Data types, infinity and equality in system AF_2

Christophe Raffalli
L.F.C.S.
Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

June 9, 1995

Abstract

This work presents an extension of system AF_2 to allow the use of infinite data types. We extend the logic with inductive and coinductive types, and show that the "programming method" is still correct. Unlike previous work in other type-systems, we only use the pure λ-calculus. Propositions about normalization and unicity of the representation of data have no equivalent in other systems. Moreover, the class of data types we consider is very large with some unusual ones.

1 Introduction

Since the work of Curry, a lot of type-systems have been created (e.g., De Bruijn's Automath [4]; Girard's system F [5]; Martin-Löf's type theory [10]; Coquand-Huet's Calculus of construction [3]; etc). One of their purposes is program extraction via the Curry-Howard isomorphism [6], which establishes a correspondence between programs and proofs of specifications.

One of these systems is AF_2 (second order functional arithmetic) due to Leivant and Krivine [9, 7, 8]. It uses equations as algorithmic specifications, and its "programming method" ensures the correctness of programs extracted from proofs of function totality.

However, the efficiency of these extracted programs is not as expected. To get the expected efficiency, Parigot introduces "inductive types" [12] , which are definable by adding a least fixed point connective to the logic.

Our work shows how this system may be extended with "coinductive types" using a greatest fixed point connective. This allows the use of infinite data types like streams. Indeed, we get a very large variety of infinite data types (some of which are unexpected and unusual), and we will see that our programming method is still correct.

The particulars of this work come from those of AF_2, including data types encoded in pure λ-calculus, equational axioms, and second order logic. Moreover, in most of the previous work [11, 13, 15], infinite data types are added through a categorical duality,

defined by their destructors (except for [15]), and use the coinduction principle. In our work, finite and infinite data types are added in a similar way, and both are defined using constructors. This similarity allows us to define a more general class of data types.

In the next section, we give a summary of Krivine and Parigot's previous work and show how it leads us to introduce the notion of "coinductive type". Next we present formally the syntax and the semantics of our system and see how the general result about AF_2 may be extended to our system. Then, we study its normalization properties. Next, we show how to define infinite data types and how extract correct programs involving them. Finally, we consider the problem of equality on infinite data types and its relation with the coinduction principle.

2 From previous work to coinductive types.

AF_2 uses a first order language $\mathcal{L}$ given by a signature. Specifications of programs are given by a set $\mathcal{E}$ of equational axioms. For instance, if we want to use natural numbers, we choose a signature $\Sigma = \{0, s(\_), add(\_,\_), \ldots\}$ and some equations: $add(0, y) = y, add(s(x), y) = s(add(x, y)) \ldots$ The logical part of the system is second order predicate logic using only first and second order universal quantification and implication. This syntax is very expressive and close to common mathematical usage. Here, for instance, is the definition of natural numbers: $N[x] = \forall X(X0 \to \forall y(Xy \to Xsy) \to Xx)$.

Now, if we use natural deduction to prove this formula, we can associate a λ-term with each proof. Moreover, there is a unique λ-term (up to β-equivalence) associated with any proof of $N(t)$, and this λ-term is Church's numeral $\overline{n}$ if $\mathcal{E} \vdash t = s^n 0$. This result about data types ensures the correctness of the program extracted from a proof of function totality. For instance, a term $\overline{add}$, extracted from a proof of $\vdash \forall x \forall y(Nx \to Ny \to Nadd(x, y))$, is a correct program that computes sums of Church numerals.

Krivine and Parigot prove that we can define most of the usual data types in computer science, and extract correct programs for any function whose termination is provable, in Peano arithmetic. This is powerful enough! But the behaviour of these programs is not the expected one. For instance, predecessor on Church numerals is in linear time! It would be nice to use only pure the λ-calculus for theoretical study. But we want our theory to be realistic. Thus, it is necessary to get the expected efficiency in our system.

To approach this goal, Michel Parigot introduces "inductive types". For instance, we could write $N[x] = \forall X(X0 \to \forall y(N[y] \to Xsy) \to Xx)$ as an inductive definition of the natural numbers. And this does give the expected behaviour for this unary representation. Allowing inductive definitions is difficult to manage. One of the solutions is to add a new connective to the logic:

$$\mu X x_1 \ldots x_n \; F \, \langle t_1, \ldots, t_n \rangle$$

where X is a predicate variable of arity n that occurs positively in the formula F (note that variables $X, x_1, \ldots, x_n$ are bound in F by μ). This connective defines a general least fixed point operator. Semantically, it is interpreted, in a model $\mathcal{M}$, as $\mathcal{K} t_1 \ldots t_n$ where $\mathcal{K}$ is the least predicate of arity n verifying $\mathcal{M}[\mathcal{K}/X] \models \forall x_1 \ldots x_n (X x_1 \ldots x_n \leftrightarrow F)$.

Parigot shows that program correctness is preserved with this extension. Moreover, the behaviour of programs on inductive data types is similar to the behaviour of everyday

programs.

Some problems still remain. For instance, some data types are still not definable, such as input and output streams. The notion of stream is of general interest in computer science. It involves the possibility of infinite objects.

An elegant way to provide infinite data types is simply to add a greatest fixed point operator symmetrical to the least one. To do this we add the following connective:

$$\nu X x_1 \dots x_n \; F \langle t_1, \dots, t_n \rangle$$

where X is a predicate variable of arity n which occurs positively in the formula F. Semantically, it is interpreted, in a model $\mathcal{M}$, as $\mathcal{K} t_1 \dots t_n$ where $\mathcal{K}$ is the greatest predicate of arity n verifying $\mathcal{M}[\mathcal{K}/X] \models \forall x_1 \dots x_n (X x_1 \dots x_n \leftrightarrow F)$.

In both definition of μ and ν operator, the first order variables $x_1 \dots x_n$ allow us to define a set of n-tuples by a least or a greatest fixed point. So, the formula $\nu X \overline{x} \; F \langle \overline{t} \rangle$ expresses that the n-tuple of terms $\overline{t}$ belongs to the greatest fixed point defined from F.

Remark: It's possible to define in AF_2 some formulas equivalent to our least and greatest fixed points. But, we don't get the expected algorithmic contents.

3 Formal definition of the system.

Formulas and logical terms are constructed using individual variables: $x, y, z, \dots$, predicate variables of arbitrary arity: $X, Y, Z, \dots$, a set of function constants given by a signature Σ, and the following symbols: $\rightarrow, \forall, (\,), \mu, \nu, \langle, \rangle$.

Logical terms (or *individual terms*) are constructed inductively as follows:

- individual variables are logical terms.
- if f is a n-ary function constant and $t_1, \dots, t_n$ are logical terms, then $f(t_1, \dots, t_n)$ is a logical term.

The set of formulas is defined in two steps. First, we construct inductively the set of *pseudo-formulas* as follows:

- if X is a n-ary predicate variable and $t_1, \dots, t_n$ are logical terms, then $X(t_1, \dots, t_n)$ is a pseudo-formula.
- if F and G are pseudo-formulas, then $F \rightarrow G$ is a pseudo-formula.
- if F is a pseudo-formula and χ is an individual or a predicate variable, then $\forall \chi \, F$ is a pseudo-formula.
- if F is a pseudo-formulas, X is a n-ary predicate variable, $x_1, \dots, x_n$ are individual variables and $t_1, \dots, t_n$ are logical terms, then $\mu X x_1 \dots x_n \; F \langle t_1 \dots t_n \rangle$ and $\nu X x_1 \dots x_n \; F \langle t_1 \dots t_n \rangle$ are pseudo-formulas.

Next we define the negative and positive occurrences of variables in a pseudo-formula. Then, we define *formulas* as follow: a pseudo-formula F is a formula iff F has no subformula of the form $\mu X x_1 \dots x_n \; G \langle t_1 \dots t_n \rangle$ or $\nu X x_1 \dots x_n \; G \langle t_1 \dots t_n \rangle$ when X has a negative occurrence in G.

For convenience, we will use the following notation:

- we denote by $F_1 \to F_2 \ldots \to F_n$ the formula $F_1 \to (F_2 \to \ldots (F_{n-1} \to F_n)\ldots)$.
- we often omit parenthesis for unary symbols.
- $\overline{t}$ (resp. $\overline{x}$) associated to any n-ary predicate variable X, denotes a vector of terms $(t_1, \ldots, t_n)$ (resp. a vector of variables $(x_1, \ldots, x_n)$).
- $F[x \Leftarrow u]$ denotes the substitution of the first order term u to the variable x.
- $F[X\overline{x} \Leftarrow G]$ denotes the substitution of each occurrence in F of an atomic formula $X\overline{t}$ by $G[\overline{x} \Leftarrow \overline{t}]$
- $F\langle X \Leftarrow G\rangle = F'$ denotes the physical substitution in F of each occurrence of the null-ary predicate variable X by the formula G. This is used only to design a sub-formula G of F' (we need physical substitution because G may have some free variables bounded in F').

To define the type-system itself, we use pure λ-calculus [1] as an algorithmic counterpart. It's very important to note that the world of programs, represented by λ-terms, is kept separated from the logical world. Thus, Logical terms should not be confused with λ-terms (though we use the same notation). We also use a set of equations on logical terms $\mathcal{E}$, which are the specifications of our programs.

A typing judgement is an expression of the form $x_1 : F_1, \ldots, x_n : F_n \vdash t : F$ where $x_1, \ldots, x_n$ are λ-variables, t is a λ-term and $F_1, \ldots, F_n, F$ are formulas. To prove such a judgement, we use some rules. The first set of rules, are those of AF_2:

$$\frac{}{x_1 : F_1, \ldots, x_n : F_n \vdash x_i : F_i} R0 \qquad \frac{\Gamma \vdash t : F[x \Leftarrow u] \qquad \mathcal{E} \vdash u = v}{\Gamma \vdash t : F[x \Leftarrow v]} R1$$

$$\frac{x : F, \Gamma \vdash t : G}{\Gamma \vdash \lambda x\, t : F \to G} R2 \qquad \frac{\Gamma \vdash t : F \to G \qquad \Gamma \vdash u : F}{\Gamma \vdash (t\ u) : G} R3$$

$$\frac{\Gamma \vdash t : F \quad x \text{ is not free in } \Gamma}{\Gamma \vdash t : \forall x F} R4 \qquad \frac{\Gamma \vdash t : \forall x F}{\Gamma \vdash t : F[x \Leftarrow u]} R5$$

$$\frac{\Gamma \vdash t : F \quad X \text{ is not free in } \Gamma}{\Gamma \vdash t : \forall X F} R6 \qquad \frac{\Gamma \vdash t : \forall X F}{\Gamma \vdash t : F[X\overline{x} \Leftarrow G]} R7$$

The next rules are for the equivalence expressing that μ and ν represent some fixed points. These rules say that we are allowed to exchange everywhere the formula $\mu X\overline{x}\ F\,\langle\overline{t}\rangle$ and $F[\overline{x} \Leftarrow \overline{t}][X\overline{y} \Leftarrow \mu X\overline{x}\ F\,\langle\overline{y}\rangle]$ because they are equivalent (the same rules are used with ν in place of μ):

- development of a fixed point (or unfolding, if $H = Y$):

$$\frac{\Gamma \vdash t : H\langle Y \Leftarrow \mu X\overline{x}\ F\,\langle\overline{t}\rangle\rangle}{\Gamma \vdash t : H\langle Y \Leftarrow F[\overline{x} \Leftarrow \overline{t}][X\overline{y} \Leftarrow \mu X\overline{x}\ F\,\langle\overline{y}\rangle]\rangle} R8 \quad \text{(resp. with } \nu\text{, R9)}$$

- factorization of a fixed point (or folding, if $H = Y$):

$$\frac{\Gamma \vdash t : H\langle Y \Leftarrow F[\overline{x} \Leftarrow \overline{t}][X\overline{y} \Leftarrow \mu X\overline{x}\ F\,\langle\overline{y}\rangle]\rangle}{\Gamma \vdash t : H\langle Y \Leftarrow \mu X\overline{x}\ F\,\langle\overline{t}\rangle\rangle} R10 \quad \text{(resp. with } \nu\text{, R11)}$$

The two last rules express that μ and ν are respectively a least and a greatest fixed point. They use a fixed point combinator as algorithmic counterpart. It is denoted by ! and must satisfy the condition that $!t$ can reduce to $(t\ !t)$ for any λ-term t. In these rules we use the notation $\forall\overline{\chi}$ to denote a sequence (eventually empty) of quantification over any kind of variables. The justification of these rules will be found in the next section on semantics.

– least fixed point rule:

$$\frac{\Gamma \vdash t : \forall \overline{\chi}(X\overline{t} \to G) \to \forall \overline{\chi}(F[\overline{x} \Leftarrow \overline{t}] \to G) \quad X \text{ is not free in } \Gamma \text{ and } G}{\Gamma \vdash\ !t : \forall \overline{\chi}(\mu X \overline{x}\ F\ \langle \overline{t} \rangle \to G)} R12$$

– greatest fixed point rule:

$$\frac{\Gamma \vdash t : \forall \overline{\chi}(G \to X\overline{t}) \to \forall \overline{\chi}(G \to F[\overline{x} \Leftarrow \overline{t}]) \quad X \text{ is not free in } \Gamma \text{ and } G}{\Gamma \vdash\ !t : \forall \overline{\chi}(G \to \nu X \overline{x}\ F\ \langle \overline{t} \rangle)} R13$$

Let us call AF_2^I (for inductive second order functional arithmetic) this new system. To be a real type system, It has to verify the property:

Proposition 1 (Subject reduction) *For any formula F and any term t we have:*

$$\vdash t : F \quad \textit{and } t \textit{ reduces to } t' \textit{ implies } \quad \vdash t' : F$$

The proof is similar to that of AF_2: we first show that we can restrict the derivation to respect some constraints about rule ordering. Then we get the result by structural induction on the derivation of $\Gamma \vdash t : F$.

4 Semantics.

Next, we define a semantic notion of type. To do this, we introduce a notion of "interpretation" which associates to each formula F a set of λ-terms. This notion doesn't match exactly the syntactic one, but we get that the set of terms of type F is included in the interpretation of F. This is sufficient to prove some properties of typed terms. For this semantic notion of type, we work up to β-equivalence, and Λ denotes the set of all λ-terms quotiented by this equivalence.

To get all the power of this notion, it's must be able to change the domain for the interpretation of formulas. Therefore, we choose a set $\mathcal{C} \subset \mathcal{P}(\Lambda)$, and we define our semantics by these definitions:

Definition 2 *A set $\mathcal{C} \subset \mathcal{P}(\Lambda)$ is* coherent *if it verifies the following conditions:*
- *If Φ and Φ' belong to $\mathcal{C}$ then $\Phi \to \Phi' = \{t \in \Lambda \mid \forall u \in \Phi, (tu) \in \Phi'\}$ belongs to $\mathcal{C}$*
- *$\mathcal{C}$ is closed under union and intersection*
- *$\emptyset$ and Λ belong to $\mathcal{C}$.*

Notation: Given a set $\mathcal{C} \subset \mathcal{P}(\Lambda)$, $\mathcal{C}_n$ denotes the set of all functions from Λ^n to $\mathcal{C}$ (So, $\mathcal{C}_0 = \mathcal{C}$).

Definition 3 *A $\mathcal{C}$-interpretation σ is defined as follows:*

- *For each individual variable x, we choose $|x|^\sigma \in \Lambda$.*
- *For each n-ary function constant f, we choose $|f|^\sigma$, a function from Λ^n to Λ.*
- *For each n-ary variable of predicate X, we choose $|X|^\sigma \in \mathcal{C}_n$.*

Notation: Given a $\mathcal{C}$-interpretation σ, $\sigma[\chi \Leftarrow \varphi]$ represent the same interpretation as σ except for the variable χ which is interpreted by φ.

Definition 4 *Given a $\mathcal{C}$-interpretation σ we define the interpretation of logical terms and formulas by induction as follows:*

- $|f(t_1,\ldots,t_n)|^\sigma = |f|^\sigma\,(|t_1|^\sigma,\ldots,|t_n|^\sigma)$
- $|X(t_1,\ldots,t_n)|^\sigma = |X|^\sigma\,(|t_1|^\sigma,\ldots,|t_n|^\sigma)$
- $|F \to G|^\sigma = |F|^\sigma \to |G|^\sigma$
- $|\forall x\, F|^\sigma = \bigcap\{|F|^{\sigma[x\Leftarrow v]} \mid v \in \Lambda\}$
- $|\forall X\, F|^\sigma = \bigcap\{|F|^{\sigma[X\Leftarrow \Phi]} \mid \Phi \in \mathcal{C}_n\}$
- $|\mu X\overline{x}\ F\,\langle\overline{t}\rangle|^\sigma = \bigcap\{\Phi(|\overline{t}|^\sigma) \mid \Phi \in \mathcal{C}_n, \forall \overline{v} \in \Lambda^n\ \ |F|^{\sigma[X\Leftarrow\Phi][\overline{x}\Leftarrow\overline{v}]} \subseteq \Phi(\overline{v})\}$
- $|\nu X\overline{x}\ F\,\langle\overline{t}\rangle|^\sigma = \bigcup\{\Phi(|\overline{t}|^\sigma) \mid \Phi \in \mathcal{C}_n, \forall \overline{v} \in \Lambda^n\ \Phi(\overline{v}) \subseteq |F|^{\sigma[X\Leftarrow\Phi][\overline{x}\Leftarrow\overline{v}]}\}$

Fact 5 *From this definition and the coherency of $\mathcal{C}$, we get that for all formulas F and all $\mathcal{C}$-interpretations σ, we have $|F|^\sigma \in \mathcal{C}$.*

We have to comment the interpretation of the fixed points. To understand the definition of $|\mu X\overline{x}\ F\,\langle\overline{t}\rangle|^\sigma$ (resp. ν), we should consider the mapping Ψ_F from $\mathcal{C}_n$ to $\mathcal{C}_n$, which associates to Φ the mapping $\overline{v} \mapsto |F|^{\sigma[X\Leftarrow\Phi][\overline{x}\Leftarrow\overline{v}]}$ (which is in $\mathcal{C}_n$ by the previous fact). Because X has no negative occurrence in F, we deduce that Ψ_F is an increasing mapping. By definition, $|\mu X\overline{x}\ F\,\langle\overline{t}\rangle|^\sigma$ (resp. ν) is the intersection (resp. union) of all $\Phi(|\overline{t}|^\sigma)$ for Φ verifying $\Psi_F(\Phi) \le \Phi$ (resp. $\ge$) [where $\Phi \le \Phi'$ (resp. $\ge$) iff for all $\overline{v} \in \Lambda^n$, $\Phi(\overline{v}) \subset \Phi'(\overline{v})$ (resp. $\supset$)]. But, this corresponds to a definition of the least (resp. greatest) fixed point of Ψ_F (in the Knaster-Tarski style). Hence, we get:

Fact 6 *If Π_F denotes the least (resp. greatest) fixed point of Ψ_F (defined above), we have $|\mu X\overline{x}\ F\,\langle\overline{t}\rangle|^\sigma = \Pi_F(|\overline{t}|^\sigma)$ (resp. ν).*

Examples: The principal notions of interpretation used here are the *full interpretation*, obtained with $\mathcal{C} = \mathcal{P}(\Lambda)$ and the *classical full model*, obtained with $\mathcal{C} = \{\emptyset, \Lambda\}$. Both of them are used in the definition of the notion of data types. If $\mathcal{M}$ is a classical full model, we will denote $\mathcal{M} \models F$ for $|F|^{\mathcal{M}} = \Lambda$.

The next proposition relates the syntactic type-system to our semantics:

Proposition 7 (Conservation or soundness) *If we prove $\vdash t : F$ then $t \in |F|^\sigma$ for any $\mathcal{C}$-interpretation σ compatible with our set of equations $\mathcal{E}$ (this means $\mathcal{E} \vdash t = u$ implies $|t|^\sigma = |u|^\sigma$).*

Proof: We prove, by induction on the derivation of $x_1 : F_1, \ldots, x_n : F_n \vdash t : F$, that for any $\mathcal{C}$-interpretation σ, for all $u_1 \in |F_1|^\sigma, \ldots, u_n \in |F_n|^\sigma$, we have $t[\overline{x} \Leftarrow \overline{u}] \in |F|^\sigma$ (the proposition is the case of an empty context):

- When the proof ends on one of the rules of AF_2, we use the same arguments.
- If we use a rule of factorization or development of a fixed point, we just use the previous fact 6. Using the same notation, we see that $|\mu X\overline{x}\ F\,\langle\overline{t}\rangle|^\sigma = \Pi_F(|\overline{t}|^\sigma)$ and $|F[\overline{x} \Leftarrow \overline{t}][X\overline{y} \Leftarrow \mu X\overline{x}\ F\,\langle\overline{y}\rangle]|^\sigma = \Psi_F(\Pi_F)(|\overline{t}|^\sigma)$ (resp. ν). Now, because Π_F is

a fixed point of Ψ_F, both interpretations are equal, and the two formulas may be exchanged everywhere without changing the interpretation.

- If the last rule is R12, we use the inductive construction of the fixed point. We define, by ordinal induction, a family of elements of $\mathcal{C}_n$ by:
  - $\Phi_0(\overline{v}) = \emptyset$ for all $\overline{v} \in \Lambda^n$.
  - $\Phi_\alpha(\overline{v}) = \bigcup\{\Psi_F(\Phi_\beta)(\overline{v}) \mid \beta < \alpha\}$ for all $\overline{v} \in \Lambda^n$.

  By induction hypothesis, we have $t \in |\forall\overline{\chi}(X\overline{t} \to G) \to \forall\overline{\chi}(F[\overline{x} \Leftarrow \overline{t}] \to G)|^\sigma$. If $\Theta(\Phi)$ denotes $|\forall\overline{\chi}(X\overline{t} \to G)|^{\sigma[X \Leftarrow \Phi]}$, from this we deduce $u \in \Theta(\Phi)$ implies $(t\,u) \in \Theta(\Psi_F(\Phi))$. Moreover, it's easy to see that for any family of mappings $\{\Phi_i\}_{i\in I}$, we have $\Theta(\cup\{\Phi_i \mid i \in I\}) = \cap\{\Theta(\Phi_i) \mid i \in I\}$. Thus, we get $!t \in \Theta(\Phi_\beta)$ for all $\beta < \alpha$ implies $(t\,!t) = !t \in \Theta(\Phi_\alpha)$. Now, from $\Phi_0 = \emptyset$, we deduce $!t \in \Theta(\Phi_0) = \Lambda$. So we get $!t \in \Theta(\Phi_\alpha)$ for all ordinals α. Hence, we have $!t \in \Theta(\Pi_F) = |\forall\overline{\chi}(\mu X\overline{x}\ F\,\langle\overline{t}\rangle \to G)|^\sigma$, which is the wanted result.
- The case of Rule R13 is similar.

♠

The cases of the rules R12 and R13, in the previous proof, give the best justification of these rules.

5 Normalization properties.

Because we use a fixed point combinator in pure λ-calculus, the terms we can type are usually not normalizable. Thus, we will only prove result about solvability and *hereditary* solvability (i.e., there is no $\perp$ in the Böhm tree of the term) of typed terms.

First, we prove this proposition, which is true in all type-systems satisfying the subject reduction and such that all sub-terms of a typed terms are typable:

Proposition 8 *If all typed terms are solvable then all typed terms are hereditary solvable.*

Proof: We define the set Ω of all hereditary solvable terms, by induction, as follow:

- $\Omega_0 = \Lambda$
- $t \in \Omega_{n+1}$ iff there exists $t_1, \ldots, t_p \in \Omega_n$ and some λ-variables $y, x_1, \ldots, x_n$ such that $t \sim_\beta \lambda x_1 \ldots \lambda x_n (y\,t_1 \ldots t_p)$
- $\Omega = \cap\{\Omega_n \mid n \in \mathbb{N}\}$

(It's easy to see that $t \in \Omega$ iff there is no $\perp$ in the Böhm tree of t).

We assume that all typed terms are solvable, and we prove by induction that all typed terms are in Ω_n. By definition the Ω_0 case is obvious. Now, we assume that all typed terms are in Ω_n.

Let us choose t such that $\Gamma \vdash t : F$. By hypothesis, t is solvable. Hence, we have t reduce to $\lambda x_1 \ldots \lambda x_n (y\ t_1 \ldots t_p)$. By subject reduction, we get $\Gamma \vdash \lambda x_1 \ldots \lambda x_n (y\ t_1 \ldots t_p) : F$. But, somewhere in the proof we type t_1, ..., t_n. Hence, by the induction hypothesis, these terms belong to Ω_n. Thus, by definition we get $t \in \Omega_{n+1}$. ♠

Now, we will describe a condition which ensures that all typed terms are hereditary solvable. First, we give this definition:

Definition 9 *We define a three valued semantics as follows: we choose a mapping* Δ *which associate to each predicate variable a value in* $\{-1,0,1\}$ *and we define the interpretation of a formula* F *by induction:*

- $|X(\overline{t})|^{\Delta} = \Delta(X)$.
- $|G \to H|^{\Delta}$ *is given by the following table:*

		$\lvert H\rvert^{\Delta}$		
		-1	0	1
$\lvert G\rvert^{\Delta}$	-1	1	1	1
	0	-1	0	1
	1	-1	0	1

- $|\forall x\, F|^{\Delta} = |F|^{\Delta}$
- $|\forall X\, F|^{\Delta} = \inf\{|F|^{\Delta[X\Leftarrow -1]}, |F|^{\Delta[X\Leftarrow 0]}, |F|^{\Delta[X\Leftarrow 1]}\}$.
- $|\mu X\overline{x}\ F\ \langle\overline{t}\rangle|^{\Delta} = \inf\{\Phi \in \{-1,0,1\} \mid |F|^{\Delta[X\Leftarrow\Phi]} \leq \Phi\}$
- $|\nu X\overline{x}\ F\ \langle\overline{t}\rangle|^{\Delta} = \sup\{\Phi \in \{-1,0,1\} \mid \Phi \leq |F|^{\Delta[X\Leftarrow\Phi]}\}$

We will say that a formula F *is* strict *if we have* $|F|^{\Delta} = 0$, *for* Δ *defined by* $\Delta(X) = 0$ *for all predicate variables. We will say that a proof is* strict *if all formula which appear in the proof (even as sub-formula) are strict.*

Remark: In this three valued semantics, we can see that -1 corresponds to the classical false and 0 and 1 are two kinds of truth. We could say that 1 corresponds to the true formulas like "false imply something" and 0 to the other true formulas. Therefore, a strict proof is a proof which doesn't use any second order definition of false. Thus, we could say that the notion of strict proof corresponds to a kind of second order minimal logic.

Proposition 10 *If we have* $\Gamma \vdash t : F$ *and if all formulas which appears in the proof (even as sub-formula) are strict, then* t *is hereditary solvable.*

Proof: To prove this, we introduce a new notion of $\mathcal{C}$-interpretation with

- $\mathcal{R} = \{t \in \Lambda \mid \ t \text{ solvable } \}$
- $\mathcal{R}_0 = \{t \mid t \sim_{\beta} (x\ t_1 \ldots t_n), \text{ with for all } i \ \ t_i \in \Lambda\}$
- $\mathcal{C}' = \{\Phi \in \mathcal{P}(\Lambda) \mid \mathcal{R}_0 \subseteq \Phi \subseteq \mathcal{R}\}$
- $\mathcal{C} = \mathcal{C}' \cup \{\Lambda, \emptyset\}$

So we can prove the following inclusions:

$$\mathcal{R}_0 \subset (\Lambda \to \mathcal{R}_0) \subset (\mathcal{R} \to \mathcal{R}_0) \subset (\mathcal{R}_0 \to \mathcal{R}) \subset \mathcal{R}$$

And from this, we deduce that $\mathcal{C}$ is coherent. Moreover, we prove, by induction on the formula F, that for any three valued interpretation Δ, then we have $|F|^{\Delta} = 0$ iff $|F|^{\sigma} \in \mathcal{C}'$ for any $\mathcal{C}$-interpretation σ verifying $|X\overline{t}|^{\sigma} \in \mathcal{C}'$ iff $\Delta(X) = 0$, $|X\overline{t}|^{\sigma} = \emptyset$ iff $\Delta(X) = -1$ and $|X\overline{t}|^{\sigma} = \Lambda$ iff $\Delta(X) = 1$,.

So, if we have $x_1 : F_1 \ldots x_n : F_n \vdash t : F$ by a strict proof, we may choose an interpretation σ such that $|X\overline{t}|^{\sigma} = \mathcal{R}$ for all predicate variables. Then, all formulas in the proof are strict, so we get $|F_1|^{\sigma} \in \mathcal{C}'$,..., $|F_n|^{\sigma} \in \mathcal{C}'$ and $|F|^{\sigma} \in \mathcal{C}'$. By the conservation lemma, and because we have for all i, $x_i \in \mathcal{R}_0 \subset |F_i|^{\sigma}$, we get $t \in |F|^{\sigma} \subset \mathcal{R}$. Therefore, t is solvable.

Now, we can apply the proposition 8 because all sub-proofs of a strict proof are strict and because strict proofs are stable by the subject reduction (i.e., if we get $\vdash t : A$ by a

strict proof and if t reduce to t', then we can get $\vdash t' : A$ by a strict proof). The first point is obvious by the definition. The second point is prooved by looking at the proof of the subject reduction: all formulas in the new proof come from substitutions of formulas from the old proof. Therefore, it's sufficient to remark that the substitution of a strict formula in a strict formula gives a strict formula. ♠

The fact we can type $\vdash \lambda z!\lambda r\lambda x(r\ z) : (\forall X X) \to (\mu X\ F) \to C$ for any formula F and C, shows that this result is quite optimum. The problem comes from the interaction of the second order quantification and the least fixed point rule. In fact, with the system using only the greatest fixed point, we can prove a simpler result:

Proposition 11 *In the system using only the greatest fixed point connective, and with the restriction that $\nu X\overline{x}\ F\langle\overline{t}\rangle$ is used only if X is not the right-most variable in F (for instance $\nu X\ X$ is not allowed), all typable terms are hereditary solvable.*

Proof: The proof of this result uses a similar notion of $\mathcal{C}$-interpretation with $\mathcal{C} = \mathcal{C}' \cup \{\Lambda\}$ (we do not need $\emptyset$). Then we prove that in such a $\mathcal{C}$ interpretation $|F| = \Lambda$ iff the right most atomic formula is free and interpreted by Λ or if there is, on the right branch of the formula, a sub-formula of the form $\nu X\overline{x}\ G\langle\overline{t}\rangle$ where X is the right most variable in G. From this, the result follows as above. ♠

6 Data types and programming method.

As we have seen in Parigot's work, familiar data types can be defined using inductive types constructed with our μ connective. For instance, we define natural numbers and lists of natural numbers as follows (we assume the constants $0, s(\_), nil$, and $cs(\_,\_)$ are present in our signature Σ):

$$\begin{aligned} N[n] &= \mu K x\ \forall X(X0 \to \forall y(Ky \to Xsy) \to Xx)\langle n\rangle \\ L[l] &= \mu K x\ \forall X(Xnil \to \forall n\forall y(N[n] \to Ky \to Xcs(n,y)) \to Xx)\langle l\rangle \end{aligned}$$

It is not difficult to see that these define the expected set. For instance, for the first one, we see that the sub-formula $F[K,x] = \forall X(X0 \to \forall y(Ky \to Xsy) \to Xx)$ may be seen as a function which associates to the predicate K the least predicate containing 0 and all successors of an element of K. So $N[x]$, being the least fixed point of this function, is the intended set.

In our work, to define streams, we take the type of lists, suppress the condition on nil (we want our stream to always be infinite) and we replace the least fixed point with the greatest one. This gives:

$$S[l] = \nu K x\ \forall X(\forall n\forall y(N[n] \to Ky \to Xcs(n,y)) \to Xx)\langle l\rangle$$

Why do we call these sets data types? In fact, we use Krivine's definition of data types:

Definition 12 *A formula $D[x]$ with one free variable (x is the only free variable in D), is a data type for a full interpretation σ if:*

$$t \in \left|D[x]\right|^{\sigma} \quad \textit{iff} \quad t \sim_{\omega} |x|^{\sigma} \quad \textit{and} \quad \mathcal{M}_{\sigma} \models D[x]$$

where $\sim_{\omega}$ is Böhm tree equivalence[1] *, and $\mathcal{M}_{\sigma}$ is any classical full model coinciding with σ on first order terms.*

The first condition in this definition ($t \sim_{\omega} |x|^{\sigma}$) expresses that for each interpretation $|x|^{\sigma}$ of the variable x, the interpretation of the formula $D[x]$ is empty or equal to the singleton $\{|x|^{\sigma}\}$. The second condition ($\mathcal{M}_{\sigma} \models D[x]$) means that the set of all terms in the interpretation of our data type is isomorphic to the set of all terms satisfying the formula $D[x]$ in an usual model. This implies that our set is the intended set.

Notation: D^{σ} denotes the set $\{u \in \Lambda \mid |D[x]|^{\sigma[x \Leftarrow u]} \neq \emptyset\}$. By definition, if $D[x]$ is a data type for σ, then $u \in D^{\sigma}$ is equivalent to $\mathcal{M}_{\sigma}[x \Leftarrow u] \models D[x]$ for any classical full model coinciding with σ on first order terms.

For instance, if D defines the set of natural numbers, the second condition implies that the set D^{σ} is isomorphic to the set of natural numbers. Whereas, the first condition implies that each natural number has a unique representation in λ-calculus.

We can show the following proposition:

Proposition 13 *The formula $S[x]$, defined above, is a data type for any interpretation such that $N[x]$ is a data type and $|cs|^{\sigma}(a,s) = \lambda f(f\,a\,s)$.*

Proof: To prove this, we choose such an interpretation σ. We define $\Delta \in \Lambda \to \mathcal{P}(\Lambda)$ by induction as follows:

- $t \in \Delta_0(u)$ for all λ-terms t and u.
- $t \in \Delta_{n+1}(u)$ iff there exist $a, b, t', u' \in \Lambda$ such that $t \sim_{\beta} \lambda f(f\,a\,t')$, $u \sim_{\beta} \lambda f(f\,b\,u')$, $a \in |N[x]|^{\sigma[x \Leftarrow b]}$ and $t' \in \Delta_n(u')$.
- $t \in \Delta(u)$ iff $t \in \Delta_n(u)$ for all integers n.

Let us show that $t \in |S[x]|^{\sigma[x \Leftarrow u]}$ iff $t \in \Delta(u)$. We use $t \in \Psi(\Phi, u)$ to denote $t \in |\forall X(\forall n \forall y(N[n] \to Ky \to Xcs(n,y)) \to Xx)|^{\sigma[x \Leftarrow u][K \Leftarrow \Phi]}$. Then, we show:

$$t \in \Psi(\Phi, u) \;\text{ iff }\; t \sim_{\beta} \lambda f(f\,a\,t'), u \sim_{\beta} \lambda f(f\,b\,u'), a \in |N[x]|^{\sigma[x \Leftarrow b]} \;\text{ and }\; t' \in \Phi(u').$$

- For the left-right implication, we assume that $t \in \Psi(\Phi, u)$, choose a λ-variable f which is not free in t, and we define Θ by $t \in \Theta(u)$ iff $t \sim_{\beta} (f\,a\,t')$, $u \sim_{\beta} \lambda f(f\,b\,u')$, $a \in |N[x]|^{\sigma[x \Leftarrow b]}$ and $t' \in \Phi(u')$. σ' denotes $\sigma[x \Leftarrow u][K \Leftarrow \Phi][X \Leftarrow \Theta]$. If we take $a \in |N[x]|^{\sigma[x \Leftarrow b]}$ and $t' \in \Phi(u')$, we get $(f\ a\ t') \in \Theta(|cs|^{\sigma}(b, u'))$. Hence, $f \in |\forall n \forall y(N[n] \to Ky \to Xcs(n,y))|^{\sigma'}$. Therefore, $(t\,f) \in |Xx|^{\sigma'}$. Then, by definition of Θ, we get $(t\,f) \sim_{\beta} (f\,a\,t')$, $u \sim_{\beta} \lambda f(f\,b\,u')$, $a \in |N[x]|^{\sigma[x \Leftarrow b]}$ and $t' \in \Phi(u')$. Because f is not free in t, we get $(t\,f) \sim_{\beta} (f\,a\,t')$, which implies $t \sim_{\beta} \lambda f(f\,a\,t')$. Thus, we get the expected result.

[1] In the original definition, β-equivalence is required. But, for finite data types both equivalences coincide. And, for infinite data types, such an equivalence is needed to express that two terms are equivalent if "their infinite normal forms" are equal.

- For the converse implication, we suppose $t \sim_\beta \lambda f(f\,a\,t')$, $u \sim_\beta \lambda f(f\,b\,u')$, $a \in |N[x]|^{\sigma[x \Leftarrow b]}$ and $t' \in \Phi(u')$. We choose any $\Theta \in \mathcal{C}_1$ and we use σ' to denote $\sigma[x \Leftarrow u][K \Leftarrow \Phi][X \Leftarrow \Theta]$. We choose $v \in |\forall n \forall y(N[n] \to Ky \to Xcs(n,y))|^{\sigma'}$. We have to prove that $(t\,v) \in |Xx|^{\sigma'}$. But, $(t\,v) \sim_\beta (v\,a\,t')$ and we have $a \in |N[x]|^{\sigma[x \Leftarrow b]}$ and $t' \in \Phi(u')$. So we get $(v\,a\,t) \in |Xcs(n,y)|^{\sigma'[n \Leftarrow b][y \Leftarrow u']}$, which implies $(v\,a\,t') \in \Theta(|cs|^\sigma\,(a,u'))$. This gives the expected result because $|cs|^\sigma\,(b,u') \sim_\beta \lambda f(f\,b\,u')$.

From this we get $\Delta_{n+1} = \Psi(\Delta_n)$. Thus the definition of Δ corresponds the definition by induction of the expected greatest fixed point.

In the same way, we define $\Theta \in \Lambda \to \{\emptyset, \Lambda\}$ by induction as follows:
- $\Theta_0(u) = \Lambda$ for all λ-terms u.
- $\Theta_{n+1}(u) = \Lambda$ iff $u \sim_\beta \lambda f(f\,b\,u')$, $\mathcal{M} \models |N[x]|^{\sigma[x \Leftarrow b]}$ and $\Theta_n(u') = \Lambda$.
- $\Theta(u) = \Lambda$ iff $\Theta_n(u) = \Lambda$ for all integers n.

And, by a similar proof, we get $\mathcal{M}[x \Leftarrow u] \models S[x]$ iff $\Theta(u) = \Lambda$.

Now, to prove the proposition, we just have to prove that $t \in \Delta(u)$ iff $t \sim_\omega u$ and $\Theta(u) = \Lambda$. But we can define $\sim_\omega$ by induction as follows:
- $u \sim_{\omega_0} v$ for all λ-terms u and v.
- $u \sim_{\omega_{n+1}} v$ iff, $u \sim_\beta \lambda\overline{x}(x_i\,u_1 \ldots u_p)$ is equivalent to there exists p λ-terms $v_1 \ldots v_p$ such that $v \sim_\beta \lambda\overline{x}(x_i\,v_1 \ldots v_p)$ and for all $i \in \{1, \ldots, p\}$, $u_i \sim_{\omega_n} v_i$
- $t \sim_\omega v$ iff $t \sim_{\omega_n} u$ for all integer n.

Hence, it's sufficient to prove by induction that $t \in \Delta_n(u)$ iff $t \sim_{\omega_n} u$ and $\Theta_n(u) = \Lambda$, Which is immediate by definition of Δ and Θ. ♠

This proof is very general, and may be extended to all the usual data types (finite of infinite).

Some data types we obtain using both least and greatest fixed points are unusual. For instance, we may consider the following formulas:

$$\begin{array}{rcl} B[x] & = & \nu Ky\ F[K,K,y]\,\langle x\rangle \\ B'[x] & = & \nu Ly\ (\mu Kz\ F[K,L,z]\,\langle y\rangle)\,\langle x\rangle \\ B''[x] & = & \nu Ky\ (\mu Lz\ F[K,L,z]\,\langle y\rangle)\,\langle x\rangle \end{array}$$

where

$$F[K,L,y] = \forall X\left(\forall y\left(Ky \to Xs_0\,(y)\right) \to \forall y\left(L \to Xs_1\,(y)\right) \to Xy\right)$$

The first one (B[x]) defines the data types of all infinite boolean sequences (s_0 and s_1 are constructors adding respectively a zero or a one to a sequence). But, using the same technique as in the proof of proposition 13, we can see that $B'[x]$ represents the data type of boolean streams using always an infinite number of ones and that $B''[x]$ represents the data type of boolean streams using a finite number of zeros.

One possible use of these types is the following: using the type $B[x]$ it is not possible to type a program which associates to a stream of booleans, a stream of integers which correspond to the position of the ones. This is because this function is not total (it's undefined if the stream has only a finite number of ones). But, using the type $B'[x]$, there is no problem. This is useful, for instance, to program the Eratosthene sieve.

In the same way, it is also possible to define the data type of infinite binary trees where any path turns a finite (or infinite) number of times to the left (or right).

Finally we prove exactly as in AF_2, using definition 12 and proposition 7, the result which ensures the correctness of the programming method for all these data types:

Proposition 14 *If $D_0, \ldots, D_n$ are data types for an interpretation σ, and if using the set of equations $\mathcal{E}$, satisfied by σ, we prove*

$$\vdash t : D_1[x_1] \to \ldots \to D_n[x_n] \to D_0[f\,(x_1, \ldots, x_n)]$$

then t computes f in the sense that for any λ-terms $u_1 \in D_1^\sigma, \ldots, u_n \in D_n^\sigma$, we have

$$(t\ u_1 \ldots u_n) \sim_\omega (|f|^\sigma\,(u_1, \ldots, u_n)).$$

Proof: Let σ be such an interpretation σ. We choose $u_1 \in D_1^\sigma, \ldots, u_n \in D_n^\sigma$. Because $D_1[x], \ldots, D_n[x]$ are data types, we get $u_1 \in |D_1[x_1]|^{\sigma[x \Leftarrow u_1]}, \ldots, u_n \in |D_n[x_n]|^{\sigma[x \Leftarrow u_n]}$. Then, using the proposition 7, we get $(t\ u_1 \ldots u_n) \in |D_0[f(x_1, \ldots, x_n)]|^{\sigma[x_1 \Leftarrow u_1] \ldots [x_n \Leftarrow u_n]}$. Thus, because D_0 is a data type, we get the expected result. ♠

Remark: As proved by Krivine and Parigot, we can always construct an interpretation σ which satisfies the equations $\mathcal{E}$, if there exists a multisorted model defined on the data types.

7 From the coinductive scheme to a new representation of data types.

A problem comes when we try to use equality on infinite data types. If we use Leibniz's equality ($x = y$ defined by $\forall X(Xx \to Xy)$) we do not get the expected equality. In fact in our interpretation, it gives β-equivalence. But to work with infinite data-types, we need the Böhm tree equivalence.

The standard solution is to use the coinductive scheme to define an equality for each infinite data type. For the type of streams of natural numbers (or any finite type), it looks like this:

$$S_N^c[s,t] = \exists X \left(Xst \wedge \forall n \forall x \forall m \forall y \left(Xcs\,(n,x)\,cs\,(m,y) \to n = m \wedge Xxy \right) \right)$$

This defines the greatest relation "stable for the destruction of streams". So it gives the intended equality. But this relation says that all objects which are not streams are related, so we need to assume that s and t are streams to really use this definition.

Another solution is to give for all data types (finite or infinite), the following kind of definitions for the equality (we give the definition for natural numbers and streams of natural numbers):

$$N'[n,m] = \mu K xy\ \forall X(X00 \to \forall n \forall m(Knm \to Xsn\,sm) \to Xxy)\,\langle n, m\rangle$$
$$S'_N[s,t] = \nu K xy\ \forall X(\forall n \forall x \forall m \forall y(N'[n,m] \to Kxy \to Xcs(n,x)cs(m,y)) \to Xxy)\,\langle s, t\rangle$$

These formulas are very similar to the definitions of the data types. We just replace all unary predicates with a binary one. We remark that $\vdash N'[n,n] \leftrightarrow N[n]$ and $\vdash S'_N[s,s] \leftrightarrow S_N[s]$. We can also prove in our system that $\vdash N'[n,m] \leftrightarrow n = m$ (which

means that our definition coincides with Leibnitz equality on finite data types) and that $\vdash S'_N[s,t] \leftrightarrow (S_N[s] \wedge S[t] \wedge S^c_N[s,t])$ (which means that on infinite data types our definition coincides with the coinduction principle).

Moreover, this kind of inductive definition, using binary relations, may be used to define simultaneously the data type itself and the intended equality on it. Thus, we define the notion of data type with equality as follows:

Definition 15 *A formula $D[x,y]$ with two free variables (x and y are the only free variables in D), is a data type with equality for an interpretation σ if:*

$$t \in \left|D[x,y]\right|^\sigma \quad \textit{iff} \quad t \sim_\omega |x|^\sigma \quad \textit{and} \quad \mathcal{M}_\sigma \models D[x,y]$$

where $\mathcal{M}_\sigma$ is any classical full model coinciding with σ on first order terms.

Using the same kind of proof as for the Krivine data type in the definition 12 and proposition 13, it's easy to prove that the above formula $N'[n,m]$ and $S'_N[s,t]$ define two data types with equality for any interpretation with $|0|^\sigma = \lambda f \lambda g\, f$, $|s|^\sigma\,(t) = \lambda f \lambda g\,(g\,t)$ and $|cs|^\sigma\,(t,u) = \lambda f\,(g\,t\,u)$. Moreover, it's obvious that if $D[x,y]$ is a data type with equality for an interpretation σ, then $D[x,x]$ is a Krivine data type for the same interpretation.

8 Quotient types.

This new representation allows us to use the power of the coinductive scheme, and keep the similarity between finite and infinite data types. Moreover, it can be used to define some quotient types. To do this efficiently, we add to the logic the connective "restriction" introduced by Parigot in [12]. This is a binary connective, denoted by $F \upharpoonright G$. This connective is a conjunction which keeps only the algorithmic content of the left formula.

Using this connective, it's easy to define a quotient type, from a data type D[x] and an equivalence relation R[x,y]. For instance, we can define:

$$D_R[x,y] = D[x] \upharpoonright R[x,y]$$

Proposition 16 *$D_R[x,y]$ defines a data type with equality for any interpretation such that D[x] is a Krivine data type (see definition 12)*

Proof: We use the definition of the interpretation of the connective $\upharpoonright$ which is $|F \upharpoonright G|^\sigma = |F|^\sigma$ if $\mathcal{M} \models |G|^\sigma$ for all classical full models $\mathcal{M}$ coinciding with σ on first order terms, and $|F \upharpoonright G|^\sigma = \emptyset$ otherwise. So $t \in |D_R[x,y]|^\sigma$ iff $t \in |D[x]|^\sigma$ and $\mathcal{M} \models R[x,y]$ for all classical full models $\mathcal{M}$ coinciding with σ on first order terms. But, because $D[x]$ is a data type and R[x,y] is an equivalence relation, we get the expected result. ♠

Then, as for Krivine data types, we can extract correct programs. Indeed, if we prove $\vdash t : \forall x \forall y (D_R[x,y] \rightarrow D_R[f(x), f(y)])$, then $u \in D^\sigma$ implies $(t\,u) \sim_\omega (|f|^\sigma\ u)$ and the function $|f|^\sigma$ is compatible with the equivalence R. Thus, we can say that t is a correct program for f.

Acknowledgements: I should like to thank Andrew Wilson and Benjamin Pierce for their comments. All mistakes are, of course, my own.

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[2] C. Böhm. Alcune proprieta delle forme $\beta\eta$-normali nel λk-calculus. Pubblicazioni 696, Instituto per le Applicazioni del Calcolo, Roma, 1968.

[3] T. Coquand and G. Huet. The calculus of construction. In *Information and Computation*, pages 241–262, 1988.

[4] N. de Bruijn. The mathematical language automath, its usage and some of its extensions. In *Symp. on automatic demonstration*, pages 29–61. Springer Verlag, 1970. Lecture Notes in Mathematics Vol. 125.

[5] J-Y. Girard. The system F of variable types: fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[6] W. Howard. The formulae-as-types notion of construction. *To H.B. Curry: Essays on combinatory logic, λ-calculus and formalism*, pages 479–490, 1980.

[7] Jean-Louis Krivine. *Lambda-Calcul : Types et Modèles*. Etudes et Recherches en Informatique. Masson, 1990. Available now in english version.

[8] Jean-Louis Krivine and Michel Parigot. Programming with proofs. *Inf. Process. Cybern.*, EIK 26(3):149–167, 1990.

[9] Daniel Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44:51–68, 1986.

[10] Per Martin-Löf. Lecture notes on the domain interpretation of type theory. In Programming Methodology Group, editor, *Workshop on the Semantics of Programming Languages*, Göteborg, Sweden, 1983. Chalmers University of Technology.

[11] Paul Francis Mendler. *Inductive definition in type theory*. PhD thesis, Cornell University, 1988.

[12] Michel Parigot. Recursive programming with proofs. *Theoritical Computer Science*, 94:335–356, 1992.

[13] C. Paulin-Mohring. Inductive definitions in the calculus of constructions. Technical report, INRIA, 1989. Technical Report Number 110.

[14] Christophe Raffalli. *Le système AF_2 avec points fixes*. PhD thesis, Université Paris 7, Février 1994.

[15] M. Tatsuta. Realizability interpretation of coinductive definitions and program synthesis using streams. In *Proceedings of the fifth generation computer systems*, pages 666–673. ICOT, 1992.