Another Implementation Technique for Functional Programming Languages

Rabéa Boulifa[1] & Mohamed Mezghiche[2]

1: INRIA Sophia Antipolis, B.P. 93 06902
Sophia Antipolis Cedex France
Rabea.Boulifa@sophia.inria.fr
2: Université de Boumerdès, Algérie
mmez@wissal.dz

Abstract

We define an abstract machine called cβ-machine. This machine implements the cβ-reduction which is analogous in combinatory logic to β-reduction in λ-calculus. The advantages of cβ-machine are that it implements β-reduction, that it transforms programs even if data are incomplete, that it partially evaluates programs, and it provides, as results, residual programs. Furthermore, these residual programs can be executed as specialized programs.

Keywords: λ-calculus, combinatory logic, β-reduction, cβ-reduction, functional programming language, partial evaluation.

1. Introduction

In the last decade, the implementation of functional programming languages has been widely investigated [7, 10, 16]. These languages have a mathematical basis, the λ-calculus [2], which is a formal basis for their implementation. Some solutions, even efficient [1, 3, 15, 16, 19], do not implement the main operation of λ-calculus, namely *strong* reduction, owing to bound variables [12]. So other solutions have been proposed to implement this reduction, for example Crégut [8] and Grégoire [9]. They give machines derived from existing ones which implement *strong* reduction and are based on the notion of environments and closures.

Using a similar approach, this paper focuses on *strong* reduction. We give an abstract machine which is also an extension of Turner's machine [17]. But our reducer uses graph reduction rather than environments and closures.

Turner's solution consists of removing from expression all variables and replacing them by some operators of combinatory logic [18], called *combinators*. He proposes also weak-reduction to reduce these expressions. This approach, although efficient, does not implement faithfully the β-reduction. Mezghiche [13] gives another reduction, cβ-reduction, which extends the weak-reduction. He proved an equivalence with β-reduction.

The aim of this paper[1] is to develop an abstract machine, an implementation of cβ-reduction. This implementation, cβ-machine, is a new implementation technique for programming languages. The contributions of our work are twofold:

- We provide another technique to implement functional languages. The advantage of this technique is that it implements the β-reduction accurately. Furthermore it retains the simplicity of Turner's machine, since it is implemented by small number operators (combinators) and the computation based on graph takes advantage of sharing reductions.

- Our machine integrates partial evaluation [5, 4, 6], which can be very useful as a practical working tool for programmers. Indeed, the machine takes as an input an expression (a program) and data, and gives as an output a term or an expression (the result) even if one does not provide all of the intended arguments. The result, residual code from the source program, and can be exploited as a specialized program to be applied to the remaining input data.

The paper is organized as follows. In section 2 we recall the definition of λ-calculus and computations of this model. Section 3 gives the definition of combinators and an overview of Turner's machine. In section 4, we develop our implementation of cβ-reduction as an abstract machine. This machine is an extension of the Turner's one. Section 5 gives semantics of our machine, and an example of its behavior. Concluding remarks and future work are in section 6.

2. λ-calculus

λ-calculus is a formal mathematical system providing the meta-language for formal definitions. It has a good claim to be the prototype programming language.
The syntax of this language is given by

$$Term : a ::= x | \lambda x.a | a_1 a_2$$

[1] written since 1998

The main operation is β-reduction

$$(\lambda x.a)a' \rightarrow_\beta a\{x := a'\} \quad (\beta)$$

Computation rules of the reduction are

- *Weak* reduction, noted $\rightarrow_w$, is defined by

$$\frac{a \rightarrow_\beta a'}{a \rightarrow_w a'} \qquad\qquad \frac{a \rightarrow_w a'}{ab \rightarrow_w a'b}$$

  The terms in normal forms are of the form $\lambda x.a$.

- *Head* reduction, noted $\rightarrow_h$, is defined by

$$\frac{a \rightarrow_w a'}{\lambda x.a \rightarrow_h \lambda x.a'}$$

  The leftmost reductions are performed and closed head normal forms are of the form $\lambda x.xa$.

- *Strong* reduction, noted $\rightarrow_s$, is defined by

$$\frac{a \rightarrow_h a'}{a \rightarrow_s a'} \qquad\qquad \frac{b \rightarrow_s b'}{\lambda x.xab\ldots \rightarrow_s \lambda x.xab'}$$

  Strong reduction is described as a sequence of head reductions allowing reduction in arbitrary sub-terms.

To facilitate implementation of this language Turner proposes to represent elegantly and simply the λ-terms by some operators in which λ-bound variables are removed.

3. SK-machine

Turner in his seminal paper [17] showed how a functional languages with lazy evaluation could be implemented simply and efficiently using operators called *combinators* [11] : $\boldsymbol{S}$, $\boldsymbol{K}$, $\boldsymbol{I}$.
First, he defines a translation which takes a λ-expression and yields its combinatory code. This translation [18] is defined by a system of rewrite rules often called "abstraction algorithm" and noted $[\ ]_\beta$. This algorithm is defined by the following clauses:

$$\begin{array}{lll}
(a) & [x]_\beta\, x & \equiv \boldsymbol{I} \\
(b) & [x]_\beta\, U & \equiv \boldsymbol{K}U \text{ if } x \notin FV(U) \\
(c) & [x]_\beta\, UV & \equiv \boldsymbol{S}([x]_\beta\, U)([x]_\beta\, V) \text{ if the preceding rules don't apply}
\end{array}$$

$FV(U)$ is the set of free variables of U.

Example 1 *Given the λ-term $\lambda xy.y$, it is written by abstraction*

$$\begin{array}{lll}
[x]_\beta\,([y]_\beta y) & \equiv & [x]_\beta\, \boldsymbol{I} \quad \textit{by applying (a)} \\
 & \equiv & \boldsymbol{KI} \quad\;\; \textit{by applying (b)}
\end{array}$$

The calculus on combinatory code is defined by the following reduction rules:

$$\begin{array}{lcl} \boldsymbol{S}xyz & \rightarrow & xz(yz) \\ \boldsymbol{K}xy & \rightarrow & x \\ \boldsymbol{I}x & \rightarrow & x \end{array}$$

The abstraction algorithm tends to produce a fast code, Turner introduced in [18] some optimizations by adding to the precedent rules the following reduction rules:

$$\begin{array}{lcl} \boldsymbol{S}(\boldsymbol{K}U)(\boldsymbol{K}\ V) & \rightarrow & \boldsymbol{K}(UV) \\ \boldsymbol{S}(\boldsymbol{K}U)\boldsymbol{I} & \rightarrow & U \\ \boldsymbol{S}(\boldsymbol{K}U)V & \rightarrow & \boldsymbol{B}UV \text{ if no earlier rule applies} \\ \boldsymbol{S}U(\boldsymbol{K}V) & \rightarrow & \boldsymbol{C}UV \text{ if no earlier rule applies} \end{array}$$

These rules can be viewed as an optimisation performed after the abstraction algorithm.

Then the calculus is extended by:

$$\begin{array}{lcl} \boldsymbol{B}xyz & \rightarrow & x(yz) \\ \boldsymbol{C}xyz & \rightarrow & xzy \end{array}$$

Currently, we give an example to see how Turner evaluates an expression in his machine named $\boldsymbol{SK}$-machine.

Example 2 *Suppose the user types an expression to be evaluated:*

$$succ\ 2\ where\ succ\ x \ = \ 1 + x$$

which is equivalent in λ*-calculus to:*

$$(\lambda succ.(succ\ 2))(\lambda x.(+\ 1\ x))$$

First, the compiler removes all the variables by applying abstraction:

$$([succ]_\beta\,(succ\ 2))([x]_\beta\,(+1x)) \equiv \underbrace{(\boldsymbol{SI}(\boldsymbol{K}2))}_{(\boldsymbol{CI}2)}\underbrace{(\boldsymbol{S}(\boldsymbol{K}(+\ 1))\boldsymbol{I})}_{(+\ 1)}$$

Second, the optimisation produces the code:

$$\boldsymbol{CI}2(+\ 1)$$

The reduction machine transforms progressively this code by applying reduction rules:

$$\begin{array}{ll} \boldsymbol{I}(+\ 1)2 & \textit{using the } \boldsymbol{C}\textit{-rule} \\ +\ 1\ 2 & \textit{using the } \boldsymbol{I}\textit{-rule} \\ 3 & \textit{using the } \boldsymbol{addition}\ \textit{(arithmetic rule)} \end{array}$$

The number 3 is then printed as system response to user's request.

Example 3 *The evaluation of the code* $M \equiv \boldsymbol{SK}$ *by the* $\boldsymbol{SK}$*-machine returns an error, since* $\boldsymbol{S}$ *needs three arguments for its weak-reduction. However, in* λ*-calculus,s* $M \equiv (\lambda xyz.xz(yz))(\lambda xy.x)$ *is reduced to* $(\lambda xy.y)$ *which corresponds to* $\boldsymbol{KI}$.

Thus, we propose a machine which is able to perform this kind of reduction.

4. Cβ-machine

In [13] an extension of the weak combinatory reduction, cβ-reduction, is defined. This reduction in combinatory logic is showed analogous to the β-reduction in λ-calculus [14] and asserts the following properties:

- The cβ-reduction is an extension of the weak reduction;
- $\rightarrow_{c\beta}$ has the Church-Rosser properties;
- M is in cβ-normal form (irreducible) iff there exists an equivalent λ-term in normal form.

In order to implement the cβ-reduction an algorithm, noted $c\beta\_m()$, is defined as a set of simple rules:

$$
\begin{array}{lll l}
0.a & c\beta\_m(\boldsymbol{S}) & \rightarrow_{c\beta} & \boldsymbol{S} \\
0.b & c\beta\_m(\boldsymbol{K}) & \rightarrow_{c\beta} & \boldsymbol{K} \\
0.c & c\beta\_m(\boldsymbol{I}) & \rightarrow_{c\beta} & \boldsymbol{I} \\
0.d & c\beta\_m(x) & \rightarrow_{c\beta} & x \text{ for any variable } x \\
1.a & c\beta\_m(\boldsymbol{S}M_1M_2\ldots M_n) & \rightarrow_{c\beta} & c\beta\_m(M_1M_3(M_2M_3)M_4\ldots M_n) \text{ if } n \geq 3 \\
1.b & c\beta\_m(\boldsymbol{K}M_1M_2\ldots M_n) & \rightarrow_{c\beta} & c\beta\_m(M_1M_3M_4\ldots M_n) \text{ if } n \geq 2 \\
1.c & c\beta\_m(\boldsymbol{I}M_1M_2\ldots M_n) & \rightarrow_{c\beta} & c\beta\_m(M_1M_2\ldots M_n) \text{ if } n \geq 1 \\
2.a & c\beta\_m(\boldsymbol{K}M_1) & \rightarrow_{c\beta} & \lambda x.c\beta\_m(M_1) \text{ where } x \notin FV(M_1) \\
2.b & c\beta\_m(\boldsymbol{S}M_1M_2) & \rightarrow_{c\beta} & \lambda x.c\beta\_m(M_1x(M_2x)) \text{ where } x \notin FV(M_1M_2) \\
2.c & c\beta\_m(\boldsymbol{S}M_1) & \rightarrow_{c\beta} & \lambda xy.c\beta\_m(M_1y(xy)) \text{ where } x,y \notin FV(M_1) \\
2.d & c\beta\_m(M_1M_2\ldots M_n) & \rightarrow_{c\beta} & (c\beta\_m(M_1))\ldots(c\beta\_m(M_2))(c\beta\_m(M_n)) \\
 & & & \text{if } M_1 \neq \boldsymbol{S},\ \boldsymbol{K},\ \boldsymbol{I} \\
3.a & \lambda x.M & \rightarrow_{c\beta} & [x]_\beta\, M
\end{array}
$$

$[.]_\beta$ is the abstract algorithm used to transform a λ-term M to its combinatory form.

Instructions of type 1, when they occur, must be executed before instructions of type 2. When instructions of type 2 occur new variables are generated in order to continue the reduction. These extra-variables will be removed at the end of the reduction by abstraction operation. The instruction of type 3.a ends the process of reduction by giving a combinatory form of result.

The cβ-machine is an implementation of this algorithm. It takes as input any combinatory logic expression M and returns as output the cβ-normal form of M if it exists.

Before defining semantics of the machine and further details of its implementation, it may be helpful to follow its processing through a simple example.

Example 4 *Coming back to the example 2 and see how the cβ-reduction performs a reduction.*

$$
\begin{array}{lll l}
c\beta\_m(\boldsymbol{SK}) & \rightarrow & \lambda xy.c\beta\_m(\boldsymbol{K}y(xy)) & \textit{by instruction 2.c} \\
 & \rightarrow & \lambda xy.c\beta\_m(y) & \textit{by instruction 1.b} \\
 & \rightarrow & \lambda xy.y & \textit{by instruction 0.d} \\
 & \rightarrow & \lambda x.([y]_\beta\, y) & \textit{by instruction 3.a} \\
 & \rightarrow & [x]_\beta\,([y]_\beta\, y) & \textit{by instruction 3.a} \\
 & \rightarrow & \boldsymbol{KI} & \textit{by applying the abstraction algorithm}
\end{array}
$$

The result is the one expected, the term $\boldsymbol{KI}$ *is the cβ-normal form of the given term* $\boldsymbol{SK}$.

Thus the cβ-machine is an extension of Turner's $\boldsymbol{SK}$-machine since it performs the $\lambda\beta$-reduction. Now let us see through another example how the cβ-machine can be viewed as a partial evaluator.

Example 5 *Let us consider the function Power which, when it is applied to the inputs n and x, returns x^n.*

In λ-calculus Power corresponds to the λ-expression:

$$Power = \lambda nx.(cond(= \ n\ 0)1(*\ x\ Power(-\ n\ 1)x))$$

The function Power is a fixed point of the function:

$$Power' = \lambda nxf.(cond(eq\ n\ 0)1(times\ x\ f(minus\ n\ 1)x))$$

Using the abstraction algorithm we can put $Power'$ in a combinatory form:

$$Power' = (\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{S}(\boldsymbol{S}(\boldsymbol{KS})(\boldsymbol{S}(\boldsymbol{KK})(\boldsymbol{S}(\boldsymbol{S}(\boldsymbol{K}\ cond)(\boldsymbol{S}\ eq(\boldsymbol{K}\ 0)))(\boldsymbol{K}$$

$$1))))))(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{S}\ times))))(\boldsymbol{S}(\boldsymbol{S}(\boldsymbol{KS})(\boldsymbol{K}))(\boldsymbol{K}(\boldsymbol{S}\ minus(\boldsymbol{K}\ 1))))))$$

The functions cond, eq, minus and times which are respectively condition, equality, subtraction and multiplication are assumed to be in combinatory forms.
If we apply the Power to the input data 2 and perform the reductions, we then have the result $Power\ 2 \rightarrow \boldsymbol{S}(times)(\boldsymbol{S}\ times(\boldsymbol{K}\ 1))$.
The code $\boldsymbol{S}\ (times)(\boldsymbol{S}\ times(\boldsymbol{K}\ 1))$ can be considered as the specialized program for computing the square of a data. It is straightforward to check that for example $\boldsymbol{S}\ (times)(\boldsymbol{S}\ times(\boldsymbol{K}\ 1))\ 5 \rightarrow 25$.

5. Cβ-machine specification

Like Turner we use binary trees to represent expressions to be evaluated. Each node of a tree contains two fields: type field and value field. The type is from the set {AP , FUNC, COMB, DATA, VAR} indicates whether the node is an application, function, combinator, data, or variable (generated variable) respectively. The values are either a pointer on node or a constant.

The cβ-machine is defined by a state transition function, noted $\Rightarrow$.

Definition 1 *A state cβ-machine is a tuple:*

$$M =< S_1, N_m, S_2 >$$

where

- *S_1 is stack of nodes ordered by the analysis;*
- *N_m is the node which is currently analysed;*
- *S_2 is stack of variables generated.*

Notations:
The empty stack is denoted by [ ], and non-empty stacks are denoted by $x :: S$ where x is topmost element of stack.
The node n is noted $[n := \text{AP}\ n_1\ n_2]$, an application of n_1 to n_2 otherwise it is noted $[n := \text{Type}\ Value]$ where $\text{Type} \in$ {FUNC, COMB, DATA, VAR}.

Before, the presentation of a formal specification of our abstract machine, it is useful to define following functions.

Definition 2 *Stack abstraction, node creation, and variable generation functions*

- $abstract^*(U,s) = \begin{cases} U & if\ s = [\,] \\ abstract^*([x]_\beta U, s') & if\ s = x :: s' \end{cases}$

  where U is an expression and the s is stack of variables.

- *node(var) creates and returns a node $[n := \text{VAR}\ var]$;*

- *genvar() automatically generates variables. The first value returned by this function, at the start of the reduction for each expression, is* 0 *and it increments automatically.*

Initially, the stack S_1 contains a pointer to the expression to be evaluated, the root of the tree, and the stack S_2 is empty.

The algorithm:

1. If $S_1 = [\,]$ then we have the following cases:

   (a) $S_2 = [\,]$, no extra-variables are generated, then current node value is returned as result;

   $$< [\,], [n := \text{TYPE}\ Result], [\,] > \Rightarrow Result$$

   (b) $S_2 \neq [\,]$, then the variables pushed onto S_2 must be removed by applying the *abstract* function before the stop of the machine;

   $$< [\,], [n := \text{TYPE}\ Result], S_2 > \Rightarrow abstract^*(Result, S_2)$$

   where TYPE $\in$ {AP, FUNC, COMB, DATA, VAR}

2. If $S_1 \neq [\,]$ and the type of node, n, currently analysed is:

   (a) an **application** (a pointer to) node, then this node is pushed onto $S1$;

   $$< S_1, [n := \text{AP}\ n_1\ n_2], S_2 > \Rightarrow < n_2 :: S1, [n := n_1], S_2 >$$

   (b) a **function**, $func\_k$, of k arguments. So k topmost operands are popped from S_1, the value $func\_k(n_1, \ldots, n_k)$ is pushed onto S_1 and execution continues;

   $$< n_1 :: n_2 : \ldots :: n_k :: S_1', [n := \text{FUNC}\ func\_k], S_2 > \Rightarrow \\ < S_1', [n := func\_k(n_1, n_2, \ldots, n_k)], S_2 >$$

   (c) a **combinator** $comb\_k$ needing k operands. Two sub-cases must be considered:

      i. There are n operands on S_1, where $n \geq k$;

      $$< n_1 : n_2 : \ldots : n_k :: S_1', [n := \text{COMB}\ comb\_k], S_2 > \Rightarrow \\ < S_1', [n := comb\_k(n_1, n_2, \ldots, n_k)], S_2 >$$

      ii. Else, there are missing arguments. So variables are generated and new nodes are created to carry out the reduction according to combinator;

      - $< n_1 :: [\,], [n := \text{COMB}\ \boldsymbol{K}\ ], S_2 > \Rightarrow < [\,], [n := n_1], var :: S_2 >$

      - $< n_1 :: n_2 :: [\,], [n := \text{COMB}\ \boldsymbol{S}\ ], S_2 > \Rightarrow \\ < [\,], [n := \text{AP}(\text{AP}\ n_1\ n_3)(\text{AP}\ n_2\ n_3)], var :: S_2 >$

- $< n_1 :: [\,] :, [n := \text{COMB } \boldsymbol{S}\ ], S_2 >\Rightarrow$
  $< [\,], [n := (\text{AP}(\text{AP } (\text{COMB } \boldsymbol{S}\ )\ n_1)\ n_3], var :: S_2 >$

  where $var = genvar()$ *and* $n_3 = node\ (var)$;

(d) a **data** error message is send back (can't apply a data to others).

(e) a **variable** (generated variables) case never occurs.

The whole specification of the $c\beta$-machine have been given, see figure 2.

Now, we illustrate the working of this one by the reduction of the expression $M \equiv \boldsymbol{SK}$ given above.

At the start the state of the machine is

$$< (\text{AP } n_1 = (\text{COMB } \boldsymbol{S}\ )\ n_2 = (\text{COMB } \boldsymbol{K}\ )) :: [\,], [n :=?], [\,] >$$

For notation convenience we omit the type of node except for the type **VAR**. So, this state is represented by

$$< \boldsymbol{SK} :: [\,], [n :=?], [\,] >$$

As long as there are reduction rules to apply the machine continues working by running the algorithm; the complete execution is given in figure 1.

$\Rightarrow< \boldsymbol{SK} :: [\,],$	$[n :=?],$	$[\,]>$
$\Rightarrow< [\,],$	$[n := \boldsymbol{SK}],$	$[\,]>$
$\Rightarrow< \boldsymbol{K} :: [\,],$	$[n := \boldsymbol{S}],$	$[\,]>$
$\Rightarrow< [\,],$	$[n := (\boldsymbol{SK})\ (\text{VAR } 0)],$	$0 :: [\,]>$
where $(\text{VAR } 0 = node(\text{VAR } genvar())$		
$\Rightarrow< (\text{VAR } 0) :: [\,],$	$[n := \boldsymbol{SK}],$	$0 :: [\,]>$
$\Rightarrow< \boldsymbol{K} :: (\text{VAR } 0) :: [\,],$	$[n := \boldsymbol{S}],$	$0 :: [\,]>$
$\Rightarrow< [\,],$	$[n := (\boldsymbol{K}(\text{VAR } 1))(\text{VAR } 0)(\text{VAR } 1))],$	$1 :: 0 :: [\,]>$
where $(\text{VAR } 1) = node(\text{VAR } genvar())$		
$\Rightarrow< (\text{VAR } 0)(\text{VAR } 1) :: [\,],$	$[n := \boldsymbol{K}(\text{VAR } 1)],$	$1 :: 0 :: [\,]>$
$\Rightarrow< (\text{VAR } 1) :: (\text{VAR } 0)(\text{VAR } 1) :: [\,],$	$[n := \boldsymbol{K}],$	$1 :: 0 :: [\,]>$
$\Rightarrow< [\,],$	$[n := (\text{VAR } 1)],$	$1 :: 0 :: [\,]>$
$\Rightarrow$	$abstract^*(1, 1 :: 0 :: [\,]) = \boldsymbol{KI}$	

Figure 1: An execution of the machine

At the end the stacks S_1 and S_2 are empty and the result is printed. We now obtain :

$$\boldsymbol{SK} \ \rightarrow_{c\beta} \boldsymbol{KI}$$

6. Conclusion

A formal abstract specification of the $c\beta$-machine is given: a machine for the implementation of functional programming languages. It is a good prototype/model to construct a concrete machine.

Indeed, it is based on few operators, performs faithfully $\lambda\beta$-reduction, and integrates partial evaluation. Many optimizations, however, must be supplied to produce efficient code. In future work, we expect to write the correctness proof for this machine, so we can prove the correctness of applications running over it.

Bibliographie

[1] M. Abadi, L. Cardelli, P.L. Curien, and J.J. Lévy. *Explicit substitutions.* In Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, page 31-46, San Francisco, California, January 1990.

[2] Barendregt. *The lambda calculus: Its syntax and its semantics.* North Holland Publishing Co, 1984.

[3] T.J.W. Clarke, P.J.S. Glastone, C.D. Maclean, and A.C. Norman. *SKIM - The S, K, I Reduction Machine.* In Proceedings LISP-80 Conference, 1980.

[4] C. Consel and O. Danvy. *Partial evaluation of pattern matching in strings.* Information Processing Letters, 1989.

[5] C. Consel and O. Danvy.*Tutorial note on partial evaluation.* In Conference Record of the Twentieeth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, 1993.

[6] N.D. Jones, C. Gomard, P.Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[7] P.L. Curien. *Categorical combinators, sequential algorithms and functional programming.* Pitman/Wiley, 1986.

[8] P. Crégut. *An abstract machine for lambda-terms normalization.* In Lisp and Functional Programming 1990, pages 333-340. ACM Press, 1990.

[9] B. Grégoire, X. Leroy. *A Compiled Implementation of strong reduction.* ICFP'02, October 2002, Pittsburgh, USA.

[10] P. G Harrison and J.F. Anthony. *Functional programming.* Addison-Wesley Publishing Co, 1989.

[11] R. Hindley and J. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[12] P.J. Landin. *The mechanical evaluation of expressions.* Computer Journal, 1964.

[13] M. Mezghiche. *Une nouvelle $c\beta$-reduction dans la logique combinatoire.* TCS 31, 1984.

[14] M. Mezghiche. *$C\beta$-machine and $\lambda\beta$-reduction.* TCS , 1997.

[15] S.L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall International (UK), 1987.

[16] S.L. Peyton Jones. *Mise en œuvre des langages fonctionnels de programmation.* Prentice Hall, 1990.

[17] D.A. Turner. *A new implementation technique for applicative languages.* Software Practice and Experience, 1979.

[18] D.A. Turner. *Another algorithm for bracket abstraction.* Journal Symbolic Logic, June 1979.

[19] C.P. Wadsworth. *Semantics and pragmatics of lambda calculus.* PhD thesis, University of Oxford, 1981.

type $Comb == \{\boldsymbol{S},\ \boldsymbol{K},\ \boldsymbol{I}\}$
type $Func == ADD,\ SUB,\ MULT, \ldots$
type $Data == INTEGER,\ REEL,\ STRING, \ldots$
type $TYPE == AP,\ FUNC,\ COMB,\ DATA,\ or\ VAR;$
type $exp == (exp, exp),\ Func,\ Comb,\ Data,\ or\ Int$
type $Var == INTEGER$
type $N == (TYPE, exp)$
type $S_1 == list(N)$
type $S_2 == list(Var)$
type $State == < S_1,\ N,\ S_2 >$

dec $abstract^* :\ exp,\ list(Var) \rightarrow exp$
$genvar() :\ \ \rightarrow INTEGER$
$node :\ TYPE, Var \rightarrow N$

dec $\Rightarrow : State \rightarrow State$
$---$ $< [\ ], [n := \textbf{TYPE}\ Result], S_2 >$
$\Rightarrow abstract^*(Result, S_2)$
$---$ $< S_1, [n := \textbf{AP}\ n_1\ n_2], S_2 >$
$\Rightarrow < n_2 :: S_1, [n := n_1], S_2 >$
$---$ $< n_1 :: n_2 :: \ldots : n_k :: S_1, [n := \textbf{FUNC}\ func\_k], S_2 >$
$\Rightarrow < S_1, [n := func\_k(n_1, n_2, \ldots, n_k)], S_2 >$
$---$ $< n_1 :: n_2 :: n_3 :: S_1, [n := \textbf{COMB}\ \boldsymbol{S}\ ], S_2 >$
$\Rightarrow < S_1, [n := (\textbf{AP}(\textbf{AP}\ n_1\ n_3)(\textbf{AP}\ n_2\ n_3))], S_2 >$
$---$ $< n_1 :: n_2 :: S_1, [n := \textbf{COMB}\ \boldsymbol{K}\ ], S_2 >$
$\Rightarrow < S_1, [n := n_1], S_2 >$
$---$ $< n_1 :: S_1, [n := \textbf{COMB}\ \boldsymbol{I}\ ], S_2 >$
$\Rightarrow < S_1, [n := n_1], S_2 >$
$---$ $< n_1 :: [\ ], [n := \textbf{COMB}\ \boldsymbol{K}\ ], S_2 >$
$\Rightarrow < [\ ], [n := n_1], Var :: S_2 >$
$Var = genvar()$
$---$ $< n_1 :: n_2 :: [\ ], [n := \textbf{COMB}\ \boldsymbol{S}\ ], S_2 >$
$\Rightarrow < [\ ], [n := (\textbf{AP}(\textbf{AP}\ n_1\ n_3)(\textbf{AP}\ n_2\ n_3))], Var :: S_2 >$
$Var = genvar()\ and\ n_3 = node(Var)$
$---$ $< n_1 :: [\ ] :, [n := \textbf{COMB}\ \boldsymbol{S}\ ], S_2 >$
$\Rightarrow < [\ ], [n := (\textbf{AP}(\textbf{AP}\ (\textbf{COMB}\ \boldsymbol{S}\ )\ n_1)\ node(Var))], Var :: S_2 >$
$Var = genvar()$

Figure 2: Whole specification of cβ_machine