Chapter 1

PLAYFUL, STREAMLIKE COMPUTATION

Pierre-Louis Curien

PPS
CNRS & Université Paris 7
Case 7014, 2 pl. Jussieu, 75251 Paris Cedex 05, France
curien@pps.jussieu.fr

Abstract

We offer a short tour into the interactive interpretation of sequential programs. We emphasize streamlike computation — that is, computation of successive bits of information upon request. The core of the approach surveyed here dates back to the work of Berry and the author on sequential algorithms on concrete data structures in the late seventies, culminating in the design of the programming language CDS, in which the semantics of programs of any type can be explored interactively. Around one decade later, two major insights of Cartwright and Felleisen on one hand, and of Lamarche on the other hand gave new, decisive impulses to the study of sequentiality. Cartwright and Felleisen observed that sequential algorithms give a direct semantics to control operators like call-cc and proposed to include explicit errors both in the syntax and in the semantics of the language PCF. Lamarche (unpublished) connected sequential algorithms to linear logic and games. The successful program of games semantics has spanned over the nineties until now, starting with syntax-independent characterizations of the term model of PCF by Abramsky, Jagadeesan, and Malacaria on one hand, and by Hyland and Ong on the other hand.

Only a basic acquaintance with λ-calculus, domains and linear logic is assumed in sections 1 through 3.

Keywords: Coroutines, sequentiality, games, abstract machines, Böhm trees

1. Prologue: playing with Böhm trees

We first make some preparations. For self-containedness, we briefly recall the relevant notions. The syntax of the untyped λ-calculus (λ-calculus for short) is given by the following three constructions: a vari-

able \( x \) is a \( \lambda \)-term, if \( M \) and \( N \) are \( \lambda \)-terms, then the application \( MN \) is a \( \lambda \)-term, and if \( M \) is a term then the abstraction \( \lambda x.M \) is a term. Usual abbreviations are \( \lambda x_1 x_2 M \) for \( \lambda x_1.(\lambda x_2.M) \), and \( MN_1N_2 \) for \( (MN_1)N_2 \), and similarly for \( n \)-ary abstraction and application. A more macroscopic view is quite useful: it is easy to check that any \( \lambda \)-term has exactly one of the following two forms:

\[
(n \geq 1, p \geq 1) \quad \lambda x_1 \cdots x_n.xM_1 \cdots M_p
\]

\[
(n \geq 0, p \geq 1) \quad \lambda x_1 \cdots x_n. (\lambda x.M)M_1 \cdots M_p .
\]

The first form is called a head normal form (hnf), while the second exhibits the head redex \( (\lambda x.M)M_1 \). The following easy property justifies the name of head normal form: any reduction sequence starting from a hnf \( \lambda x_1 \cdots x_n.xM_1 \cdots M_p \) consists of an interleaving of independent reductions of \( M_1, \ldots, M_p \). More precisely, we have:

\[
(\lambda x_1 \cdots x_n.xM_1 \cdots M_p \to^* P) \Rightarrow
\]

\[
\exists N_1, \ldots, N_p \left\{ \begin{array}{l}
P = \lambda x_1 \cdots x_n.xN_1 \cdots N_p \\
\forall i \leq p \ M_i \to^* N_i .
\end{array} \right.
\]

Here, reduction means the replacement in any term of a sub-expression of the form \( (\lambda x.M)N \), called a \( \beta \)-redex, by \( M[x \leftarrow N] \). A normal form is a term that contains no \( \beta \)-redex, or equivalently that contains no head redex. Hence the syntax of normal forms is given by the following two constructions: a variable \( x \) is a normal form, and if \( M_1, \ldots, M_p \) are normal forms, then \( \lambda x_1 \cdots x_n.xM_1 \cdots M_p \) is a normal form.

Now, we are ready to play. Consider the following two normal forms:

\[
M = zM_1M_2(\lambda z_1z_2.z_1M_3M_4) \quad N = \lambda x_1x_2x_3.x_3(\lambda y_1y_2.y_1N_1)N_2 .
\]

The term \( M[z \leftarrow N] = NM_1M_2(\lambda z_1z_2.z_1M_3M_4) \) is not a normal form anymore, and can be \( \beta \)-reduced as follows:

\[
NM_1M_2(\lambda z_1z_2.z_1M_3M_4) \to (\lambda z_1z_2.z_1M_3M_4)(\lambda y_1y_2.y_1N_1)N_2' \\
\to (\lambda y_1y_2.y_1N_1')M_3'M_4' \\
\to M_3'N_1'' ,
\]

where \( N_1' \), etc... are suitable substitution instances of \( N_1 \) etc... But there is a more geometric way of describing the interaction of \( M \) and \( N \). First, we represent \( M \) and \( N \) explicitly as trees (read from left to right), as follows:

\[
\begin{align*}
z & \left\{ \begin{array}{l} \\
M_1 & M_2 \\
\lambda z_1z_2. & M_3 \\
M_1 & M_4 \\
\end{array} \right. \\
\lambda x_1x_2x_3. & \left\{ \begin{array}{l} \\
\lambda y_1y_2. & y_1 \\
N_1 & N_2 \end{array} \right.
\end{align*}
\]