Typing streams in the $\Lambda\mu$-calculus: extended abstract

Alexis Saurin

INRIA-Futurs & École polytechnique (LIX)
saurin@lix.polytechnique.fr

Abstract. $\Lambda\mu$-calculus [11] has been built as an untyped extension of the $\lambda\mu$-calculus [10] in order to recover Böhm theorem which was known to fail in $\lambda\mu$-calculus since a result by David & Py [2]. An essential computational feature of $\Lambda\mu$-calculus for separation to hold is the unrestricted use of abstractions over continuation variables that provides the calculus with a construction of streams.
In [11], $\Lambda\mu$-calculus was studied as a purely untyped calculus. Moreover, when one types $\Lambda\mu$-terms using standard $\lambda\mu$-calculus type system, the stream mechanism is desactivated because of typing constraints. This is deeply related with the failure of separation property in $\lambda\mu$-calculus and with the fact that $\lambda\mu$-calculus was first designed as a typed calculus for classical logic.
In this paper, we introduce a type system, $\Lambda_{\mathcal{S}}$, that unveils the stream mechanism at work in $\Lambda\mu$-calculus in the typed setting. We discuss some of its properties and argue it is a good candidate to study a typed separation result [5] for classical λ-calculi.

Keywords: Böhm Theorem, Classical λ-calculus, $\lambda\mu$-calculus, Streams, Typed Separation.

1 Introduction

Curry-Howard in classical logic. Curry-Howard isomorphism was first designed as a correspondance between simply typed λ-calculus and intuitionnistic logic. Extending the isomorphism to classical logic resulted in strong connections with control operators in functionnal programming languages as first noticed by Griffin [6]. In particular, $\lambda\mu$-calculus [10] was introduced by Michel Parigot as an extension of λ-calculus isomorphic to an alternative presentation of classical natural deduction (known as free deduction) in which one can encode usual control operators and in particular the `call/cc` operator.

$\lambda\mu$-calculus and Separation. $\lambda\mu$-calculus became one of the most standard ways to study classical lambda-calculi. As a result, the calculus has been more and more studied and more fundamental questions arose. Among them, knowing whether separation property (also called Böhm theorem [1, 8]) holds for $\lambda\mu$-calculus was one of the important question in the study of $\lambda\mu$-calculus, since

separation is a very fundamental property for such a calculus. In 2001, David & Py proved that separation fails in $\lambda\mu$-calculus by exhibiting a counter-example to separation [2]. In a previous work of 2005, we introduced an extension to $\lambda\mu$-calculus, $\Lambda\mu$-calculus, for which we could prove that separation holds [11]. $\Lambda\mu$-calculus is fairly close to standard presentations of $\lambda\mu$-calculus (see [3, 4] for instance), but is definitely a different calculus. In particular, an essential computational feature of $\Lambda\mu$-calculus for separation to hold is the unrestricted use of abstractions over continuations that provides the calculus with a construction of streams.

Typed separation. In [11], $\Lambda\mu$-calculus was studied as a purely untyped calculus. Moreover, when one types $\Lambda\mu$-terms using standard $\lambda\mu$-calculus type system, the stream mechanism is cancelled because of typing constraints and the separating power is lost. This is closely related with the failure of separation property Parigot's $\lambda\mu$-calculus and with the fact that $\lambda\mu$-calculus was first designed as a typed calculus for classical logic. Actually, when restricted to the typed case, $\Lambda\mu$-calculus is essentially the standard $\lambda\mu$-calculus. This leads us to wonder where the additional constructions of $\Lambda\mu$-calculus live, what they really do from the operational point of view and what they mean from the typed/logical viewpoint. In order to do so, one way is to look for a type system for $\Lambda\mu$-calculus in which one might prove a typed separation result such as the ones existing for simply typed λ-calculus [5, 8] and then to try and understand it from the logical point of view.

Structure of the paper. We first recall some basic elements about $\lambda\mu$-calculus, its extension, $\Lambda\mu$-calculus, and the separation property in Section 2 before introducing a type system for $\Lambda\mu$-calculus, $\Lambda_{\mathcal{S}}$, in Section 3 and discussing some of its properties. We finally conclude with some remarks about typed separation and discuss related and future works.

Remark about notations. In the following, we use the following notational conventions: we will as usual consider the application to be left associative that is fab shall be read as $((f)a)b$. Instead of the usual notation $f(a)$ we will write $(f)a$ as in [9].

2 $\lambda\mu$-calculus, streams and Separation: $\Lambda\mu$-calculus

David & Py counter-example to Separation in $\lambda\mu$-calculus. In their 2001 paper [2], David & Py adressed the question of separation property in $\lambda\mu$-calculus by exhibiting a counter-example to separation, the $\lambda\mu$-term $W = \lambda x.\mu\alpha.[\alpha]((x)\ \mu\beta.[\alpha](x)\ U_0\ y)\ U_0$ with $U_0 = \mu\delta.[\alpha]\lambda z_1.\lambda z_2.z_2$. Separation property fails in this setting because there is no way to put variable y in head position. The key point is that the entire applicative context in which this term is placed is transmitted through $\mu\alpha$ to subterms; as a consequence, the usual technique (which consists in building a context that shall explore the part of the term we

$$
\begin{array}{ll}
\triangleright\ (\lambda x.M)N & \longrightarrow_{\beta}\ M[N/x] \\
\triangleright\ \lambda x.(Mx) & \longrightarrow_{\eta}\ M \\
\triangleright\ (\mu\alpha.M)\beta & \longrightarrow_{\beta_s}\ M[\beta/\alpha] \\
\triangleright\ \mu\alpha.(M\alpha) & \longrightarrow_{\eta_s}\ M \\
\triangleright\ \mu\alpha.M & \longrightarrow_{fst}\ \lambda x.\mu\alpha.M[(U)x\alpha/(U)\alpha]
\end{array}
$$

Proviso:
In η, fst, $x \notin FV_t(M)$; in η_s, $\alpha \notin FV_s(M)$

Fig. 1. $\Lambda\mu$-calculus reduction rules

want) cannot be applied.

Recovering Separation in $\lambda\mu$-calculus: relaxing implicit (underlying) typing constraints. What we do by introducing $\Lambda\mu$-calculus is precisely to be more liberal with the construction of terms in order to provide the calculus with more applicative contexts and retrieve the ability to realize the needed exploration paths. In particular, Parigot's $\lambda\mu$-calculus syntax has a constraint of naming a term right before it is μ-abstracted (terms have the form $\mu\alpha.[\beta]\_$) which can actually be seen as a typing constraint directly built in the syntax of the untyped calculus. $\Lambda\mu$-calculus is basically the result of removing this constraint. By doing so, we obtain a calculus which is close to de Groote's presentation of $\lambda\mu$-calculus but it is not equivalent to this calculus since de Groote's presentation also contains a typing constraint which is built in the syntax, namely the ϵ rule that is absent from $\Lambda\mu$-calculus[1].

More $\Lambda\mu$-terms in order to separate. $\Lambda\mu$-calculus has more contexts and more terms than $\lambda\mu$-calculus by allowing arbitrary μ-abstractions and continuation-applications. $\Lambda\mu$-terms are given by the following syntax[2]:

$$M ::= \ x \mid \lambda x.M \mid (M)M \mid \mu\alpha.M \mid (M)\alpha$$

where x ranges over a set $\mathcal{V}_t$ of term variables and α ranges over a set $\mathcal{V}_s$ of stream variables. $\Lambda\mu$-calculus reduction rules are shown in figure 1. These rules are discussed in [11].

μ as a stream abstraction. In $\Lambda\mu$-calculus, μ can be seen as an abstraction over streams of terms[3]. For instance, while $\lambda x.\lambda y.\lambda z.((z)(M)xy)(N)xy$ may duplicate

[1] The result of the ϵ rule in $\Lambda\mu$-calculus would actually be to cancel multiple stream abstractions which would be problematic with respect to separation.

[2] We abandon notation $[\alpha]M$ and use notation $(M)\alpha$ instead, see [11].

[3] Streams as first-class citizens are consequences of more extensionality in $\Lambda\mu$-calculus than in $\lambda\mu$-calculus, due to the fact that it is possible to use the extentionality rules η and η_s where $\lambda\mu$-calculus syntax forbids to do so, for isntance: $\mu\alpha.(M)\beta \rightarrow_{\eta} \mu\alpha.(\lambda x.(M)x)\beta$.

$$\frac{}{\Gamma, x : T \vdash x \ : T|\Delta} \ Var$$

$$\frac{\Gamma, x : T \vdash M \ : U|\Delta}{\Gamma \vdash \lambda x.M \ : T \to U|\Delta} \ \lambda Abs \qquad \frac{\Gamma \vdash M \ : T \to U|\Delta \quad \Gamma \vdash N \ : T|\Delta}{\Gamma \vdash (M)N \ : U|\Delta} \ \lambda App$$

$$\frac{\Gamma \vdash M \ : \bot|\Delta, \alpha : A}{\Gamma \vdash \mu\alpha.M \ : A|\Delta} \ \mu Abs \qquad \frac{\Gamma \vdash M \ : A|\Delta, \alpha : A}{\Gamma \vdash (M)\alpha \ : \bot|\Delta, \alpha : A} \ \mu App$$

Fig. 2. $\Lambda\mu$-calculus Classical Type System.

two terms passed through x and y, $\Lambda\mu$-term $\mu\alpha.\mu\beta.\lambda z.((z)(M)\alpha\beta)(N)\alpha\beta$ can duplicate two *streams* of terms, these streams being for instance the applicative context: $[]M_1 \dots M_k\gamma N_1 \dots N_l\delta$.

Compared to $\lambda\mu$-calculus where the effect of μ is only to redirect the computation flow, in $\Lambda\mu$-calculus, one can manage to deal with streams as first-class citizens: for instance, $\mu\alpha.\mu\beta.\lambda x.\lambda y.x$ is a term that erases two streams of terms and returns the boolean value `true`. As previously said, $\Lambda\mu$-calculus has been designed in order to recover the separation property. The original counter-example to separation by David & Py [2], W, is solved by the following $\Lambda\mu$-context: $\mathcal{C} = []\mathcal{P}x_0x_1\alpha 0\alpha 1\alpha$ where $\mathcal{P} = \lambda z_0, z_1.\mu\gamma.\lambda u.((u)\mu\beta.z_1)z_0$: $\mathcal{C}(W) \to^\star y$ (see [11] for more details).

3 Typing $\Lambda\mu$-calculus

Typing $\Lambda\mu$-calculus as $\lambda\mu$-calculus. One could think of typing $\Lambda\mu$ using a standard type system for classical lambda-calculi as shown in figure 2. However, this approach is not satisfactory considering our motivations in developping the new calculus, that is from the point of view of separation. Indeed, the main structures used in [11] in order to obtain separation would not be typable in the system of figure 2 and for very fundamental reasons. Any term of the form $\mu\alpha.\lambda x.M$ would not be typable whereas this is the typical term used in the proof of separation for $\Lambda\mu$-calculus. In fact, the typing system originally introduced in order to connect the calculus with free deduction [10] precisely forbids such terms: $\lambda x.M$ is a λ-abstracted term and thus shall be of an $\to$-type whereas the fact that it is μ-abstraced through stream variable α forces the term to be of type $\bot$ which is incompatible (see rule μAbs in figure 2).

Making streams first-class citizens in the typed setting. The stream mechanism that was used in the untyped calculus in order to obtain separation is thus desactivated when classical types are reintroduced. We shall look for a variant of this type system that would reflect in types the stream construction. In particular, since μ is seen as a stream abstraction, one might think of a functional type for streams: if term M is of type $\mathcal{T}$ when stream α is of stream type $\mathcal{S}$, then $\mu\alpha.M$ would be of the type of a stream function from $\mathcal{S}$ to $\mathcal{T}$ (that we write

$\mathcal{S} \Rightarrow \mathcal{T}$). We can thus think of the following typing rules for μ-abstracted terms in $\Lambda\mu$-calculus:

$$\frac{\Gamma \vdash M \ : \mathcal{T} | \Delta, \alpha : \mathcal{S}}{\Gamma \vdash \mu\alpha.M \ : \mathcal{S} \Rightarrow \mathcal{T} | \Delta} \ Abs_{\mathcal{S}} \qquad \frac{\Gamma \vdash M \ : \mathcal{S} \Rightarrow \mathcal{T} | \Delta, \alpha : \mathcal{S}}{\Gamma \vdash (M)\alpha \ : \mathcal{T} | \Delta, \alpha : \mathcal{S}} \ App_{\mathcal{S}}$$

A type mismatch. Rule *fst* does complicate the definition of a type system for $\Lambda\mu$ that would take streams into account: whereas $\mu\alpha.M$ is of a stream type, say $\mathcal{S} \Rightarrow \mathcal{T}$, the term resulting from $\mu\alpha.M$ by applying the *fst* rule once (namely $\lambda x.\mu\beta.M[(U)x\beta/(U)\alpha]$) should be of a standard function type $A \to B$ (more precisely $A \to (\mathcal{S}' \Rightarrow \mathcal{T}')$). Moreover, things should not be as simple as in the previous paragraph since streams are streams of terms and thus they should be related to each other and they should not leave in distinct worlds: one should be allowed to apply a term to a stream function (for instance $(\mu\alpha.M)N$) and conversely, one might want to apply a stream to a λ-abstracted term (for instance $(\lambda x.M)\alpha$). $\Rightarrow$-types and $\to$-types should be related in some way. *fst* gives the key to this connection; we thus analyze more carefully this rule in the following paragraph.

A relation over stream types. We recall that *fst* synthesized in $\Lambda\mu$-calculus as the result of a η-expansion and a μ-reduction. In the typed case, the η-expansion can occur only on $\to$-type terms. This restriction adapted to $\Lambda\mu$-calculus results in the condition that $\mu\alpha.M$ is of a stream type of the form $(\mathcal{T} \to \mathcal{S}) \Rightarrow \mathcal{T}'$. After an application of *fst*, we have term $\lambda x.\mu\beta.M[(U)x\beta/(U)\alpha]$ that should be of type $\mathcal{T} \to (\mathcal{S} \Rightarrow \mathcal{T}')$.

Simply typed streams: $\Lambda_{\mathcal{S}}$. We now define more formally the type system $\Lambda_{\mathcal{S}}$ for $\Lambda\mu$-calculus. The types are produced by the following grammar of simple types:

$$\begin{array}{ll} \textbf{Term types:} & \mathcal{T}, A, B, \ldots ::= o \mid A \to B \mid \mathcal{S} \Rightarrow \mathcal{T} \\ \textbf{Stream types:} & \mathcal{S}, P, Q, \ldots ::= \bot \mid \mathcal{T} \to \mathcal{S} \end{array}$$

In addition, we consider the congruence relation $\equiv_{fst}$ over Term types which is the symmetric, reflexive and transitive closure of relation $\succ_{fst}$ defined by $(\mathcal{T} \to \mathcal{S}) \Rightarrow \mathcal{T}' \succ_{fst} \mathcal{T} \to (\mathcal{S} \Rightarrow \mathcal{T}')$ and we always consider the types of $\Lambda_{\mathcal{S}}$ up to this congruence relation. We show in figure 3 the type system $\Lambda_{\mathcal{S}}$ for $\Lambda\mu$-calculus.

In the typed case, application of the *fst* rule to term M requires that M has a type in relation with a type of shape $(\mathcal{T}_1 \to \mathcal{S}) \Rightarrow \mathcal{T}_2$ (this requirement is similar to the condition on the η-expansion application in simply typed λ-calculus).

More detailed study of properties of type system $\Lambda_{\mathcal{S}}$ is postponed to expanded versions of this works. Let us just note before going to our conclusion that $\Lambda_{\mathcal{S}}$ type system is very much related with a type system that has been independently introduced by Herbelin and Ghilezan [7] when they were studying the relationships between $\Lambda\mu$-calculus and programming languages with delimited control operators. They avoid having an equivalence relation over types by using more complexed typing judgements so that $\Rightarrow$ is not really a connective in their

$$\frac{}{\Gamma, x : \mathcal{T} \vdash x \ : \mathcal{T} | \Delta} \ Var_{\mathcal{T}}$$

$$\frac{\Gamma, x : \mathcal{T} \vdash M \ : \mathcal{T}' | \Delta}{\Gamma \vdash \lambda x.M \ : \mathcal{T} \to \mathcal{T}' | \Delta} \ Abs_{\mathcal{T}} \qquad \frac{\Gamma \vdash M \ : \mathcal{T} \to \mathcal{T}' | \Delta \quad \Gamma \vdash N \ : \mathcal{T} | \Delta}{\Gamma \vdash (M)N \ : \mathcal{T}' | \Delta} \ App_{\mathcal{T}}$$

$$\frac{\Gamma \vdash M \ : \mathcal{T} | \Delta, \alpha : \mathcal{S}}{\Gamma \vdash \mu\alpha.M \ : \mathcal{S} \Rightarrow \mathcal{T} | \Delta} \ Abs_{\mathcal{S}} \qquad \frac{\Gamma \vdash M \ : \mathcal{S} \Rightarrow \mathcal{T} | \Delta, \alpha : \mathcal{S}}{\Gamma \vdash (M)\alpha \ : \mathcal{T} | \Delta, \alpha : \mathcal{S}} \ App_{\mathcal{S}}$$

Fig. 3. Λ_S, a type system for $\Lambda\mu$-calculus.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathcal{T}_x \vdash x \ : \mathcal{T}_x |} \ Var_{\mathcal{T}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{y : \mathcal{S}_\alpha \Rightarrow \mathcal{A} \vdash y \ : \mathcal{S}_\alpha \Rightarrow \mathcal{A}|} \ Var_{\mathcal{T}}}{y : \mathcal{S}_\alpha \Rightarrow \mathcal{A} \vdash (y)\alpha \ : \mathcal{A} | \alpha : \mathcal{S}_\alpha} \ App_{\mathcal{S}}}{y : \mathcal{S}_\alpha \Rightarrow \mathcal{A} \vdash \mu\beta.(y)\alpha \ : \mathcal{S}_\beta \Rightarrow \mathcal{A} | \alpha : \mathcal{S}_\alpha} \ Abs_{\mathcal{S}}}{\vdash \lambda y.\mu\beta.(y)\alpha \ : (\mathcal{S}_\alpha \Rightarrow \mathcal{A}) \to (\mathcal{S}_\beta \Rightarrow \mathcal{A}) | \alpha : \mathcal{S}_\alpha} \ Abs_{\mathcal{T}}}{x : ((\mathcal{S}_\alpha \Rightarrow \mathcal{A}) \to (\mathcal{S}_\beta \Rightarrow \mathcal{A})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B}) \vdash (x)\lambda y.\mu\beta.(y)\alpha \ : \mathcal{S}_\alpha \Rightarrow \mathcal{B} | \alpha : \mathcal{S}_\alpha} \ App_{\mathcal{T}}}{x : ((\mathcal{S}_\alpha \Rightarrow \mathcal{A}) \to (\mathcal{S}_\beta \Rightarrow \mathcal{A})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B}) \vdash ((x)\lambda y.\mu\beta.(y)\alpha)\alpha \ : \mathcal{B} | \alpha : \mathcal{S}_\alpha} \ App_{\mathcal{S}}}{x : ((\mathcal{S}_\alpha \Rightarrow \mathcal{A}) \to (\mathcal{S}_\beta \Rightarrow \mathcal{A})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B}) \vdash \mu\alpha.((x)\lambda y.\mu\beta.(y)\alpha)\alpha \ : \mathcal{S}_\alpha \Rightarrow \mathcal{B}|} \ Abs_{\mathcal{S}}}{\vdash \texttt{call/cc} \ : (((\mathcal{S}_\alpha \Rightarrow \mathcal{A}) \to (\mathcal{S}_\beta \Rightarrow \mathcal{A})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B})|} \ Abs_{\mathcal{T}}$$

with $\mathcal{T}_x = ((\mathcal{S}_\alpha \Rightarrow \mathcal{A}) \to (\mathcal{S}_\beta \Rightarrow \mathcal{A})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B})$.

Fig. 4. $\Lambda_{\mathcal{S}}$ type derivation for `call/cc`.

type system.

Remarks on the $\Rightarrow$ connective and the *fst* rule. Contrarily to what the notation $\Rightarrow$ may suggest, no duality is involved with this connective. The rule $Abs_{\mathcal{S}}$ would rather suggest the $\Rightarrow$ connective to be related with the $\parr$ connective of linear logic. This is precisely what we evidence in a current work with Michele Pagani: when translating Λ_S into polarized proof nets, $T_1 \to T_2$ becomes as usual $?T_1^\perp \parr T_2$ while $S \Rightarrow T$ is translated into $S \parr T$. The $\equiv_{fst}$ is thus an associativity property (namely $(?T^\perp \parr S) \parr T \equiv_{fst} ?T^\perp \parr (S \parr T)$) of $\parr$ which is perfectly sound logically.

Typing `call/cc` in $\Lambda_{\mathcal{S}}$. The $\Lambda\mu$-calculus encoding of `call/cc` is the term $\lambda x.\mu\alpha.((x)\lambda y.\mu\beta.(y)\alpha)\alpha$. In the classical type system presented in figure 2, this term is typed by the Peirce's Law: $((A \to B) \to A) \to A$. In $\Lambda_{\mathcal{S}}$, `call/cc` can be assigned type $(((\mathcal{S}_\alpha \Rightarrow \mathcal{A}) \to (\mathcal{S}_\beta \Rightarrow \mathcal{A})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B})) \to (\mathcal{S}_\alpha \Rightarrow \mathcal{B})$ as shown by the type derivation in figure 4. One may notice that the structure of the the Peirce's Law is now to be found in the stream type (see the alternation of $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ types).

4 Conclusion: Towards typed separation in $\Lambda\mu$-calculus

In the present paper, we introduced $\Lambda_{\mathcal{S}}$, a simple type system for $\Lambda\mu$-calculus that does not desactivate the stream constructions which are at the heart of the untyped separation result for $\Lambda\mu$-calculus [11]. Until now, the only type system available for $\Lambda\mu$-calculus did not allowed to use $\Lambda\mu$-calculus stream mechanism: indeed, the standard type system for $\lambda\mu$-calculus which can serve as a type system for $\Lambda\mu$-calculus forbids the most interesting computational behaviours of $\Lambda\mu$-calculus.

Future works: towards typed separation. We consider the introduction of $\Lambda_{\mathcal{S}}$ as the first step towards obtaining a typed separation result for $\Lambda\mu$-calculus (for instance using techniques of the kind Dosen and Petric use in their proof of typed separation for λ-calculus [5]). Indeed, while classical $\lambda\mu$-calculus typing forbids constructions that are essential for separation to hold, $\Lambda_{\mathcal{S}}$ is a type system that takes $\Lambda\mu$-calculus streams as first-class citizens. On the other hand, we are currently investigating with Pagani the links between $\Lambda\mu$-calculus and polarized proof nets.

Acknowledgments: The author wishes to thank Hugo Herbelin and Michele Pagani for valuable discussions and helpful comments.

References

1. Corrado Böhm. Alcune proprietà delle forme $\beta\eta$-normali nel λk-calcolo. *Pubblicazioni dell'Istituto per le Applicazioni del Calcolo*, 696, 1968.
2. René David and Walter Py. $\lambda\mu$-calculus and Böhm's theorem. *Journal of Symbolic Logic*, 2001.
3. Philippe de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *LPAR'94*, volume 822 of *LNAI*, 1994.
4. Philippe de Groote. An environment machine for the $\lambda\mu$-calculus. *MSCS*, 8, 1998.
5. Kosta Dosen and Zoran Petrić. The typed Böhm theorem. *ENTCS*, 2001. in Proceedings of BOHM00.
6. Timothy Griffin. A formulae-as-types notion of control. In *POPL'90*, 1990.
7. Hugo Herbelin and Silvia Ghilezan. An approach to call-by-name delimited continuations. submitted, july 2007.
8. Thierry Joly. *Codages, séparabilité et représentation de fonctions en λ-calcul simplement typé et dans d'autres systèmes de types.* PhD thesis, Université Paris VII, 2000.
9. Jean-Louis Krivine. *Lambda-calculus, Types and Models.* Ellis Horwood, 1993.
10. Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *ICLP'92*, LNCS, 1992.
11. Alexis Saurin. Separation with streams in the $\Lambda\mu$-calculus. In *LICS'05*, 2005.