A call-by-name lambda-calculus machine

Jean-Louis Krivine
University 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 λ-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 It has been, since, used and implemented by several authors, but remained unpublished.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 λ-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 λ-calculus, which means that the active redex must be at the very beginning of the λ-term. Thus, computation stops if there is no redex at the head of the λ-term. In fact, we reduce at once a whole chain $\lambda x_1 \ldots \lambda x_n$. Therefore, execution also stops if there are not enough arguments.

The first example of a λ-calculus machine is P. Landin's celebrated SECD-machine [8]. 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 λ-calculus if the proof is intuitionistic and only uses logical axioms ;

in λ-calculus extended with a control instruction, if one uses the law of excluded middle [4] and the axioms of Zermelo-Frænkel set theory [6], which is most often the case.

Other instructions are necessary if one uses additional axioms, such as

the Axiom of Choice [7].

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 is, in itself, 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 $p \geq n$.

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 [6, 7] 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. [2]). 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 λ-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 *term area* where are written the λ-terms to be performed, the *stack* and the *heap*. We denote by $\&t$ the address of the term t in the term area.

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

— *environment* : a finite sequence $(e, \xi_1, \ldots, \xi_k)$ where e is the address of an environment (in the heap), and $\xi_1, \ldots, \xi_k$ 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 λ-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 λx with λ and each variable occurrence with an ordered pair of integers $<\nu, k>$ (it is a variant of the *de Bruijn* notation [3], see the definition below). We assume that t_0 is a closed term. Thus, the term area contains a sequence of closed terms.

Nevertheless, terms may contain symbols of constant, which are performed with some predefined programs. For example :

— a constant symbol which is the name of another closed term ; the program consists in the execution of this term.

— constant symbols for programs in 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 (which is not closed, in general) : 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 $\lambda x\, t$ 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 $\lambda x_1 \dots \lambda x_n\, t$ where t does not begin with a λ ; thus, T points to λx_1.
A new environment $(e, \xi_1, \dots, \xi_n)$ is created : e is the address of E, $\xi_1, \dots, \xi_n$ are "popped" : we take the n top entries off the stack. We put in E the address of this new environment in the heap, and we go on by performing t : thus T points now to t.

– Execution of x (a λ-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 $<\nu, k>$. If $\nu = 0$, the value we need is the k-th closure of the environment E. If $\nu \geq 1$, 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_ν. 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 $\lambda x, (, x$ of λ-calculus as elementary instructions :
• "λx" 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 ν, k for each occurrence of a variable x, i.e. how we "compile" a closed λ-term t. More generally, we compute ν for an occurrence of x in an arbitrary λ-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 $\nu = 0$. If $t = uv$, the occurrence of x we consider is in u (resp. v). We compute ν, and possibly k, in u (resp. v).
Let now $t = \lambda x_1 \dots \lambda x_n\, u$ with $n > 0$, u being a term which does not begin with a λ. If the occurrence of x we consider is free in t, we compute ν in t by computing ν in u, then adding 1. If this occurrence of x is bound in u, we compute ν and k in u. Finally, if this occurrence is free in u and bound in t, then we have $x = x_i$. We compute ν in u, and we set $k = i$.

2. Formal definitions and correction proof

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

- A constant a or an ordered pair $<\nu, k>$ ($k \geq 1$) of integers is a λ_B-term (*atomic* term).
- If t, u are λ_B-terms, then so is $(t)u$.
- If t is a λ_B-term which does not begin with λ^i and if $n \geq 1$, then $\lambda^n t$ is a λ_B-term.

Let us consider, in a λ_B-term t, an occurrence of a constant a or of $<\nu, k>$ (ordered pair of integers). We define, in an obvious way, the *depth* of this occurrence, which is the number of λ^n symbols above it. The definition is done by induction on the length of t :
If there is no λ^i 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 $t = \lambda^n u$, we compute the depth of this occurrence in the subterm u and we add 1 to it.

An occurrence of $<\nu, k>$ in t is said to be *free* (resp. *bound*) if its depth in t is $\leq \nu$ (resp. $> \nu$).
Of course, each occurrence of a constant a is free. Thus, we could write constants as ordered pairs $<\infty, k>$.
Consider a bounded occurrence of $<\nu, k>$ in a λ_B-term t ; then there is a unique λ^n in t which bounds this occurrence. If $k > n$, we say that this occurrence of $<\nu, k>$ in t is *dummy*. A λ_B-term without dummy bound occurrences will be called *good*. We can easily transform a λ_B-term into a good one : simply substitute each dummy occurrence with a (unique) new constant symbol d.

ALPHA-EQUIVALENCE

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

THEOREM 1. *A λ_B-term τ is good iff there exists a λ-term t such that $\tau = \mathsf{B}[t]$.*

We omit the easy proof.

□

The compiled form $\mathsf{B}[t]$ of a λ-term t is a variant of the de Bruijn notation for t. Its main property, expressed by theorem 2, is that it depends only on α-equivalence class of t. This property is not used in the following, but the simplicity of the proof below convinced me to give it here.

THEOREM 2. *Two closed λ-terms t, t' are α-equivalent (which we denote by $t \simeq_\alpha t'$) if and only if $\mathsf{B}(t) = \mathsf{B}(t')$.*

The proof is done by induction on t. The result is clear if $t = a$ or $t = uv$. So, we assume now that $t = \lambda x_1 \ldots \lambda x_n u$ where u does not begin with λ. If $t \simeq_\alpha t'$ or if $\mathsf{B}(t) = \mathsf{B}(t')$, then $t' = \lambda x'_1 \ldots \lambda x'_n u'$ where u' does not begin with λ. Let $a_1, \ldots, a_n$ be new constants ; then, by definition of α-equivalence [5], we have :
$t \simeq_\alpha t' \Leftrightarrow u[a_1/x_1, \ldots, a_n/x_n] \simeq_\alpha u'[a_1/x'_1, \ldots, a_n/x'_n]$.
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 λ^n and replacing $<\nu, i>$ with a_i for every occurrence of $<\nu, i>$ the depth of which is precisely equal to ν.
Therefore, we have $\mathsf{B}(u[a_1/x_1, \ldots, a_n/x_n]) = \mathsf{B}(u'[a_1/x'_1, \ldots, a_n/x'_n])$ $\Leftrightarrow \mathsf{B}(t) = \mathsf{B}(t')$ and finally $t \simeq_\alpha t' \Leftrightarrow \mathsf{B}(t) = \mathsf{B}(t')$.

□

WEAK HEAD REDUCTION

Consider a λ_B-term of the form $(\lambda^n t)u_1 \ldots u_p$ with $p \geq n$. Then, we can carry out a *weak head reduction step* : we get the λ_B-term $t'u_{n+1} \ldots u_p$ (or t', if $n = p$) ; the term t' is obtained by replacing in t each *free* occurrence of $<\nu, i>$ with :

$<\nu - 1, i>$ if ν is strictly greater than the depth of this occurrence ;

u_i (resp. d) if ν is equal to the depth of this occurrence and $i \leq n$ (resp. $i > n$) ; d is a fixed constant, which replaces dummy bound occurrences in $\lambda^n t$.

We write $t \succ u$ if u is obtained from t by a finite (possibly null) number of weak head reduction steps.

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

Closures, environments and stacks

We now define recursively *closures* and *environments* :
$\emptyset$ is an environment (the empty environment) ; if e is an environment and $\phi_1, \ldots, \phi_n$ are closures ($n \geq 0$), then the finite sequence $(e, \phi_1, \ldots, \phi_n)$ is an environment.
A closure is an ordered pair (t, e) composed with a λ_B-term t and an environment e.
A *stack* is a finite sequence $\pi = (\phi_1, \ldots, \phi_n)$ of closures.
We denote by $\phi.\pi$ the stack $(\phi, \phi_1, \ldots, \phi_n)$ obtained by "pushing" the closure ϕ on the top of the stack π.

Execution rules
A *state* of the machine is a triple (t, e, π) where t is a λ_B-term, e an environment and π a stack. We now give the execution rules, by which we pass from a state (t, e, π) to the next one (t', e', π') :

• If $t = (u)v$, then $t' = u$, $e' = e$ and $\pi' = (v, e).\pi$.

• If $t = \lambda^n u$, the length of the stack π must be $\geq n$, otherwise the machine stops. Thus, we have $\pi = \phi_1 \ldots \phi_n.\pi'$, which defines π'. We set $t' = u$ and $e' = (e, \phi_1, \ldots, \phi_n)$.

• If $t = <\nu, k>$: 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 $e_i = \emptyset$ for an $i \leq \nu$, then the machine stops.
Otherwise, we have $e_\nu = (e_{\nu+1}, (t_1, \varepsilon_1), \ldots, (t_p, \varepsilon_p))$.
If $k \leq p$, we set $t' = t_k$, $e' = \varepsilon_k$ and $\pi' = \pi$.
If $k > p$, then the machine stops.

The value of a closure
Given any closure $\phi = (t, e)$, we define a *closed* λ_B-term which is denoted by $\bar{\phi}$ or $t[e]$; it is defined by induction on the environment e as follows :

If $e = \emptyset$, we obtain $t[\emptyset]$ by replacing in t each *free* occurrence of $<\nu, i>$ (i.e. its depth is $\leq \nu$) with the constant d.

If $e = (\varepsilon, \phi_1, \ldots, \phi_n)$, we set $t[e] = u[\varepsilon]$ where u is the λ_B-term we obtain by replacing in t each *free* occurrence of $<\nu, i>$ with :
$<\nu - 1, i>$ if ν is strictly greater than the depth of this occurrence ;
$\bar{\phi}_i$ (resp. d) if ν is equal to the depth of this occurrence and $i \leq n$ (resp. $i > n$) ; d is a fixed constant.

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

THEOREM 3. *Let (t, e, π), (t', e', π') be two consecutive states of the machine, with $\pi = (\phi_1, \ldots, \phi_m)$ and $\pi' = (\phi'_1, \ldots, \phi'_{m'})$.*
Then, we have : $t[e]\bar{\phi}_1 \ldots \bar{\phi}_m \succ t'[e']\bar{\phi}'_1 \ldots \bar{\phi}'_{m'}$.

Recall that the symbol $\succ$ denotes the weak head reduction. We shall use the notation $t[e]\bar{\pi}$ for $t[e]\bar{\phi}_1 \ldots \bar{\phi}_m$ when π is the stack $(\phi_1, \ldots, \phi_m)$. 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 $\pi' = (v, e).\pi$. Therefore $t[e]\bar{\pi} = t'[e']\bar{\pi}'$.

• $t = <\nu, k>$: let $e_0 = e$ and e_{j+1} the environment which is the first element of e_j, if $e_j \neq \emptyset$. Then, by the reduction rules of the machine, we have $(t', e') = \psi_k$ where ψ_k is the k-th closure of the environment $e_\nu = (e_{\nu+1}, \psi_1, \ldots, \psi_p)$ (and k is necessarily $\leq p$). Now, by definition of $t[e]$, we have $t[e] = \psi_k$. Therefore, $t[e]\bar{\pi} = \bar{\psi}_k\bar{\pi} = t'[e']\bar{\pi}'$, since $\pi' = \pi$.

• $t = \lambda^n u$: then we have $n \leq m$ and
$t' = u$, $e' = (e, \phi_1, \ldots, \phi_n)$, $\pi' = (\phi_{n+1}, \ldots, \phi_m)$. We must show that $(\lambda^n u)[e]\bar{\phi}_1 \ldots \bar{\phi}_n \succ u[(e, \phi_1, \ldots, \phi_n)]$.
By the very definitions of the value of a closure and of the weak head reduction, we have $u[(e, \phi_1, \ldots, \phi_n)] = v[e]$, where v is obtained by one step of weak head reduction in $(\lambda^n u)\bar{\phi}_1 \ldots \bar{\phi}_n$. We denote this by $(\lambda^n u)\bar{\phi}_1 \ldots \bar{\phi}_n \succ_1 v$, and we now show that $(\lambda^n u)[e]\bar{\phi}_1 \ldots \bar{\phi}_n \succ_1 v[e]$ (which will give the result).
We obtain $v[e]$ by replacing, in u, the free occurrences of $<\nu, i>$
i) with $\bar{\phi}_i$ (or d if $i > n$) if $\nu =$ the depth of this occurrence ;
ii) with $<\nu - 1, i>$ if $\nu >$ the depth of this occurrence ; and after that, we replace this occurrence of $<\nu - 1, i>$ with the closed term given by the environment e.
Now, we obtain $(\lambda^n u)[e]$ (which is closed) by the substitution (ii) on the free occurrences of $<\nu, i>$ in u such that $\nu >$ the depth of this occurrence in u. Indeed, they are exactly the free occurrences in $\lambda^n u$.
Then, one step of weak head reduction on $(\lambda^n u)[e]\bar{\phi}_1 \ldots \bar{\phi}_n$ performs the substitution (i) on the occurrences of $<\nu, i>$ in u such that $\nu =$ the depth in u of this occurrence. This shows that this step of reduction gives $v[e]$.

$\square$

This theorem shows that the machine which has been described above computes correctly in the following sense : if $t \succ at_1 \ldots t_k$, where t is a closed λ-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 $t \succ a$, 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 $((A \rightarrow B) \rightarrow A) \rightarrow A$ (see [4]).

We insist on the fact that we use a *call-by-name* version of `call/cc`, which is rather unusual. The reason for this was explained in the introduction.

An interesting study of the connection of such a machine with context sensitive calculi (calculi of explicit substitutions) is done in [1].

As we did before, we give first an informal description of the machine, then mathematical definitions.

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 (in the heap) ; or the address $\&\gamma$ of a *continuation*.

- continuation : it is a sequence $\gamma = (\xi_1, \ldots, \xi_n)$ of closures (the same as a stack).

Execution of a term

The execution consists in constantly updating the *current closure* Ξ and the stack. There are now two possible forms for the current closure : $(\&\tau, e)$ (where τ is a term) or $\&\gamma$ (where γ is a continuation).

Consider the first case : $\Xi = (\&\tau, e)$. There are now four possibilities for the term τ : an application $(t)u$, an abstraction $\lambda x\, t$, a variable x or the constant `cc`. Nothing is changed during execution in the first two cases.

- Execution of x (λ-calculus variable).
  As before, we fetch the value of the variable x in the environment e, which gives a closure ξ which becomes the current closure Ξ. The stack does not change.

> – Execution of cc.
>
> We pop a closure ξ which becomes the current closure Ξ. We save the current stack in a continuation γ and we push the address of γ (this address is a closure) on the top of the stack.
>
> Therefore, the stack, which was of the form $(\xi, \xi_1, \ldots, \xi_n)$, has become $(\&\gamma, \xi_1, \ldots, \xi_n)$ with $\gamma = (\xi_1, \ldots, \xi_n)$.

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

Formal definitions

λ_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*) :

$\emptyset$ is an environnement (the empty environnement) ; if e is an environment and $\phi_1, \ldots, \phi_n$ are closures ($n \geq 0$), then the finite sequence $(e, \phi_1, \ldots, \phi_n)$ is an environnement.

A closure is either *a stack*, or an ordered pair (t, e) composed with a λ_B-term t and an environment e.

A *stack* (or *continuation*) is a finite sequence $\gamma = (\phi_1, \ldots, \phi_n)$ of closures. We denote by $\phi.\gamma$ the stack $(\phi, \phi_1, \ldots, \phi_n)$ which is obtained by "pushing" the closure ϕ on the top of the stack γ.

Execution rules

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

• If ϕ is a continuation (i.e. a stack), then ϕ' is the closure which is on the top of the stack π (if π is empty, the machine stops) and $\pi' = \phi$.

• Else, we have $\phi = (t, e)$ and there are four possibilities for the λ_B-term t :

> • If $t = (u)v$, then $\phi' = (u, e)$ and $\pi' = (v, e).\pi$.
>
> • If $t = \lambda^n u$, then the length of the stack π must be $\geq n$, otherwise the machine stops. Thus, we have $\pi = \phi_1 \ldots \phi_n.\pi'$, which defines π'. We set $\phi' = (u, e')$ with $e' = (e, \phi_1, \ldots, \phi_n)$.
>
> • If $t = <\nu, k>$: 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 $e_i = \emptyset$ for an $i \leq \nu$, then the machine stops. Else, we have $e_\nu = (e_{\nu+1}, \phi_1, \ldots, \phi_p)$.

If $k \leq p$, we set $\phi' = \phi_k$ and $\pi' = \pi$. If $k > p$, the machine stops.

- If $t = \mathsf{cc}$, then ϕ' is the closure which is on the top of the stack π (if π is empty, the machine stops). Thus, we have $\pi = \phi'.\rho$ where ρ is a stack. Therefore, ρ is also a closure, which we denote by ϕ_ρ. Then, we set $\pi' = \phi_\rho.\rho$.

References

1. M. Biernacka, O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. BRICS Report series, RS-05-38, 2005.
2. V. Danos, J.-L. Krivine. Disjunctive tautologies and synchronisation schemes. Computer Science Logic'00, Lecture Notes in Computer Science no. 1862, pp. 292-301, 2000.
3. 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, pp. 381-392, 1972.
4. 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.
5. J.-L. Krivine. Lambda-calculus, types and models. Ellis Horwood, 1993.
6. J.-L. Krivine. Typed λ-calculus in classical Zermelo-Frænkel set theory. Archiv for Mathematical Logic, 40, 3, pp. 189-205, 2001.
7. J.-L. Krivine. Dependent choice, 'quote' and the clock. Theoretical Computer Science, 308, pp. 259-276, 2003.
8. P. J. Landin. The mechanical evaluation of expressions. The Computer Journal, vol. 6, pp. 308-320, 1964.