Investigations into the Duality of Computation

Hugo Herbelin
LIX - INRIA-Futurs - PCRI

Abstract

The work presented here is an extension of a previous work realised jointly with Pierre-Louis Curien [CH00]. The current work focuses on the pure calculus of variables and binders that operates at the core of the duality between call-by-name and call-by-value evaluations. A Curry-Howard-de Bruijn correspondence is given that shed light on some aspects of Gentzen's sequent calculus. This includes a sequent-free presentation of it.

Introduction

Call-by-name and call-by-value When evaluating a program $f(e)$ where f is a function of some parameter x, there are basically two different strategies. Either f is evaluated and, each time its argument x is needed, e is evaluated, or, e is evaluated first, and, if ever f needs x, the value of e is used. In the first case, the argument e is passed by its name x: it is a call-by-name strategy of reduction. In the second case, the value of e is passed: it is a call-by-value strategy of reduction. Most of the actual programming languages obey a call-by-value strategy of reduction.

Church's λ-calculus is a model of computation which is both universal (it is assumed to be complete with respect to what is effectively computable) and remarkably simple to formulate. Long before the design of effective programming languages, call-by-name was studied as an equational theory by Church and Rosser. Call-by-value evaluation as a theory has first been investigated by Plotkin, then by Moggi. Note that in λ-calculus, the difference between call-by-value and call-by-name is observable only when considering non-terminating objects.

Control operators From 1965, Landin, then Reynolds, explored programming constructs able to manipulate the execution stack. Some of these operators, called control operators, have been implemented in effective programming languages. This is the case of the operator `call-with-current-continuation` to be found in Scheme. The theory of control operators have been first investigated by Felleisen *et al* in 1986.

In presence of control operators, the semantical difference of call-by-name and call-by-value can be observed, even in presence of terminating programs only.

Continuation-passing-style (CPS) transformations provide a way to simulate the behaviour of control operators within Church's call-by-name lambda-calculus. CPS-transformations are the computational counterparts of Kolmogorov-style double negation translations [Mur92]. There are CPS for simulating call-by-name and CPS for simulating call-by-value.

The call-by-name/call-by-value duality In 1989, Filinski designed a symmetric syntax for λ-calculus that he equipped with two dual CPS semantics that express the choices between call-by-name and call-by-value. This was the first explicit work showing (at a semantical level) a duality between the two notions.

In 2000, Selinger designed dual notions of control and co-control categories that characterise the equational theory of Parigot's $\lambda\mu$-calculus, in its call-by-name and call-by-value variants.

In 2000, Curien and Herbelin designed a symmetric λ-calculus exhibiting a duality between terms and evaluation contexts. This calculus was equipped with a symmetric but non-deterministic reduction system such that solving the non-determinism amounted to falling either in the call-by-name theory or in its syntactically dual call-by-value theory.

The main kind of logical formalisms The first formal systems were axiomatic: the only inference rule in the propositional case was modus ponens and the properties of connectives were expressed by axioms. These are "Hilbert-style" formalisms.

In 1935, Gentzen introduced two major alternative formalisms. The first class of formalism is Natural Deduction. In Natural Deduction, the properties of connectives are expressed by means of introduction and elimination rules plus an extra axiom rule to express the fact that under some hypothetical reasoning, the result stated by the hypothesis holds. Gentzen represented the proofs in natural deduction by trees of formulae, each node being justified by an introduction or elimination rule and each leaf by an axiom rule.

The second class of formalism was called Logistic Calculus but, following Prawitz, it remains in the history with the name Gentzen's sequent calculus. The definition of Logistic Calculus relies on sequents that consist of a list of hypotheses and a list of conclusions. Sequents have become a standard way to represent the context of a derivation, whatever the logical formalism is, but at the time of Prawitz, "Logistic Calculus" was the only formalism to be expressed this way. Nowadays, it is no longer a discriminating feature. The properties of connective in "Gentzen's sequent calculus" are expressed by left introduction rules and by right introduction rules.

The property of sequent calculus we are interested in is its deep symmetry between the left and right hand side of the sequents.

Minimal, intuitionistic and classical logic Classical logic is a logic in which $\neg\neg A \rightarrow A$ holds for any formula A. The principle $\neg\neg A \rightarrow A$ is equivalent to the conjunction of Peirce's law, $((A \rightarrow B) \rightarrow A) \rightarrow A$ for any A and B and of the principle *ex falso quodlibet*, namely $\bot \rightarrow A$ for any A. Intuitionistic logic is a logic that does not validate Peirce's law but that validates *ex falso quodlibet* while minimal is usually considered as the logic that validates neither Peirce's law nor *ex falso quodlibet*. Following [AH03], we call minimal classical logic the logic that validates Peirce's law without validating *ex falso quodlibet*.

The Curry-Howard-de Bruijn's proofs-as-programs correspondence A major breakthrough of the late 70's is the emergence of the consciousness of an intimate identity between the structure of proofs and the one of programs.

This identity was first observed in 1958 by Curry who remarked a coincidence, both between the types of the basic combinators of combinatory logic and the form of the basic axioms of Hilbert's propositional calculus, and between the application construction of combinatory logic and the principle of modus ponens in logic.

The identity was then made explicit in 1969 by Howard for the proofs of intuitionistic Gentzen's natural deduction and Church's simply-typed λ-calculus. Independently, de Bruijn used λ-calculus for denoting the proofs of his AUTOMATH system.

In 1990, Griffin discovered that the $\mathcal{C}$ control operator, designed in 1986 by Felleisen, Friedman, Kohlberger and Duba, was typable with type $\neg\neg A \rightarrow A$, when used in a context of type $\bot$. This was the starting point for investigating the proof-as-programs correspondence beyond minimal logic. In 1992, Parigot designed the $\lambda\mu$-calculus which is a "clean" formulation of λ-calculus with control.

The computational content of sequent calculus Griffin's result brought the stimulus not only to investigate the computational content of classical logic, but also of other logical structures and axioms. In the mid 90's, various works proposed computational interpretations of Gentzen's sequent calculus. Kesner *et al* proposed an interpretation of the left introduction rules as pattern constructors [KPT96] while we proposed an approach based on the interpretation the left introduction rules as constructors of argument lists [Her95].

Outline Our work is at the intersection of these different subjects. We first exhibit a minimal syntax, the $\mu\tilde{\mu}$-subsystem, that expresses a syntactic duality between call-by-name and call-by-value. Then, we investigate the connections with $\lambda\mu$-calculus. Finally, we show that the $\mu\tilde{\mu}$-subsystem provides with a proof-as-correspondence with Gentzen's sequent calculus.

1 The $\mu\tilde{\mu}$-Subsystem: the Core of Computation

1.1 Syntax of the $\mu\tilde{\mu}$-subsystem

The $\mu\tilde{\mu}$-subsystem is a structure based on three syntactic categories: terms, evaluation contexts and commands. We call expressions the elements in the union of these categories.

We assume the existence of two infinite sets of variables $\mathcal{X}$ and $\mathcal{A}$ called the sets of term variables and of evaluation context variables respectively. We respectively denote these variables by names based on the Roman letters $x, y, z, \ldots$ and on the Greek letter $\alpha, \beta, \gamma, \ldots$

Terms and evaluation contexts are dual objects while commands link terms and evaluation contexts together. The syntax of the $\mu\tilde{\mu}$-subsystem is the following:

Commands	c	$::=$	$\langle v \| e\rangle$
Terms	v	$::=$	$\mu\alpha.c \mid x$
Evaluation contexts	e	$::=$	$\tilde{\mu}x.c \mid \alpha$

The $\mu\tilde{\mu}$-subsystem is a "classical" calculus. It is also a pure calculus without λ-abstraction nor λ-application. Up to the absence of the λ-constructions, the terms of the $\mu\tilde{\mu}$-subsystem are the named terms of $\lambda\mu$-calculus and the commands of the $\mu\tilde{\mu}$-subsystem have the same role as the unnamed terms of $\lambda\mu$-calculus. Synonyms for commands are *programs* or *executables* (see e.g. Krivine [Kri01]) or *states* (by analogy with the role of states in abstract evaluation devices, see e.g. Landin).

In the $\mu\tilde{\mu}$-subsystem, any computation is the result of an interaction between a term and an evaluation context. The expression $\mu\alpha.c$, when, in interaction with some evaluation context, binds this context to α and continues the computation with c.

The evaluation context variables play in the $\mu\tilde{\mu}$-subsystem the same role as they play in the $\lambda\mu$-calculus. They are bound to evaluation contexts and as such are the basic components of the syntactic category of evaluation contexts. The construction $\tilde{\mu}x.c$ builds a "let-in" evaluation contexts. When put into interaction with a term v, it binds v to x and continues the computation with c.

The binders $\mu\alpha.c$ and $\tilde{\mu}x.c$ induce a notion of α-conversion on the expressions of the $\mu\tilde{\mu}$-calculus and we reason from now up to α-conversion. The notions of free and bound variables are defined accordingly.

In the context of λ-calculus and $\lambda\mu$-calculus, evaluation contexts are often written using terms with holes and interactions come from the graft of a term in the hole of an evaluation context. Using this kind of notations, the syntax of the $\mu\tilde{\mu}$-subsystem can be rewritten as follows:

Syntactic rephrasing in $\lambda\mu$-calculus

Commands	c	$::=$	$e[v]$
Terms	v	$::=$	$\mu\alpha.c \mid x$
Evaluation contexts	$e[\,]$	$::=$	$\mathbf{let}\ x = [\,]\ \mathbf{in}\ c \mid [\alpha]([\,])$

This syntax may be useful to keep in mind for comparison but we prefer the symmetric syntax. First because of the elegance provided by its symmetry. Secondly because the interpretation of evaluation contexts as terms with holes is not structural and will not scale when the $\mu\tilde{\mu}$-subsystem will be extended with new constructions. Refining an evaluation context that is represented as a term with some hole needs to refine the hole deep inside the term, while we really expect in the extensions of the $\mu\tilde{\mu}$-subsystem to have algebraic constructors of evaluation contexts.

1.2 Semantics of the $\mu\tilde{\mu}$-subsystem

Evaluation is the result of an interaction between a term and an evaluation context. Either the term binds its evaluation context (this is the typical behaviour of a control operator), or the evaluation context binds the term that is given to it, as a "let-in context" does. Hence, the reduction rules of the $\mu\tilde{\mu}$-subsystem are the following:

$$\begin{array}{lccc} (\mu) & \langle \mu\alpha.c \| e\rangle & \rightarrow & c[\alpha \leftarrow e] \\ (\tilde{\mu}) & \langle v \| \tilde{\mu}x.c\rangle & \rightarrow & c[x \leftarrow v] \end{array}$$

where $c[\alpha \leftarrow e]$ and $c[x \leftarrow v]$ are capture-free substitutions.

The rule $(\tilde{\mu})$ is the "standard" reduction rules of the "let-in" construction. This is obvious from the rephrasing of the reduction rules in the syntax of $\lambda\mu$-calculus:

Syntactic rephrasing in $\lambda\mu$-calculus

$$\begin{array}{lccc} (\mu) & e[\mu\alpha.c] & \rightarrow & c[[\alpha]v \leftarrow e[v]] \\ (\tilde{\mu}) & \mathbf{let}\ x = v\ \mathbf{in}\ c & \rightarrow & c[x \leftarrow v] \end{array}$$

In the $\lambda\mu$-calculus, evaluation contexts are moved around piece by piece. Contrastingly, the rule (μ) of the $\mu\tilde{\mu}$-subsystem moves a whole evaluation context at once. Polonovski [Pol03] proved the following:

Proposition: *Rules (μ) and $(\tilde{\mu})$ together are terminating.*

1.3 The fundamental dilemma of computation

The two symmetric reduction rules (μ) and $(\tilde{\mu})$ overlap. The resulting theory, defined as the reflexive-symmetric-transitive congruent closure of (μ) and $(\tilde{\mu})$, is inconsistent as we can prove that any two commands are equal:

$$\langle v \| e \rangle \quad \leftarrow \quad \langle \mu\alpha.\langle v \| e \rangle \| \tilde{\mu}x.\langle v' \| e' \rangle \rangle \quad \rightarrow \quad \langle v' \| e' \rangle$$

where x and α are fresh variables, not occurring in v, v', e and e'.

The two canonical, symmetrical, ways to solve the critical pair lead to either call-by-value or call-by-name reduction. The terminology comes from the point of view of the term variables. In the critical pair $\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle$, if the term is substituted as such without first evaluating it, i.e. if rule $(\tilde{\mu})$ is applied, then the reference to x is "by name". Otherwise, if we require that the term is first evaluated before binding it to x, then it is a call "by value". Of course, if the evaluation of the term does not terminate (assuming the language is rich enough to have non termination), or the evaluation of the term removes its evaluation context (as control operators can do), then the term never gets bound to x. For comparison matters, here is how the critical pair is expressed in the syntax of $\lambda\mu$-calculus:

$$c[[\alpha]v \leftarrow \textbf{let } x = v \textbf{ in } c'] \quad \overset{(\mu)}{\leftarrow} \quad \textbf{let } x = \mu\alpha.c \textbf{ in } c' \quad \overset{(\tilde{\mu})}{\leftarrow} \quad c[x \leftarrow \mu\alpha.c]$$

1.4 Call-by-name and call-by-value reduction

Call-by-name is obtained by restricting the rule (μ) to a new category of linear evaluation contexts. Call-by-value is dually obtained by restricting the rule $(\tilde{\mu})$ to a new class of linear terms, or synonymously, values. The syntax and semantics of the call-by-name and call-by-value $\mu\tilde{\mu}$-subsystems are the following (the dots indicate where the extensions of the $\mu\tilde{\mu}$-subsystem take place):

Call-by-name

Commands	c	$::=$	$\langle v \| e \rangle$
Terms	v	$::=$	$\mu\alpha.c \mid x \mid \ldots$
Linear ev. contexts	E	$::=$	$\alpha \mid \ldots$
Evaluation contexts	e	$::=$	$\tilde{\mu}x.c \mid E$

$$\begin{array}{lllll} (\mu_n) & \langle \mu\alpha.c \| E \rangle & \rightarrow & c[\alpha \leftarrow E] \\ (\tilde{\mu}) & \langle v \| \tilde{\mu}x.c \rangle & \rightarrow & c[x \leftarrow v] \end{array}$$

Call-by-value

Commands	c	$::=$	$\langle v \| e \rangle$
Terms	v	$::=$	$\mu\alpha.c \mid V$
Values	V	$::=$	$x \mid \ldots$
Evaluation contexts	e	$::=$	$\tilde{\mu}x.c \mid \alpha \ldots$

$$\begin{array}{lllll} (\mu) & \langle \mu\alpha.c \| e \rangle & \rightarrow & c[\alpha \leftarrow e] \\ (\tilde{\mu}_v) & \langle V \| \tilde{\mu}x.c \rangle & \rightarrow & c[x \leftarrow V] \end{array}$$

1.5 Extensional equalities

We take as equality the reflexive-symmetric-transitive congruent closure of $\rightarrow$. In any interaction with a context, v and $\mu\alpha.\langle v \| \alpha \rangle$, for α not free in v, are equal. Similarly, for e and $\tilde{\mu}x.\langle x \| e \rangle$ for x not free in e. This justifies to define the following η-reduction rules:

$$\begin{array}{lllll} (\eta_\mu) & \mu\alpha.\langle V \| \alpha \rangle & \rightarrow & V & \alpha \text{ not free in } V \\ (\eta_{\tilde{\mu}}) & \tilde{\mu}x.\langle x \| E \rangle & \rightarrow & E & x \text{ not free in } E \end{array}$$

1.6 Extension with implication

Implication is characterised by a value constructor and a linear evaluation context constructor. The value constructor is the usual λ-abstraction, written $\lambda x.t$. Abstraction is intended to interact with an applicative context. An applicative context is any context that has at least one argument ready to be bound. Hence, an applicative context has the form $v \cdot e$ reminiscent of the list structure of stacks in abstract computation devices. The extra syntactic constructions and the extra reduction rule characterising the implication connective are the followings:

Terms	v	$::=$	$\ldots \mid \lambda x.v$
Evaluation contexts	e	$::=$	$\ldots \mid v \cdot e$

$$(\rightarrow) \quad \langle \lambda x.v \| v' \cdot e \rangle \quad \rightarrow \quad \langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle$$

The resulting λ-calculus is the $\overline{\lambda}\mu\tilde{\mu}$-calculus [CH00]. A canonical name for it could have been $\mu\tilde{\mu}^{\rightarrow}$-calculus. As a matter of comparison, the rephrasing of the constructions and rule for implication in standard λ-calculus syntax are:

Syntactic rephrasing in λ-calculus

$$\begin{array}{lccl} \text{Terms} & v & ::= & \ldots \mid \lambda x.v \\ \text{Evaluation contexts} & e[\,] & ::= & \ldots \mid e[[\,]\,v] \end{array}$$

$$(\rightarrow) \quad e[(\lambda x.v)v'] \quad \rightarrow \quad \textbf{let } x = v' \textbf{ in } e[v]$$

Note that the reduction rule for implication introduces no extra critical pair. Depending on how the fundamental dilemma is solved, we fall either on the $\mu\tilde{\mu}_n^{\rightarrow}$-calculus or on the $\mu_v\tilde{\mu}^{\rightarrow}$-calculus. These calculi respectively correspond to the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus and to the $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus in [CH00], to the exception of the exact characterisation of the evaluation context constructor. The following η-equality is satisfied by implication

$$(\eta_{\rightarrow}) \quad \lambda x.\mu\alpha.\langle V \| x \cdot \alpha\rangle \quad = \quad V \qquad\qquad \alpha \text{ not free in } V$$

Embeddings from Natural Deduction to Sequent Calculus have been given by Gentzen and Prawitz. Both embeddings can be generalised to embeddings from $\lambda\mu$-calculus to $\mu\tilde{\mu}^{\rightarrow}$-calculus.

1.7 The "canonical" call-by-name and call-by-value $\overline{\lambda}$-calculi: the $\overline{\lambda}\mu_n$- and $\overline{\lambda}\tilde{\mu}_v$-calculi

It can be shown that the theory of $\mu_n\tilde{\mu}^{\rightarrow}$-calculus (with η-rules) is equivalent, when observed on terms and commands, to the theory of a restricted calculus called $\overline{\lambda}\mu_n$-calculus. Similarly, it can be shown that the theory of $\mu\tilde{\mu}_v^{\rightarrow}$-calculus (with η-rules) is equivalent, when observed on values, evaluation contexts and commands, to the theory of a restricted calculus called $\overline{\lambda}\tilde{\mu}_v$-calculus (this calculus comes from [CH00], up to the η-reduction rule). The calculi are defined as follows (the usual occurrence restrictions of variables on η-rules applies):

$\overline{\lambda}\mu_n$-calculus

$$\begin{array}{lcl} c & ::= & \langle v \| E\rangle \\ v & ::= & \mu\alpha.c \mid x \mid \lambda x.v \\ E & ::= & \alpha \mid v \cdot E \end{array}$$

$$\begin{array}{llcl} (\mu_n) & \langle \mu\alpha.c \| E\rangle & \rightarrow & c[\alpha \leftarrow E] \\ (\rightarrow^{\beta}) & \langle \lambda x.v \| v' \cdot E\rangle & \rightarrow & \langle v[x \leftarrow v'] \| E\rangle \\ (\eta_{\mu}) & \mu\alpha.\langle v \| \alpha\rangle & \rightarrow & v \\ (\eta^R_{\rightarrow n}) & v & \rightarrow & \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle \end{array}$$

$\overline{\lambda}\tilde{\mu}_v$-calculus

$$\begin{array}{lcl} c & ::= & \langle V \| e\rangle \\ V & ::= & x \mid \lambda(x,\alpha).c \\ e & ::= & \alpha \mid V \cdot e \mid \tilde{\mu}x.c \end{array}$$

$$\begin{array}{llcl} (\tilde{\mu}_v) & \langle V \| \tilde{\mu}x.c\rangle & \rightarrow & c[x \leftarrow V] \\ (\rightarrow^{\beta}_v) & \langle \lambda(x,\alpha).c \| V \cdot e\rangle & \rightarrow & c[x \leftarrow V][\alpha \leftarrow e] \\ (\eta_{\tilde{\mu}}) & \tilde{\mu}x.\langle x \| e\rangle & \rightarrow & e \\ (\eta^R_{\rightarrow v}) & \lambda(x,\alpha).\langle V \| x \cdot \alpha\rangle & \rightarrow & V \end{array}$$

It is easy to show that the theory of the $\overline{\lambda}\mu_n$-calculus is isomorphic to the theory of $\lambda\mu$-calculus (as described e.g. in David and Py [DP01]. Conversely, the theory of call-by-value $\lambda\mu$-calculus is not as well studied (and as simple) as the theory of call-by-name $\lambda\mu$-calculus. We believe that the $\overline{\lambda}\tilde{\mu}_v$-calculus provides an interesting alternative to call-by-value $\lambda\mu$-calculus.

2 The $\mu\tilde{\mu}$-Subsystem and Sequent Calculus

2.1 A proof-as-program correspondence for Sequent Calculus

The $\mu\tilde{\mu}$-subsystem and its extensions provide a proof-as-program correspondence with Gentzen's sequent calculus. More precisely, it provides with a proof-as-program correspondence with a variant of sequent calculus that has the following specificities: 1) it has two axiom rules 2) contraction rule are not primitive but simulated by cuts 3) only the cuts not used to simulate contraction can be eliminated.

Following [CH00], we call $LK_{\mu\tilde{\mu}}$ this variant of sequent calculus. We use the letters A, B, C, $\ldots$ to denote formulae. Contexts, written Γ or Δ are sets of named formulae[1]. The calculus has three kinds of judgements,

[1]If contexts were multisets of formulae, the correspondence with simply-typed $\mu\tilde{\mu}$-subsystem would not be one-to-one.

written $\Gamma \vdash A; \Delta$, and $\Gamma; A \vdash \Delta$, and $\Gamma \vdash \Delta$. The inference rules, together with their corresponding denotation in the $\mu\tilde{\mu}$-subsystem, are the following:

$$Ax_R \quad \frac{}{\Gamma, x : A \vdash x : A \,|\, \Delta} \qquad\qquad Ax_L \quad \frac{}{\Gamma \,|\, \alpha : A \vdash \alpha : A, \Delta}$$

$$\mu \ \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \,|\, \Delta} \qquad Cut \ \frac{\Gamma \vdash v : A \,|\, \Delta \quad \Gamma \,|\, e : A \vdash \Delta}{c : (\Gamma \vdash \Delta)} \qquad \tilde{\mu} \ \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \,|\, \tilde{\mu}x.c : A \vdash \Delta}$$

Sequent calculus has more normal proofs than (call-by-name) natural deduction. Its interpretation along the lines of the $\mu\tilde{\mu}$-calculus shows that the extra proofs it has are call-by-value normal proof.

2.2 A sequent-free presentation of sequent calculus

Thanks to the absence of contraction, the variant of sequent calculus whose underlying structure is the $\mu\tilde{\mu}$-subsystem can be presented without sequents, in the same way as natural deduction was originally presented by Gentzen. We use the notation $\vdash A$ to indicate a formula that is asserted and $A \vdash$ for a formula that is refuted. We write $\perp\!\!\!\perp$ for a contradiction. The rules of the sequent-free presentation of sequent calculus (with implication) are the following:

$$\to_R \ \frac{\begin{matrix}[\vdash A] \\ \vdots \\ \vdash B\end{matrix}}{\vdash A \to B} \qquad \mu \ \frac{\begin{matrix}[A \vdash] \\ \vdots \\ \perp\!\!\!\perp\end{matrix}}{\vdash A} \qquad Cut \ \frac{\vdash A \qquad A \vdash}{\perp\!\!\!\perp} \qquad \tilde{\mu} \ \frac{\begin{matrix}[\vdash A] \\ \vdots \\ \perp\!\!\!\perp\end{matrix}}{A \vdash} \qquad \to_L \ \frac{\vdash A \qquad B \vdash}{A \to B \vdash}$$

References

[AH03] Zena M. Ariola and Hugo Herbelin. Minimal classical logic and control operators. In *Thirtieth International Colloquium on Automata, Languages and Programming , ICALP'03, Eindhoven, The Netherlands, June 30 - July 4, 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 871–885. Springer-Verlag, 2003.

[CH00] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, Montreal, Canada, September 18-21, 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000.

[DP01] René David and Walter Py. Lambda-mu-calculus and Böhm's theorem. *J. Symb. Log.*, 66(1):407–413, 2001.

[Her95] Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1995.

[KPT96] Delia Kesner, Laurence Puel, and Val Tannen. A typed pattern calculus. *Inf. Comput.*, 124(1):32–61, 1996.

[Kri01] Jean-Louis Krivine. Typed lambda-calculus in classical zermelo-fraenkel set theory. *Arch. Math. Log.*, 40(3):189–205, 2001.

[Mur92] Chetan R. Murthy. A computational analysis of girard's translation and lc. In *Proc. of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 90–101, Santa Cruz, CA, 1992.

[Pol03] Emmanuel Polonovski. *Subsitutions explicites, logique et normalisation*. Thèse de doctorat, Université Paris 7, June 2003.

Note: the references to the foundational works mentioned in the paper are standard and have been omitted.