Reflection, Non-Determinism and the λ-Calculus

François-René RIDEAU ĐẶNG-VŨ Bân

CNET DTL/ASR (France Télécom)
38–40 rue du general Leclerc
92794 Issy Moulineaux Cedex 9, FRANCE
francoisrene.rideau@cnet.francetelecom.fr
http://www.tunes.org/~fare

Abstract We present a conceptual path we followed striving towards defining a computational and logical architecture integrating internal reflection. We start from a fundamental limitation theorem about Turing-equivalent systems, and justify partial non-deterministic functions as a natural framework to reach our goal. We then introduce non-determinism first in the applicative λ-calculus, and consider the semantic implications of introducing it with evaluation strategies other than call-by-value; we see how sharing of reductions in presence of non-determinism implies that normal evaluation must be modeled by call-by-future rather than by call-by-name. We then study the integration of a reflection primitive in the obtained non-deterministic λ-calculus, and try to determine how it effects the expressive power of the system; we see this could be a model of Curry's illative combinatory logic. Finally, we propose directions to extend the above theory of reflection so to actually encompass useful real-life computational systems, which naturally leads to reflection in distributed calculi.

1 Introduction

In a previous paper [27], we have motivated from a cybernetical point of view the utility of a reflective platform to manage the complexity of long-lived and widely-used programs. When the developed programs have such additional constraints of quality of service as have telecommunication networks, to deliver permanent uninterrupted real-time evolving adaptable services, it is required that the reflective capabilities be fully deployable dynamically at runtime (since any time is runtime), as a "normal" first-class feature. That is, reflection must be *internal* to the calculus modeling those services.

In this paper, we explore basic elements of theory required to achieve the goal of defining a formal computational architecture that provide for full reflection, that is, arbitrary internal manipulation of programs themselves by metaprograms. So as to extract the essence of the problem at stake, we study how to integrate this reflection in the simplest known foundational formalisms for computation, recursion theory and λ-calculi. Along the trail of our exploration, we stumble on a few undeservedly ignored but elementary theorems that result immediately from formalizing of our approach; however, for the sake of conciseness, we only give the key ideas behind the proofs, from which they could be easily formalized using well-known techniques from relevant domains of theoretical computer science, to which we refer[1].

In a first part (2), we start with a limitation theorem regarding the consistent formalization of internal reflection in computational systems, in forms that enable full dynamic manipulation of code, and are led to the study non-determinism in computational systems. Since we weren't able to find a satisfying formal framework for non-deterministic computational systems in available literature, we introduce, in a second part (3), non-determinism in λ-calculi, and study its impact on various well-known properties of their semantics, focusing on evaluation strategies. We may then, in a third part (4), add a reflection primitive to a non-deterministic computational system; we see that it actually achieves our initial goal beyond expectations, by providing with internal logical reasoning on programs as well as with mere internal metaprogramming control. Finally (5), we discuss the expressive power of a reflective platform, and how taking advantage of it requires extending the system with external interaction within a concurrent computation framework, which poses some interesting problems.

2 From Reflection to Non-Determinism

2.1 A Fundamental Limitation

The first result we have in studying the semantics of reflection is the impossibility of a "perfect" internal reification function that would map functions each to a valid source code for it.

Theorem 1 *Given a data structure $\mathbb{D}$ (recursively enumerable set with computable equality test), and a Turing-equivalent computational domain $\mathbb{C}$ (set given by Σ_1-definable "semantics" over some "source language" data structure, and able to express all partial recursive functions) there is no computable injective total mapping $\mathbb{R}$ from $\mathbb{C}$ to $\mathbb{D}$.*

Proof Sketch for Theorem 1 With such a mapping, we could easily compare partial recursive functions for equality, by checking that they have the same image by $\mathbb{R}$. This would solve the Turing halting problem, which we know has no computable solution.

[1] We are convinced that readers familiar with the domain will have no problem with the semi-formal arguments proposed. If a reader has serious doubts about the validity or the relevance of any given argument, we will gladly welcome criticism and try to answer as best as we can; hesitating readers are invited to consult the bibliography.

Actually, we think that complete formalizations are to be done in a computerized proof system, not in an article written for human beings, that must remain concise within its space limits. We apologize for lack of having actually implemented our proofs in a computerized formal reasoning system before we publish.

We cannot say who first discovered this theorem; actually, we've never seen it written[2], though it looks like common implicit lore among people studying Reflection. Most likely, every person newly interested in Reflection quickly rediscovers it, but doesn't dare write it down, since the underlying proof is such a trivial elaboration on Turing's Fundamental Theorem [33], yet its correct formalization takes quite a lot of hassle. Indeed, this theorem is actually a metatheorem on computational domains; it relies on our being in some formal metasystem in which are already formalized notions of formal systems and of computability, that will be external to those formal systems the theorem talks about.

The crucial point in the theorem is that we require the image of every abstract program in $\mathbb{C}$ to be uniquely defined by a computable metaprogram. However, the notion of "abstract program" is not a computable notion; in other words, $\mathbb{C}$ is not a "computable structure", but results from taking the quotient of source programs with respect to some non-computable notion of observational equivalence; it can be defined in usual first-order logic presentations [7] with a $\forall\exists$-quantified predicate, but not with a computable ($\exists$-quantified) predicate. A computable metaprogram must thus be defined as a computation over program sources, not one directly over abstract programs; it may be considered as correctly defining a mapping over abstract programs if and only if it is compatible with the semantic equivalence, that is, if and only if two program sources representing the "same" abstract program have the "same" image.

2.2 Bypassing the Limitation

There have been many approaches to cope with this limitation. They all involve dropping at least one hypothesis from the limitation theorem. Since we're interested in achieving Reflection in a powerful language, we'll ignore those approaches that drop $\mathbb{C}$ being Turing-equivalent, $\mathbb{D}$ being a data structure, or $\mathbb{R}$ being total and injective.

One way, used by type theorists [4] or daring application programmers [23] is to manipulate only abstract objects, have leave reification be an oracle external to the calculus, yielding representations whose interpretation "magically" coincides with desired abstract objects. This does not actually achieve *internal* reification, that be usable dynamically within the calculus, but only some kind of static metaprogramming from an external metasystem, and pushes back the problem of having satisfying internal reification in the considered system+metasystem combination. The fact that the metaprogramming be static means that the coincidence between object and representation is fragile, since its consistency is not dynamically enforced by the system.

Another way, used by recursion theorists [31] or system implementors [29], is to manipulate only representations as first-class objects, and leave their interpretation into abstract programs be external to the actual calculus. The calculus is

[2] We are very conscious of the incomplete character of our bibliography, and will gratefully welcome pointers to earlier works (and further works as well). We will more generally welcome any kind of constructive criticism.

thus an “explicit calculus” of low-level representations rather than an “abstract algebra” where you could actually manipulate high-level objects This breaks “referential transparency” in the programming language, since two objects that should have the “same” meaning according to the intended declared semantics, will actually behave in quite distinct ways. Actual computing systems following this path have their guts revealed, and users are forced to manually manage low-level representation details and constantly adapt them for use-dependent performance or adequation, instead of focusing on high-level abstract concepts independently from the underlying representation. Reflective features developed on top of such systems tend to lock the representation and induce bloat, limit adaptability, and make formal reasoning overly complex.

These two ways avoid the problem without solving it, by privileging either the concrete or the abstract aspect of objects in the calculus, and resorting to external magic so as to relate the two aspects. The complexity is thus shifted towards an outside metacomputational or metalogical system, in which the same problem in turn happens, which makes metareasoning exponentially more complex as metalevels are explored, while increasing development cost and system fragility by requiring manually-managed redundancy between metalevels.

Since we want internal reification, we need to remove a different hypothesis than chosen above from the impossibility theorem. The only one that seems to remain is that $\mathbb{R}$ be a deterministic function; indeed, since associating a behavior to a source program (the semantics) is intrinsically non-injective (many-to-one), then the inverse operation (the reification) is intrinsically non-deterministic (one-to-many). Thus, we can achieve our goals if we use a framework in which non-deterministic functions are first-class.

2.3 Non-Deterministic Partial Functions

While partial functions are of common use in recursion theory [7,31], and have been cleanly expressed in formal systems [11], non-deterministic functions as such have been mostly ignored, to the point that the word “function” is generally understood only for deterministic functions.

Functions that are not forcibly total will be said to be partial, and functions that are not forcibly deterministic will be said to be non-deterministic. Herein, when we say “function”, we’ll mean “partial non-deterministic function”. In our non-deterministic framework, traditional “functions” are called “deterministic functions”; traditional “mappings” are called “total deterministic functions”.

An interesting approach is then to reuse and extend the classical paradigm of functions-as-relations, where functions are identified to binary relations of same graph. Actually, since we accept non-determinism and partiality in functions, we now have an exact isomorphism between functions and relations, whereas this was not the case when only deterministic functions were accepted:

Given a binary relation R from E to F,

R is *deterministic* if and only if $\forall x \in E\ \forall y \in F\ \forall z \in F\ \ xRy \wedge xRz \implies y = z$
R is *total* if and only if $\forall x{\in}E\ \exists y{\in}F\ xRy$
R is *injective* if and only if $\forall x{\in}E\ \forall y{\in}E\ \forall z{\in}F\ xRz \wedge yRz \implies x = y$
R is *surjective* if and only if $\forall y{\in}F\ \exists x{\in}E\ xRy$
R is *bijective* if and only if if it is deterministic total injective and surjective.
Moreover, the *inverse* of R is the binary relation R^{-1} from F to E such that

$$\forall x{\in}E\ \forall y{\in}F\ yR^{-1}x \Leftrightarrow xRy$$

Taking the inverse of a binary relation is an involutive operation, and that a binary relation is deterministic if and only if its inverse is injective, while it is total if and only if its inverse is surjective.

We then identify a partial non-deterministic function with a binary relation of same graph: the graph Γ_T of a function or binary relation T is defined[3] such that:

$$\Gamma_T = \{\langle x, y\rangle \in X{\times}Y \mid T : x \mapsto y\}$$

That is, a function ϕ is identified to the relation $\overset{\phi}{\mapsto}$ that relates its inputs to their respective outputs (if any), and a binary relation R is identified to the function that maps elements of the first set to those of the second set to which it is related. In other words, a function ϕ and a relation R are identified if and only if

$$\forall x\ \forall y\ \ y = (\phi\ \ x) \Leftrightarrow xRy.$$

Only, the scripture $y = (\phi\ \ x)$ above is misleading at best in a non-deterministic context, since the expression $(\phi\ \ x)$ (application of function ϕ to argument x) may take several values, unless ϕ is deterministic; thus we have to refine the concept of equality.

2.4 Logical Semantics of partial non-deterministic functions

Hence, we'll write $y{\lhd}(\phi\ \ x)$ the fact that the evaluation of expression $(\phi\ \ x)$ may yield value y (which again holds if and only if $\phi : x \mapsto y$). More generally, we'll write $a \lhd b$ the fact that the evaluation of expression b may yield any value that the evaluation of expression a may yield. We'll reserve symbol $=$ for equality of expressions, so that $a = b$ holds if and only if expressions a and b can both yield the exact same values (any value resulting from evaluating a may result from evaluating b, and conversely). $\lhd$ is a reflexive transitive binary metarelation between expressions; we can make it antisymmetric by adding an axiom of "extensionality" or otherwise *defining* $=$ such that $a = b$ be equivalent to $(a \lhd b) \wedge (b \lhd a)$.

The interpretation of $(\phi\ \ x)$ when ϕ is a general partial non-deterministic function, as defined by its graph Γ_ϕ, is thus that of being an expression that can yield any single result y such that the pair $\langle x, y\rangle$ is in the graph Γ_ϕ of ϕ, or else that won't return any result. Given a non-deterministic version of Hilbert's

[3] Actually, which of graphs, sets, functions, relations, or total mappings are "more primitive", and which are defined from the others, depends on the formalism in use, and is of little importance as long as the properties that interrelate them hold.

choice quantifier ε (which is to $\exists$ what λ is to $\forall$), this can be formalized in the following way:

$$\phi(x) = \varepsilon y \ . \ \langle x, y\rangle \in \Gamma_\phi$$

This formula is well-known in many formal logic systems, except that in the present framework, the non-determinism (as well as partiality) associated to this formula is *internal* to the logic, not external to it. This difference means that a model of the calculus needs not (and must not) statically choose a one value for every ε-witness, but instead must maintain an account of all dynamic values possible for the expression; this subtle difference is what will later allow reflection to happen, by not requiring consistency between static choices, but only dynamic consistency.

The obvious semantics to consider for non-deterministic expressions, that seamlessly generalizes the semantics already considered for expressions with partial functions, is (denotational) "may" semantics. That is, the meaning of an expression is the set of possible outcomes of its evaluation, the set of the values that evaluation *may* yield after possible termination. The meaning of complex expressions can be inductively defined from their structure as terms, with the set of possible values of a term being synthesized from the set of possible values of its subterms. This is also compatible with extending to non-deterministic functions the well-known elimination of functional constructors in favor of their associated relation, as used in classical first-order logic, and in some logic programming languages as well. Hence, an assertion $(P\ \ (f\ \ x)\ \ (g\ \ y))$ will be translated into

$$\exists z\ \ \exists t\ \ (P\ \ z\ \ t) \wedge (R_f\ \ x\ \ z) \wedge (R_g\ \ y\ \ t)$$

An interesting property of this transformation is that after replacing non-deterministic functional constructors with relations, the first-order logical substratum is the same in the non-deterministic case as in the well-known deterministic case! Hence, we can reuse all known theorems of first-order logic that apply to logical structures with a purely relational presentation.

In particular, we may define computability for partial non-deterministic functions in pretty much the same way as for traditional partial functions: in the framework of recursive functions, a partial non-deterministic function is computable if and only if its graph can be defined from primitive relations, constructors, and variables by a formula whose only possible quantifiers are outermost existential quantifier. We immediately see that, assuming the graph of the primitive constructors are recursively enumerable in the traditional meaning, a partial non-deterministic function is computable if and only if its graph is a recursively enumerable set in the traditional meaning. As the reference for computability, we may choose a standard presentation of the set $\mathbb{N}$ of natural integers, with usual axioms, and a non-determinism generator $flip$ that takes no argument and returns either 0 or 1, and can be eliminated into the unary relation R_{flip} that holds for 0 and for 1 but for no other integer. Computability of functions from a data structure to another data structure in general may either be defined directly from primitive operations as above, or more conveniently perhaps, from the above case, through otherwise axiomatically computable encodings of the domain and codomain sets of these functions with natural integers.

An easy result is that, by a simple symmetry of the graphs, every computable (partial non-deterministic) function has a computable (partial non-deterministic) inverse; this of course says nothing about the complexity of the inverse function, since the method used in the above proof consists in using a recursive enumeration or the graph, which in general is a particularly bad algorithm. Such a result suggests that while a logic axiomatization of computability is useful, it (voluntarily) lacks discrimination power, and for finer results, we need study a computational system in more details, with its operational semantics.

3 From Non-Determinism to λ-Calculus

3.1 Non-Determinism in Applicative λ-Calculus

A simple intrinsic way to formalize classical deterministic functions in a way directly compatible with first-order logical presentations is with applicative λ-calculus: it provides a skeletal Turing-equivalent computational system where the semantics of expression matches that of first-order logic. We may thus hope that it will be as easy to extend it with non-determinism as was the case for logical systems, so that, having a formal computational system in which to express partial non-deterministic functions, we may build a minimalistic internal reflection upon it.

The λ-calculus [9,10] is defined as usual by the grammar:

$$\begin{array}{lr} B ::= \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \ldots & \text{(basic constants)} \\ V ::= x \mid y \mid z \mid \ldots & \text{(variables)} \\ M ::= B \mid V \mid \lambda V.M \mid M\ M \mid \ldots & \text{(lambda terms)} \end{array}$$

We unhappily lack space to recall the theory of λ-calculus in this paper, so that we refer the reader to standard texts on the topic [2,**?**]; good such texts are available on the web [**?**].

Recursion, numbers, pairing, booleans, can all be expressed in most λ-calculi (applicative or not) either intrinsically (with the "paradoxical" fix-point combinator Y, Church numerals, and generally selector-constructing λ-terms [**?**]), or extrinsically (from a set of basic constants). With the restriction by a Church-like simple type system with proper axioms, the exact setting of recursive function theory can be retrieved.

In the applicative λ-calculus, the evaluation is done in applicative order (call-by-value), that is, functions and arguments must be "fully" evaluated before a function call happens. As said above, the semantics of applicative evaluation express exactly the semantics of functions in usual (partial) recursive function theory [7], or in usual programming languages [18].

Not surprisingly, we must rely on a distinction between expressions and values to formalize the semantics; only, since we don't have an a-priori extrinsic model for the calculus, we must define values in an intrinsic way. Values are defined from normal forms, or rather, observable forms, as the quotient structure of observable forms considered up to observational equivalence. The denotational semantics of an expression is the set of values it may take. The notion of observable form is computable (provided all added reduction rules are computable), but the notion of value isn't, since it is not usually decidable whether two observable forms are semantically "equal".

Non-determinism will appear simply with the addition of non-deterministic operators (syntactic constructors for terms in the M grammar, with their associated evaluation rules), though we'll see that we could do it with special constants (nullary syntactic constructors for terms in the B grammar, with their associated δ-rules [9]). By definition, however, non-determinism amounts to add rewrite rules that break the sacrosankt confluence of the calculus, which is why it is deliberately avoided by theorists.

The simplest non-deterministic operator to be introduced in λ-calculus was John McCarthy's ambiguity special form `amb` [21], also known as `either` in Screamer [30], whose semantics is that `(amb` M N`)` (for which we'll also use the infix syntax $M \cup N$) would be rewritten in either M or N when evaluated. This operator suffices to introduce all recursive partial non-deterministic functions, as we'll prove in Theorem 2 below, which we'll latter see is not such great a feat (non-deterministic version of the Turing tar-pit [26]).

For convenience, we'll also introduce a special form $\bot$ whose evaluation rule is to always diverge. This form is mostly for syntactic convenience, since macro-expressible [12] in traditional λ-calculi as $(\lambda x.x\ x)(\lambda x.x\ x)$. Actually, we could avoid special forms, and achieve the same expressive power with only basic constants and their δ-rules, by doing all the dirty job under the protection of λ's: hence, instead of $\cup$, we'd have a function $\hat{\cup}$ such that $\hat{\cup}\ f\ g = \lambda x.(f\ x) \cup (g\ x)$; instead of $\bot$, we'd have a function $\hat{\bot}$ such that $\hat{\bot} = \lambda\_.\bot$; etc. However, we will use special forms instead of basic constants for the sake of readability.

Theorem 2 *Any computable partial non-deterministic function f from a data-structure to another data-structure can be expressed in the applicative λ-calculus extended with the ambiguity operator (assuming some convention to express natural numbers, pairing, and those data-structures within the calculus).*

Proof Sketch for Theorem 2 By definition of computability, the graph Γ_f of a partial non-deterministic function f is recursively enumerable set by a deterministic index function ϕ: $\Gamma_f = \{(\phi\ i)\}_{i=0}^{+\infty}$. We can thus define f with the following recursive definitions:

```
(define (try x i)
   (if (equal? (car (φ i)) x)
      (cdr (φ i))
      ⊥))
(define (enum x i)
   (amb (try x i) (enum x (1+ i))))
(define (f x)
   (enum x 0))
```

3.2 Generalizing Normal Order Evaluation in Presence of Non-Determinism

The semantics of vanilla λ-calculus [2], is such that reductions happen in any order, and the confluence of the calculus ensures that the result (successful convergence toward a irreducible form, or divergence) does not depend on that order. A valid and complete evaluation strategy for λ-calculus in this deterministic case is normal order evaluation (call-by-name or call-by-need), whereby textual substitution of formal parameters with given arguments happens at function call sites systematically before any reduction within the arguments, reductions matter (and happen) only when needed. In contrast, applicative order evaluation (call-by-value) appears in this setting as a valid but incomplete strategy, since it fails to converge to a normal form when evaluating such terms such as $(\lambda\_.\top)\bot$ (where $\top$ is a converging term) by exploring unneeded diverging sub-terms.

Now, non-determinism precisely breaks the confluence of the calculus, so that generalizing traditional λ-calculus to a non-deterministic setting requires particular attention to issues such as order of evaluation, whereas, the semantics of applicative λ-calculus constrains evaluation order enough so as to have clearly defined, easily understood semantics in presence of non-determinism. If the semantics doesn't impose any constraint on β-reductions, then call-by-name will still model it completely, and will still generate strictly more convergent evaluations than call-by-value. However, such semantics is not quite satisfactory [6], and doesn't extend the call-by-value semantics to more terms, but modifies it in not-so-subtle ways, since some terms have strictly more, undesirable, values, under the former "evaluation strategy" than under the latter: indeed, consider a data-structure (intrinsic or extrinsic, for instance the set $\mathbb{N}$ of natural integers) with equality test `equal?`, and an ambiguous expression E with many possible results, all within the data-structure (for instance, an E such that $E = 0 \cup$ `(1+` E`)`); then the form `((`λ x`.(equal?` x x`))` E`)` will always succeed and yield a truth value under call-by-value, whereas it may report either truth or false under call-by-name.

The failure of call-by-name to provide satisfactory semantics comes from its being unable to model the *sharing* of reductions between multiple occurrences of a "same" variable. In contrast, call-by-value provides sharing by forcing all reductions in a term to happen *before* any duplication, by fully evaluating arguments of function applications; it is not a subtle way to share reductions, and it requires that "early decisions" be taken in the choice of ambiguous reduction during the evaluation process; still it provides with a limited notion of "identity" of variables. If we want to correctly generalize normal order evaluation from deterministic settings to the non-deterministic settings, in a way that still extends applicative order evaluation in these new settings, it is necessary to discard call-by-name and replace it with call-by-future [1,13]. Futures are a mechanism to mandate sharing of reductions among multiple instances of a replicated argument, without imposing an order on those reductions. The fact that arguments are shared rather than copied means that substitution must no more be considered as textual, replacing variables each by an expression tree, but as "structural", replacing variables each by the shared node of an expression graph. This brings another evidence, if was needed anymore, that the λ-calculus is intrinsically a calculus on *graphs* [?], not a calculus on trees. The fact call-by-future produces more values than call-by-value is obvious, since any reduction path for a term under the latter is valid under the former (much like with call-by-name); the fact that call-by-future produces no more values than call-by-value for terms where the latter converges (criterion where call-by-name failed) follows from a double inductive reasoning on the structure of terms and the sequence of reductions, with every reduction in call-by-future being valid in call-by-value assuming previous reductions forced by call-by-value could converge (by hypothesis of induction).

It is known that in a deterministic setting, we can macro-express call-by-name λ-calculus within the applicative call-by-value λ-calculus (conserving function application) through λ-thunkification [24,16], technique also known as "protecting by a λ": $\lambda\_.M$ (where $\_$ is a variable name that does not appear free in M) delays the evaluation of M and partially "reifies" the term M, allowing it to be passed as "text"; evaluation of delayed text may be forced when needed, by applying the λ-thunk as a function to a dummy argument. The same transformation works in non-deterministic settings as well as deterministic settings, with call-by-name being thus emulated with call-by-value; however, we'd be more interested in emulating not call-by-name but rather call-by-future, since the latter is the Right Thing^{TM}. This is not possible with λ-thunkification (that emulates call-by-name), however, this is possible if a new syntactic construct is added for first-class "thunks" such as the `(delay ...)` or the `(future ...)` constructs of some LISP systems [18], that ensure sharing of the "future result", and may have to be explicitly `force`'d or `touch`'ed to extract that value, when needed. Such a construct appears quite naturally when factoring the transformation of normal λ-calculus into applicative λ-calculus [16]. Actually, addition of "forcing" primitive is also required to emulate call-by-value in call-by-future, for no application-preserving transformation may change the fact that applicative semantics prevent sharing of any information that be not fully reduced, whereas normal semantics prevent forcing of reductions that are not needed in the final result (again, a formal proof of these assertions would involve induction on the structure of terms and of reductions).

Remarkably, its semantic nuisances do not prevent call-by-name non-deterministic λ-calculus to express every individual recursive partial non-deterministic function from $\mathbb{N}$ to $\mathbb{N}$ (or similarly, from any usual data-structure to any other one) by pretty much the same trick as exposed above for call-by-value. What it cannot correctly express is their dynamic composition as first-class objects in way that conserves composition of higher-order functions, which again suggests that the set of implementable functions is poor way to judge expressive power of a language [12,27].

3.3 Combining call-by-value and call-by-future

In the same papers as above [24,16], it is shown how a transformation in CPS allows call-by-value to be simulated by either call-by-name or call-by-value itself in the deterministic case. Though we haven't fully studied yet what the properties of the transformation become in presence of non-determinism, it looks like that they are mostly conserved, as far as denotational may semantics is respected. In particular, the transformation indeed allows simulation of call-by-future (not call-by-name) by call-by-value. Non-determinism is transformed so that alternatives be spawned in advance as a choice between continuations, for every case of reduction that may later be found in a shared delayed expression. This means that a big tree of all possibilities of reductions will be built, which induces a blow-up in the size of the CPS term, unless an oracle can decide early enough what successful path will later be taken so as to always keep only the right alternative.

Now call-by-value and call-by-future typically involve different point of views on operational resolution of choices: it is expected that under call-by-value, a choice will be made "immediately", whereas under call-by-future, it will be made "no sooner than needed", and meanwhile some "fairness" will be provided among possibilities. This doesn't show in denotational may semantics, but does show in the operational semantics; that is, if some kind of communication makes it possible to observe computation in progress, they lead to completely different processes, with different implications on the way the choice tree is pruned. Thus, the fact that neither can directly express the other, but that CPS can simulate both of them at the same time suggests that we should be considering a calculus that has both of them, for instance, the call-by-value calculus with thunks of [16]. We'll call λ_{ND} such a calculus.

In such a language, it becomes possible to express one-shot communication and unification between parallel parts of the source expression tree, by creating a shared object (using call-by-future), and later "forcing" its value (using call-by-value) by failing execution (in effect hanging/killing the continuation) if the object fails to have the expected value. Fairness ensures that the case in which the value is propagated will be considered, whereas non-termination in the other cases ensures that other cases will not be observable. For instance, assuming `(any)` is a form that can return any value, at least among a rich enough domain (we'll later axiomatize $\top$ as such a form), and `(assert` *bool*`)` is a form macro-equivalent to `(if` *bool* $\top$ $\bot$`)`, then the following statement will create a value `x`, and later ensure that it is equal to 42:

```
    (let ((x (future (any))))
       (... (assert (equal? x 42)) ...))
```

This gives one-shot communication, or delocalized initialization of bindings, but doesn't allow side-effect, since the communicated value cannot change (however, in another continuation rooted upward in the tree of choices, the binding may have a different value; an assertion only forces its "own" future). Multiple attempts at "setting it" (in a same time-line) will result in unification of values: either the attempts match, and evaluation continues, or attempts do not match, and evaluation fails. The matching can very well be partial, with a binding being constrained with partial information, so that it may be later possible to constrain it further. For instance, in the following expression, assuming `cons` is any `any`-valued injective binary function, then the `assert` statement will match `x` with a non-empty list, and bind `y` to its `car`, and then invoke function `FOO` under these constraints:

```
(letrec ((list (λ _ . (either (cons (any) (list)) nil)))
         (x (future (list)))
         (y (future (any)))
  (FOO x (assert (equal? x (cons y (list))))))
```

Multiple cases may be handled as follows, with `(seq` x y`)` being a form macro-equivalent to `((λ _ . (`y`))` x`)`:

```
(amb (seq (assert (equal? x nil)) case_if_nil...)
     (seq (assert (equal? x (cons y (list)))) case_if_cons...))
```

All in all, the resulting language, considered with its may semantics, looks as much like a logic programming language as it looks like a functional programming language. This shouldn't be such a surprise, since we basically identified partial non-deterministic functions with arbitrary relations, and such relations are precisely what logic programming language manipulate. And indeed, the programming language that matches most the model of this non-deterministic λ-calculus is the pure logic programming language Mercury [32], in which our call-by-future corresponds to the normal mode of non-determinism (implemented through backtracking), whereas call-by-value corresponds to committed-choice non-determinism.

However, the operational semantics of Mercury and other logic programming languages induce a gap between their actual semantics and this abstract model: logic programming languages rely on some criterion of finite-time failure to prune the decision tree through backtracking, whereas λ_{ND} explores all subexpressions and their decision branches in parallel (only branches that can provably not succeed may be pruned). Since it cannot be decided in general which branches will fail, the precise criterion of finite-time failure used in logic-programming languages is always an incomplete approximation, that the language implementor forces upon users; on the other hand, implementation of the full parallel-evaluating semantics, while possible (by recursive enumerability of valid terminating computation processes), may be arbitrarily costly.

The may semantics does not discriminate between an object whose operational semantics is such that its evaluation must terminate from an object that may operationally diverge, but, when its evaluation terminates, may take the same value as the first one. This is purposeful, since non-termination is not a computable concept, and hence cannot be observed. The calculus may be enriched with a notion of an observable finite-time failure [**?**], that may be useful to define efficiently implementable operational semantics (at the cost of making reasoning about programs a bit more complex); but as long as the language is powerful enough, there will always unobservable failures where a program runs indefinitely.

4 From λ-Calculus to Reflection

4.1 Adding a Reification Primitive

Now that we have a bit of understanding of what simple non-deterministic computation systems are like, and what their may semantics allow to express, we may consider having an internal reification function $\mathbb{R}$ into one of them, that would map values into an internal data structure.

First off, we can easily check that a reification function $\mathbb{R}$ cannot be "expressed in terms of $\cup$", that is, isn't macro-expressible in the λ_{ND} calculus, since it wouldn't be able to correctly differentiate $\hat{\perp}$ from other λ-abstractions: by induction on the contexts around terms to be reified, any value towards which $(\mathbb{R}\ \hat{\perp})$ may converge will be a value towards which any λ-abstraction may converges. On the other side, $\cup$ may be macro-expressed in terms of $\mathbb{R}$, (more exactly, $\hat{\cup}$ may be directly expressed), since $(\mathbb{R}\ x)$ will have to choose among the multiple (possibly infinitely many) valid representations for given object x, as soon as there is a value with multiple possible sources. Reification thus requires that we add a new primitive.

The addition of a reification function $\mathbb{R}$ to the calculus leads to a question: what happens when the value to be reified was itself defined using $\mathbb{R}$? An easy answer would that $\mathbb{R}$ be a partial function that may well fail to return a result on such terms, and that we already did enough of going around the limitation theorem. But this answer is not satisfactory, as this solution is basically the one we rejected earlier, of having a two-level architecture, where reification is external to the actually reifiable domain. The correct answer is that $\mathbb{R}$ being a primitive, the source representation ranged by $\mathbb{R}$ may itself contain reified calls to $\mathbb{R}$. Actually, adding any primitive to the calculus implies that a suitable representation for this primitive be reserved in the data structure (most likely a simulated abstract syntax tree) in which the values are represented.

Now, the big problem is that the set of functions that have same may semantics as a given function is not recursively enumerable: its definition requires a $\forall\exists$-quantified formula instead of a computable $\exists$-quantified formula. Hence, it is impossible to implement a system where the semantics are completely respected. But this limitation is not actually as bad as it sounds; it may even be a source of creativity.

One way out is to consider the abstract high-level system as a specification, and only mandate correctness with respect to it from the implementation, without requiring fairness among possible outcomes (which would be an unobservable feature of the calculus, anyway). The implemented calculus is thus lower-level than specified, and the reification function discriminates more than should. This is apparently not much better than the "explicit calculus" way around the initial limitation. However, it is crucial that we be able to properly declare the abstract semantics that we care about independently from the peculiarities a particular implementation: in this way, we can isolate as such static limitations of implementations, and enable the human operator to dynamically modify and adapt the actual implementation while preserving the high-level semantics. This abstract semantics thus acts as a logical contract specification, and non-determinism appears as a way to express partial knowledge in these contracts (which is our initial limitation theorem states is a necessary feature so as to have powerful enough self-knowledge).

Another way out is to see that the limitations stem from classical logic: it is the principle of the excluded middle that induces the existence of “true” and “false” facts about the system, among those that are neither provable nor refutable. If we use intuitionist logic, then the paradox disappears: are considered “true” facts that are provable, and “false” facts that are refutable, other facts are just unobservable. Now, provability and refutability are recursively enumerable concepts: you can enumerate all valid proofs and refutations, and the facts they prove or refutate. The set of objects that provably have the same semantics as a given object is thus recursively enumerable, and it is possible to implement a reification function that completely reify the semantics with respect to intuitionist logic. Even if we consider such a system from a classical logic point of view, any discrepancy between the reified semantics and the “real” semantics is unobservable.

4.2 Metalogical Reflection

The latter solution to having correct reification in a computational system suggests that such a computational system would not only reify computation, but also logical reasoning about computation: indeed, to enumerate provably equivalent sources for a given programs means that there is virtually a full-fledged proof system within the system. At least, it is possible to “semi-compare” two programs as far as may logic is concerned, by taking action when they have a common source representation (that is, there is a computation that *may terminate* if and only if two programs have same semantics). Let us thus consider reification of the logic, and see if it requires more (finitely or indefinitely) than reification of the computation, or if it comes granted with “mere” reification into source.

Since the most simple non-trivial observer in our framework is convergence or divergence of terms, it makes a natural choice when it comes to modeling that an assertion “holds”, all the more since convergence is a recursively enumerable concept. We have described in [28] how a to express intuitionist logic in a non-deterministic call-by-value λ-calculus extended with $\cap$ (non-deterministic conjunction) and $\lhd$ (semantic comparison of expression as in §2.4): every expression can be considered as a logical statement that holds if and only if evaluation of said expression may terminate. $a \cap b$ can be used to express the fact that a and b are both hold at once (and yield a common value), while $a \lhd b$ can be used to express the fact that a implies b (since b takes all values that a can take). A truth expression $\top$ can be conveniently expressed as a new primitive one that may take any value that any other expression may take (so that for any expression E, $E \lhd \top$ holds). For convenience, it can be also decided that $a \lhd b$ behaves like $\top$ when it holds (and obviously like $\bot$ when it doesn't). The result is a logic framework that integrates well with the computational theory of untyped non-deterministic λ-calculus, just like logic frameworks based on Type Theories integrate well with typed λ-calculi (see for instance Coq [8]).

Such a system, if it can be consistently modeled, would be very attractive. Now, can it? If we already had a computable reification function in a system $\mathbb{C}$ with computable semantics, within a finitarily presented logical metasystem $\mathbb{M}$ (which all assumes we are in a metametasystem), then we could achieve full metalogical reflection within $\mathbb{C}$ (with respect to $\mathbb{M}$) using the following slogan:

metalogical reflection = metacomputational reflection + Gödel encoding.

Indeed, there would exist an algorithm (a priori external to $\mathbb{C}$) that would enumerate proofs within $\mathbb{M}$ that one term of $\mathbb{C}$ may yield more values than another one; now, since $\mathbb{C}$ can do any Turing-equivalent computation on an internal data-structure, it can in particular implement this very algorithm with respect to the image of $\mathbb{R}$, so that $\lhd$ (and similarly other logical primitives) are implementable in terms of $\mathbb{R}$! Of course, the adequation between this internal algorithm is some kind of magic due to a metametatheorem external to $\mathbb{C}$. This all proves that metalogical reflection requires no more power of expression than metacomputational reflection already grants. However, this doesn't (yet) provide with a reflective computational system, since we supposed the existence of such a system to begin with!

4.3 Bootstrapping Reflection

If a computable reflective system is possible, the theory of computability ensures that it can be bootstrapped from any Turing-equivalent system (though perhaps a bit more easily in systems with suitable pre-reflective properties than in other ones) by writing a suitable evaluator.

We haven't discussed the case for evaluators much in the rest of the article, being satisfied with hypotheses of existence, that we know are satisfiable in many systems like usual λ-calculus or recursive function theory. Indeed, for any computable system, it is possible by definition to have a computable interpreter in a suitable metasystem; some systems are not "regular enough" for such interpreter to be internal to the calculus, although, by the Turing universal property, we can always embed irregular systems within regular ones, as long as they exist. The art of the metacircular (internal) interpreter [25] ensures that the regular case indeed exists, where an evaluator is expressible within the system itself without a need for an additional primitive, though with a touch of "magic" for bootstrap purposes (actually a metametahypothesis relating the system to its semantics as observable in a metasystem).

Now, we're precisely trying to achieve the magic of reflection, a strong static magic that makes a few "universal constants" have some miraculous fitness properties, and that we want to embed once and for all inside the system, instead of having to appeal to renewed external magic, to repeated miracles and continuous divine intervention, every time we need to do metaprogramming (an art that is nevertheless important per se, even in absence of reflection, as we argue in [27]).

To build a simple computable reflective system $\mathbb{C}'$, consider within a universe $\mathbb{U}$ a finitary logical system $\mathbb{M}$ capable of expressing computability, and calculus $\mathbb{C}$ with suitable properties that we know are implementable, such as the existence of eval and quote functions as in original interpreted LISP systems [20,15]. Since $\mathbb{M}$ is finitary, its proofs are recursively enumerable, and it has model $\mathbb{M}_0$ in $\mathbb{C}$ that itself includes a copy $\mathbb{C}_0$ of $\mathbb{C}$; it is thus possible to computably express within $\mathbb{C}$ the provability of logical statements in $\mathbb{M}_0$ about programs in $\mathbb{C}_0$. We may then consider a language $\mathbb{C}'$ with representation $\mathbb{C}'_0$ that enriches the signature of $\mathbb{C}$ with for logical primitives $\lhd$, $\cap$, $\bot$, $\top$, etc, and define its semantics through an interpreter internal to $\mathbb{C}$. Evaluation rules for logical primitives consists in applying a function that tests provability of the corresponding predicate to the reified representation of the arguments; that is, we inductively transform statements from $\mathbb{C}'$ into statements in $\mathbb{C}$, replacing logical statements by their explicit implementations.

It is important that in the above construction, the low-level representation itself is not directly accessible in $\mathbb{C}'$ which would break referential transparency and yield an explicit calculus; instead, it is only used within the implementation of logical primitives, that, by definition, preserve the semantics as observable by $\mathbb{M}$. A referentially transparent reification $\mathbb{R}$ can be constructed as the computable inverse of explicit evaluation as explained in earlier sections. All in all, terms of $\mathbb{C}'$ are made to be denotationally equal by the observational semantics if and only if they are provably equal by $\mathbb{M}$, and else unequal, because they are distinguishable by difference of mutual and self comparison! So indeed, the semantics is completely and correctly reified with respect to $\mathbb{M}$.

Of course, completeness and correctness of reflection in $\mathbb{C}'$ are proven inside a universe $\mathbb{U}$ that contains $\mathbb{M}$ as an object and serves as a metametasystem for $\mathbb{C}$. Now, $\mathbb{U}$ might have more axioms than $\mathbb{M}$ can prove, at which point the semantics of $\mathbb{C}'$ would distinguish more objects than that of $\mathbb{C}$ (as can be observed within $\mathbb{U}$ but not $\mathbb{M}$), so that objects in $\mathbb{C}'$ be lower-level than $\mathbb{C}$ (from the stand-point of someone in $\mathbb{U}$). Beyond philosophical considerations about logic and truth in a reflective system, the practical problem is that one cannot easily extend the constructed reflective system $\mathbb{C}'$ with new primitives and new axioms, because its semantics relies precisely on the correctness and completeness of a given metacircular interpreter in a given logic. This suggests that a useful reflective system should have a way to internally express modalities within which different axioms hold and different primitives are available, which could be implemented with an open-interpreter that can be explicitly extended and "sealed" in differently extended contexts.

5 From Reflection to Distributed Systems

5.1 Opening the System

Now, internal reflection seems quite a feat to achieve (if possible), but what does it bring, besides philosophical interest? Indeed, by definition of computability, given any couple of computable data-structures, computation systems in which they are internally modeled can have no more functions from the former to the latter than has any Turing-equivalent system, reflective or not. What kind of expressive power, then, does reflection bring?

We have given answers to such a question in a previous article [27]; the idea is that Turing-equivalence is a very important concept, but that the set of statically computable functions it considers do not take into account the dynamic development process. By considering the approximate cost of development as the quantity of human-computer interactions needed between two iterations of development process, we proved that metaprogramming could cut down the cost of development towards the Kolmogorov Complexity of what it was without it. This suggests that any enhancement in internal expressiveness brought to a language by metaprogramming and reflection would be observable by dynamic interactions only.

The natural way to add dynamic interactions as an observable behavior of programs in a formal calculus is to consider concurrent process calculi, the most promising of which seems to us to be the join-calculus [14] (unlike what suggests the title of the article, the join-calculus is *not* a reflective calculus, but instead a higher-order calculus, that eliminates some nuisances of π-calculi and can model actors in a nice and straightforward way). Such calculi have intrinsic non-determinism due to asynchronicity of concurrent parallel reduction, and most interestingly, they are a natural target for CPS transformations such as we used earlier [3]; indeed, CPS terms obviously do not use the full power of the λ-calculi in which they are traditionally expressed. Also, communication allows for observation of the time when non-deterministic decisions are taken, which was one of our earlier concerns. Finally, they correspond more closely than static calculi to the way people interact with computers, and computers (and computer processes) interact with each others.

In such systems, we expect as we do in the above-mentioned article, that the expressive power of a language lies in the set of contracts that may be passed by people (or actors) exchanging data over a given set of channels, that is, by the achievable states of trusted mutual knowledge in the system. The richer the algebra of expressible contracts, the more expressive the language. We conjecture that using such a criterion, reflective systems might be proven (much) more expressive than non-reflective systems, by allowing negotiation of high-level features without tying them to a particular low-level implementation (reflection may be viewed as a mechanism to achieve quotienting and non-deterministic selection of a representent for every considered element of a quotiented structure).

5.2 Metaobjects and Semantic attachment

When considering communicating systems, there comes the question of what to reify. An obvious and easy answer would be to reify the whole running program, that is, the whole world of running agents, into a datastructure, and pass it as an argument to a whole-world continuation. Only this means that we would consider all interactions as internal, and the whole world as a same static program observable by its only eventual outcome; that is, despite change in language structure, we would be actually facing the exact same reification as before, without much interest as far as expressiveness is concerned. Also, reifying the whole world has implementation and security issues that make it unsuitable in any non-trivial (and possibly world-wide) distributed system. At the other end of the continuum of ways to reify, the system would be divided into "atomic" agents, and reification of an agent would allow to reify its internal state and intercept its incoming and outgoing connections. Between these two extremes, there are as many ways to reify as there are ways to split the system into a set of "internal" and "external" communications (which in the case of mobile calculi, might be a dynamic notion), that is, to define the "limits" of an agent. Things get only more interesting as multiple concurrent reifications would attempt to reify overlapping sets of communications.

Another issue with reification in communicating systems is the consistency between the representation and the agent: depending on the uses, the representation could be read-only or modifiable; the agent itself being read-only or modifiable, there comes the question of whether changes to one should or not be automatically reflected into the other, with what coherence. A commonly adopted solution, that of "metaobjects", is that the reified object is given a standard continuation and replaces the object. Another solution is that of semantic attachment, as introduced in Weyrauch's works about reflection in FOL [34]. By semantic attachment, the operator can create correspondences between abstract objects of a formal logic system and concrete objects in the underlying (or an external) system. It looks like the two approaches are related as linear logic is related to usual monotonic logic: in the first case, the metaobject replaces the object; in the second, both have indefinite extent, and are mostly interchangeable, except that one form is preferred for efficient ground reductions while the other is preferred for powerful abstract reasoning.

All in all, there seems to be a very rich algebra of possible reflection concepts within communicating systems; a logic suited to reify the semantics of such systems will likely include non-monotonicity, in the form of linear logic or otherwise modalities. A whole exciting field of research awaits us, that we have hardly begun to explore.

6 Conclusion

6.1 Achievements

We have pin-pointed a fundamental limitation regarding reflection in computational systems. We have proposed non-determinism as a framework to specify partial knowledge in a computational system, and in particular partial self-knowledge. We have explored the basic semantics of non-deterministic λ-calculi as compared to the deterministic case, and suggested relations with logic programming. We have shown that faithful reification of the semantics of high-level objects provides not only with a computational reflection but also a logical reflection, and have consequently constructed a reflective system starting from reified finitary logic. Finally, we have justified how further work on reflection in general should naturally be done in dynamically communicating systems, not mere statically computing systems, and have tried to overview the difficulties that await us there.

6.2 Further Research

Most importantly, we still have to implement formal proofs of our theorems within a trustable computerized logic system. Although we think our proof sketches are basically correct, the precise difficulties on which we'd stumble might bring some insight on the implementation of actual useful reflective systems.

We have only begun to study how source code reification, in a fully reflective system or in more modest ones, interacts with side-effects, and more generally parallelism and communication. We have only guessed the difficulty that resides in the specification of the “limits” up to which an object is to be reified.

We intend to formally specify and effectively implement a reflective concurrent programming system as suggested in the article.

6.3 Related Works

Though it's always been obvious to all people involved that reification is intrinsically a one-to-many concept, the idea of using non-determinism to cleanly formalize computational reification was initially suggested to us by Fergus Henderson [17] whose logic programming language Mercury [32], allows for reification in an evaluation mode of committed-choice non-determinism (equivalent of the call-by-value non-determinism above). We weren't convinced at first, until we found quite similar results in our Master's Thesis, under Jacques Chazarain's guidance, while building a logical framework around applicative λ-calculus (as embodied by a pure subset of Scheme) [28]: termination naturally appeared as an intrinsic truth value, and logical or as McCarthy's ambiguity operator [21], whence the rest followed.

As for earlier work concerning non-determinism in theoretical λ-calculus, there is an interesting review by Meldal and Walicki [22]. As far as actual computational systems go, it looks like there is a taboo against breaking the foundational theorems of confluence in λ-calculi and most works deal with “don't care” non-determinism, that is clustered in ways such that it is not relevant to the result of the overall computation. Practical implementations generally focus on deterministic semantics, too, since traditional monoprocessors are intrinsically deterministic. However, non-determinism does appear in even the most formalized programming language standards, where for instance, the order of evaluation of arguments is not specified [18]. Non-determinism is also a fact of life when using the logic programming paradigm as found in languages such as Prolog, Oz, Mercury, or extensions to existing programming languages such as Screamer [30]. Furthermore, it is inevitable as soon as asynchronism is present, in the form of multithreading, multiprocessing, and distributed computing, or even just interrupt-based input/output.

As for reflection in logical systems, it looks like the idea of fully integrating a logical framework in a Turing-equivalent computational system dates back to Curry's illative combinatory logic [9,10]. Curry wanted to encode a self-standing logic based on a λ-calculus enriched with logical operators (or more precisely an equivalent calculus with a finitary combinator-based presentation). But, as far as we know, he could never propose a model for such a theory, and could only give a combinatory presentation to what are now well-known as type theories for λ-calculi (in particular, we are not aware that he would have ever considered non-determinism). If it is indeed possible to specify reflective systems as we conjecture, such systems would be the first models to somehow match Curry's expectations for an illative logic. More recent work on the general topic include the reflective frameworks added to NuPRL, as tower of universes [19]. We are personally not satisfied with such static towers, that require the user to manually give a static bound for the level of the tower in which resides a term, which we feel is like asking him to manually give a static bound for the depth to which a function may recurse. Another promising framework to simply express reflection is rewrite logic [5], that seems to make the concept of an open interpreter much easier to implement, by allowing incremental addition of rewrite rules within existing rule-sets without having to break and remake explicit fix-points.

A The Authors

Department DTL/ASR at CNET, headed by Jean-Bernard Stefani, is involved in the formalization and implementation of the Architecture of (asynchronous) Distributed Systems, such as those to be used in a telecommunications operator company. The stringent requirements for continuous real-time service of the systems we consider imply that hardware and software evolution must be regarded as dynamic implementation of a single system, rather than stopping a system and starting a new one every time.

Metaprogramming appears to us as a natural tool to manage the complexity of the task, and Reflection as its natural extension for pushing useful metaprograms into the runtime for automatic program adaptation. However, such techniques currently lack any formalized semantics, hence any large scale communicability and trustability. Thus, we have undertaken a study of such semantics, beginning with the simplest case in programming languages, untyped λ-calculus, although we intend to use it in concurrent process algebras with a variety of implicit analyses and their induced typesystems.

References

1. Henry G. Baker and Carl Hewitt. The Incremental Garbage Collection of Processes. In *SIGPLAN Notices*, volume 12, 8, pages 52–59. ACM, August 1977.
2. H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1985.
3. Gérard Boudol. The π-calculus in direct style. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, January 1997.
4. Samuel Boutin. *Réflexions sur les quotients*. PhD thesis, Université Paris 7, November 1996.
5. Manuel G. Clavel and José Meseguer. Axiomatizing Reflective Logics and Languages. In *Reflection 96*, pages 262–288, 1996. `http://www.csl.sri.com/rewriting/`.
6. Will Clinger. Nondeterministic call-by-need is neither lazy nor by name. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 226–234. ACM, 1982.
7. René Cori and Daniel Lascar. *Logique Mathématique*. Masson, 1993.
8. C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Mu noz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual*, 1996. `http://coq.inria.fr/`.
9. Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland, 1958.
10. Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume II. North-Holland, 1972.
11. Solomon Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995. `http://math.stanford.edu/~feferman/`.
12. Matthias Felleisen. On the expressive power of programming languages. Technical report, Rice University, 1991. `http://www.cs.rice.edu/CS/PLT/Publications/`.
13. Cormac Flanagan and Matthias Felleisen. The semantics of futures. Computer Science Technical Report 94-238, Rice University, October 1994. `http://www.cs.rice.edu/CS/PLT/Publications/`.
14. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM. `http://para.inria.fr/join/`.
15. Jean Goubault. On Computational Interpretations of the Modal Logic S4. Technical report, University of Karlsruhe, 1996. `http://hypatia.dcs.qmw.ac.uk/data/G/GoubaultJ/S4/`.
16. John Hatcliff and Olivier Danvy. Thunks and the λ-calculus (extended version). *Journal of Functional Programming*, March 1997. `http://www.brics.dk/RS/97/7/index.html`.
17. Fergus Henderson. Private communications. started as discussion on comp.lang.functional, 1996.
18. Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revisedˆ5 Report on the Algorithmic Language Scheme, February 1998. `http://www-swiss.ai.mit.edu/~jaffer/Scheme.html`.
19. Todd B. Knoblock and Robert L. Constable. Formalized meta-reasoning in type theory. Technical Report TR86-742, Cornell University, Computer Science Department, March 1986.

20. John McCarthy. Recursive functions of symbolic expressions and their computation by machine (part I). *Communications of the ACM*, April 1960. `http://www-formal.stanford.edu/jmc/recursive.html`.
21. John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*. North-Holland, 1963. `http://www-formal.stanford.edu/jmc/basis.html`.
22. Sigurd Meldal and Michal Antonin Walicki. Nondeterministic Operators in Algebraic Frameworks. Technical report, Stanford CS Laboratory, 1995. `http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs/CSL-TR-95-664`.
23. Ben Olmstead. The Quines list, 1999. `http://magma.mines.edu/students/b/bolmstea/quines/`.
24. G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science, 1(2)*, pages 125–159, 1975.
25. Christian Queinnec. *Les Langages LISP*. InterEditions, 1994.
26. Eric S. Raymond, Editor. The on-line hacker Jargon File, version 4.0.0, 1996. `ftp://ftp.gnu.org/pub/gnu/jarg*`.
27. François-René Rideau. Metaprogramming and free availability of sources, January 1999. Translated from the french article "Métaprogrammation et libre disponibilité des sources" published in conference "Autour du Libre 1999". `http://fare.tunes.org/articles/ll99/index.en.html`.
28. François-René Rideau. Système de preuves pour un langage purement fonctionnel. Rapport de DEA, Université de Nice, 1997.
29. Emmanuel Saint-James. *La programmation applicative (de LISP à la machine en passant par le lambda-calcul)*. Hermès, 1993.
30. Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic Common Lisp. Technical report, University of Pennsylvania, Institute for Research in Cognitive Science, 1993. `http://www.neci.nj.nec.com/homepages/qobi/`.
31. Raymond Smullyan. *Recursion Theory for Metamathematics*. Oxford University Press, 1993.
32. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, February 1995. `http://www.cs.mu.oz.au/research/mercury/index.html`.
33. Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 45:161–228, 1936. `http://www.abelard.org/turpap2/turpap2.htm`.
34. Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. Technical Report CS-TR-78-687, Stanford University, Department of Computer Science, December 1978.