquigarrow M & \hat{\imath} :: K \\
\\
\lambda\{\ldots, i.M_i, \ldots\} & \hat{\imath} :: K \\
\rightsquigarrow M_i & K \\
\\
V\text{`}M & K \\
\rightsquigarrow M & V :: K \\
\\
\lambda\texttt{x}.M & V :: K \\
\rightsquigarrow M[V/\texttt{x}] & K \\
\\
\texttt{print}\ c.\ M & K \\
\overset{c}{\rightsquigarrow} M & K
\end{array}$$

Terminal Configurations

$$\begin{array}{ll}
\texttt{return}\ V & \texttt{nil} \\
\lambda\{\ldots, i.M_i, \ldots\} & \texttt{nil} \\
\lambda\texttt{x}.M & \texttt{nil}
\end{array}$$

Fig. 3. CK-Machine For CBPV

This reading is made, in the call-by-name setting, in [Kri85]—see Sect. 7.

The contexts on the stack that are of the form $[\cdot]$ `to x`. M are called *frames*. In general, the stack will consist of frames, values and tags. For a call-by-value language, the stack would consist only of frames.

4 Jumping Semantics: An Informal Account

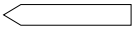
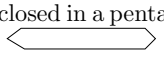
4.1 Requirements

Putting the ideas of Sect. 1.1 into a CBPV form, we require a jumping machine that embodies the following intuitions.

- A thunk is a point. When we force the thunk, we jump to it.
- A frame is a point. When we return a value to a frame, we pop the frame from the stack and jump to it.

4.2 Graphical Syntax

We write a program using a graphical syntax, depicted in Fig. 4, in which

- we write `thunk` as •, because it is a point
- each instruction, other than sequencing, is enclosed in a pentagon 
- each sequencing `to` is enclosed in a hexagon 
- binding occurrences of identifiers are placed on edges, enclosed in 

The *link-point* of a polygon is its leftmost vertex, which usually leads to the next instruction. In certain cases (e.g. conditional branching), there is more than one possibility, and we tag the edges accordingly. In other cases (e.g. jump), there are none. The *frame-point* of a hexagon is its rightmost vertex. We give the name *jumpabout* to this kind of tree of pentagons, hexagons, edges and points (again, this is informal at this stage).

4.3 Principles of Execution

During execution, there are two jumpabouts:

- the *code*, which does not change
- the *trace*, which grows throughout execution.

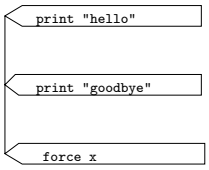
The cycle of execution can be described (in the von Neumann idiom) as "fetch, decode, execute".

fetch We copy a polygon, including its inscription, from the code to the trace.
decode We decode the inscription in the newly created trace polygon by
- replacing each • by `pt` i, where i is the position of the •
- replacing each identifier by the value it is bound to, determined by looking up the branch of the trace.

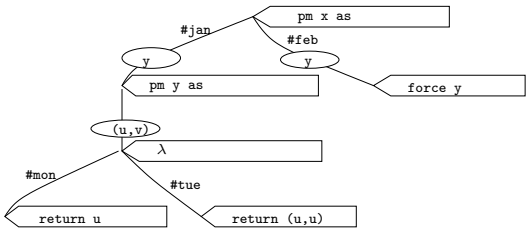
This gives us an instruction.

Term syntax	Graphical syntax

```
print "hello".
print "goodbye".
force x
```




```
pm x as {
      (#jan,y). (
            pm y as (u,v)
            λ {
                  #mon. return u
                  #tue. return (u,u)
            }
      )
      (#feb,y). force y
}
```




```
print "hello".
let thunk (
      λ x.
      return x.
) be u.
(
      (#jan,()) ‘
      force u
) to y.
return y
```

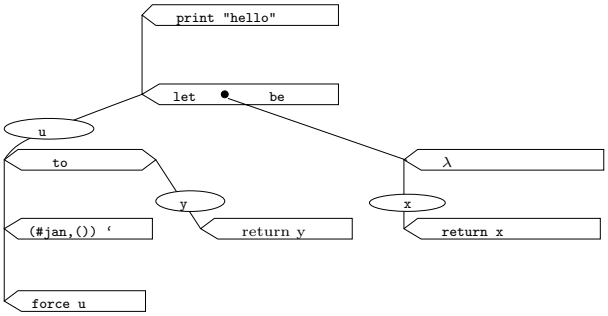


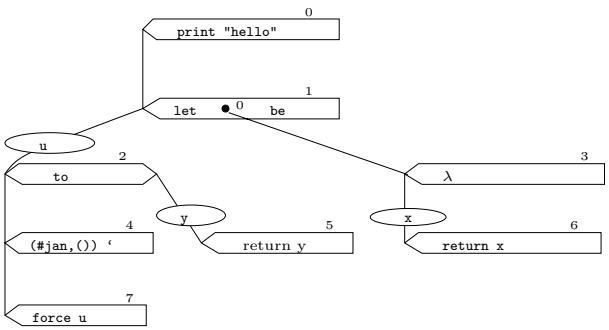
Fig. 4. Examples of Graphical Syntax

execute We execute the instruction. At the same time, we draw an edge from the link-point, unless the instruction is a jump i.e. `force` or `return`, in which case we draw an edge from the destination of the jump.

Every point in the trace has a *teacher*, which is the point in the code it was copied from; similarly for pentagons, hexagons and edges. The function mapping each point, polygon and edge to its teacher is a jumpabout homomorphism, and it grows as the trace jumpabout grows.

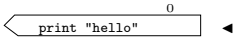
4.4 Example

To illustrate how this works, we take the last example from Fig. 4. For ease of reference, we have numbered all the polygons, and numbered all the points (though there is only one).



Initially, the code polygon is the root (numbered 0 in our example). As in the CK-machine, the stack is `nil`.

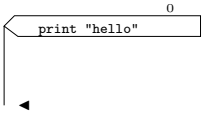
Cycle 0: fetch We copy code polygon 0 to the trace, so the trace looks like this:



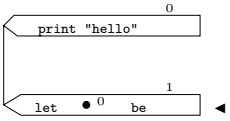
Thus the teacher of trace polygon 0 is code polygon 0. We use the symbol ◀ for “where we are now”.

Cycle 0: decode We obtain the instruction `print "hello"`.

Cycle 0: execute We print `hello`, and draw an edge from the link-point.



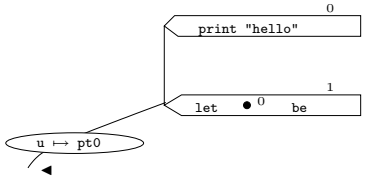
Cycle 1: fetch We copy code polygon 1 to the trace, which now looks like this:



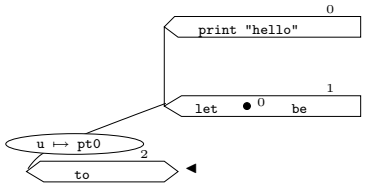
Thus teacher of trace polygon 1 is code polygon 1, and the teacher of trace point 0 is code point 0.

Cycle 1: decode To decode the inscription `let • be`, we replace • by `pt0`, and obtain the instruction `letpt0be`.

Cycle 1: execute We make a binding to `pt0`, on an edge drawn from the link-point.



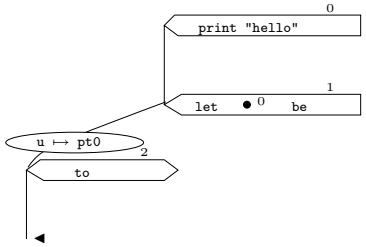
Cycle 2: fetch We copy code polygon 2 to the trace, so the trace looks like this:



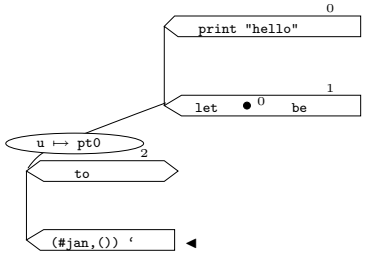
where the teacher of trace polygon 2 is code polygon 2.

Cycle 2: decode We obtain the instruction `to`.

Cycle 2: execute We place trace hexagon 2 on the stack, which becomes `hgon2 :: nil`, and draw an edge from the link-point.



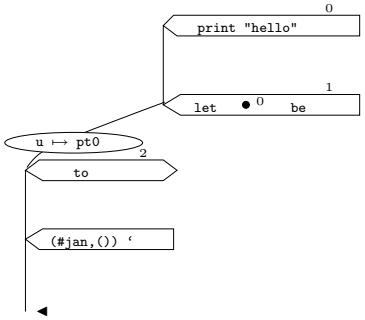
Cycle 3: fetch We copy code polygon 4 to the trace, so the trace looks like this:



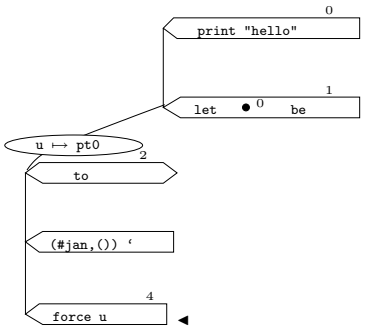
where the teacher of trace polygon 2 is code polygon 2

Cycle 3: decode We obtain the instruction `(#jan,()) ‘`.

Cycle 3: execute We push `(#jan,())`, making the stack `(#jan,()) ::hgon2 ::nil`, and draw an edge from the link-point.



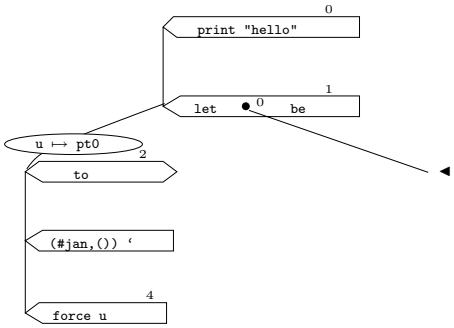
Cycle 4: fetch We copy code polygon 7 to the trace, which now looks like this:



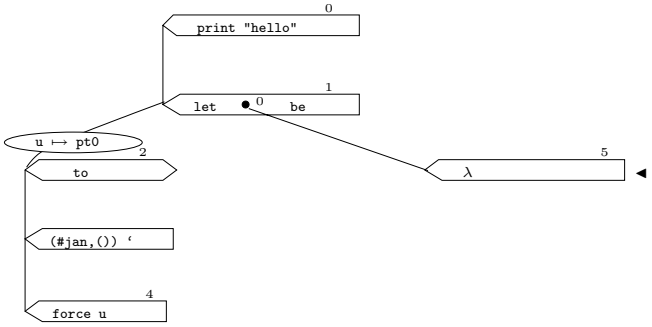
where the teacher of trace polygon 4 is trace polygon 7.

Cycle 4: decode To decode the inscription `force u`, we must replace `u` by its binding. Looking up the branch of the trace, we see that `u` is bound to `pt0`. So we obtain the instruction `force pt0`.

Cycle 4: execute We jump to trace point 0, and draw an edge from it:



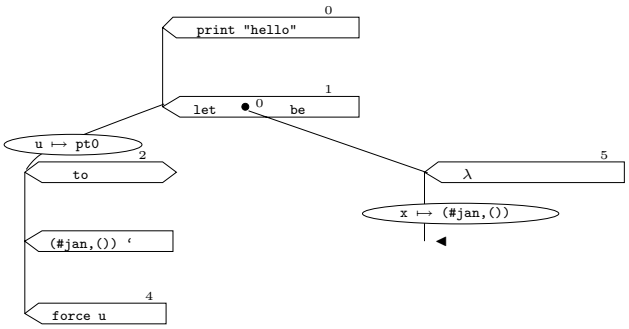
Cycle 5: fetch We copy code polygon 3 to the trace:



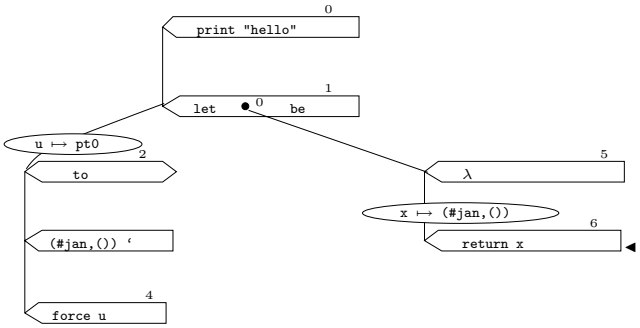
where the teacher of trace polygon 5 is code polygon 3.

Cycle 5: decode We obtain the instruction λ.

Cycle 5: execute We pop the value `(#jan,())` from the stack, which becomes `hgon2 ::nil`. We make a binding to this value on the edge drawn from the link-point:



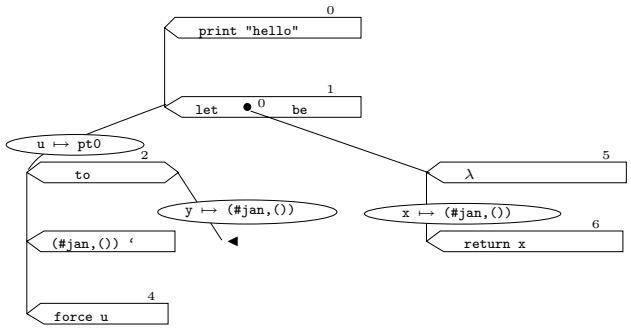
Cycle 6: fetch We copy code polygon 6 to the trace:



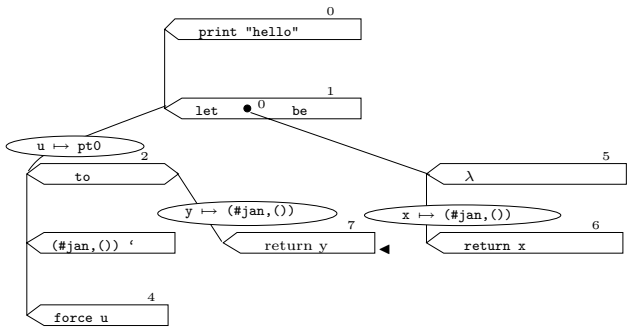
where the teacher of trace polygon 6 is code polygon 6.

Cycle 6: decode Replacing `x` by its binding, which is `(#jan,())`, we obtain the instruction `return (#jan,())`.

Cycle 6: execute We remove `hgon2` from the stack, which becomes `nil`, jump to the frame-point of hexagon 2, and draw an edge from it. We make a binding to `return (#jan,())` on this edge.



Cycle 7: fetch We copy code polygon 5 to the trace:



where the teacher of trace polygon 7 is code polygon 5.

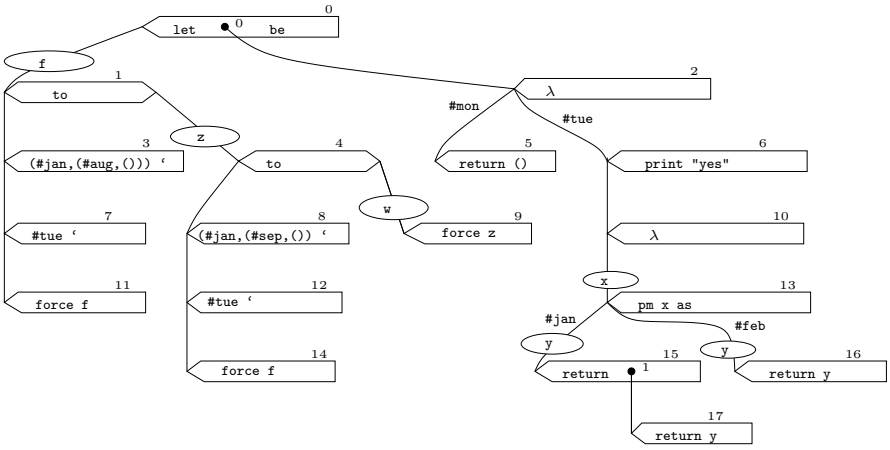
Cycle 7: decode Replacing `y` by its binding, which is `(#jan,())`, we obtain the instruction `return (#jan,())`.

Cycle 7: execute Since the stack is empty, we terminate.

The final instruction is thus `return (#jan,())`.

5 Exercise

The reader is invited to try executing the following example (21 cycles), which could be used to illustrate to students the concept of static binding.



This example makes it clear how easy it is to execute a program on paper using the jumping machine.

6 Correctness

There is a lock-step correspondence between the jumping machine and the CK-machine. More precisely, suppose we take a computation M, and create the trace using the jumping machine. Then to each trace polygon r we can associate a closed computation $\theta(r)$ of type $\underline{B}$. Similarly to each stack k that appears in the jumping execution, we can associate a stack $\theta(k)$ of the CK-machine. If the sequence of trace points and stacks is

$$(\text{polygon } r_0, \text{ stack } k_0), (\text{polygon } r_1, \text{ stack } k_1), \ldots$$

and the sequence of the CK-machine is

$$M, K = M_0, K_0 \rightsquigarrow M_1, K_1 \rightsquigarrow \cdots$$

then the two sequences have the same length and $\theta(r_i) = M_i$ and $\theta(k_i) = K_i$.

The computation $\theta(r)$ is obtained from the (decoded) instruction of r by substituting for points (including link-points and frame-points), and likewise the stack $\theta(k)$. For the example in Sect. 4.4, we therefore know not only that the CK-machine execution has 7 transitions, but also that it terminates in the configuration

$$\texttt{return}(\#\texttt{jan}, ()) \qquad\qquad \texttt{nil}$$

7 Comparison With CEK And Krivine Machine

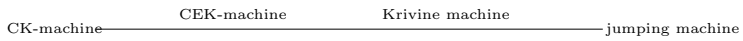
Both the Krivine machine [Kri85] and the CEK-machine [FF86] can be seen as lying on a spectrum between the CK-machine and the jumping machine. Alhough the Krivine machine was presented for CBN, and the CEK-machine for CBV, both styles of machine can be adapted for CBPV.

The intuition underlying the Krivine machine is described in [Kri85] as follows, slightly paraphrased:

- $\lambda \mathtt{x}.M$ means: pop $\mathtt{x}$, then do M
- MN [translated into CBPV as $(\mathtt{thunk}\ N)$‘M] means: push the address of N, then do M
- $\mathtt{x}$ [translated into CBPV as $\mathtt{force}\ \mathtt{x}$] means: go to the address that $\mathtt{x}$ is bound to.

The Krivine machine contains a “T” component, which points into the code. In our terminology, it is the teacher of the current polygon. So the jumping about the *code* is made clear. But the jumping about the *trace* is not apparent. Instead, the machine contains an “environment” component, which is changed with every jump.

The CEK machine is closer still to the CK-machine. Instead of the “T” component pointing into the code, it contains a “C” component which is the subterm itself. So there is no jumping at all, not even about the code. We can therefore think of these machines as lying on a spectrum:



References

[ABDM03] M. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proc., 5th ACM-SIGPLAN Int. Conf. on Principles and Practice of Declarative Programming*, 2003.

[DR99] V. Danos and L. Regnier. Reversible, irreversible and optimal λ-machines. *Theoretical Comp. Sci.*, 227(1–2), 1999.

[FF86] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Prog. Concepts*. North-Holland, 1986.

[Kri85] J.-L. Krivine. Un interpréteur de λ-calcul. Unpublished, 1985.

[Lev99] P. B. Levy. Call-by-push-value: a subsuming paradigm (extended abstract). In J.-Y Girard, editor, *Typed Lambda-Calculi and Applications*, volume 1581 of *LNCS*, 1999.

[Lev04] P. B. Levy. *Call-By-Push-Value*. Semantic Structures in Computation. Kluwer, 2004.

[Rey72] John Reynolds. Definitional interpreters for higher order programming languages. *ACM Conference Proceedings*, pages 717–740, 1972.