A foundational calculus for computing with streams

Marco Gaboardi[1] and Alexis Saurin[2*]

[1] Dip. di Scienze dell'Informazione, Università di Bologna & INRIA Focus, Italy
gabordi@cs.unibo.it
[2] Laboratoire PPS, Université Paris-Diderot & INRIA πr^2, France
alexis.saurin@pps.jussieu.fr

Abstract. Computing with streams is a topic of increasing interest in modern computer science. Several proposals have suggested the use of programming languages and rewriting systems in order to formally describe programs computing over stream data. However, no stream calculus (in the sense of λ-calculus) has been exhibited for such studies yet.
The challenge of providing a calculus for streams is to make possible a more general study than what is possible with stream programming languages (since no reduction strategy is fixed to start with) and more modular and compositional than stream rewriting systems (indeed a new rewriting system shall be designed for each problem). In the present work, starting from the $\Lambda\mu$-calculus (an extension of λ-calculus for classical logic) we design $\Lambda_{\mathcal{S}}$ which is a calculus where streams are first-class citizens. This calculus enjoys several interesting syntactical and computational properties and, moreover, it is well adapted to the use of types in order to prove stream program properties.

Extended Abstract

Stream-like data structures are ubiquitous in computer science. By representing discrete potentially infinite information flows, streams can be used to formalize and study several situations of increasing significance in modern computer science, such as network communication flows, audio and video signals flows, etc. Additionally, stream-like data can be used in order to model the potentially infinite interactions between a program and its environment.
For these reasons it is important to study and develop formal techniques for the specification and the analysis of programs working on streams. Several interesting results in this direction have been already obtained in the rewriting systems and functional programming settings respectively. In the rewriting system setting many techniques have been developed in order to study infinitary rewriting and its properties. Analogously, in the setting of functional programming languages, many investigations on the notion of laziness has brought to an elegant

* This work started while the second author was a postdoctoral fellow at Università di Torino and was funded by a grant Lavoisier.

treatment of stream programs and their properties.
at the present a foundational functional calculus, in the spirit of λ-calculus, where streams are first class citizens is missing. One such a calculus could be a unified framework where to study stream programs in a modular and compositional way, without restricting attention to a pre-determined evaluation mechanism.
The starting point of our investigation is $\Lambda\mu$-calculus [3], a calculus extending Parigot's $\lambda\mu$-calculus in order to obtain separability. In 1992, Parigot proposed an extension of λ-calculus providing "an algorithmic interpretation of classical natural deduction" [2]: $\lambda\mu$-calculus is in Curry-Howard correspondence with classical natural deduction. In addition to bridging classical reasoning with computation allowing (non-delimited) control primitives, $\lambda\mu$-calculus is a simple and elegant extension of Church's λ-calculus which retains most of the standard properties of λ-calculus such as confluence,subject reduction and SN. A noticeable exception to this was Böhm theorem [1] which fails in Parigot's calculus. This led the second author to introduce an extension to Parigot's calculus, $\Lambda\mu$-calculus, for which he proved Böhm theorem [3].
$\Lambda\mu$-calculus has a more liberal syntax than Parigot's $\lambda\mu$-calculus. This results in more computational contexts and thus allows to achieve a Böhm out process in $\Lambda\mu$. $\Lambda\mu$-terms ($t, u, v \dots \in \Sigma_{\Lambda\mu}$) are defined by the following syntax:

$$\boxed{\Sigma_{\Lambda\mu} \qquad t, u ::= \; x \mid \lambda x.t \mid (t)u \mid \mu\alpha.t \mid (t)\alpha}$$

where x (resp. α) ranges over an infinite set $\mathcal{V}_t$ (resp. $\mathcal{V}_s$) of term (resp. stream) variables. $\mathcal{V}_t$ and $\mathcal{V}_s$ are disjoint. The set of closed $\Lambda\mu$-terms is denoted by $\Sigma^c_{\Lambda\mu}$. The $\Lambda\mu$-reduction, written $\longrightarrow_{\Lambda\mu}$, is induced by the following rules:

$$\boxed{\begin{array}{llll}
(\lambda x.t)u & \longrightarrow_{\beta_T} & t\{u/x\} & \\
\lambda x.(t)x & \longrightarrow_{\eta_T} & t & \text{if } x \notin FV(t) \\
(\mu\alpha.t)\beta & \longrightarrow_{\beta_S} & t\{\beta/\alpha\} & \\
\mu\alpha.(t)\alpha & \longrightarrow_{\eta_S} & t & \text{if } \alpha \notin FV(t) \\
\mu\alpha.t & \longrightarrow_{fst} & \lambda x.\mu\alpha.t\{(v)x\alpha/(v)\alpha\} & \text{if } x \notin FV(t)
\end{array}}$$

where $n\{(v)u\alpha/(v)\alpha\}$ substitutes (without variable-capture) every named term $(v)\alpha$ in n by $(v)u\alpha$.
The $\Lambda\mu$-calculus can be viewed under a stream operation interpretation. In particular, a term $\mu\alpha.t$ can be considered as an abstraction over streams of terms (*i.e.* $\lambda x_1^\alpha \dots x_n^\alpha \dots t$) while a term $(t)\alpha$ can be view as the construction passing the stream α as an argument to term t (*i.e.* $(t)x_1^\alpha \dots x_n^\alpha \dots$). Thus, μ can be viewed as a sort of infinitary λ-abstraction but, as suggested by the second author in [3, 4] in a much wider sense than what Parigot remarked in his seminal paper [2].

Under this interpretation one wants to show that $\Lambda\mu$ constructions permits to deal with streams as first class citizens. For instance, one wants to deal with standard functions working in finite time on streams and proving their properties by means of types. Unfortunately, in order to do this, one needs to overcome some difficulties. In the $\Lambda\mu$-calculus syntax, streams objects cannot be in head position

since they are permitted only in argument position. As a consequence we have that all the standard stream functions acting on stream and returning streams cannot directly be recovered.
To better understand this point consider the destructors `head (x:xs) = x` and `tail (x:xs) = xs`. Clearly we can easily define a term for the `head` function as follows: `head` $= \lambda x.\mu\alpha.x$. Analogously, we would define a term for the `tail` function as `tail` $= \lambda x.\mu\alpha.\alpha$ but unfortunately this is forbidden by $\Lambda\mu$-calculus grammar. In fact, in $\Lambda\mu$-calculus, there are two distinct syntactical categories: *streams* and *terms*. The former is only represented by stream variables α that can be used in the construction $(t)\alpha$. If we wish really to compute with streams, we shall be able to return a stream as the result of a computation. Since this is not directly possible in $\Lambda\mu$, so an extension is needed. For this reason we introduce the $\Lambda_{\mathcal{S}}$-calculus. The syntax of $\Lambda_{\mathcal{S}}$-calculus is defined as follows:

$$\boxed{\begin{array}{lcr} \Sigma_{\Lambda\mathcal{S}} & \ni & t, u ::= x \mid \lambda x.t \mid (t)u \mid \mu\alpha.t \mid (t)\mathbb{S} \mid \tau(\mathbb{S}) \\ & & \mathbb{S} ::= \alpha \mid [t \mid \mathbb{S}] \mid \sigma(t) \end{array}}$$

The grammar of $\Lambda_{\mathcal{S}}$ differs from the one of $\Lambda\mu$ since we now have a new construction $[- \mid -]$ that permits to build actual streams $[u \mid \mathbb{S}]$ and two term/stream coercions $\tau(-)$ and $\sigma(-)$. The coercion $\tau(-)$ builds a term from a stream, for instance as $\tau(\alpha)$, and $\sigma(-)$ which builds a stream from a term, for instance $\sigma((t)u)$. The reduction for $\Lambda_{\mathcal{S}}$-calculus is obtained by the following rules:

$$\boxed{\begin{array}{rll} (\lambda x.t)u & \longrightarrow_{\beta_{(\mathcal{T})\mathcal{T}}} & t\,\{u/x\} \\ (\mu\alpha.t)\mathbb{S} & \longrightarrow_{\beta_{(\mathcal{S})\mathcal{S}}} & t\,\{\mathbb{S}/\alpha\} \\ (\lambda x.t)[u \mid \mathbb{S}] & \longrightarrow_{\beta_{(\mathcal{T})\mathcal{S}}} & (t\,\{u/x\})\mathbb{S} \\ (\mu\alpha.t)u & \longrightarrow_{\beta_{(\mathcal{S})\mathcal{T}}} & \mu\beta.t\,\{[u \mid \beta]/\alpha\} \\ \mu\alpha.t & \longrightarrow_{fst} & \lambda x.\mu\beta.t\,\{[x \mid \beta]/\alpha\} \text{ if } x, \beta \notin FV(t) \\ \tau(\sigma(t)) & \longrightarrow_{\tau\sigma} & t \\ \hline \sigma(\tau(\mathbb{S})) & \longrightarrow_{\sigma\tau} & \mathbb{S} \end{array}}$$

Thanks to the extensions given above we can now define a term acting as `tail` by using the coercion $\tau(-)$ turning a stream into a term: `tail` $= \lambda x.\mu\alpha.\tau(\alpha)$. Analogously the coercion $\sigma(-)$ turns a term into a stream. Moreover, the construction $[- \mid -]$ can be used to build explicitly streams, *i.e.* given a term t and a stream $\mathcal{S}$ it returns a stream $[t \mid \mathcal{S}]$. To better illustrate their behaviour consider a toy function `F` defined as: `F (x:xs)` $=$ `(x:(tail xs))` A first way to define in $\Lambda_{\mathcal{S}}$-calculus a function acting as `F` is by using the term $\lambda x.\lambda y.\mu\alpha.\tau([x \mid \alpha])$ Such a term has the expected behaviour but it does not reflect the compositional meaning of the above definition, *i.e.* it reflects instead a direct definition as `F' (x:y:xs)` $=$ `x:xs`. So, in order to have a term reflecting the above definition in the $\Lambda_{\mathcal{S}}$-calculus we can consider the term: `F` $= \lambda x.\mu\alpha.\tau([x \mid \sigma((\texttt{tail})\alpha)])$ Since $(\texttt{tail})\alpha$ is a term, we need the coercion σ in order to turn it in a stream.

$$\frac{}{\Gamma, x:\mathcal{T} \vdash_t x \ : \mathcal{T}|\Delta}\,Var_{\mathcal{T}} \qquad \frac{}{\Gamma \vdash_s \alpha \ : \mathcal{S}|\Delta, \alpha:\mathcal{S}}\,Var_{\mathcal{S}}$$

$$\frac{\Gamma \vdash_t t \ : \mathcal{T}|\Delta}{\Gamma \vdash_t t \ : \mathcal{T}'|\Delta}\equiv_{fst} \quad \text{(provided } \mathcal{T} \equiv_{fst} \mathcal{T}'\text{)}$$

$$\frac{\Gamma, x:\mathcal{T} \vdash_t t \ : \mathcal{T}'|\Delta}{\Gamma \vdash_t \lambda x^{\mathcal{T}}.t \ : \mathcal{T} \to \mathcal{T}'|\Delta}\,Abs_{\mathcal{T}} \qquad \frac{\Gamma \vdash_t t \ : \mathcal{T} \to \mathcal{T}'|\Delta \qquad \Gamma \vdash_t u \ : \mathcal{T}|\Delta}{\Gamma \vdash_t (t)u \ : \mathcal{T}'|\Delta}\,App_{\mathcal{T}}$$

$$\frac{\Gamma \vdash_t t \ : \mathcal{T}|\Delta, \alpha:\mathcal{S}}{\Gamma \vdash_t \mu\alpha^{\mathcal{S}}.t \ : \mathcal{S} \Rightarrow \mathcal{T}|\Delta}\,Abs_{\mathcal{S}} \qquad \frac{\Gamma \vdash_t t \ : \mathcal{S} \Rightarrow \mathcal{T}|\Delta \qquad \Gamma \vdash_s \mathbb{S} \ : S|\Delta}{\Gamma \vdash_t (t)\mathbb{S} \ : \mathcal{T}|\Delta}\,App_{\mathcal{S}}$$

$$\frac{\Gamma \vdash_t t \ : \mathcal{T}|\Delta \qquad \Gamma \vdash_s \mathbb{S} \ : \mathcal{S}|\Delta}{\Gamma \vdash_s [t \mid \mathbb{S}] \ : \mathcal{T} \to \mathcal{S}|\Delta}\,\text{Stream}$$

$$\frac{\Gamma \vdash_s \mathbb{S} \ : \mathcal{S}|\Delta}{\Gamma \vdash_t \tau(\mathbb{S}) \ : \mathtt{Str}(\mathcal{S})|\Delta}\,\text{Coerc}_{\tau} \qquad \frac{\Gamma \vdash_t t \ : \mathtt{Str}(\mathcal{S})|\Delta}{\Gamma \vdash_s \sigma(t) \ : \mathcal{S}|\Delta}\,\text{Coerc}_{\sigma}$$

$$\frac{\Gamma \vdash_s \mathbb{S} \ : \mathcal{S}\left\{\mu\mathcal{X}.\mathcal{S}/\mathcal{X}\right\}|\Delta}{\Gamma \vdash_s \mathbb{S} \ : \mu\mathcal{X}.\mathcal{S}|\Delta}\,\mu\text{-intro} \qquad \frac{\Gamma \vdash_s \mathbb{S} \ : \mu\mathcal{X}.\mathcal{S}|\Delta}{\Gamma \vdash_s \mathbb{S} \ : \mathcal{S}\left\{\mu\mathcal{X}.\mathcal{S}/\mathcal{X}\right\}|\Delta}\,\mu\text{-elim}$$

Fig. 1. The type system for $\Lambda_{\mathcal{S}}$-calculus.

Analogously, since $[x \mid \sigma((\mathtt{tail})\alpha)]$ is a stream, we need the coercion τ in order to turn it in a term.

Besides the definition of the $\Lambda_{\mathcal{S}}$-calculus, we prove that it enjoys some important syntactic and computational properties. In particular, confluence and standardization with respect to head reduction. The fact that such properties hold shows that we can use the $\Lambda_{\mathcal{S}}$-calculus concretely to model stream program computations. Moreover, we investigate the $\Lambda_{\mathcal{S}}$-calculus expressivity by showing several standard examples of both functions defining streams and functions working on streams. Finally, we show that $\Lambda_{\mathcal{S}}$-calculus terms can be typed by means of the type system depicted in Figure 1 and we show that it enjoys standard properties as subject reduction.

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. Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of the 1992 International Conference on Logic Programming and Automated Reasonning*, volume 624 of *Lecture Notes in Computer Science*, London, UK, 1992. Springer-Verlag.
3. Alexis Saurin. Separation with streams in the $\Lambda\mu$-calculus. In *Logic In Computer Science*, pages 356–365, Chicago, 2005. IEEE Computer Society Press.
4. Alexis Saurin. Typing streams in the $\Lambda\mu$-calculus. *ACM Transactions on Computational Logic*, 2009. to appear.