Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 286 (2012) 337–350

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Continuation Models for the Lambda Calculus With Constructors

Barbara Petit[1]

Focus - INRIA
University of Bologna
(Italy)

Abstract

The lambda calculus with constructors decomposes the pattern matching *à la ML* into some atomic rules. Some of them do not match with the usual computational intuitions (in particular with typing intuitions). However it is possible to define an abstract notion of model for the untyped calculus, that has a trivial syntactic instance.
Nevertheless, the question of devising a non-syntactic model for this calculus was still unresolved. In this paper we answer this question in the untyped setting, by going back to the first motivation of the lambda-calculus with constructors: the simulation of an abstract machine with two independent stacks. This provides immediately a CPS translation into the usual lambda calculus. At the semantic level, it appears that this translation transforms any *continuation model* of the untyped lambda calculus into a model of the lambda calculus with constructors. In particular, any Scott domain can be turned into such a model.

Keywords: Lambda calculus, Pattern matching, *Continuation Passing Style* transformation, Categorical semantics, Continuation model.

Introduction

Pattern matching is a key feature in modern functional programming languages (Haskell, Ocaml) and proof assistants (Agda, Coq, Twelf). Since the late 90's, many formalisms have been proposed to integrate it with lambda calculus [3,9,1,2]. The syntactic properties of these calculi have been thoroughly studied, in both typed and untyped settings, and this led Jay to implement a programming language centred on pattern matching [8].

A more abstract approach to these formalisms could allow a deeper understanding of them, and possibly a comparison between them. As far as we know, no (non syntactical) denotational model has been defined for any of these calculi.

Owing to its simple syntax, the *lambda calculus with constructors* (or $\lambda_{\mathscr{C}}$-calculus) may be the best one to start with. Indeed, whereas most calculi with

[1] Email:barbara.petit@ens-lyon.org

pattern matching require the definition of a powerful operation of pattern substitution, the operational semantics of the $\lambda_{\mathscr{C}}$-calculus is composed of atomic rules: the pattern matching *à la* ML is decomposed into a simple analysis on constants (like the `case` instruction of Pascal), and a commutation rule between the case construction and the application (Sec. 1.2). Although this last rule is rather counter intuitive at first sight (it was presented as "ill-typed" in the introducing paper), the calculus is confluent and enjoys the *separation property* (in the spirit of Böhm's theorem), and a type system has also been defined for it [13].

A naive definition of a model can be given in category theory for the untyped lambda calculus with constructors [12]. However it seems difficult to build non syntactic instantiations of this definition. This sends us back to one of the main challenges of theoretical computer science in the late 60's: to build a denotational model for the pure lambda calculus (*i.e.* a mathematical structure with a reflexive object $D \cong D^D$). This problem was solved by Scott [16] in 1970, with the construction of a so-called D_∞ domain. It appeared later that such domains are in fact *continuation models* (characterised by two objects R and C such that $C \cong R^C \times C$) of the pure lambda calculus [15].

The idea underlying these continuation models is to use a CPS translation of the pure lambda calculus into the simply typed lambda calculus with only two basic types (one for the continuations, and one for the responses), and then to use the standard interpretation of the simply typed lambda calculus in a Cartesian closed category. We use the same method in this paper: we define a CPS transformation of the lambda calculus with constructors into the lambda calculus, and then interpret the translated terms in a CCC (with some required isomorphisms). The main difficulty is to interpret the pattern analysers (called *case bindings*). Indeed, to keep the definition of the models conceptually simpler, we use an operation of composition of case bindings, that has a non trivial translation in the CPS. However, the translation is correct, and provides as expected a sound definition of *continuation models* for the lambda calculus with constructors.

Outline: In the first section, we give an intuitive presentation of the lambda calculus with constructors, by defining an abstract machine for it. A CPS translation naturally results from this machine; we formalise it in Sec. 2. In the last section, we give the categorical definition of models (Sec. 3.1), and of *continuation models* (Sec. 3.2) for the lambda calculus with constructors, and we show that the second ones form a subclass of the first ones (Sec. 3.3). Finally we show that good candidates for continuation models of the $\lambda_{\mathscr{C}}$-calculus already exist (like Scott's domains for instance).

1 Lambda calculus with constructors

1.1 First approach: a two stack abstract machine

We extend the syntax of the lambda calculus with a finite set $\mathscr{C}$ of *constructors* (c, d *etc.*) and a *case construct* $\{\!|\theta|\!\} \cdot t$, where t is a term and θ a *case binding*, *i.e.* a partial function from constructors to terms:

$$\theta := \{\mathsf{c}_1 \mapsto t_1; \cdots ; \mathsf{c}_k \mapsto t_k\}$$

The *domain* $\{\mathsf{c}_1; \cdots ; \mathsf{c}_k\}$ of this case binding is denoted by $\mathrm{dom}(\theta)$, and θ_{c_i} represents the term t_i. Pattern matching occurs when such a case binding is associated to a constructor of its domain, just like a case analysis on constants:

$$\{\!|\theta|\!\} \cdot \mathsf{c} \;\rightarrow\; t \qquad \text{if} \quad \mathsf{c} \mapsto t \in \theta$$

The conditional branching, testing a Boolean and returning t or u if it is true or false respectively (where true and false are constructors), is then written

$$\mathtt{if}_{t,u} = \lambda x.\{\!|\mathsf{true} \mapsto t; \mathsf{false} \mapsto u|\!\} \cdot x.$$

In this language, there are now two different kinds of values: the functions (the usual λ-abstractions) and the constructors[2]. Each of them can be evaluated by the corresponding construction in a context: the argument of an application and the case binding of a case construct respectively. Also we can extend the Krivine abstract machine [5] to this syntax, by replacing the stack of arguments originally composing the evaluation context by two stacks: one (say the "right stack") for the arguments, and the other one (the "left stack") for the case bindings. When a term is evaluated in this machine, it then interacts with the left stack if it is a case construct or a case binding, and with the right stack if it is an application or a λ-abstraction. This machine is formally defined in Fig. 1. Evaluating the term $(\mathtt{if}_{t,u}\ \mathsf{false})$ in this machine (starting with two empty stacks) will indeed lead to the configuration $\diamond \;\star\; u \;\star\; \diamond$. But this machine can also simulate the pattern matching on compound data structures.

$$\begin{array}{lrl}
\text{Terms:} & t, u := & x \mid tu \mid \lambda x.t \mid \mathsf{c} \mid \{\!|\theta|\!\} \cdot t \\
\text{Case bindings:} & \theta, \phi := & \{\mathsf{c}_1 \mapsto t_1; \cdots ; \mathsf{c}_k \mapsto t_k\} \qquad (k \geq 0) \\
\text{Application stacks:} & \pi := & \diamond \mid t \cdot \pi \\
\text{Case stacks:} & \tau := & \diamond \mid \tau \cdot \theta \\
\text{Processes:} & s := & \tau \star t \star \pi
\end{array}$$

Execution rules:

$$\begin{array}{ccccccccccc}
\tau & \star & \lambda x.t & \star & u \cdot \pi & \blacktriangleright & \tau & \star & t[x := u] & \star & \pi & (\mathtt{Pop}) \\
\tau & \star & tu & \star & \pi & \blacktriangleright & \tau & \star & t & \star & u \cdot \pi & (\mathtt{Push}) \\
\tau \cdot \theta & \star & \mathsf{c} & \star & \pi & \blacktriangleright & \tau & \star & \theta_{\mathsf{c}} & \star & \pi & (\mathtt{Pop}_{\mathsf{c}}) \\
\tau & \star & \{\!|\theta|\!\} \cdot t & \star & \pi & \blacktriangleright & \tau \cdot \theta & \star & t & \star & \pi & (\mathtt{Push}_{\mathsf{c}})
\end{array}$$

Figure 1. A two stacks abstract machine.

[2] This second kind of values will be elaborated in Sec. 1.2.

AppLam	(al)	$(\lambda x.t)\ u \to t[x := u]$	
LamApp	(la)	$\lambda x.\, tx \to t$	$(x \notin fv(t))$
CaseCons	(co)	$\{\!\|\theta\|\!\} \cdot \mathsf{c} \to t$	$((\mathsf{c} \mapsto t) \in \theta)$
CaseApp	(ca)	$\{\!\|\theta\|\!\} \cdot (tu) \to (\{\!\|\theta\|\!\} \cdot t)u$	
CaseLam	(cl)	$\{\!\|\theta\|\!\} \cdot \lambda x.t \to \lambda x.\{\!\|\theta\|\!\} \cdot t$	$(x \notin fv(\theta))$
CaseCase	(cc)	$\{\!\|\theta\|\!\} \cdot \{\!\|\phi\|\!\} \cdot t \to \{\!\|\theta \circ \phi\|\!\} \cdot t$	

with $\theta \circ \{\mathsf{c}_1 \mapsto t_1; \ldots; \mathsf{c}_n \mapsto t_n\} = \{\mathsf{c}_1 \mapsto \{\!\|\theta\|\!\} \cdot t_1; \ldots; \mathsf{c}_n \mapsto \{\!\|\theta\|\!\} \cdot t_n\}$

Figure 2. The λ-calculus with constructors.

1.2 ML-style pattern matching

Notice that constructors, just as any terms, can be applied to any number of arguments (they are *variadic*). We call a *data structure* a constructor possibly applied to some arguments. For instance, one can represent the natural numbers by data structures, using two constructors S and $\mathsf{0}$ and the unary encoding of natural numbers. The predecessor function is then written $\mathtt{pred} := \lambda x.\{\!\|\mathsf{0} \mapsto \mathsf{0}; \mathsf{S} \mapsto \lambda z.z\|\!\} \cdot x$. Its application to a number $\mathsf{S}n$ is actually evaluated to n (we skip the first β-reduction steps):

$$\begin{array}{ccccccccccc}
\diamond & \star & \mathtt{pred}\ (\mathsf{S}n) & \star & \diamond & \blacktriangleright & \diamond & \star & \{\!\|\theta\|\!\} \cdot \mathsf{S}n & \star & \diamond \\
 & & & & & \blacktriangleright & \diamond \cdot \theta & \star & \mathsf{S}n & \star & \diamond \\
 & & & & & \blacktriangleright & \diamond \cdot \theta & \star & \mathsf{S} & \star & n \cdot \diamond \\
 & & & & & \blacktriangleright & \diamond & \star & \lambda z.z & \star & n \cdot \diamond \\
 & & & & & \blacktriangleright & \diamond & \star & n & \star & \diamond
\end{array}$$

(where $\theta = \{\mathsf{0} \mapsto \mathsf{0}; \mathsf{S} \mapsto \lambda z.z\}$). More generally, any pattern matching with a branch " `C(x1,...,xk) -> t` " in a ML-like program behaves like a term with a branch $\mathsf{c} \mapsto \lambda x_1 \ldots x_k.t$ evaluated in our machine. In this sense, the machine presented above is able to simulate pattern matching on elaborated data structures with a simple rule of constant analysis. The same idea is underlying the lambda calculus with constructors (or $\lambda_{\mathscr{C}}$-calculus).

1.3 Operational semantics of the $\lambda_{\mathscr{C}}$-calculus

The ML-style pattern matching is achieved in the double stack abstract machine by giving a double status to the constructors: they can be applied to some arguments to form a compound data structure (in this case they interact with the application stack), but they can also be seen as a constant to analyse by the case bindings (and then interact with the case stack). In the semantics setting, this context switching corresponds to the following commutation rule between case and application constructs:

$$\{\!\|\theta\|\!\} \cdot (tu) \quad \simeq \quad (\{\!\|\theta\|\!\} \cdot t)\ u\ .$$

This is a crucial rule of the lambda calculus with constructors (Fig. 2), called CaseApp (or ca for short). In addition to this rule, the calculus supports the usual β and η-reductions (*resp.* al and la), and the rule of constant analysis that we have seen earlier (CO). There are also a commutation rule between case construct and λ-abstractions (CL), and a composition rule for case bindings (CC) so that the $\lambda_{\mathscr{C}}$-calculus enjoys confluence and separation properties [1]. Writing $\rightarrow^*$ the transitive closure of the reduction relation $\rightarrow$, one can check that $\texttt{pred}\ (\mathsf{S}n) \rightarrow^* n$, using rules AL, CA and CO.

Whereas the rule CaseApp does not match with the usual typing intuitions (the same subterm can be applied like a function, or pattern matched like a data structure), the case composition corresponds to a commutative conversion in logic [6, Sec. 10.4]:

$$\{\!|\theta|\!\} \cdot \{\!|\mathsf{c}_1 \mapsto u_1; \ldots;\ \mathsf{c}_n \mapsto u_n|\!\} \cdot t \ \rightarrow\ \{\!|\mathsf{c}_1 \mapsto \{\!|\theta|\!\} \cdot u_1; \ldots; \mathsf{c}_n \mapsto \{\!|\theta|\!\} \cdot u_n|\!\} \cdot t$$

Concerning evaluation contexts, this rule amounts to merging all case bindings of a case stack $\diamond \cdot \theta_1 \cdots \theta_k$ in only one (optional) case binding $\theta_1 \circ \cdots \circ \theta_k$ (with left associativity of $\circ$). Also we consider the following alternative abstract machine for the $\lambda_{\mathscr{C}}$-calculus, that we call the $\mathsf{KAM}_{\lambda_{\mathscr{C}}}$:

Definition 1.1 (The $\mathsf{KAM}_{\lambda_{\mathscr{C}}}$). A *process* is a triple $\langle\theta\rangle \ \star\ t\ \star\ \pi$, where $\langle\theta\rangle$ is an optional case binding ($\diamond$ or θ), t is a term and π a stack of terms. The four *execution rules* are

$$\begin{array}{ccccccccccc}
\langle\theta\rangle & \star & \lambda x.t & \star & u\cdot\pi & \blacktriangleright & \langle\theta\rangle & \star & t[x:=u] & \star & \pi \\
\langle\theta\rangle & \star & tu & \star & \pi & \blacktriangleright & \langle\theta\rangle & \star & t & \star & u\cdot\pi \\
\theta & \star & \mathsf{c} & \star & \pi & \blacktriangleright & \diamond & \star & \theta_{\mathsf{c}} & \star & \pi \\
\langle\theta\rangle & \star & \{\!|\phi|\!\}\cdot t & \star & \pi & \blacktriangleright & \langle\theta\rangle\circ\phi & \star & t & \star & \pi
\end{array}$$

where $\langle\theta\rangle \circ \phi$ is ϕ if $\langle\theta\rangle = \diamond$, and $\theta \circ \phi$ if $\langle\theta\rangle = \theta$.

Actually the rule CaseCase is not absolutely necessary from the computational point of view (it is only necessary for the separation property). Hence we could also consider the $\lambda_{\mathscr{C}}$-calculus without CC (the $\lambda_{\mathscr{C}}^{-}$-*calculus*), and the first version of abstract machine we have presented. This would also lead to a slightly different notion of models (see the footnotes 3 and 5).

In the next section, we will use this machine to translate the lambda calculus with constructors into the pure lambda calculus.

2 CPS translation

Plotkin [14] used stack abstract machines to define *continuation passing style* (CPS) translations between the *call-by-name* and *call-by-value* λ-calculi. Indeed, the stack of the the machine can be encoded with pairs in the λ-calculus. In the same way, we will define a CPS translation of the $\lambda_{\mathscr{C}}$-calculus into the lambda calculus with pairs (or λ_{P}-calculus) based on the $\mathsf{KAM}_{\lambda_{\mathscr{C}}}$. A $\lambda_{\mathscr{C}}$-term t will be translated by a λ_{P}-term t^* that takes an evaluation context k (the *continuation*) in argument, and that returns the result of the evaluation of t with context k in the machine.

$$
\begin{aligned}
|M_\theta \star x \star M_\pi| &= x \,\langle\!|M_\theta, M_\pi|\!\rangle \\
|M_\theta \star tu \star M_\pi| &= |M_\theta \star t \star \langle\!|u^*, M_\pi|\!\rangle| \\
|M_\theta \star \lambda x.t \star M_\pi| &= \mathsf{let}\ \langle\!|x, x_{\pi'}|\!\rangle = M_\pi\ \mathsf{in}\ |M_\theta \star t \star x_{\pi'}| \quad (\text{if } x \notin fv(M_\theta, M_\pi)) \\
|M_\theta \star \mathsf{c}_i \star M_\pi| &= \mathsf{let}\ \langle\!|x_1; \dots; x_n|\!\rangle_n = M_\theta\ \mathsf{in}\ |* \ \star x_i \star M_\pi| \\
|M_\theta \star \{\!|\phi|\!\} \cdot t \star M_\pi| &= |\langle\!|N_1; \cdots; N_n|\!\rangle_n \star t \star M_\pi| \\
&\left(\text{where } \begin{array}{ll} N_i = \lambda k'.\mathsf{let}\ \langle\!|z_\theta, z_\pi|\!\rangle = k'\ \mathsf{in}\ |M_\theta \star u_i \star z_\pi| & \text{if } \mathsf{c}_i \mapsto u_i \in \phi \\ N_i = * & \text{if } \mathsf{c}_i \notin \mathrm{dom}(\phi) \end{array}\right)
\end{aligned}
$$

Figure 3. Translation of $\lambda_{\mathscr{C}}$-calculus into λ_{P}-calculus

2.1 Target calculus

A continuation is a pair $\langle\!|M_\theta, M_\pi|\!\rangle$ where M_θ and M_π are two λ_{P}-terms representing respectively the case binding and the application stack of the evaluation context. From now on, we write $\{\mathsf{c}_1, \cdots, \mathsf{c}_n\}$ the set $\mathscr{C}$ of constructors, and a case binding θ of the $\lambda_{\mathscr{C}}$-calculus will be translated by a the n-tuple $\langle\!|M_1, \cdots, M_n|\!\rangle_n$ where M_i is t_i^* if $(\mathsf{c}_i \mapsto t_i) \in \theta$, or a special constant $*$ (meaning here match failure) if $\mathsf{c}_i \notin \mathrm{dom}(\theta)$.

Terms of the λ_{P}-calculus are given by the following grammar and rules:

$$M, N, P \ := \quad x \mid \lambda x.M \mid MN \mid * \mid \langle\!|M, N|\!\rangle \mid \pi_i(M) \quad (i \in \{1,2\})$$

$$
\begin{array}{rclcrcl}
(\lambda x.M)\, N & \to_{\mathrm{P}} & M[x := M] & ; & \lambda x.(Mx) & \to_{\mathrm{P}} & M \quad (x \notin fv(M)) \\
\pi_i(\langle\!|M_1, M_2|\!\rangle) & \to_{\mathrm{P}} & M_i & ; & \langle\!|\pi_1(M), \pi_2(M)|\!\rangle & \to_{\mathrm{P}} & M
\end{array}
$$

We use the same names for variables than in the $\lambda_{\mathscr{C}}$-calculus, although we may also write k for some λ_{P}-variables (representing a continuation). We use the notations $\langle\!|M_1, \dots, M_\ell|\!\rangle_\ell$ and $\pi_i^\ell(M)$ for the usual encoding of tuples and generalised projections with pairs. We also write $\mathsf{let}\ \langle\!|x_1, \dots, x_\ell|\!\rangle_\ell = P\ \mathsf{in}\ M$ for the term $(\lambda x_1, \dots, x_\ell.M)\pi_1^\ell(P) \dots \pi_\ell^\ell(P)$ (when ℓ is not specified it is 2), so that $\mathsf{let}\ \langle\!|x_1, \dots, x_\ell|\!\rangle_\ell = \langle\!|N_1, \dots, N_\ell|\!\rangle_n\ \mathsf{in}\ M \to_{\mathrm{P}}^* M[x_1 := N_1] \dots [x_\ell := N_\ell]$.

2.2 The CPS translation

The translation of a $\lambda_{\mathscr{C}}$-term t in the λ_{P}-calculus is then given by

$$t^* \ := \quad \lambda k.\mathsf{let}\ \langle\!|x_\theta, x_\pi|\!\rangle = k\ \mathsf{in}\ |x_\theta \star t \star x_\pi|$$

where the *result of* t *in context* $\langle\!|M_\theta, M_\pi|\!\rangle$ (where M_θ an d M_π are two λ_{P}-terms), denoted by $|M_\theta \star t \star M_\pi|$, is defined by induction in Fig. 3. The translations of a variable, a λ-abstraction and an application exactly correspond (after forgiving the "case" part of the continuation) to the translation *c.b.v.-c.b.n.* of Plotkin [14]. The translation of a constructor c_i consists in giving the application context to the i^{th} component of the case context (x_i). Remark that no case context is given (we use the term $*$), since x_i comes with its own case context (Ex. 2.1). The translation of a case construct amounts to composing the (translated) case binding with the case context.

Example 2.1 Let $\psi = \{\mathsf{c}_i \mapsto u_i / i \in \mathcal{S} \subseteq [1..n]\}$ and $\phi = \{\mathsf{c}_i \mapsto s_i / i \in \mathcal{S}' \subseteq [1..n]\}$. Then the result of the term $u = \{\!|\phi|\!\} \cdot \{\!|\psi|\!\} \cdot (\mathsf{c}_j\ t)$ (with $j \in \mathrm{dom}(\psi)$) is:

$$\begin{aligned}
|M_\theta \star u \star M_\pi| &= |\langle\!| N_1, \ldots, N_n |\!\rangle_n \star \{\!|\psi|\!\} \cdot (\mathsf{c}_j\ t) \star M_\pi| \\
&\qquad\qquad \text{(with } N_i{=}\lambda k.\mathsf{let}\ \langle\!| z_\theta, z_\pi |\!\rangle{=}k\ \mathsf{in}\ |M_\theta \star s_i \star z_\pi| \text{ if } i{\in}\mathrm{dom}(\phi)) \\
|M_\theta \star u \star M_\pi| &= |\langle\!| P_1, \ldots, P_n |\!\rangle_n \star \mathsf{c}_j\ t \star M_\pi| \\
&\qquad\qquad \text{(with } P_i{=}\lambda k'.\mathsf{let}\ \langle\!| z'_\theta, z'_\pi |\!\rangle{=}k'\ \mathsf{in}\ |\langle\!| N_1,\ldots,N_n |\!\rangle_n \star u_i \star z'_\pi| \text{ if } i{\in}\mathrm{dom}(\phi)) \\
|M_\theta \star u \star M_\pi| &= |\langle\!| P_1, \ldots, P_n |\!\rangle_n \star \mathsf{c}_j \star \langle\!| t^*, M_\pi |\!\rangle| \\
&= \mathsf{let}\ \langle\!| x_1; \ldots; x_n |\!\rangle_n = \langle\!| P_1, \ldots, P_n |\!\rangle_n\ \mathsf{in}\ |* \ \star x_j \star \langle\!| t^*, M_\pi |\!\rangle| \\
&= \mathsf{let}\ \langle\!| x_1; \ldots; x_n |\!\rangle_n = \langle\!| P_1, \ldots, P_n |\!\rangle_n\ \mathsf{in}\ x_j\ \langle\!| *, \langle\!| t^*, M_\pi |\!\rangle |\!\rangle \\
&\to_{\mathrm{P}}^* P_j\ \langle\!| *, \langle\!| t^*, M_\pi |\!\rangle |\!\rangle \\
&\to_{\mathrm{P}} \mathsf{let}\ \langle\!| z'_\theta, z'_\pi |\!\rangle = \langle\!| *, \langle\!| t^*, M_\pi |\!\rangle |\!\rangle\ \mathsf{in}\ |\langle\!| N_1, \ldots, N_n |\!\rangle_n \star u_j \star z'_\pi| \\
&\to_{\mathrm{P}}^* |\langle\!| N_1, \ldots, N_n |\!\rangle_n \star u_j \star \langle\!| t^*, M_\pi |\!\rangle|
\end{aligned}$$

This translation enables the simulation of the lambda calculus with constructors in the lambda calculus with pairs (Theo. 2.4). This result derives from the following lemmas (proved by a trivial induction on t):

Lemma 2.2 *Let t, t', u be three $\lambda_{\mathscr{C}}$-terms, and x a variable* not free *in t'. Then for any λ_{P}-terms N, M_θ, M_π,*

$$\begin{array}{rlcl}
\text{(i)} & |M_\theta \star t' \star M_\pi|\ [x := N] & = & |\ M_\theta[x := N] \star t' \star M_\pi[x := N]\ | \\
\text{(ii)} & |M_\theta \star t \star M_\pi|\ [x := u^*] & \to_{\mathrm{P}}^* & |\ M_\theta[x := u^*] \star t[x := u] \star M_\pi[x := u^*]\ |
\end{array}$$

$$\begin{array}{rlcl}
\text{(iii)} & M_\theta \to_{\mathrm{P}} M'_\theta & \Longrightarrow & |M_\theta \star t \star M_\pi| \to_{\mathrm{P}} |M'_\theta \star t \star M_\pi| \\
& M_\pi \to_{\mathrm{P}} M'_\pi & \Longrightarrow & |M_\theta \star t \star M_\pi| \to_{\mathrm{P}}^* |M_\theta \star t \star M'_\pi|
\end{array}$$

Lemma 2.3 *For any $\lambda_{\mathscr{C}}$-terms t, u, any case-bindings ϕ, ψ, and any λ_{P}-terms M_θ, M_π,*

$$\begin{array}{rcll}
|M_\theta \star (\lambda x.t)u \star M_\pi| & \to_{\mathrm{P}}^+ & |M_\theta \star t[x := u] \star M_\pi| & \\
|M_\theta \star \lambda x.tx \star M_\pi| & \to_{\mathrm{P}}^+ & |M_\theta \star t \star M_\pi| & \textit{if } x \notin fv(t) \\
|M_\theta \star \{\!|\phi|\!\} \cdot \mathsf{c} \star M_\pi| & \to_{\mathrm{P}}^+ & |M_\theta \star u \star M_\pi| & \textit{if } c \mapsto u \in \phi \\
|M_\theta \star \{\!|\phi|\!\} \cdot tu \star M_\pi| & = & |M_\theta \star (\{\!|\phi|\!\} \cdot t)u \star M_\pi| & \\
|M_\theta \star \{\!|\phi|\!\} \cdot \lambda x.t \star M_\pi| & = & |M_\theta \star \lambda x.\{\!|\phi|\!\} \cdot t \star M_\pi| & \textit{if } x \notin fv(\phi) \\
|M_\theta \star \{\!|\phi|\!\} \cdot \{\!|\psi|\!\} \cdot t \star M_\pi| & = & |M_\theta \star \{\!|\phi \circ \psi|\!\} \cdot t \star M_\pi| &
\end{array}$$

Notice that the only $\lambda_{\mathscr{C}}$-rules that are actually simulated by some reduction steps in the λ_{P}-calculus are the β and η-reductions, and the constant analysis. The other rules correspond to the management of the stacks by the machine, and are simulated *during* the CPS translation.

Theorem 2.4 (Correct simulation) *For any $\lambda_{\mathscr{C}}$-terms t, t',*

$$t \to t' \quad \textit{implies} \quad t^* \to^*_{\mathrm{P}} t'^* \ .$$

2.3 Consequences for denotational models

The simulation theorem provides a sound interpretation of the $\lambda_{\mathscr{C}}$-calculus in any model of the λ_{P}-calculus. Indeed, if $[\cdot]$ is an interpretation of the λ_{P}-terms that equalises the equivalent ones, then $t \simeq_{\lambda_{\mathscr{C}}} t'$ implies $t^* \simeq_{\lambda_{\mathrm{P}}} t'^*$ (by Theo. 2.4 and the Church-Rösser property) and thus $[t^*] = [t'^*]$ in the model (for each calculus $\mathcal{L}$ presented in this paper, we write $\simeq_{\mathcal{L}}$ the reflexive symmetric and transitive closure of its reduction rules).

In the next section, we give a categorical definition of models for the $\lambda_{\mathscr{C}}$-calculus, and we show how to transform a model of the lambda calculus with pairs into a $\lambda_{\mathscr{C}}$-model. This transformation of models will directly come from the CPS translation we have just presented.

3 Classical models for the $\lambda_{\mathscr{C}}$-calculus

In this section we briefly present what is a categorical model for the $\lambda_{\mathscr{C}}$-calculus (more details and proofs can be found in [12]), and we show that the continuation models of the pure lambda calculus have the good structure to be seen as $\lambda_{\mathscr{C}}$-models.
Notations: In a Cartesian closed category (CCC), we write Id_A the identity morphism on A, and $f;g$ the composition of f and then g. We denote by $A \times B$ the product of two objects A and B, and by B^A their exponent, and by $\mathbf{1}$ the terminal object. The i^{th} projection morphisms over k is written π^k_i (or π_i if $k = 2$), the pairing of f and g is $\langle f, g\rangle$, $\mathtt{ev}$ is the evaluation morphisms and $\Lambda(f)$ the curried form of f.

3.1 Categorical models of the $\lambda_{\mathscr{C}}$-calculus

In category theory, a model for the pure lambda calculus is a CCC with a reflexive object $D \cong D^D$. Indeed, λ-terms are interpreted in D, and points of D^D are functions from terms to terms (*i.e.* open terms, abstracted over a free variable). Then a morphism $\mathtt{lam} : D^D \to D$ enables to construct the denotation of $\lambda x.t$, from the representation of the function mapping x to t. In the same way, a morphism $\mathtt{app}$: $D \to D^D$ allows to interpret the application of any term to an other one. Also the equality $\mathtt{app} \circ \mathtt{lam} = \mathit{Id}$ ensures that the interpretation respects β-equivalence.

To interpret the $\lambda_{\mathscr{C}}$-calculus in such a category, some extra morphisms are necessary (for the interpretation of the constructors and the case construct), as well as some equalities between them to validate the CASE rules. A case binding θ will be interpreted in D^n: the i^{th} component corresponds to θ_{c_i} if it is defined, and is a special point $\lightning$ (meaning match failure) of D otherwise. Then a morphism $\mathtt{case} : (D^n \times D) \to D$ is required to interpret the case construct $\{\!|\theta|\!\} \cdot t$, given the denotation of θ in D^n and the one of t in D. We also need a point c^*_i of D for every constructor $\mathsf{c}_i \in \mathscr{C}$.

Definition 3.1 **($\lambda_{\mathscr{C}}$-model)** A categorical model for the untyped $\lambda_{\mathscr{C}}$-calculus is a structure $(\mathbb{C}\,, D\,, \mathtt{app}\,, \mathtt{lam}\,, (c_i^*)_{i=1}^n, \lightning\,, \mathtt{case})$ where

- $\mathbb{C}$ is a Cartesian closed category, and D is one of its object.
- $\mathtt{app} : D \to D^D$ and $\mathtt{lam} : D^D \to D$ form an isomorphism: $D \cong D^D$.
- All the c_i^*'s and $\lightning$ are points of D, and $\mathtt{case}$ is a morphism of $D^n \times D \to D$,
- The four diagrams of Fig.4 commute (($D1$) commutes for every $i \in [\![1..n]\!]$).

	CaseCons		CaseApp
($D1$)	$$\begin{tikzcd} D^n \arrow[r, leftrightarrow, "\cong"] \arrow[d, "\pi_i^n"'] & D^n \times \mathbf{1} \arrow[d, "Id \times c_i^*"] \\ D & D^n \times D \arrow[l, "\mathtt{case}"] \end{tikzcd}$$	($D2$)	$$\begin{tikzcd} (D^n \times D) \times D \arrow[rr, leftrightarrow, "\cong"] \arrow[d, "\mathtt{case} \times Id"'] & & D^n \times (D \times D) \arrow[d, "Id \times (\mathtt{app} \times Id)"'] \\ D \times D \arrow[d, "\mathtt{app} \times Id"'] & & D^n \times (D^D \times D) \arrow[d, "Id \times \mathtt{ev}"'] \\ D^D \times D \arrow[dr, "\mathtt{ev}"'] & & D^n \times D \arrow[dl, "\mathtt{case}"] \\ & D & \end{tikzcd}$$

	CaseCase		
($D3$)	$$\begin{tikzcd} D^n \times \mathbf{1} \arrow[r, "Id_{D^n} \times \lightning"] \arrow[d, "\pi_2"'] & D^n \times D \arrow[d, "\mathtt{case}"] \\ \mathbf{1} \arrow[r, "\lightning"'] & D \end{tikzcd}$$	($D4$)	$$\begin{tikzcd} (D^n \times D^n) \times D \arrow[rr, leftrightarrow, "\cong"] \arrow[d, "\bullet \times Id"'] & & D^n \times (D^n \times D) \arrow[d, "Id \times \mathtt{case}"'] \\ D^n \times D \arrow[dr, "\mathtt{case}"'] & & D^n \times D \arrow[dl, "\mathtt{case}"] \\ & D & \end{tikzcd}$$

Figure 4. Commuting diagrams in a $\lambda_{\mathscr{C}}$-model

The equalities of morphisms described in Fig. 4 ensure that the interpretation we have informally presented respects $\lambda_{\mathscr{C}}$-equivalence. It is pretty clear how the commutation of diagrams ($D4$) and ($D2$) entail the validity of rules CaseCons and CaseApp respectively. The validity of the rule CaseCase is expressed through a morphism $\bullet : D^n \times D^n \to D^n$ ($D4$), that represents the case binding composition in the categorical framework (Lem. 3.3). It is defined as the pairing of the morphisms $(Id_{D^n} \times \pi_i^n); \mathtt{case}$, for $1 \leq i \leq n$.

The diagram ($D3$) is the only one that does not directly translate a reduction rule of the $\lambda_{\mathscr{C}}$-calculus. It expresses the equivalence between a match failure and the matching of a match failure[3], and is necessary for the soundness *w.r.t.* the rule CaseCase.

[3] If we enrich the $\lambda_{\mathscr{C}}$-calculus with explicit match failure (that is, a special constant $\lightning$ and the rule $\{\!|\theta|\!\} \cdot \mathsf{c} \to \lightning$ if $\mathsf{c} \notin \mathrm{dom}(\theta)$), then we need an extra rule $\{\!|\theta|\!\} \cdot \lightning \to \lightning$ for confluence (to close the critical pair

Notice that there is no diagram corresponding to the rule CaseLam in the definition of a $\lambda_{\mathscr{C}}$-model. In the same way that the rule CL closes a critical pair between CA and AL, the commutation of the diagram corresponding to it $(D2')$ is induced by the commutation of $(D2)$ and the reflexivity of D, as expressed by Lem. 3.2. This diagram uses a morphism $\texttt{case}^\circ : D^n \times D^D \to D^D$, that abstracts the case construct over a variable: it turns a case binding θ and a function mapping x to t into the function mapping x to $\{\!|\theta|\!\} \cdot t$, and it is formally defined as the curried form of

$$(D^n \times D^D) \times D \cong D^n \times (D^D \times D) \xrightarrow{Id_{D^n} \times \texttt{ev}} D^n \times D \xrightarrow{\texttt{case}} D \ .$$

Lemma 3.2 (Diagram $(D2')$) *If $(\texttt{app}, \texttt{lam})$ form an isomorphism between D and D^D, then the commutation of $(D2)$ is equivalent to the commutation of the following diagram:*

$$\begin{tikzcd} D^n \times D^D \arrow[r, "\texttt{case}^\circ"] \arrow[d, "Id \times \texttt{lam}"'] & D^D \arrow[d, "\texttt{lam}"] \\ D^n \times D \arrow[r, "\texttt{case}"'] & D \end{tikzcd} \qquad (D2')$$

This enables to define a sound interpretation of $\lambda_{\mathscr{C}}$-terms in any $\lambda_{\mathscr{C}}$-model: if t has its free variables included in $\Gamma = (x_1, \ldots, x_k)$, then its interpretation $[t]_\Gamma$ is defined by induction in Fig. 5.

$$\begin{array}{rcl}
[x_i]_\Gamma & = & \pi_i^k : D^k \to D \\
[tu]_\Gamma & = & D^k \xrightarrow{\langle [t]_\Gamma, [u]_\Gamma \rangle} D \times D \xrightarrow{\texttt{app} \times Id_D} D^D \times D \xrightarrow{\texttt{ev}} D \\
[\lambda x_{k+1}.t]_\Gamma & = & D^k \xrightarrow{\Lambda(f_t)} D^D \xrightarrow{\texttt{lam}} D \\
\text{where } f_t & = & D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{[t]_{\Gamma, x_{k+1}}} D \\
[\mathsf{c}]_\Gamma & = & D^k \xrightarrow{!_{D^k}} \mathbf{1} \xrightarrow{c^*} D \\
[\{\!|\theta|\!\} \cdot t]_\Gamma & = & D^k \xrightarrow{\langle [\theta]_\Gamma, [t]_\Gamma \rangle} D^n \times D \xrightarrow{\texttt{case}} D \\
[\theta]_\Gamma & = & \langle f_1, \cdots, f_n \rangle : D^k \to D^n \ , \quad \text{where } f_i = \begin{cases} [u_i]_\Gamma & \text{if } \mathsf{c_i} \mapsto u_i \in \theta \\ !_{D^k}; \lightning & \text{if } \mathsf{c_i} \notin \text{dom}(\theta) \end{cases}
\end{array}$$

Figure 5. Interpretation of $\lambda_{\mathscr{C}}$-terms in a categorical model

Lemma 3.3 (Categorical case composition) *If θ and ϕ are two case bindings whose free variables are all in Γ, then their interpretation in any $\lambda_{\mathscr{C}}$-model satisfies*

$$[\theta \circ \phi]_\Gamma = \quad D^k \xrightarrow{\langle [\theta]_\Gamma, [\phi]_\Gamma \rangle} D^n \times D^n \xrightarrow{\bullet} D^n \ .$$

with CC). The model we present here would still be sound for the extended calculus, and the diagram $(D3)$ corresponds to this last rule. Alternatively, if we remove the rule CaseCase form the calculus, then the commutation of $(D1)$ and $(D2)$ is sufficient.

Proposition 3.4 (Soundness) *If* $(\mathbb{C}, D, \texttt{app}, \texttt{lam}, (c_i^*)_{i=1}^n, \lightning, \texttt{case})$ *is a* $\lambda_{\mathscr{C}}$*-model, then Fig. 5 interprets each closed* $\lambda_{\mathscr{C}}$*-term* t *by a point* $[t]$ *of* D *such that*

$$t \simeq_{\lambda_{\mathscr{C}}} t' \quad \Longrightarrow \quad [t] = [t'] \ .$$

In fact $\lambda_{\mathscr{C}}$-models are even *complete* for the sub calculus with no match failure [12].

3.2 Continuation models

A Cartesian closed category is a model of the pure λ-calculus if it has an object D equivalent to its function space, in which we can interpret the λ-terms. Among them are the *continuation models* [4] (see the excellent introduction of [15]): CCC with two objects C and R satisfying the equation $C \cong C \times R^C$. Indeed, taking $D = R^C$ fulfils the condition $D \cong D^D$, and leads to interpret the λ-terms by points of R^C, *i.e.* informally by functions taking a *continuation argument* in C, and returning a *response* in R. A functional term (*i.e.* a point of D^D) is interpreted in $(R^C)^{R^C} \cong R^{C \times R^C}$, also the continuation argument of a function is a point of $C \times R^C$. It represents a pair composed of *later continuation* (in C) and a term (in R^C) that is the argument of the function. That is why we can see the continuations as *stacks of arguments*, and the term interpretations in a continuation model as *processes* in a stack abstract machine.

As we have seen earlier (Sec. 2.1), a continuation for the $\lambda_{\mathscr{C}}$-calculus should not be only a stack of arguments, but a pair composed of a case binding (*i.e.* a point of D^n) and a stack (*i.e.* a point of some object S satisfying the stack equation). This gives rise to the following definition.

Definition 3.5 (Continuation $\lambda_{\mathscr{C}}$-model) A CCC is a *continuation $\lambda_{\mathscr{C}}$-model* (or *classical* $\lambda_{\mathscr{C}}$-model) if it has four objects R, C, S [5] and D satisfying the following equations:

$$D \cong R^C \qquad ; \qquad C \cong D^n \times S \qquad ; \qquad S \cong D \times S$$

In the next section, we show that every continuation $\lambda_{\mathscr{C}}$-model is actually a $\lambda_{\mathscr{C}}$-model in the sense of Def. 3.1. It might be a bit tedious to describe the morphisms $\texttt{app}$, $\texttt{lam}$, c^* and $\lightning$ in a continuation model (and still more to prove the diagrams commutation) with compositions and curried forms; and so we use the λ-calculus with pairs (as an *internal language* for CCCs) to define those morphisms. Given a Cartesian closed category $\mathbb{C}$, we call its internal language the $\lambda_{\mathbb{C}}$-calculus, and we write $\simeq_{\mathbb{C}}$ the equivalence of terms in this language.

3.3 From continuation $\lambda_{\mathscr{C}}$-models to $\lambda_{\mathscr{C}}$-models

Let $\mathbb{C}$ be a continuation $\lambda_{\mathscr{C}}$-model (Def. 3.5). We write $\uparrow_s$, $\downarrow_s$, $\uparrow_c$ and $\downarrow_c$ the terms (*resp.* of type $S \to D \times S$, $D \times S \to S$, $C \to D^n \times S$ and $D^n \times S \to C$) corresponding

[4] Also called *classical models*, as their underlying logic is the classical logic [10].

[5] Without the case composition, the case context would be not only a case binding (in D^n), but a stack of case bindings. Hence we would need a fourth object S' satisfying the equation $S' \cong D^n \times S'$, and the interpretation of terms in such a model would be more complex.

to the morphisms that guarantee $S \cong D \times S$ and $C \cong D^n \times S$. We show that $\mathbb{C}$ can be provided with the structure of a $\lambda_{\mathscr{C}}$-model (Def. 3.1). To do so, we refer to the $\lambda_{\mathbb{C}}$-terms defined in Fig. 6. The terms $M_{\texttt{lam}}$, $M_{\texttt{app}}$ and $M_{\texttt{case}}$ have a free

$$\begin{aligned}
M_{\texttt{lam}} &= \lambda k.\mathsf{let}\ \langle\!| x_\theta, x_\pi |\!\rangle =\uparrow_c k\ \mathsf{in}\ \ \mathsf{let}\ \langle\!| x, x'_\pi |\!\rangle =\uparrow_s x_\pi\ \mathsf{in}\ z\ x\ \left(\downarrow_c \langle\!| x_\theta, x'_\pi |\!\rangle\right)\\
M_{\texttt{app}} &= \lambda x.\lambda k.\mathsf{let}\ \langle\!| x_\theta, x_\pi |\!\rangle =\uparrow_c k\ \mathsf{in}\ z\ \left(\downarrow_c \langle\!| x_\theta, \downarrow_s \langle\!| x, x_\pi |\!\rangle |\!\rangle\right)\\
M_{\mathsf{c_i}} &= \lambda k.\mathsf{let}\ \langle\!| x_\theta, x_\pi |\!\rangle =\uparrow_c k\ \mathsf{in}\ \ \mathsf{let}\ \langle\!| x_1; \dots; x_n |\!\rangle_n = x_\theta\ \mathsf{in}\ x_i\ k\\
M_{\texttt{case}} &= \lambda k.\mathsf{let}\ \langle\!| x_\theta, x_\pi |\!\rangle =\uparrow_c k\ \mathsf{in}\\
&\qquad \mathsf{let}\ \langle\!| y_\phi, y |\!\rangle = z\ \mathsf{in}\ y\ \left(\downarrow_c \langle\!| \langle\!| M_1, \dots, M_n |\!\rangle_n, x_\pi |\!\rangle\right)\ ,\\
&\qquad\qquad \text{where } M_i = \lambda k'.\ \mathsf{let}\ \langle\!| z_\theta, z_\pi |\!\rangle =\uparrow_c k'\ \mathsf{in}\\
&\qquad\qquad\qquad \mathsf{let}\ \langle\!| x_1; \dots; x_n |\!\rangle_n = y_\phi\ \mathsf{in}\ x_i\ \left(\downarrow_c \langle\!| x_\theta, z_\pi |\!\rangle\right)\\
M_{\lightning} &= \lambda k.\mathsf{let}\ \langle\!| x_\theta, x_\pi |\!\rangle =\uparrow_c k\ \mathsf{in}\ \mathsf{let}\ \langle\!| x, x'_\pi |\!\rangle =\uparrow_s x_\pi\ \mathsf{in}\ x\ \left(\downarrow_c \langle\!| \langle\!| x, \dots, x |\!\rangle_n, x_\pi |\!\rangle\right)
\end{aligned}$$

Figure 6. Terms for the morphisms of a continuation $\lambda_{\mathscr{C}}$-model

variable z, that will correspond (through the mapping from the $\lambda_{\mathbb{C}}$-calculus to $\mathbb{C}$) to the arguments of `lam`. `app` and `case` respectively. Remark that there is a direct connection between the terms defined in Fig. 6 and the CPS translation of Sec. 2 (Fig. 3), given that $|M_\theta \star t \star M_\pi| \simeq_{\lambda_{\mathrm{P}}} t^* \langle\!| M_\theta, M_\pi |\!\rangle$.

Definition 3.6 The morphisms defining a $\lambda_{\mathscr{C}}$-model are given by the following derivable judgements: $\lightning \;=\; \vdash M_{\lightning} : D$

$$\begin{array}{lclclcl}
\texttt{lam} & = & z : D \to D \vdash M_{\texttt{lam}} : D & & \texttt{app} & = & z : D \vdash M_{\texttt{app}} : D \to D\\
c^* & = & \vdash M_{\mathsf{c_i}} : D & & \texttt{case} & = & z : D^n \times D \vdash M_{\texttt{case}} : D
\end{array}$$

Theorem 3.7 ($\mathbb{C}$ is a $\lambda_{\mathscr{C}}$-model) *In a Cartesian closed category $\mathbb{C}$ that is a continuation $\lambda_{\mathscr{C}}$-model, the morphisms defined in Def. 3.6 satisfy the diagrams in Fig. 4, and the morphisms* `lam` *and* `app` *form an isomorphism between D and D^D. Also $(\mathbb{C}\,, D\,, \texttt{app}\,, \texttt{lam}\,, (c_i^*)_{i=1}^n, \lightning\,, \texttt{case})$ is a $\lambda_{\mathscr{C}}$-model.*

Proof (sketch). All the equalities on morphisms can be proved using the internal language: two morphisms are equal if their corresponding terms are convertible. For instance, the equality $\texttt{lam}; \texttt{app} = Id_{D^D}$ follows from the $\lambda_{\mathbb{C}}$-convertibility of $\texttt{lam}; \texttt{app}$ (*i.e.* $\lambda z.M_{\texttt{app}}[z := M_{\texttt{lam}}]$) and $\lambda z.z$, and inverse equation comes from $\lambda z.M_{\texttt{app}}[z := M_{\texttt{lam}}] \simeq_{\mathbb{C}} \lambda z.z$. In the same way, the commutation of the diagrams come from the following equivalences:

$$\lambda z.M_{\texttt{case}}[z := \langle\!| \pi_1(z), M_{\mathsf{c_i}} |\!\rangle] \simeq_{\mathbb{C}} \lambda z.\pi_i^n((\pi_2(z))) \qquad (D1)$$

$$\begin{aligned}
&\lambda y.(M_{\texttt{app}}[z := M_{\texttt{case}}[z := \pi_1(y)]])\pi_2(y) \simeq_{\mathbb{C}}\\
&\qquad \lambda y.M_{\texttt{case}}[z := \langle\!| \pi_{11}(y), (M_{\texttt{app}}[z := \pi_{21}(y)] |\!\rangle)\pi_2(y)] \qquad (D2)
\end{aligned}$$

$$\begin{aligned}
&\lambda y.M_{\texttt{case}}[z := \langle\!| M_\bullet\ \pi_1(y), \pi_2(y) |\!\rangle] \simeq_{\mathbb{C}}\\
&\qquad \lambda y.M_{\texttt{case}}[z := \langle\!| \pi_{11}(y), M_{\texttt{case}}[z := \langle\!| \pi_{21}(y), \pi_2(y) |\!\rangle] |\!\rangle] \qquad (D4)
\end{aligned}$$

$$\lambda y.M_{\lightning} \simeq_{\mathbb{C}} \lambda y.M_{\texttt{case}}[z := \langle\!| \pi_1(z), M_{\lightning} |\!\rangle] \qquad (D3)$$

In the commutation of ($D4$), the term $M_\bullet$ corresponds to $\bullet$, the pairing of the morphisms $(Id_{D^n} \times \pi^n_i)$; `case`:

$$M_\bullet = \lambda z.\langle\!| M_{\texttt{case}}[z := \langle\!| \pi_1(z), \pi^n_1(z) |\!\rangle]; \ldots ; M_{\texttt{case}}[z := \langle\!| \pi_1(z), \pi^n_n(z) |\!\rangle] |\!\rangle_n \qquad \Box$$

3.4 Non syntactical $\lambda_{\mathscr{C}}$-model

Although it is *a priori* not easy to construct a $\lambda_{\mathscr{C}}$-model [6] (using Def. 3.1), some well-known categories are in fact continuation $\lambda_{\mathscr{C}}$-models. Indeed, every *continuation model* (*i.e.* every CCC with two objects R and C such that $C \cong R^C \times C$) happens to be a continuation $\lambda_{\mathscr{C}}$-model if we take (by definition) $S = C$ and $D = R^C$: we immediately have $S \cong D \times S$, and

$$C \;\cong\; R^C \times C \;\cong\; R^C \times (R^C \times C) \;\cong\; \ldots \;\cong\; (R^C)^n \times C \;=\; D^n \times S.$$

Corollary 3.8 *There is a sound interpretation of the $\lambda_{\mathscr{C}}$-calculus in any* CCC *with two objects R and C such that $C \;\cong\; R^C \times C$.*

This is good news, since we know how to construct such mathematical structures. In particular, any Scott's D_∞ domain is suitable [15, Theo. 3.1].

Notice that conversely, every continuation $\lambda_{\mathscr{C}}$-model is a continuation model:

$$C \;\cong\; D^n \times S \;\cong\; D^n \times (D \times S) \;\cong\; (D^n \times S) \times D \;\cong\; C \times R^C.$$

By plugging this decomposition of isomorphism into the usual interpretation of the pure lambda calculus in continuation models, one actually obtains the morphisms `lam` and `app` as defined in Def. 3.1. Hence, using this isomorphism to transform a continuation $\lambda_{\mathscr{C}}$-model into a continuation model, and then interpreting pure λ-terms in it, amounts to the same as interpreting directly the λ-terms (seen as $\lambda_{\mathscr{C}}$-terms) in the continuation $\lambda_{\mathscr{C}}$-model.

Conclusion and further work

We have shown how to construct an interpretation of the $\lambda_{\mathscr{C}}$-calculus in any continuation model (for instance in Scott's domain): first use the isomorphism $C \cong R^C \times C$ to define the isomorphism $C \cong D^n \times S$ (with $S = C$ and $D = R^C$), and then use the isomorphims in Def. 3.6 to define the morphisms `lam`, `app`, c^*, `case` and $\lightning$. Finally interpret the $\lambda_{\mathscr{C}}$-terms as in Fig. 5.

This work raises several questions. The first one concerns the interaction of the $\lambda_{\mathscr{C}}$-calculus with the $\lambda\mu$-calculus of Parigot [11], as a calculus corresponding to classical models. Is there a well-behaved calculus including both of them? Such a calculus would be of particular interest, since the $\lambda\mu$-calculus corresponds to classical logic, whereas pattern matching on data structures is usually associated to constructive proofs [4].

The second one is about the completeness of categorical models for the lambda calculus with constructors. Indeed, the $\lambda_{\mathscr{C}}$-models are complete for the $\lambda_{\mathscr{C}}$-calculus with no match failure (or with identification of all of them, as explained in the footnote p. 9). It is then natural to ask whether the continuation $\lambda_{\mathscr{C}}$-models are

[6] Except the syntactical model in the category of Partial Equivalence Relations [12].

complete for the calculus; in other words, whether every $\lambda_{\mathscr{C}}$-model (in particular, the syntactic model of PERs) is equivalent to continuation $\lambda_{\mathscr{C}}$-model. If they are not, what would be an internal language for these categories? Maybe a kind of "$\lambda\mu_{\mathscr{C}}$-calculus", since continuation categories are complete for the $\lambda\mu$-calculus [7].

Last, the question of a denotational model for the typed $\lambda_{\mathscr{C}}$-calculus is still pending. The syntax of this calculus is quite simple but the type system proposed for it [13] is not. To give a categorical definition of the data types seems especially not easy (the only denotational model for the typed calculus so far is the syntactic model of reducibilty candidates).

Acknowledgement

To Alexandre Miquel, for giving the idea of a double stack abstract machine for the $\lambda_{\mathscr{C}}$-calculus, and for all the fruitful discussions we had on this work.

References

[1] Ariel Arbiser, Alexandre Miquel, and Alejandro Ríos. The lambda-calculus with constructors: Syntax, confluence and separation. *Journal of Functional Programming*, 19(5):581–631, 2009.

[2] Horatiu Cirstea and Germain Faure. Confluence of pattern-based lambda-calculi. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 78–92, Paris, France, June 2007. Springer.

[3] Horatiu Cirstea and Claude Kirchner. The rewriting calculus as a semantics of elan. In *ASIAN*, pages 84–85, 1998.

[4] Thierry Coquand. Pattern matching with dependant types. Proceedings of the Workshop on Types for Proofs and Programs, June 1992.

[5] Pierre Crégut. An abstract machine for lambda-terms normalization. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 333–340, 1990.

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

[7] Martin Hofmann and Thomas Streicher. Continuation models are universal for lambda-mu-calculus. In *LICS*, pages 387–395, 1997.

[8] Barry Jay. *Pattern Calculus: Computing with Functions and Structures.* Springer, 2009.

[9] C. Barry Jay and Delia Kesner. Pure pattern calculus. In *ESOP*, pages 100–114, 2006.

[10] Yves Lafont, Bernhard Reus, and Thomas Streicher. Continuation semantics or expressing implication by negation. Technical report, University of Munich, 1993.

[11] Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, pages 190–201, 1992.

[12] Barbara Petit. Categorical model for the lambda calculus with constructors. http://arxiv.org/abs/1202.4678, 2011.

[13] Barbara Petit. Semantics of typed lambda-calculus with constructors. *Logical Methods in Computer Science*, 7(1:2), 2011.

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

[15] Bernhard Reus and Thomas Streicher. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[16] Dana Scott. Outline of a mathematical theory of computation. Technical report, Princeton University, 1970.