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

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

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` $\mathtt{x}.\ N$ 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 $\to$.

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

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

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

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

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

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

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

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

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

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

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

and the interpreter is shown in Fig. 2.

To evaluate

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

$$\begin{array}{lll} & M & \texttt{nil} \end{array}$$

Transitions

$$\begin{array}{lll}
 & \texttt{let}\ V\ \texttt{be}\ \texttt{x}.\ M & K \\
\rightsquigarrow & M[V/\texttt{x}] & K \\[1ex]
 & M\ \texttt{to}\ \texttt{x}.\ N & K \\
\rightsquigarrow & M & [\cdot]\ \texttt{to}\ \texttt{x}.\ N :: K \\[1ex]
 & \texttt{return}\ V & [\cdot]\ \texttt{to}\ \texttt{x}.\ N :: K \\
\rightsquigarrow & N[V/\texttt{x}] & K \\[1ex]
 & \texttt{force}\ \texttt{thunk}\ M & K \\
\rightsquigarrow & M & K \\[1ex]
 & \texttt{pm}\ (\hat{\imath}, V)\ \texttt{as}\ \{\ldots,(i,\texttt{x}).M_i,\ldots\} & K \\
\rightsquigarrow & M_{\hat{\imath}}[V/\texttt{x}] & K \\[1ex]
 & \texttt{pm}\ (V, V')\ \texttt{as}\ (\texttt{x},\texttt{y}).M & K \\
\rightsquigarrow & M[V/\texttt{x}, V'/\texttt{y}] & K \\[1ex]
 & \hat{\imath}\text{`}M & K \\
\rightsquigarrow & M & \hat{\imath} :: K \\[1ex]
 & \lambda\{\ldots, i.M_i, \ldots\} & \hat{\imath} :: K \\
\rightsquigarrow & M_{\hat{\imath}} & K \\[1ex]
 & V\text{`}M & K \\
\rightsquigarrow & M & V :: K \\[1ex]
 & \lambda\texttt{x}.M & V :: K \\
\rightsquigarrow & M[V/\texttt{x}] & K \\[1ex]
 & \texttt{print}\ c.\ M & K \\
\overset{c}{\rightsquigarrow} & M & K
\end{array}$$

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 $\bullet$, 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.

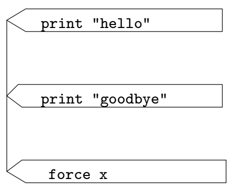
The cycle of execution can be described (in the von Neumann idiom) as "fetch, decode, execute".

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 $\bullet$ by `pt` i, where i is the position of the $\bullet$
  - replacing each identifier by the value it is bound to, determined by looking up the branch of the trace.

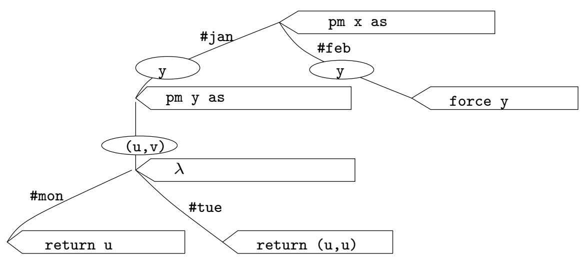
  This gives us an instruction.

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
```

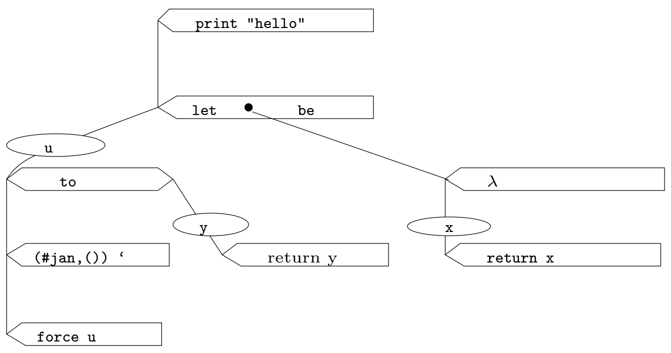


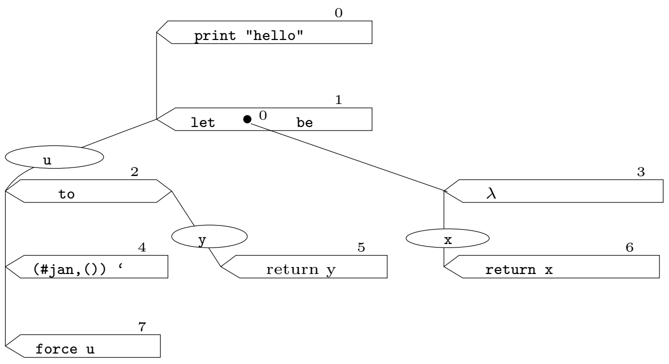
Fig. 4. Examples of Graphical Syntax

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.

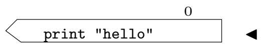
4.4 Example

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`.

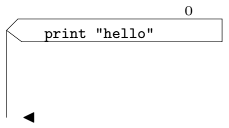
Cycle 0: fetch We copy code polygon 0 to the trace, so the trace looks like this:



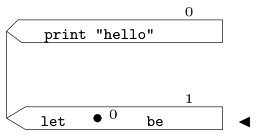
Thus the teacher of trace polygon 0 is code polygon 0. We use the symbol ◀ for "where we are now".

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

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



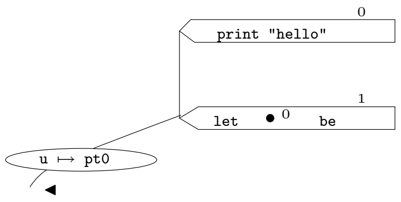
Cycle 1: fetch We copy code polygon 1 to the trace, which now looks like this:



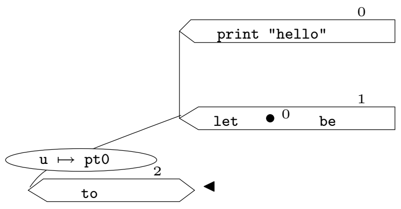
Thus teacher of trace polygon 1 is code polygon 1, and the teacher of trace point 0 is code point 0.

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.



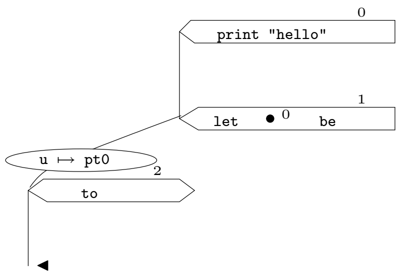
Cycle 2: fetch We copy code polygon 2 to the trace, so the trace looks like this:



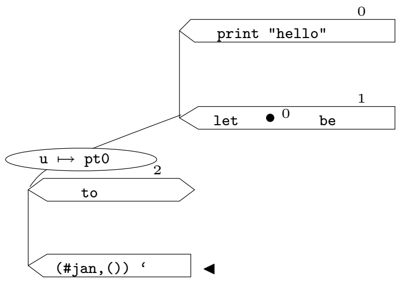
where the teacher of trace polygon 2 is code polygon 2.

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.



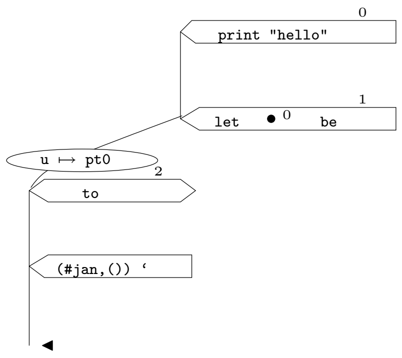
Cycle 3: fetch We copy code polygon 4 to the trace, so the trace looks like this:



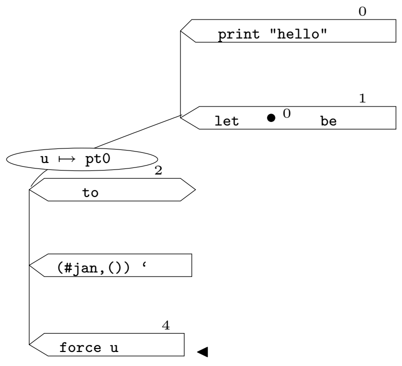
where the teacher of trace polygon 2 is code polygon 2

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.



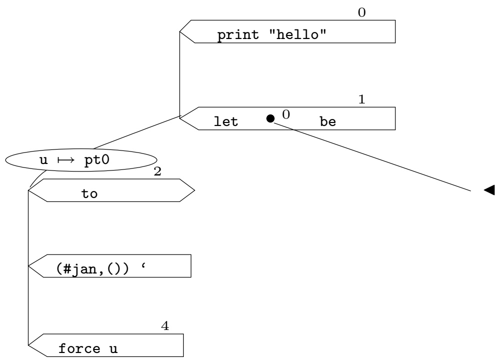
Cycle 4: fetch We copy code polygon 7 to the trace, which now looks like this:



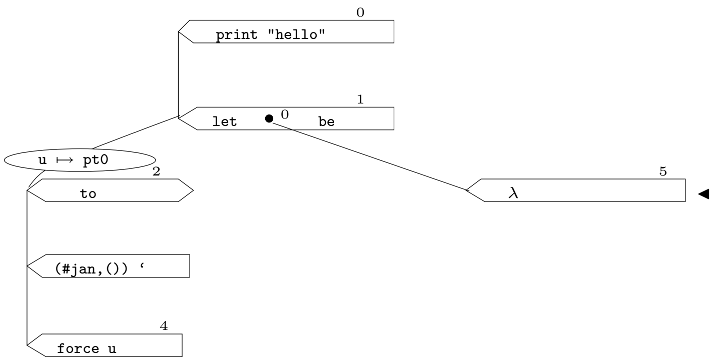
where the teacher of trace polygon 4 is trace polygon 7.

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:



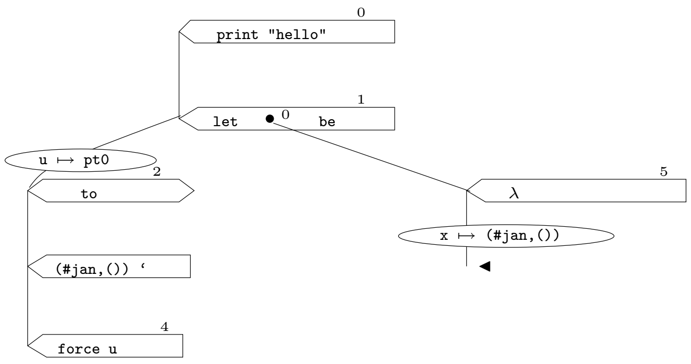
Cycle 5: fetch We copy code polygon 3 to the trace:



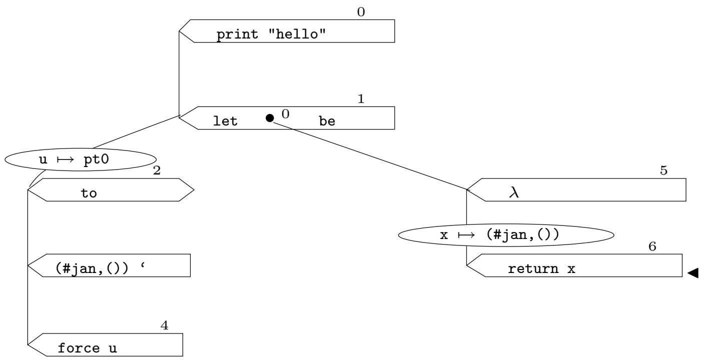
where the teacher of trace polygon 5 is code polygon 3.

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:



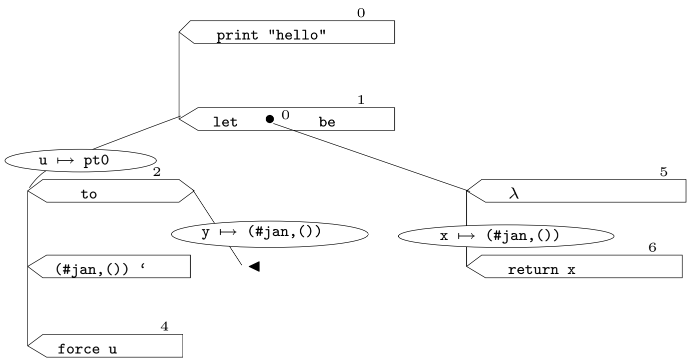
Cycle 6: fetch We copy code polygon 6 to the trace:



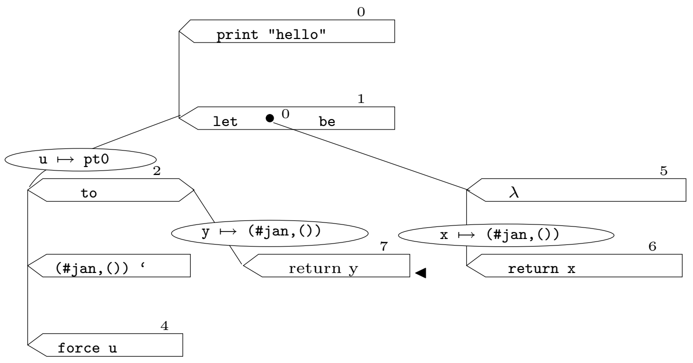
where the teacher of trace polygon 6 is code polygon 6.

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.

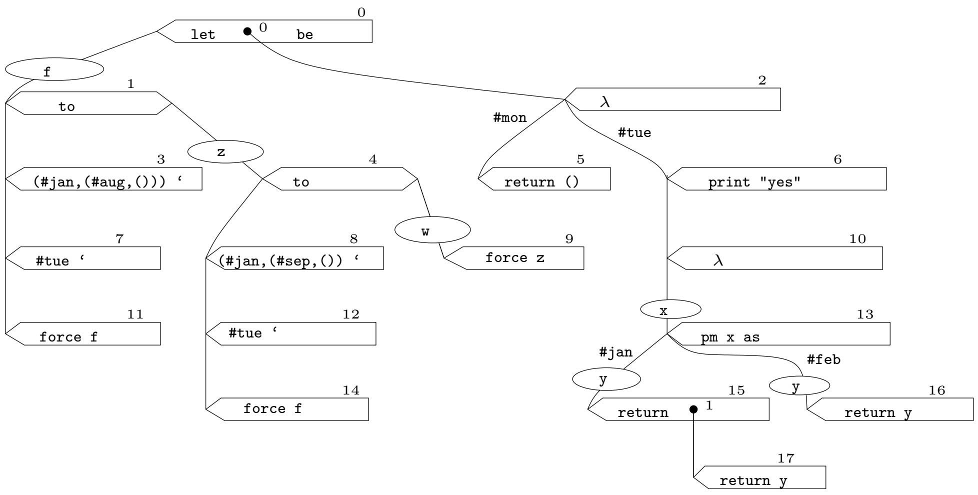
Cycle 7: decode Replacing `y` by its binding, which is `(#jan,())`, we obtain the instruction `return (#jan,())`.

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

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

and the sequence of the CK-machine is

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

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

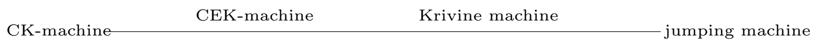
Both the Krivine machine [Kri85] and the CEK-machine [FF86] can be seen as lying on a spectrum between the CK-machine and the jumping machine. Alhough the Krivine machine was presented for CBN, and the CEK-machine for CBV, both styles of machine can be adapted for CBPV.

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

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