RAPIDO:
REASONING AND PROGRAMMING WITH INFINITE DATA OBJECTS

Défi *Société de l'information et de la communication* – 2014

Contents

1 Summary of the project

RAPIDO aims at gathering young researchers to investigate the applicability of proof-theoretical methods to reason and program on infinite data objects. The goal of the project is to develop logical systems capturing infinite proofs (proof systems with least and greatest fixed points as well as infinitary proof systems), to design and to study programming languages for manipulating infinite data such as streams both from a syntactical and semantical point of view. Moreover, the ambition of the project is to apply the fundamental results obtained from the proof-theoretical investigations (i) to the development of software tools dedicated to the reasoning about programs computing on infinite data, *e.g.* stream programs (more generally coinductive programs), and (ii) to the study of properties of automata on infinite words and trees from a proof-theoretical perspective with an eye towards model-checking problems.

2 Participants

Category	Name	Firstname	Current position	Impl in months	Role & resp.
Scientific Coordinator	Saurin	Alexis	CR CNRS (PPS)	32	scientific coordinator
Permanent members	Baelde	David	MCF ENS Cachan (LSV)	16	1, 2, 3, 4, coordinator of task 1
	Clairambault	Pierre	CR CNRS (LIP)	16	1, 2, 3, coordinator of task 3
	Pous	Damien	CR CNRS (LIP)	16	3, 4
	Régis Gianas	Yann	MCF Univ. Paris 7 (PPS)	16	1, 4, coordinator of task 4
	Riba	Colin	MCF ENS Lyon (LIP)	16	2, 3
	Tasson	Christine	MCF Univ. Paris 7 (PPS)	16	1, 2, coordinator of task 2
PhD Students	Pédrot	Pierre-Marie	PhD student at PPS	6	1, 3, 4
	Doumane	Amina	PhD student at PPS	36	1, 2, 3
ANR funded participants	PhD student		PPS	36	
	Post-doc A		PPS & LSV	24	
	Post-doc B		PPS & LIP	24	

3 Scientific context, positioning and objectives

3.1 Introductory examples

In order to motivate our project, let us begin with three examples which illustrate various problems encountered with infinite data. Those examples will be used later in the proposal to motivate and to illustrate some of our project challenges.

Sieve. A simple non trivial example of computing and reasoning on streams comes from ancient greek mathematics by presenting Eratosthenes' sieve (for computing prime numbers) as an algorithm working on streams. An elegant formulation of the sieve goes as follows:

```
filter(x:s,0,m)  = 0:filter(s,m,m)      sieve(0:s)   = sieve(s)
filter(x:s,n+1,m)= x:filter(s,n,m)      sieve(n+1:s)= n+1:sieve(filter(s,n,n))
from m           = m:(from (m+1))       primes       = sieve(from 2)
```

Here, `x`, `m`, `n` denote natural numbers and `s` denotes a stream while `x:s` denotes the stream with head `x` and tail `s`. `from n` denotes the stream of natural numbers greater or equal to n, `filter(s,j,m)` traverses the stream and turns to 0 one element out of $m+1$ while `sieve(s)` acts depending on the head of the stream: if it is a 0 it erases the head and sieves the tail of the stream and otherwise, it filters the tail of the stream accordingly and sieves to resulting stream.

When considering such a specification, one is naturally led to address the questions: are the elements resulting from the sieve all prime numbers? Is `sieve` really producing the stream of prime numbers, *i.e.* will it produce any arbitrarily large prefix of the stream of prime numbers in a finite amount of time?

Those questions correspond to the ***correctness*** and ***productivity*** of `sieve`. Moreover, since we are computing on infinite data, the ordering of subcomputations is crucial: if, for instance, one stubbornly filters the whole stream even when it is not needed to produce the next prime (that is if `filter` is unconditionally given priority over `sieve`), there is no hope to produce more than the first prime:

```
primes = sieve(from 2) = sieve(2:(from 3)) = 2:sieve(filter(from 3,1,1))
       = 2:sieve(filter(3:(from 4),1,1) = 2:sieve(filter(3:(from 4),1,1)
       = 2:sieve(3:filter(from 4,0,1) = 2:sieve(3:filter(4:(from 5),0,1)
       = 2:sieve(3:0:filter(from 5,1,1) = ...
```

An even more trivial situation would have been to expand the whole stream of natural numbers by forcing the evaluation of `from n` even if it is not demanded: `from 2 = 2:(from 3) = 2:3:(from 4)`... Therefore, evaluation has to be organized ***lazily*** if we wish to produce arbitrarily large primes.

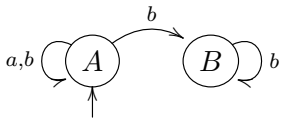
Automata and proof theory. One important tool to address the issues outlined above is the use of decidable logics (such as Monadic Second Order Logic (MSO) or the Modal μ-Calculus) to specify and reason about infinite data over finite alphabets. However, a successful marriage of these with our proof-theoretic perspective will require us to overcome subtle constructivity issues; this is the subject of our next introductory example.

Consider functions $f : \{a,b\}^\omega \to \{0,1\}^\omega$ such that

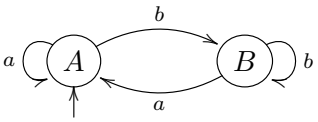
$$f(\alpha) \quad = \quad \begin{cases} 0^\omega & \text{if } \alpha \text{ contains only finitely many } a\text{'s} \\ 1^\omega & \text{if } \alpha \text{ contains only finitely many } b\text{'s} \end{cases} \tag{1}$$

There exist functions realizing the specification (1), since a stream $\alpha \in \{a,b\}^\omega$ contains either infinitely many a's or infinitely many b's, but there is no computable function realizing it. However, the specification (1) can be expressed by a formula of MSO, and the existence of functions realizing it can

be decided by translating that formula to a finite state automaton, and then checking its non-emptyness. In the process of translating (1) to a finite state automaton, one has to manipulate the following automata (initial states are marked with an arrow (↑)):



$\mathcal{B}$



$\mathcal{R}$

We equip the automaton $\mathcal{B}$ on the left with a *Büchi acceptance condition*, by stipulating that state B is final. Hence a run of $\mathcal{B}$ (on some input $\alpha \in \{a,b\}^\omega$) is accepting iff it goes through state B infinitely often, and $\mathcal{B}$ recognizes the $\alpha \in \{a,b\}^\omega$ such that α contains only finitely many a's. An accepting run ρ of $\mathcal{B}$ on input α starts from state A, and may loop in A while reading a's and b's in α, but eventually has to go and stay in state B. After that, α can no longer produce a's. The point where ρ switches to state B cannot be computed from α because there is no computable partial function mapping a stream $\alpha \in \{a,b\}^\omega$ to some $n \in \mathbb{N}$ such that no a's occurs in the infinite sequence $\alpha(n) \cdot \alpha(n+1) \cdots$ whenever such n exists. Hence reasoning on the runs of Büchi automata involves non-constructive principles.

To implement the specification (1), one also has to ***complement*** the automaton $\mathcal{B}$, typically by first ***determinizing*** it. A typical deterministic automaton equivalent to $\mathcal{B}$ is the automaton $\mathcal{R}$ (on the right). It is known that there is no deterministic Büchi automaton equivalent to $\mathcal{B}$. We thus have to equip $\mathcal{R}$ with a different acceptance condition, for instance a Rabin acceptance condition stipulating that an accepting run must see state B infinitely often and state A only finitely often.

The automata $\mathcal{R}$ and $\mathcal{B}$ recognize the same language, but there is no computable function mapping the run of $\mathcal{R}$ on some accepted input α to some accepting run of $\mathcal{B}$. In other words, even if automata on ω-words can be determinized and complemented, ***proving*** the correctness of these constructions involves some non-constructive reasoning.

Infinite proofs. Our final example will involve formal proof systems which, as we shall see in the next sections, are key ingredients to relate logic and computation as seen in the previous two examples. Assume we want to reason about arithmetic predicates such as $\mathsf{nat}(n)$ (resp. $\mathsf{even}(n)$) expressing that an individual n is a natural number (resp. a natural number of even parity). For instance, we should be able to establish that $\forall n.\ \mathsf{even}(n) \Rightarrow \mathsf{nat}(n)$. Given that both nat and even are read inductively, they should be equipped with ***induction principles***. The induction principle for nat is the well-known axiom of Peano arithmetic. For even we obtain the similar axiom schema, where the ***invariant*** P ranges over an arbitrary predicate:

$$P(0) \Rightarrow (\forall x.\ P(x) \Rightarrow P(x+2)) \Rightarrow (\forall x.\mathsf{even}(x) \Rightarrow P(x))$$

We can now choose nat as the invariant, and show both $\mathsf{nat}(0)$ and $\forall x.\ \mathsf{nat}(x) \Rightarrow \mathsf{nat}(x+2)$, to obtain a proof that every even natural number is indeed a natural number.

When writing proofs by induction, the key difficulty is to come up with the right invariant. Unlike in the previous example, one often has to craft invariants more general than the formula to be established. In technical terms, this corresponds to the fact that proof systems with induction do not satisfy the subformula property. While this is ultimately unavoidable when dealing with such rich logics, this obfuscates the study of proofs. In particular, the computational behavior of proofs is encapsulated in a too rigid way in the use of the induction principle. Another approach is to consider ***infinite proofs***. An

infinite but regular proof for the above theorem is given by the following cyclic presentation:

$$\dfrac{\dfrac{}{\vdash \mathsf{nat}(0)}\ \mathsf{constr} \qquad \dfrac{\dfrac{\dfrac{\vdots\ (\textit{cycle to root})}{\mathsf{even}(y) \vdash \mathsf{nat}(y)}}{\mathsf{even}(y) \vdash \mathsf{nat}(y+1)}\ \mathsf{constr}}{\mathsf{even}(y) \vdash \mathsf{nat}(y+2)}\ \mathsf{constr}}{\mathsf{even}(x) \vdash \mathsf{nat}(x)}\ \mathsf{case}$$

Here we made use of a case analysis rule which gave rise to two branches corresponding to the two constructions from which even can be derived: 0 is even; $y + 2$ is even if y is. In a similar but dual fashion we used constr in each branch to derive nat using one of its two defining constructions. It is very easy to come up with *invalid* infinite proofs, which derive false theorems. The key observation which makes the above proof ***valid*** is that with every iteration of the cycle we ***consume*** one more constructor of the even predicate in hypothesis. Since this predicate is inductive it must be finitely derived from its constructors, and thus the *infinite* branch corresponding to our cycle will only be *finitely* explored for each concrete instance of even.

This simplistic example illustrates a few important points. First, the cyclic presentation used to present infinite regular proofs explicits the recursive computation that underlies proofs by induction. Second, we see that, even though we are reasoning only on *finite* objects in our example, *infinite* proofs and computations are involved — all the above considerations would in fact apply in a symmetric fashion to infinite objects, the infinite ***consumption*** of an inductive assumption becoming an infinite ***production*** of a coinductive goal. Third, since proofs are infinite objects, they become subject to the kind of questions we had on streams. In our example, it is quite obvious that the cyclic presentation corresponds to a unique infinite proof tree, and that the infinite branches of that tree satisfy our infinite consumption condition. But this question can become quite complex as one starts to deal with several formulas interleaving inductive and coinductive predicates, or when computation at the level of proofs (by means of the *cut* rule) is used to construct derivations. In the latter case, we will need to ensure that such (lazy) computations are productive.

3.2 Context of the proposal

Our project consists in developping proof-theoretical methods to reason and program on infinite data objects. We review below the Curry-Howard approach in which we set our work, some elements of context about programming and reasoning about infinite objects as well as issues in this area, in particular current limitations with the Curry-Howard approach.

3.2.1 The Curry-Howard approach

Well-typed programs cannot go wrong. A type is a syntactic object representing an invariant of a computation. As such, it is a relevant ingredient to define static semantics, which are formal systems that assign meaning to programs without executing them. Type systems ensure that all programs that can be written meet certain correctness properties and ease the static analysis for other properties. Typically, the type-based approach to designing programming languages has been very successful in ensuring safety properties, as summed up by Robin Milner's slogan: "Well-typed programs cannot go wrong."

The type-based method has several characteristics that distinguish it from *e.g.* model-checking:

(i) It is an approach to the correctness of software which is *a priori*, contrasting with model-checking for instance. Here we recall the fundamental motto of programs which are *correct by construction*.

(ii) A well-designed programming language will enable programmers to write better programs[1].

(iii) It puts compositionality properties to the center of the picture[2].

(iv) By constraining how programs are written, it raises a new challenge: the constraints imposed on the structure of programs must meet the idiomatic way of expressing computations (here, computations on infinite data) without putting unbearable restrictions on the language expressiveness.

Well-typed programs meet their specification. A major step forward was taken by Martin-Löf, who designed in 1971 a formalism that was both a very expressive logical system and a rich programming language, referred to as dependent type theory. In contrast to earlier logical systems based on λ-calculus, such as Church's Higher-Order-Logic (HOL), Martin-Löf's dependent type theory was axiom-free, and made explicit the principle of proofs as programs and propositions as types, the so-called Curry-Howard correspondence.

Inspired from this seminal work, Huet and Coquant designed an axiom-free formalism, the Calculus of Construction [38], which expressiveness was in principle sufficient to tackle the mechanization of realistic mathematical and computer science developments. A refinement of this language, the Calculus of Inductive Constructions of Paulin [100] is the underlying logic of the Coq proof assistant [124], one of the most successful tools to write and to verify formal proofs on a computer. Amongst all the achievements that were made possible thanks to Coq, the proof of the Feit-Thompson theorem by Gonthier [59] and the proof of the Compcert semantic-preserving C compiler by Leroy [89] have been recognized as major evidences that the certification of complex mathematical proofs and programs can be mechanized with the help of a type-based formal system.

A correspondence serving as a fruitful methodological guideline. During the 90s, the Curry-Howard correspondence was extended beyond intuitionistic logic and has been used a methodological guideline to better understand logical mechanisms by looking at them from computational perspective and to better understand computational mechanisms from a proof-theoretical point of view. For instance, certain control operators present in programming languages (*e.g.* the call with current continuation operator) were shown to correspond to some laws used in classical reasoning (*e.g.* the double-negation elimination). Linear logic is also a typical example of a logic that has a deep connection with programming languages since it considers the usage of hypotheses as the consumption of (possibly limited) resources. Looking for the computational contents of mathematical proofs is a fruitful approach for reverse mathematics, *i.e.* the study of the dependencies between proofs and the axioms they effectively use.

Computational restrictions entailed by type systems. Type systems usually constrain the way programs are expressed mostly because they use syntactical criteria to accept or reject user programs. In a dependently-typed setting, the decidability of type-checking crucially depends on the ability to decide the equivalence between two computational terms. If the type system is to be used as a logic, the consistency proof of this logic usually relies on the strong normalization property: every program evaluation must terminate.

These computational restrictions imply challenging problems when designing a typed programming language because there is a constant tension between the expressiveness of the computational language and our ability to capture its static semantics using syntactical tools.

[1]The structure of programming languages impacts the correctness of programs: indeed, a poorly designed programming language can be an endless source of bugs while a well-designed programming language helps a lot in meeting a specification.

[2]In such a way, it naturally allows to verify parts of program developments agreeing with the fact that programs are developed in an incremental way and reuse preexisting code. As such, there is also hope that when an error is recognised, it can be tracked back to the programmer in a way which is meaningful and can help him to modify her development adequately.

3.2.2 Infinite objects in programming and logic

Infinite data are everywhere. The present section opened with the example of a small program computing Eratosthenes' sieve of prime numbers. This illustrates how naturally infinite data can enter the picture of computing with finite devices.

More generally, stream-like and infinite data structures are ubiquitous in computer science. By representing discrete potentially infinite information flows, streams can be used to formalize and study several situations of increasing significance in modern computer science, such as network communication flows, audio and video signals flows, etc. Additionally, stream-like data can be used in order to model the potentially infinite interactions between a program and its environment (typically like an OS, a web server and more genrally reactive programs).

Having strong methods and tools to represent and to reason about infinite and stream-like data and computations is of crucial importance in order to ensure the correctness of these kinds of software.

Actual versus potential infinity. With the sieve, streams of natural numbers were structured as potentially infinite lists. This view, which is common in *lazy* programming languages (such as Haskell), simply considers those infinite data structures as usual data structures where the common well-foundedness requirements have been dropped.

On the other hand, the coalgebraic approach to infinite data structures (see below) reveals a different picture by considering objects which are truly infinite. This discrepancy is reminiscent of Aristotle's distinction between potential and actual infinity and this explains that different conceptual tools were developed in each direction.

Infinite data as non-well-founded data structures. In this view, infinite data can be viewed as resulting from a non-terminating process of constructing non-well-founded data. Infinite λ-terms arise naturally in the very basic framework of pure λ-calculus, for instance from the consideration of Böhm trees or other Böhm-like trees [43] which can be understood as infinite normal forms for various notions of convergence. An infinite Böhm tree corresponds to the infinite normal forms of a finite λ-term, that is to a program which ***produces*** more and more information but never stops computing, contrarily to a program which runs forever without producing the least bit of information.

Infinite objects are then typically added by an algebraic or topological completion process from finite objects, typically by ideal or metric completion. For instance, Böhm trees for untyped λ-terms are typically obtained by an ideal completion from finite $\lambda\Omega$-normal forms and indeed they serve as infinite normal forms of finite λ-terms.

Infinitary rewriting. In this direction, ***infinitary rewriting*** [75] is an extension of usual rewriting where reduction sequences of ordinal length greater than ω are allowed. A notion of convergence of an infinite reduction is thus given by means of a metric topology induced by the usual tree distance (and infinite terms can be viewed as the metric completion of finite terms): convergent reductions are then Cauchy sequences of rewritings[3] of first-order terms or λ-terms.

Fibonacci sequence can thus be phrased in the setting of infinitary rewriting as follows:

```
zipadd(x:s,y:t) = (x+y):zipadd(s,t)     tail(x:s) = s
fib = 0:1:zipadd(fib,tail(fib))
```

In the setting of infinitary rewriting, infinite data is treated just like finite data and special conditions are imposed on reductions of infinite length; this is typically the case of infinitary λ-calculus [76, 13] which extends traditional λ-calculus with infinite terms and possibly transfinite reductions.

[3] Possibly satisfying additional requirements, *eg* for strongly convergent sequence, the depths of the redexes is required to tend to infinity.

Productivity. With the description of the computation of a Böhm tree given above, one is faced with two different types of infinite behaviours, depending on the presence of Ω nodes of infinite branches: Ω nodes correspond to infinitely long sequences of head reduction, while infinite branches correspond to an infinite computation reaching infinitely often head normal forms (thus constructing the infinite branch). This last case is a case of productivity [119] which replaces termination property when dealing with infinite data. Indeed, when producing infinite data (typically streams), one does require to terminate after finitely many steps but one requires than every finite prefix can be computed in finite time. For instance, `sieve` and `fib` are productive.

Productivity is undecidable in general, exactly in the same way as termination is, but productivity is far less understood than termination in the sense that designing fragments of programs on which productivity can be decided (just as usual type systems ensure termination for programs working on inductive data) results in cumbersome restrictions on languages.

In proof assistants, for example, productivity is usually ensured by a syntactic condition called ***guard condition*** that restricts recursive occurrences of the defined object to be placed immediately below constructors. For instance, the definitions of `sieve` and `fib` are not guarded (while `zipadd` is guarded).

Causality and Reactive Programming. Stronger than productivity is causality, which requires that the nth element of the result shall only be determined by the first n elements of the input streams (for instance, *fib* is causal while `sieve` is not causal.

This notion of causality is typical from reactive programming. In functional reactive programming [51], programs are functions over signals or behaviours, which are values varying with time. In such a model with discrete time, signals can be viewed as streams and the above notion of causality actually correspond to requiring that the value of a function does not depend on the future.

Infinite data as coinductive data structures. On the other hand, the coalgebraic approach [69] to infinite data structures reveals a different picture. While finite data structures (booleans, natural numbers, finite lists or trees) are modelled by inductive types in the typed setting and categorically as initial algebras, infinite data structures (such as streams, infinite trees, ...) are modelled by coinductive types and categorically modelled by final coalgebras.

While in the former approach, finite and infinite objects are dealt with using the same methods (infinite tree are just trees... which happen not to be finite), the latter, coalgebraic view offers a different treatment for finite and infinite objects: finite objects are manipulated by means of their constructors (which serve to define them), infinite objects are manipulated by means of destructors (which serve to observe them). Specific reasoning principles come with this coinductive modelling, namely the coinduction proof principle using bisimulations. For instance, `fib` can be given an alternative definition by its destructors:

```
hd(fib) = 0     hd(tl(fib)) = 1     tl(tl(fib)) = zipadd(fib,tl(fib))
```

Such kinds of definitions, reminiscent of differential equations, allows one to use bisimulations or bisimulations "up-to", to prove the equivalence of two programs (see, e.g., the work by Jan Rutten on the differential stream calculus [110])

Proof-theoretical dualities. In the last 25 years, the notion of proof-theoretical duality emerged as a central theme in proof theory, in particular in the context of linear logic [56]. This notion of proof-theoretical duality turned out to be an extremely fruitful and successful concept for structuring logic and for studying programming language theory.

These dualities are one of the stepping stones for various interactive modelling of logic and programming languages, with game semantics which developed from the early nineties with the fully abstract game-model of PCF [68, 3] or ludics [58].

Moreover, they provided logical and mathematical tools on which to make formal some of the computational dualities which are central to programming studies (input/output, program/environment, result of a computation/continuation) which were crucial in understanding on the one hand the computational content of classical logic and the logical formalization of control operators, for instance with the design of $\lambda\mu$-calculus [98], and dualities between evaluation strategies such as call-by-name and call-by-value [39, 62] on the other hand.

Polarity in logic. About the same time, the notion of polarity was recognized as an essential concept in the study and the design of logic. In the context of linear logic, this began with Andreoli's study of focalization [5] and Girard analysis of classical logic [57], but the concept of polarity in logic is far from being restricted to linear logic and, quite the contrary, they can be said to have emerged in the proof theory of constructive logic[4].

This notion of polarity turned out to be particularly fruitful for programming language theory, in particular: (i) positive polarities/types therefore represent data that are characterized by their right introduction rules, ie the way their are constructed. Finite data falls in this category; (ii) negative polarities/-types represent data that are characterized thanks to their left introduction rules, that is the possible ways to use them, or how to observe them in a computation. Functions, environments, but more generally infinite, coinductive objects fall in this category. Typically, a function is defined by pattern matching on its input and a stream by its behaviour when heads or tails are requested.

Interactive semantics. In the same time frame and following this activity around the notions of polarity and duality in proof theory, interactive semantics of programming languages emerged. On the one hand, Game semantics [68, 3] were driven by semantic motivations, and in particular by the long-standing open problem known as full abstraction for PCF aiming at characterising abstractly the notion of sequentiality. On the other hand, Girard introduced ludics [58] as a research programme aiming to reconstruct logic from the notion of interaction: one starts with an untyped notion of interaction, and later reconstructs types through a realizability construction.

Interactive semantics are, in particular, denotational semantics: this means that they provide an abstract and compositional description of programs. But in constrast to many approaches to denotational semantics (such as domains), games do more than just describing the input-output behaviour of programs: they describe exhaustively their behaviour under a notion of evaluation. This key property means that beyond properties about the *result* of a program, games provide a framework in which one can formalize – and prove – non-functional properties, such as execution time or memory consumption. For instance, plays in strategies interpreting λ-terms in Hyland-Ong games [68] are in strong correspondence with their execution through the Krivine Abstract Machine [40].

Besides being a central object of study in a Curry-Howard perspective, games are in general of paramount importance when it comes to the study of infinite objects. For instance, model-checking relies heavily on game-theoretic tools: a desirable property of a system is often formalized as a winning condition on a game obtained from the system, and the satisfaction of this desirable property amounts to the existence of a winning strategy for one of the players. For instance, parity winning conditions are decidable on finite games, and are expressive enough to encode properties expressed in the modal μ-calculus (see below). It is worth noting that in a compositional perspective, parity winning conditions can also be mixed with game semantics of programming languages to give a composition account of termination and productivity for a language with inductive and coinductive types [31].

[4] In some sense, one could say that linear logic made formal and internalized principles that were guiding principles in the design of constructive logic.

Logics for specifying and verifying infinite objects. Automata on infinite objects (words and trees) and their connection to logics (such as Monadic Second-Order Logic (MSO) and the Modal μ-Calculus) now form a mature field (see e.g. [60, 101]). Central results, in particular *w.r.t.* our objectives, are determinization of automata on infinite words (also known as McNaughton's Theorem) and complementation of automata on infinite words (also known as Büchi's Theorem). Concerning infinite trees, the central point is the connection of automata and logic with infinite two-players games, in particular the positional determinacy of parity games, as well as the simulation of alternating tree automata by non-deterministic automata (the Simulation Theorem [92]), used to complement non-deterministic tree automata and to interpret quantifier alternations of MSO. All these automata constructions involve non-constructive arguments in their correctness proofs, most notably Weak Koening Lemma (WKL) for complementation.

Linear-Time Temporal Logic (LTL) [102] is a widely used propositional modal logic with temporal operators. Its expressive power corresponds to first-order logic on ordering (a first-order fragment of MSO on infinite words), or equivalently to ω-regular languages definable by star-free rational expressions [101]. Operationally, it is equivalent to a very weak form of alternating automata [126]. Though less expressive than other logics, LTL has proved to be very useful for ***model checking***. The idea of model-checking is to come up with a formal specification of the system of interest, to build the corresponding automaton $\mathcal{A}_{\mathcal{S}}$ and to check wether it accepts all possible computation traces generated by the system. To model-check systems modeled by finite-state automata against specifications expressible in LTL [127], one simply tests the emptyness of the product of the model with the automaton $\mathcal{A}_{\neg\mathcal{S}}$ corresponding to the negation of the specification.

Curry-Howard and non-constructivity. Incorporating logics such as MSO or the modal μ-calculus into type theory could allow both to extend automation in proof assistants to reason about infinite computation, and to extend the application of these logics to more expressive programming languages.

The Curry-Howard correspondence has been extended to classical logic thanks to Griffin's discovery that some control operators can be typed by classical laws [61]. However, the concrete use of this extension is still problematic for two reasons. Firstly, there is up to now no classical extension of *dependent types* (such as used e.g. in the proof assistant Coq). Secondly, the behaviour of extracted programs using control operators is in general far from straightforward. In particular, computational interpretations of proofs obtained with Krivine's *Classical Realizability* [86] are sensitive to the non-constructive arguments used, and to their polarities.

It follows that incorporating the computational interpretation of MSO and the Modal μ-Calculus into the Curry-Howard proofs-as-programs correspondence, in particular to obtain extraction procedures, involves a proper proof-theoretical understanding of the underlying non-constructive principles and their uses in computational interpretations of these logics. This point is also important *w.r.t.* formalizations in the proof assistant Coq, since its metatheory makes it problematic to build data according to non-constructive principles (typically the excluded middle).

3.3 State of the art

Coinduction in proof assistants. The Coq proof assistant [124] allows the definition of coinductive types and of infinite proofs using cofixpoints [55]. To preserve the strong normalization property, a corecursive definition must fullfill a syntactic criterion of productivity: every corecursive reference must appear as an argument of a coinductive constructor. For instance, the following definitions are accepted:

```
CoInductive LList := LCons : nat → LList → LList | LNil : LList.
CoFixpoint from (n: nat) := LCons O (from (S n)).
```

while the following one is not:

```
CoFixpoint filter (p : nat → bool) (l : LList) : LList :=
```

```
match l with
| LNil ⇒ LNil
| LCons a l' ⇒ if p a then LCons a (filter p l') else filter p l'
end.
```

(Indeed, the second corecursive call to `filter` is not *guarded* by a constructor.)

On the one hand, this last example is rejected legitimately since the term `filter (fun _ ⇒ false)` cannot be applied to an infinite stream without diverging. On the other hand, the programmer should be able to add an extra logical argument to `filter` to justify, for instance, that there is only a finite number of steps before the predicate `p` is verified by an element of the stream. Such a logical argument can be expressed in Coq by nesting an induction inside the coinduction [37, 19] which requires a significative reformulation of the program. More recent work [67] introduces a semantically justified productivity argument based on lattice theory formalized inside Coq itself.

Beyond mere usability issues, deeper metatheoretical problems arise because of coinductive predicates: as pointed out by Gimenez and Oury, and later explained by McBride[90], coinduction in Coq breaks type preservation. A line of work by Abel and Pientka consists in the unification of induction and coinduction in a way that is compatible with dependent types. By introducing a concept of copatterns [2, 1] – a language to define computations over infinite structures through "observations", a form of copattern-matching – they indeed recovered subject reduction. Besides, they have shown that size annotations put on types can provide a better criterion to check productivity.

Infinite proofs. One of the oldest and most obvious goals of proof theory is to provide simple certificates to justify theorems. Those certificates should be easily checkable and are thus required to be finite objects. When reasoning about (co)inductive specifications, induction and coinduction principles provide a well-established solution to that goal, from the early work of Kozen [82, 83] to recent investigations in linear logic [11]. Another approach, somehow starting from the semantics of our infinite (co)inductive objects, is to consider infinite proofs. This goes in two steps: one first naively extends proofs from finite to infinite trees, which breaks the consistency of the logical system; then one devises a *validity* or *guard* condition on infinite branches of proofs in order to restore consistency. Once this is done, one might worry again about finite proof objects by considering finitely presented infinite proofs, e.g. regular ones, and compare the expressivity of these finite proof objects with that of proofs by (co)induction. We note here the work of Luigi Santocanale on cyclic proofs for a simple, purely additive logic, their relationship with the category of parity games [113, 114] and more recently a cut elimination result for (an extension of) these proofs [52]. Brotherston has also studied infinite and circular proofs for arithmetic [28, 27], establishing the completeness of the infinite calculus and leaving a (still open) conjecture regarding the equi-expressivity of cyclic proofs and proofs by induction. More broadly, tableaux for μ-calculi can be considered as infinite proof systems, even though their goal is to establish satisfiability rather than validity.

Classical logic and streams. The $\Lambda\mu$-calculus [115] is the Böhm completion of Parigot's original $\lambda\mu$-calculus [99], a term calculus for classical natural deduction. In [115] conjectured that $\Lambda\mu$ was connected with stream calculi. Indeed, μ-abstraction abstracts over evaluation contexts, or continuations, and can thus be viewed as potentially infinitary λ-abstractions. This is emphasized by reduction:

$$\boxed{\mu\alpha.t \longrightarrow_{\mathsf{fst}} \lambda h.\mu\theta.t\{(u)\ h\theta/(u)\ \alpha\} \qquad h,\theta \notin \mathsf{FV}(t)}$$

where h abstracts over the head of the stream α while θ abstracts over the tail of the stream. This viewpoint on μ-abstraction as *stream abstraction* [115] was since then supported by several evidences, from the study of $\Lambda\mu$-Böhm trees [117] to the definition of Nakazawa's Stream models for the $\Lambda\mu$-calculus [95, 94].

Automata and proof theory. Ludics provides an elegant treatment of the theory of automata on finite words [123] in particular due to the fact that languages and machines leave in the same world (ludics is said to adopt a monist view of logic) and that acceptance of a word by an automaton is modelled *via* a proof-theoretical duality: automata and words are modelled as proofs and a word w is accepted by an automaton $\mathfrak{A}$ if the interaction (or cut-elimination) of their proof-theoretical counterparts is successful. As a result, one can extract, from a set of proofs interacting properly with an (encoding of an) automaton $\mathfrak{A}$, the language recognized by $\mathfrak{A}$.

Fundamental tools for proof theoretical approaches to automata are complete deductions systems for MSO and the Modal μ-Calculus. It is known since the early 70's [118] that MSO on ω-words can be axiomatized as a subsystem of second-order Peano's arithmetic (PA2). This result has been simplified using model-theoretic tools [108]. Completeness of the Modal μ-Calculus is a difficult result in the general case [128], but the special case of infinite words is substantially simpler [74].

A Curry-Howard interpretation of MSO can be obtained by a direct application of Krivine's *Classical Realizability* [86]. However, it seems difficult to directly get interesting results this way since proofs seem to lack structure in order to reflect the intuitions given by automata. A possibility is to devise proof-theoretical interpretations similar to translations of formulas to automata. Preliminary results have already been obtained in [109], which proposes a interpretation based on Cohen's forcing of MSO on ω-words in Weak MSO, motivated by the recent Curry-Howard interpretation of forcing by Krivine [87]. Usual reductions of MSO to Weak MSO are consequences of McNaughton's Theorem (determinization of automata on ω-words).

Higher-Order Model Checking: Ong's breakthrough. Following Ong's celebrated result [97] on the decidability of MSO on infinite trees generated by Higher-Order Recursion Schemes (an abstraction of functional programs, and a generalization of automata), a community developed around the theory and practice of the verification of higher-order programs through MSO model-checking. Amid an intense theoretical activity around the extension and improvement of the decision algorithms, this led to practical model-checking tools, such as MoCHi [79] for a fragment of Ocaml.

Decidability proofs for higher-order model-checking rely on a fine-grained analysis of computation, provided in particular by interactive semantics. Ong's original solution used Hyland-Ong game semantics. Since then, various alternative proofs have been proposed, in particular using *type systems* [78]. A *compositional* approach (following the notion of composition of programs provided by the λ-calculus) was recently proposed using a combination of types and game semantics [125]. Among complementary approaches, let us mention [112] based on denotational semantics, as well as the powerful MSO-compatibility of evaluation [111].

Connections between FRP and LTL. In the past three years, people started to realize that Functional reactive programming and constructive versions of LTL [102] formed a Curry-Howard correspondence [70, 72]. This move was coincident with development of denotational semantics for FRP (see for instance [84]). This induced recent refinements on type systems for FRP in order to ensure more subtle properties on FRP programs, such as liveness of fairness [71, 30].

Moreover, these logic and programming language have been equipped with semantical counterparts [84, 73, 72] that allows to abstract from particular syntax or implementation and to use new tools to prove properties on the logical side or the programming side.

4 Scientific objectives and program

4.1 Objectives of the project

Overall, the objectives of **RAPIDO** consist in the design of good proof-theoretical tools to reason and program with infinite data. More precisely, our detailed objectives are organized in the following four tasks.

1/ Designing proof systems and calculi for infinite data. We will design and investigate proof systems and programming languages dealing with infinite objects. More specifically, we will develop a study of notions such as productivity, laziness, causality and non-determinism, in the computational behavior of proofs and programs. We will approach this question from various angles. On the proof-theoretical side, we will develop further infinite proofs in sequent calculus, and work on ludics on the question of non-determinism. On the computational side, we shall build calculi for computing with streams and in particular will investigate transfinite calculi which naturally arise from classical λ-calculi. We also plan to address issues regarding lazy evaluation, its integration with control and effects, and applications to functional reactive programming.

2/ Investigating the semantics of infinite proofs and computations. In this direction, we will study the interactive and denotational semantics of our logical and computational objects. It shall be noted that we are interested in proof-theoretic semantics, not only validity semantics. We are confident that, once again, semantics will provide useful guidelines in developing new syntaxes as well as powerful proof methods.

3/ Approaching automata theory via proof theoretical methods. As part of the exciting convergence of verification and proof theory that has started with the work of Ong on higher-order model-checking, we will develop a Curry-Howard approach to automata on infinite objects, in which we view a language of infinite words or trees as a *type* (or an object in categorical settings). This direction deals with axiomatic aspects, applications of interactive semantics and proof-theoretical tools to automata, as well as algorithmic benefits of concrete semantics and Curry-Howard enhencements to Higher-Order Model-Checking.

4/ Developing software to ease reasoning about programs using infinite data. Finally, we plan a number of software developments. Our main goal in doing so is to validate and spread our results by building tools that will be usable, e.g. in the Coq proof assistant, to help other researchers working on proofs or programs involving infinite objects or behaviors. But we also view the development of prototypes as part of the research process, as it allows to quickly experiment with ideas and test them on significant examples. Finally, we will also apply some of our theoretical investigations to end-user software in the domain of multimedia streaming.

4.2 Scientific program

Our work program is divided in four scientific tasks which are described below. Two tasks of a foundational nature in terms of proof-theoretical studies and two tasks which apply the results of the first two tasks, to the theory of automata on infinite objects (this task is still a theoretical one but several subtasks depend on the first task) and a software development task.

TASK 1. DESIGNING PROOF SYSTEMS AND CALCULI FOR INFINITE DATA.

This task is devoted to the design and investigation of proof systems and programming languages dealing with infinite objects. More specifically, we will develop a study of notions such as productivity, laziness, causality, in the computational behavior of proofs and programs. The subtasks below approach this question from various angles: we shall consider sequent calculus, ludics, and several relevant variants of the λ-calculus.

Who does what. Task coordinator: D. Baelde – (1.1) D. Baelde, P. Clairambault, A. Saurin, (1.2) A. Saurin, A. Doumane, (1.3) A. Saurin, (1.4) A. Saurin, Y. Régis-Gianas, P.-M. Pédrot, (1.5) C. Tasson, A. Saurin, postdoc/PhD student. (1.6) Y. Régis-Gianas, A. Saurin

Subtask 1.1 – ***Infinite proofs.***

Subtask 1.2 – ***Non-determinism in Ludics.***

Subtask 1.3 – ***Infinitary λ-calculi and classical logic.***

Subtask 1.4 – ***Lazy evaluation and computing with infinite objects: Dual of lazyness - Resource Calculus for lazy evaluation - Lazyness and control.***

Subtask 1.5 – ***Proof systems for LTL and correspondence with Functional Reactive Programming: cut-elimination and causality***

Subtask 1.6 – ***Dependent copatterns in $\overline{\lambda}\mu\tilde{\mu}$***

TASK 2. INVESTIGATING THE SEMANTICS OF INFINITE PROOFS AND COMPUTATIONS.

In parallel with the syntactic and proof-theoretic investigations of the previous task, we will also examine semantic aspects of computation with infinite data objects. Semantics here will also provide guidelines and insights in developing new syntaxes (in particular with respect to non-determinism) as well as powerful proof methods.

Our semantic developments will follow two main themes. The first one is the developments of interactive semantics of computation on infinite data, which proved in the past to be a fitting complement for the more syntactic investigations. This will be done both with respect to standard notions of evaluations, and with respect to lazy evaluation. Our second theme will be the semantics of Functional Reactive Programming (FRP). In this setting, we will in particular be addressing the challenge of developing new models of FRP, adapted to *continuous* time.

Who does what. Task coordinator: C. Tasson – (2.1) A. Saurin, P. Clairambault, C. Tasson, P.-M. Pédrot, (2.2) A. Saurin, D. Baelde, P. Clairambault, A. Doumane, (2.3) A. Saurin, C. Tasson.

Subtask 2.1 – ***Game semantics of lazy evaluation.***

Subtask 2.2 – ***Interactive semantics of μMALL.***

Subtask 2.3 – ***Functional Reactive Programming (FRP), from discrete time to continuous time.***

TASK 3. APPROACHING AUTOMATA THEORY VIA PROOF THEORETICAL METHODS.

This task is devoted to the Curry-Howard interpretation of automata on infinite objects. The common pattern is that a language of infinite words or trees is a *type* (or an object in categorical settings).

The first subtask concerns the *Ludic* approach to ω-automata. Ludics provides an elegant treatment of proof-theory, allowing a groundbreaking "automata-as-proof" approach, in which automata and data live in the same world, their interaction beeing modeled by cut-elimination. This development will be based on what we have learnt in the two previous objectives, in particular on non-determinism in ludics to properly handle Büchi automata. Subtasks 3.2 – 3.4 are based on a more traditional "automata-as-formula" approach: Subtask 3.2 concerns axiomatizations and deduction systems for MSO and the Modal μ-Calculus, of fundamental importance in our proof-theoretical approach; subtask 3.3 deals with categorical treatements of tree automata as models of intuitionistic linear logic, while substask 3.4 concerns the application to automata of proof-theoretical tools used for (weak) fragments of second-order arithmetic. The coalgebraic approach to automata has proved to provide new efficient algorithms for finite words, that we plan to extend to ω-words in subtask 3.5. Last but not least, subtask 3.6 proposes enhencements to Higher-Order Model-Checking based on Curry-Howard, and applications of the algorithmic benefits of game semantics.

Who does what. Task coordinator: P. Clairambault – (3.1) A. Saurin, A. Doumane (3.2) D. Baelde, C. Riba, (3.3) P. Clairambault, C. Riba, (3.4) C. Riba, P.-M. Pédrot, (3.5) D. Pous, Postdoc B, (3.6) P. Clairambault, C. Riba, Postdoc/PhD Student.

Subtask 3.1 – ***Ludics and ω-automata: Automata as proofs.***

Subtask 3.2 – ***Axiomatizations***

Subtask 3.3 – ***Categories of Tree Automata***

Subtask 3.4 – ***Applications to automata of proof-theoretical tools from arithmetic: Forcing interpretation of MSO - Dialectica interpretation of MSO.***

Subtask 3.5 – ***Algorithms for automata on infinite datastructures***

Subtask 3.6 – ***Higher-Order Verification: Higher-order Model-Checking - Algorithmic game semantics.***

TASK 4. DEVELOPING SOFTWARE TO EASE REASONING ABOUT PROGRAMS USING INFINITE DATA.

The development of software prototypes serves two roles in our project. It is obviously a way to validate our results, and increase our visibility by making our theoretical work useful for people outside our immediate community. But software will also directly support our research by providing a valuable guide and helping us to experiment with more interesting examples. We present below several projects that we have in mind, mostly targetting other researchers (users of proof assistants) but also end-user software in the domain of multimedia streaming.

Who does what. Task coordinator: Y. Régis-Gianas – (4.1) Y. Régis-Gianas, P.-M. Pédrot, (4.2) Y. Régis-Gianas, P.-M. Pédrot, (4.3) D. Baelde, postdoc/PhD, (4.4) P.-M. Pédrot, D. Pous.

Subtask 4.1 – ***CoFunction/CoEquation***

Subtask 4.2 – ***Prototyping***

Subtask 4.3 – ***Application to multimedia streaming.***

Subtask 4.4 – ***Decision procedures for mechanized formalization***

References

[1] Andreas Abel and Brigitte Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *ICFP*, pages 185–196. ACM, 2013.

[2] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *POPL*, pages 27–38. ACM, 2013.

[3] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[4] R. M. Amadio and P.-L. Curien. *Domains and Lambda-Calculi*. cttcs. CUP, 1998.

[5] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[6] Andrew W Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[7] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In *FLOPS*, volume 7294 of *LNCS*, pages 32–46. Springer, 2012.

[8] Zena M. Ariola, Hugo Herbelin, and Alexis Saurin. Classical call-by-need and duality. In *Typed Lambda Calculi and Applications*, volume 6690 of *lncs*, 2011.

[9] J. Avigad. Formalizing Forcing Arguments in Subsystems of Second-Order Arithmetic. *Annals of Pure and Applied Logic*, 82(2):165–191, 1996.

[10] David Baelde. On the proof theory of regular fixed points. In *TABLEAUX'09*, volume 5607 of *LNCS*, pages 93–107, Oslo, Norway, July 2009. Springer.

[11] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1), January 2012.

[12] David Baelde, Romain Beauxis, and Samuel Mimram. Liquidsoap: a high-level programming language for multimedia streaming. In *SOFSEM'11*, volume 6543 of *LNCS*, pages 99–110, Nový Smokovec, Slovakia, January 2011. Springer.

[13] Henk Barendregt and Jan Willem Klop. Applications of infinitary lambda calculus. *Inf. Comput.*, 207(5):559–582, May 2009.

[14] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In *FLOPS*, pages 114–129, 2006.

[15] Michele Basaldella and Claudia Faggian. Ludics with repetitions (exponentials, interactive types and completeness). *Logical Methods in Computer Science*, 7(2), 2011.

[16] Michele Basaldella, Alexis Saurin, and Kazushige Terui. From focalization of logic to the logic of focalization. *Electr. Notes Theor. Comput. Sci.*, 265:161–176, 2010.

[17] Michele Basaldella and Kazushige Terui. Infinitary completeness in ludics. In *LICS*, pages 294–303, 2010.

[18] Michele Basaldella and Kazushige Terui. On the meaning of logical completeness. *Logical Methods in Computer Science*, 6(4), 2010.

[19] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *LNCS*, pages 102–115. Springer, 2005.

[20] V. Blot and C. Riba. On Bar Recursion and Choice in a Classical Setting. In *APLAS*, volume 8301 of *LNCS*, pages 349–364. Springer, 2013.

[21] Richard Blute, Thomas Ehrhard, and Christine Tasson. A convenient differential category. *Cah. Topol. Géom. Différ. Catég.*, 53(3):211–232, 2012.

[22] Filippo Bonchi, Daniela Petrisan, Damien Pous, and Jurriaan Rot. Coinduction up-to in a fibrational setting. In *Proc. CSL-LICS*. ACM, 2014. to appear.

[23] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *POPL*, pages 457–468. ACM, 2013.

[24] Bianca Boretti. *Proof Analysis in Temporal Logic*. PhD thesis, Universita degli studi di Milano, 2009.

[25] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In *Proc. CAV*, volume 3114 of *LNCS*, pages 372–386. Springer, 2004.

[26] Thomas Braibant and Damien Pous. Deciding Kleene algebras in Coq. *LMCS*, 8(1):1–16, 2012.

[27] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In Bernhard Beckert, editor, *TABLEAUX*, volume 3702 of *LNCS*, pages 78–92. Springer, 2005.

[28] James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *LICS*, pages 51–62. IEEE Computer Society, 2007.

[29] Kai Brünnler and Martin Lange. Cut-free sequent systems for temporal logic. *J. Log. Algebr. Program.*, 76(2):216–225, 2008.

[30] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. Fair reactive programming. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 361–372. ACM, 2014.

[31] Pierre Clairambault. Least and greatest fixpoints in game semantics. In *FOSSACS*, pages 16–31, 2009.

[32] Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and martin-löf type theories. In *TLCA*, pages 91–106, 2011.

[33] Pierre Clairambault, Julian Gutierrez, and Glynn Winskel. The winning ways of concurrent games. In *LICS*, pages 235–244, 2012.

[34] Pierre Clairambault and Andrzej Murawski. Böhm trees as higher-order recursive schemes. In *FSTTCS*, 2013.

[35] Guillaume Claret, Lourdes Del Carmen Gonzalez Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *Interactive Theorem Proving*, Rennes, France, 2013.

[36] Thomas Colcombet. The Theory of Stabilisation Monoids and Regular Cost Functions. In *ICALP (2)*, volume 5556 of *LNCS*, pages 139–150, 2009.

[37] Thierry Coquand. Infinite objects in type theory. In *TYPES'93*, volume 806 of *LNCS*, pages 62–78. Springer, 1994.

[38] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[39] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *International Conference on Functional Programming*, pages 233–243, 2000.

[40] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game semantics & abstract machines. In *LICS*, pages 394–405, 1996.

[41] V. de Paiva. A Dialectica-like Model of Linear Logic. In *Category Theory and Computer Science*, volume 389 of *LNCS*, pages 341–356. Springer, 1989.

[42] U. de'Liguoro and A. Saurin, editors. *Proceedings First Workshop on Control Operators and their Semantics*, volume 127. EPTCS, September 2013.

[43] Mariangiola Dezani-Ciancaglini and Elio Giovannetti. From Böhm's theorem to observational equivalences: an informal account. *Electr. Notes Theor. Comput. Sci.*, 50(2):83–116, 2001.

[44] Thomas Ehrhard. Finiteness spaces. *Mathematical Structures in Computer Science*, 15(4):615–646, 2005.

[45] Thomas Ehrhard, Michele Pagani, and Christine Tasson. The Ccomputational Meaning of Probabilistic Coherence Spaces. In Martin Grohe, editor, *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS 2011)*, IEEE Computer Society Press, pages 87–96, 2011.

[46] Thomas Ehrhard, Michele Pagani, and Christine Tasson. Probabilistic coherence spaces are fully abstract for probabilistic PCF. In P. Sewell, editor, *The 41th Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL14, San Diego, USA. ACM, 2014.

[47] Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.

[48] Thomas Ehrhard and Laurent Regnier. Böhm trees, Krivine's machine and the Taylor expansion of lambda-terms. In *Logical Approaches to Computational Barriers*, pages 186–197. Springer, 2006.

[49] Thomas Ehrhard and Laurent Regnier. Differential interaction nets. *Theor. Comput. Sci.*, 364(2):166–195, 2006.

[50] Thomas Ehrhard and Laurent Regnier. Uniformity and the Taylor expansion of ordinary lambda-terms. *Theoretical Computer Science*, 403(2):347–372, 2008.

[51] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.

[52] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: semantics and cut-elimination. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 248–262. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[53] Alain Frisch and Keisuke Nakano. Streaming XML transformation using term rewriting. In *PLAN-X*, pages 2–13, 2007.

[54] Dan R. Ghica and Guy McCusker. Reasoning about idealized algol using regular languages. In *ICALP*, pages 103–115, 2000.

[55] Eduardo Giménez and Pierre Castéran. A tutorial on [co-] inductive types in coq.

[56] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[57] Jean-Yves Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

[58] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[59] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

[60] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

[61] T. Griffin. A Formulae-as-Types Notion of Control. In *POPL'90*, pages 47–58. ACM Press, 1990.

[62] Hugo Herbelin. C'est maintenant qu'on calcule. In *Habilitation à diriger les reserches*, 2005.

[63] Martin Hofmann and Wei Chen. Büchi types for infinite traces and liveness. *CoRR*, abs/1401.5107, 2014.

[64] Martin Hofmann and Harald Ruess. Certification for mu-calculus with winning strategies. *CoRR*, abs/1401.1693, 2014.

[65] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell Univ., December 1971.

[66] David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. A fragment of ML decidable by visibly pushdown automata. In *ICALP (2)*, pages 149–161, 2011.

[67] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 193–206. ACM, 2013.

[68] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[69] B. Jacobs and J.J.M.M. Rutten. An introduction to (co)algebras and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, pages 38–99. Cambridge University Press, 2011.

[70] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Koen Claessen and Nikhil Swamy, editors, *PLPV*, pages 49–60. ACM, 2012.

[71] Alan Jeffrey. Functional reactive programming with liveness guarantees. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 233–244. ACM, 2013.

[72] Wolfgang Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard, editors, *PLPV*, pages 69–78. ACM, 2013.

[73] Wolfgang Jeltsch. An abstract categorical semantics for functional reactive programming with processes. In Nils Anders Danielsson and Bart Jacobs, editors, *PLPV*, pages 47–58. ACM, 2014.

[74] Roope Kaivola. Axiomatising linear time mu-calculus. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, volume 962 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 1995.

[75] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Transfinite reductions in orthogonal term rewriting systems. *Inf. Comput.*, 119(1):18–38, 1995.

[76] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theor. Comput. Sci.*, 175(1):93–125, 1997.

[77] Jeroen Ketema, Stefan Blom, Takahito Aoto, and Jakob Grue Simonsen. Rewriting transfinite terms. In *Liber Amicorum for Roel de Vrijer*, pages 129–144. Lulu, Raleigh, NC, September 2009.

[78] Naoki Kobayashi and C.-H. Luke Ong. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *LICS*, pages 179–188. IEEE Computer Society, 2009.

[79] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI*, pages 222–233, 2011.

[80] U. Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*. Springer Monographs in Mathematics. Springer, 2008.

[81] Kensuke Kojima and Atsushi Igarashi. Constructive linear-time temporal logic: Proof systems and Kripke semantics. *Inf. Comput.*, 209(12):1491–1503, 2011.

[82] Dexter Kozen. On induction vs. *-continuity. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *LNCS*, pages 167–176. Springer, 1981.

[83] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[84] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs. In *LICS*, pages 257–266. IEEE Computer Society, 2011.

[85] J.-L. Krivine. *Théorie des ensembles*. Cassini, 1998.

[86] J.-L. Krivine. Realizability in Classical Logic. *Panoramas et synthèses*, pages 197–229, 2009.

[87] J.-L. Krivine. Realizability algebras : a program to well order R. *LMCS*, 7(3), 2011.

[88] Pavel Labath and Joachim Niehren. A Functional Language for Hyperstreaming XSLT. Rapport de recherche, Commenius University, Bratislava , Laboratoire d'Informatique Fondamentale de Lille - LIFL , LINKS - INRIA Lille - Nord Europe, March 2013.

[89] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[90] Conor McBride. Let's see how things unfold: Reconciling the infinite with the intensional (extended abstract). In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *CALCO*, volume 5728 of *LNCS*, pages 113–126. Springer, 2009.

[91] Paul-André Melliès. Asynchronous games 4: A fully complete model of propositional linear logic. In *LICS*, pages 386–395. IEEE Computer Society, 2005.

[92] D. E. Muller and P. E Schupp. Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of the Theorems of Rabin, McNaughton and Safra. *Theor. Comput. Sci.*, 141(1&2):69–107, 1995.

[93] Andrzej S. Murawski, C.-H. Luke Ong, and Igor Walukiewicz. Idealized algol with ground recursion, and dpda equivalence. In *ICALP*, pages 917–929, 2005.

[94] Koji Nakazawa and Tomoharu Nagai. Reduction system for extensional lambda-mu calculus. In Gilles Dowek, editor, *RTA-TLCA*, LNCS, 2014. to appear.

[95] Koji Nakazawa and Shin ya Katsumata. Extensional models of untyped lambda-mu calculus. In Herman Geuvers and Ugo de'Liguoro, editors, *CL&C*, volume 97 of *EPTCS*, pages 35–47, 2012.

[96] Paulo Oliva. Functional interpretations of linear and intuitionistic logic. *Inf. Comput.*, 208(5):565–577, 2010.

[97] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90, 2006.

[98] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR 92*, pages 190–201. Springer-Verlag, 1992.

[99] Michel Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, pages 190–201, 1992.

[100] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.

[101] Dominique Perrin and Jean-Éric Pin. *Infinite Words*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004. ISBN 0-12-532111-2.

[102] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[103] François Pottier and Yann Regis-Gianas. Stratified type inference for generalized algebraic data types. In *POPL'06 - Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, pages 232–244, Charleston, South Carolina, United States, 2006.

[104] Damien Pous. Up-to Techniques for Weak Bisimulation. In *ICALP*, volume 3580 of *LNCS*, pages 730–741. Springer, 2005.

[105] Damien Pous. Untyping typed algebras and colouring cyclic Linear Logic. *LMCS*, 8(2), 2012.

[106] Damien Pous. Kleene Algebra with Tests and Coq tools for while programs. In *Proc. ITP*, volume 7998 of *LNCS*, pages 180–196. Springer, 2013.

[107] Damien Pous and Davide Sangiorgi. *Advanced Topics in Bisimulation and Coinduction*, chapter about "Enhancements of the coinductive proof method". CUP, 2011.

[108] C. Riba. A Model Theoretic Proof of Completeness of an Axiomatization of Monadic Second-Order Logic on Infinite Words. In *Proceedings of IFIP-TCS'12*, volume 7604 of *LNCS*, pages 310–324. Springer, 2012.

[109] C. Riba. Forcing MSO on Infinite Words in Weak MSO. In *LICS*, pages 448–457. IEEE Computer Society, 2013.

[110] Jan J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theor. Comput. Sci.*, 308(1-3):1–53, 2003.

[111] Sylvain Salvati and Igor Walukiewicz. Evaluation is MSOL-compatible. In *FSTTCS*, volume 24 of *LIPIcs*, pages 103–114. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[112] Sylvain Salvati and Igor Walukiewicz. Using Models to Model-Check Recursive Schemes. In *TLCA*, volume 7941 of *LNCS*, pages 189–204. Springer, 2013.

[113] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2002.

[114] Luigi Santocanale. From parity games to circular proofs. *Electr. Notes Theor. Comput. Sci.*, 65(1):305–316, 2002.

[115] Alexis Saurin. Separation with streams in the $\Lambda\mu$-calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 356–365. IEEE Computer Society, 2005.

[116] Alexis Saurin. A hierarchy for delimited continuations in call-by-name. In *FOSSACS 2010*, LNCS, March 2010.

[117] Alexis Saurin. Böhm theorem and Böhm trees for the $\Lambda\mu$-calculus. *Theor. Comput. Sci.*, 435:106–138, 2012.

[118] D. Siefkes. *Decidable Theories I : Büchi's Monadic Second Order Successor Arithmetic*, volume 120 of *LNM*. Springer, 1970.

[119] Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, 1989.

[120] S.G. Simpson. *Subsystems of Second Order Arithmetic*. Perspectives in Logic. Cambridge University Press, 2nd edition, 2010.

[121] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In *ITP*, volume 6172 of *LNCS*, pages 419–434. Springer, 2010.

[122] Christine Tasson and Lionel Vaux. Transport of finiteness structures and applications. *To be published in MSCS*, 2010.

[123] Kazushige Terui. Computational ludics. *Theor. Comput. Sci.*, 412(20):2048–2071, 2011.

[124] The Coq Development Team. *The Coq Proof Assistant: Reference Manual*. INRIA, 2012.

[125] T. Tsukada and C.-H.L. Ong. Compositional Higher-Order Model Checking via ω-Regular Games over Böhm Trees. In *Proc. CSL-LICS*. ACM, 2014. to appear.

[126] M. Y. Vardi and T. Wilke. Automata: from logics to algorithms. In *Logic and Automata*, volume 2 of *Texts in Logic and Games*, pages 629–736. Amsterdam University Press, 2008.

[127] Moshe Y. Vardi. Automata-theoretic model checking revisited. In *VMCAI*, volume 4349 of *LNCS*, pages 137–150. Springer, 2007.

[128] I. Walukiewicz. Completeness of Kozen's Axiomatisation of the Propositional μ-Calculus. *Information and Computation*, 157(1-2):142–182, 2000.