Infinite Objects in Type Theory

Thierry Coquand

Programming Methodology Group. Department of Computer Sciences. Chalmers University of Technology and University of Göteborg. S-412 96 Göteborg, Sweden
e-mail coquand@cs.chalmers.se

Abstract. We show that infinite objects can be constructively understood without the consideration of partial elements, or greatest fixed-points, through the explicit consideration of proof objects. We present then a proof system based on these explanations. According to this analysis, the proof expressions should have the same structure as the program expressions of a pure functional lazy language: variable, constructor, application, abstraction, case expressions, and local let expressions.

1 Introduction

The usual explanation of infinite objects relies on the use of greatest fixed-points of monotone operators, whose existence is justified by the impredicative proof of Tarski's fixed point theorem. The proof theory of such infinite objects, based on the so called co-induction principle, originally due to David Park [21] and explained with this name for instance in the paper [18], reflects this explanation. Constructively, to rely on such impredicative methods is somewhat unsatisfactory (see for instance the discussion in [13]) and this paper is a tentative attempt for a more direct understanding of infinite objects. Interestingly, the explicit consideration of proof objects plays an essential rôle here and this approach suggests an alternative reasoning system. In particular, the notion of constructors, or introduction rules, keeps the fundamental importance it has for proof systems about well-founded objects [15], while it appears as a derived notion in proof systems based on co-induction (where this notion is secondary to the notion of destructors, or elimination rules). As a consequence, the strong normalisation property does not hold any more, but it is still the case that any closed term reduces to a canonical form.

Briefly, we can describe our approach as follows. A co-inductive predicate, relation, ... is defined by its introduction rules. Following the proofs as programs principle, we represent them as constructors of a functional language with dependent types and each proof is now represented as a functional expression. Like

* This research has been done within the ESPRIT Basic Research Action "Types for Proofs and Programs". It has been paid by NUTEK, Chalmers and the University of Göteborg.

in a programming language, we can define a function by recursion, which corresponds to a proof where the result we want to prove is used recursively. This cannot be considered to be a valid proof in general, and has to satisfy some conditions in order to be correct. We describe a simple syntactical check that ensures this correctness, which we believe leads to a natural style of proofs about infinite (or lazy) objects.

Since one important application we have in mind is the mechanisation of reasoning about programs and processes, we analyse in our formalism some concrete examples from the literature [22, 18].

Besides to illustrate further the increasingly recognized importance of infinite proofs for programming language semantics, we hope to show also that the addition of infinite objects is an interesting extension of Type Theory. In particular, we can now represent a notion of processes in Type Theory.

2 General presentation

2.1 Type Theory of Well-Founded Objects

We recall briefly some basic notions of type theory of well-founded objects, that will be important for the extension to infinite objects. We use the word "expressions" or "terms" for designing syntactical representations of such objects. The books [20, 15] contain more detailed explanations, and the reference [3] describes the addition of case expressions and pattern-matching. We present first these definitions in general terms, and will explicit them more in details in a special instance when we present the guarded induction principle.

Computation Tree Semantics A(n inductive) **set** A is defined by its **constructors**. A closed term of type A can be thought of as a well-founded tree, built out of constructors. We identify sets and **propositions**. The constructors can be interpreted as **introduction rules**, and a closed proof of the proposition A is a well-founded proof tree built out of introduction rules.

Besides terms purely built out of constructors, one needs also to consider **noncanonical** expressions [15, 7]. The addition of such expressions is done in such a way however that any closed term of a closed set can be reduced to a **canonical form**, i.e. a term of the form $c(a_1, \ldots, a_n)$ where c is a constructor[2]. We can then associate in a natural way to any term a tree built out of constructors, and we require this tree to be well-founded. This tree is called the **computation tree** of a term. A **component** of a closed term is a (closed) term of the same type that appears in its computation tree. This defines an order relation on closed terms, called the **component ordering**.

What is essential is the fact that the component ordering is well-founded.

These notions can be traced back to Brouwer's idea of the "fully analysed" form of a proof [7].

---

[2] Our notations will follow [20].

Examples The set N of integers is defined by its constructors $\mathsf{0} : \mathsf{N}$ and $\mathsf{s} : (\mathsf{N})\mathsf{N}$. A closed element of type N is thus a finite object of the form $\mathsf{s}^k(\mathsf{0})$.

Let us consider a type P with constructors $\mathsf{out} : (\mathsf{N})(\mathsf{P})\mathsf{P}$, $\mathsf{in} : ((\mathsf{N})\mathsf{P})\mathsf{P}$ and $\mathsf{nil} : \mathsf{P}$. A closed element $p : \mathsf{P}$ has to be thought of as a well-founded tree built with the constructors out, in and nil. For instance, if $u(n) = \mathsf{out}(n, \mathsf{nil})$, the term $\mathsf{in}(u)$ has for components all the instances $\mathsf{out}(\mathsf{s}^k(\mathsf{0}), \mathsf{nil})$ and nil.

The requirement that we should be able to think of all closed elements as a tree, with a definite branching (that may be infinite), imposes strong restriction on the type of the constructors. Thus, we cannot have a set X with a constructor of type $((X)X)X$ or of type $(((X)\mathsf{N})\mathsf{N})X$. However, a condition of strict positivity [8] on the type of the constructors is enough to ensure that we can think of elements as trees built out of constructors.

Noncanonical Constants We now give a general way of adding new noncanonical constant. These additions will be such that it will be possible at each "stages" to associate a well-founded proof tree to any closed object. A new constant f is first given a type $(x_1 : A_1, \ldots, x_n : A_n)A$, and then by its definition $f(x_1, \ldots, x_n) = e$, where e is an expression built on previously defined constants and case expressions. The definition may be recursive, but, using the semantics of a term as a well-founded tree, we can ensure that the recursive calls are well-founded and justify in such a way this recursivity. We notice, as in [6], that there is a simple syntactical check that ensures this: there exists a lexicographic ordering of the arguments of f, such that all recursive calls are well-founded for the lexicographic extension of the component ordering.

Examples The Ackerman function $A : (\mathsf{N})(\mathsf{N})\mathsf{N}$ defined by the equation

$$A(\mathsf{0}, n) = \mathsf{s}(n),\ A(\mathsf{s}(m), \mathsf{0}) = A(m, \mathsf{s}(\mathsf{0})),\ A(\mathsf{s}(m), \mathsf{s}(n)) = A(m, A(\mathsf{s}(m), n)),$$

follows the schema of definition, since the recursive calls are always smaller for the lexicographic ordering. We can thus add it as a noncanonical constant.

Soundness As noticed in [15], to follow this semantics of well-founded trees will ensure that there is *no* closed term of type $\perp$, which is defined as a set with no constructor. Indeed, there is by definition no canonical element of this type, and hence no element that reduces to a canonical form.

This simple remark is important if we look at this set theory as a proof system. Indeed, it expresses a form of consistency of this proof system. So, as long as we add new rules that are justified w.r.t. this semantics in term of well-founded trees, we are sure of the consistency of our rules.

2.2 Infinite Objects

Analogy between proofs and processes It is tempting to think of an object of type P as a process p which has three possible behaviours: it can either emit an integer and becomes p_1, when it is of the form $\mathsf{out}(n, p_1)$, or express that it needs an integer as input, if it is of the form $\mathsf{in}(u)$, or show that it is inert, if it is of the form nil. In this reading, the computation tree of an element is the "behaviour tree" [19] of the process associated to it.

With this reading, the restriction to well-founded objects seems too strong. For the type P as defined above, this will mean that we consider only processes that eventually become inert. This forbids for instance a process $p = \mathsf{in}([n]\mathsf{out}(\mathsf{s}(n), p))$ that interactively asks for an integer and outputs its successor.

It is thus quite natural to consider also **lazy** elements that can be thought of as arbitrary, not necessarily well-founded, trees built out of constructors. In particular, a lazy term eventually reduce to constructor form, and there cannot be any lazy proof of $\perp$.

As we have seen, the consideration of such objects is common in the analysis of processes [19]. The consideration of not necessarily well-founded objects arouse also in proof theory, for the study of proofs in ω-logic [9].

The process $p = \mathsf{in}([n]\mathsf{out}(\mathsf{s}(n), p))$ recursively defined is a lazy element of the set P. It makes also sense of considering lazy elements of the set Ω, which has only one constructor $s : (\Omega)\Omega$. The well-founded version of this type is empty, but the set Ω contains the recursively defined lazy element $\omega = s(\omega)$. An object is called **productive** if we can associate a computation tree to it, without requiring this computation tree to be well-founded. If x is a (productive) object of type Ω, it should reduce to an element $x = s(x_1)$ because s is the only constructor of the set Ω, and similarly x_1 should reduce to an element $x_1 = s(x_2)$, and so on.

We can now see well-founded objects as special cases of productive objects. They are productive objects that are *accessible* for the component relation. If A is a data type, we will write $a \epsilon A$ for stressing that a is a well-founded element of A, and, in general, $a : A$ for expressing only that a is a productive element of A. Sometimes, we consider only well-founded elements of a data type A, for instance if A is the data type N of natural numbers, and it is then understood that $a : A$ means that a is well-founded.

Though this notion of productivity seems clear, at least in the case of finitely branching trees, the main problem will be to give a finitary precise definition of productivity. We will give this definition after reviewing some attempts in adding infinite objects to type theory. Though simple, it is surprising that the definition we shall present achieves this goal without infinitary considerations based on greatest fixed-points or infinite ordinals[3].

[3] This definition can be extracted from the paper [11], where the notion of "convergence" corresponds to our notion of productivity.

Problem with the addition of infinite objects Some problems in adding infinite objects in Type Theory are analysed by Martin-Löf in the reference [16]. One basic problem can be expressed as follows: how to add infinite objects without also adding partial objects, that is objects that do not reduce to a canonical form? We recall that this condition was indeed crucial as a guarantee of consistency of Type Theory seen as a proof system.

For instance, it is not correct to define a function $f : (\Omega)\Omega$ by the equation $f(s(x)) = f(x)$, because then $f(\omega)$ does not reduce to canonical form. In contrast, the definition $f(s(x)) = s(f(x))$ should be clearly allowed, because the element $f(x)$ is then productive if $x : \Omega$ is productive. Indeed, if x is productive, we have a chain of equalities

$$x = s(x_1),\ x_1 = s(x_2),\ x_2 = s(x_3), \ldots$$

which will give the chain equalities

$$f(x) = s(f(x_1)),\ f(x_1) = s(f(x_2)),\ f(x_2) = s(f(x_3)), \ldots$$

Is their a simple syntactical criteria that ensures the preservation of productivity, which is not too restrictive?

In our analysis, a definition of the primitive recursive form

$$f(s(x)) = g(x, f(x))$$

cannot be justified in general. Indeed, the justification of such a definition relies ultimately on the fact that we consider only well-founded objects [15].

In [16], a different view is followed, based on an unexpected analogy between the addition of infinite objects in type theory and non-standard extensions in non-standard analysis. This explanation rejects circular definitions such as $\omega = s(\omega)$, but allow non well-founded definitions such as

$$\omega_0 = s(\omega_1),\ \omega_1 = s(\omega_2), \ldots$$

In this approach, a definition like $f(s(x)) = f(x)$ is allowed. This implies the existence of closed terms that have no canonical form, namely $f(\omega_0)$. Despite this problem, it is still possible however to establish the consistency of Type Theory with such an extension [16].

In the next paragraph, we will suggest a proof principle which can also be seen as a way of defining functions over not necessarily well-founded objects. This new proof principle relies directly on the semantics of an object as a not necessarily well-founded tree built out of constructors.

A key example At this point, the basic difficulty is to find a way of defining functions that ensures that any instances of such functions on productive elements are productive. For this, the first step is of course to have a precise notion of productivity.

In order to find this definition, let us analyse a key example. We consider the function $f : (\mathsf{P})\mathsf{P}$ defined by the equations

$$f(\mathsf{nil}) = \mathsf{nil}, \quad f(\mathsf{in}(u)) = \mathsf{in}([n]f(u(n))), \quad f(\mathsf{out}(n, p)) = \mathsf{out}(n, f(p)).$$

It should be clear intuitively that $f(p)$ is productive if p is productive. How can we be convinced of this fact in a clear and rigourous way? One answer may be a definition of productivity as a greatest fixed-point. While this answer is formally satisfactory, it can be argued that its impredicative use of Tarski's fixed point theorem is not a satisfactory finitary explanation of infinite objects.

It can be noticed however that it is directly clear that $f(p)$ reduces to a canonical form if p is productive. Furthermore, we can see that all components of $f(p)$ are then of the form $f(q)$, for some productive $q : \mathsf{P}$, or nil. This remark suggests the definitions of the next section.

2.3 Guarded induction principle

Reducible elements In order to simplify the discussion, we suppose that we have introduced only two data types, the data type N of expressions built on the constructors $\mathsf{s} : (\mathsf{N})\mathsf{N}$ and $\mathsf{0} : \mathsf{N}$, and the data type of lazy expressions P built on constructors $\mathsf{nil} : \mathsf{P}$, $\mathsf{in} : ((n\epsilon\mathsf{N})\mathsf{P})\mathsf{P}$, and $\mathsf{out} : (n\epsilon\mathsf{N})(\mathsf{P})\mathsf{P}$. We hope that it is clear how this discussion extends to the consideration on any inductively defined data types.

Definition: We define what are the **direct components** of a closed expression $p : \mathsf{P}$. If p reduces to nil, it has no direct component. If p reduces to $\mathsf{out}(\mathsf{s}^k(\mathsf{0}), q)$, it has for direct component q. If p reduces to $\mathsf{in}(u)$, it has for direct components all $u(\mathsf{s}^k(\mathsf{0}))$. A **component** of p is p itself or a component of one of its direct component.

Definition: An element of type P is **productive** iff all its components reduce either to nil, or to an element of the form $\mathsf{in}(u)$, or to an element of the form $\mathsf{out}(\mathsf{s}^k(\mathsf{0}), p)$.

We can then define when a close expression is **reducible** of type A, where A is a type built from the data type N and P. An expression c of type $(A)B$ is reducible iff the expression $c(a)$ is reducible of type B when a is reducible of type A. For the type N, it is simply to be convertible to a finite expression $\mathsf{s}^k(\mathsf{0})$. For the type P, it is to be productive.

Guarded definitions Let f be a constant of type

$$(x_1 : A_1, \ldots, x_p : A_p)A,$$

where A is a set (ground data type). We will give a sufficient condition on a recursive definition $f(x_1, \ldots, x_p) = e$ of f to ensure that f is a reducible expression. For this, we define when f is **guarded by** at least n constructors an expression e. It means intuitively that all occurences of f in e are of the form $f(u_1, \ldots, u_p)$ where f does not occur in any u_i, and are all guarded by only constructors and at least n constructors. This is by case analysis on e :

- if f does not occur in e, then f is guarded by at least n constructors in e, for all n,
- if e is of the form $c(u_1, \ldots, u_k)$ where c is a constructor, then f is guarded by at least n constructors in e iff $n \geq 1$ and f is guarded by at least $n-1$ constructors in all u_i, or $n = 0$ and f is guarded by at least 0 constructors in all u_i,
- if e is of the form $[x]u$, then f is guarded by at least n constructors in e iff f is guarded by at least n constructors in u,
- if e is a case expression $case(v, p_1 \rightarrow e_1, \ldots, p_k \rightarrow e_k)$, then f is guarded by at least n constructors in e iff f does not occur in v and f is guarded by at least n constructors in all e_i,
- if e is of the form $f(u_1, \ldots, u_p)$ and f does not occur in $u_1, \ldots, u_p$, then, f is guarded by at least 0 constructor in $f(u_1, \ldots, u_p)$.

Finally, we say that f is guarded in e iff f is guarded by at least n constructors in e for some $n \geq 1$.

Guarded induction principle The guarded condition is well-known for the recursive definition of processes [19]. The two important points here are first, its justification based on an inductive notion of productivity, and second, its use as a proof principle. In our setting, the importance of this notion comes from the following result.

Theorem: If $f : (x_1 : A_1, \ldots, x_n : A_n)A$, where A is a ground data type, has a guarded recursive definition

$$f(x_1, \ldots, x_n) = e,$$

where e is an expression built out only from f and reducible constants, then f is reducible.

Proof: We illustrate this proof on the previous example, which is hopefully generic enough. We consider $f : (\mathsf{P})\mathsf{P}$ defined by the equation

$$f(p) = case(p, \mathsf{nil} \rightarrow \mathsf{nil}, \mathsf{in}(u) \rightarrow \mathsf{in}([n]f(u(n))), \mathsf{out}(n, q) \rightarrow \mathsf{out}(n, f(q))).$$

Since p is reducible, $f(p)$ either reduces to nil, or to $\mathsf{in}([n]f(u(n)))$ if p reduces to $\mathsf{in}(u)$ or to $\mathsf{out}(n, f(q))$ if p reduces to $\mathsf{out}(n, q)$. Hence either $f(p)$ reduces to nil, or the direct components of $f(p)$ are all of the form $f(q)$, where q is reducible. Hence $f(p)$ is productive if p is productive. This means that f is reducible. **Q.E.D.**

This theorem can be read as a proof principle. In order to establish that a proposition ϕ follows from other propositions $\phi_1, \ldots, \phi_q$, it is enough to build a proof term e for it, using not only natural deduction, case analysis, and already proven lemmas, but also using the proposition we want to prove recursively, provided such a recursive call is guarded by introduction rules. We call this proof principle the "guarded induction principle". We hope to show by the examples given below that this reasoning principle is quite flexible and intuitive in practice.

The guarded induction principle will ensure that all closed expressions are reducible, and hence that they reduce to a canonical form. In particular, this implies that there will be no closed proof of $\perp$. This is a way of expressing the correctness of the guarded induction principle.

Some remarks on this proof principle First, it has to be noticed that this criteria cannot accept nested occurences of the function, contrary to the well-founded cases. Thus, we cannot define a function $f : (\Omega)\Omega$ by the equation

$$f(s(s(x))) = s(f(f(x))),$$

since the nested occurence of f in the right handside is not guarded. Indeed, in this case, it can be checked that $f(\omega)$ is not productive: since ω reduces to $s(s(\omega))$, the term $f(\omega)$ has for component $f(f(\omega))$ and this term does not reduce to canonical form.

Another remark is that we can combine this test with the previous test on well-founded recursive calls, if some arguments are explicitely assumed to be well-founded. This situation will occur in one example [18] analysed below, where an infinite proof is defined by well-founded recursion over an evaluation relation.

Finally, this guarded condition may seem too restrictive, especially in the definition of functions over infinite objects. Several programs on streams, even if they preserve productivity, do not obey in general this guarded condition [26]. Here is a simple example. If we consider the set of streams of integer S with only one constructor $\mathsf{cons} : (\mathsf{N})(\mathsf{S})\mathsf{S}$, we can define of the function $map : ((\mathsf{N})\mathsf{N})(\mathsf{S})\mathsf{S}$ by the guarded equation

$$map(f, \mathsf{cons}(x, l)) = \mathsf{cons}(f(x), map(f, l)),$$

and thus consider the equation

$$u = \mathsf{cons}(\mathsf{0}, map(\mathsf{s}, u)),$$

which should represent the stream $\mathsf{cons}(\mathsf{0}, \mathsf{cons}(\mathsf{s}(\mathsf{0}), \ldots))$. This definition is not allowed because it is not guarded. Indeed, the occurence of u in the right handside appears in $map(\mathsf{s}, u)$ and map is *not* a constructor.

We think that the situation is similar to the one of well-founded objects, where the condition on structurally smaller recursive calls does not capture all usual definitions of programs defined over well-founded objects (though its scope is surprisingly large [3, 6]).

Though this does not seem to be the general case, some non guarded definitions can be turned easily in definitions that are guarded. For the previous attempt of the definition of the stream $\mathsf{cons}(\mathsf{0}, \mathsf{cons}(\mathsf{s}(\mathsf{0}), \ldots))$, we can instead first introduce the function $v : (\mathsf{N})\mathsf{S}$ by the guarded definition

$$v(n) = \mathsf{cons}(n, v(\mathsf{s}(n))),$$

and then $u = v(\mathsf{0})$[4].

Furthermore, the first intended application is for reasoning about infinite objects, and not for programming on them. For this application, the guarded condition is enough to give a proof system at least as powerful as the one based on co-induction, and seems more flexible on the examples we have tried. It is actually by trying to understand intuitively what was going on in proofs by co-induction that the guarded condition came out as a proof principle.

To summarize, what is important about the guarded condition is that it can be ensured by a simple syntactical check, that it can be directly justified, and that it seems to provide a powerful enough proof principle for reasoning about infinite objects.

2.4 Reformulation with rule sets

In this section we express in an abstract way how one can understand inductively a greatest fixed-point. We follow the terminology of [1].

We start with a set U of atoms and a set Φ of **rules**, which are pairs (X, x) such that $X \subseteq U$ and $x \in U$. We write $\Phi : X \mapsto x$ to mean that $(X, x) \in \Phi$. An element $(X, x) \in \Phi$ is called a rule of **conclusion** x and of **premisses** X. There is a monotone operator ϕ associated to Φ, given by

$$\phi(Y) = \{x \in U \mid \Phi : X \mapsto x \text{ for } X \subseteq Y\}.$$

The **kernel** of ϕ is given by

$$K(\phi) = \bigcup \{X \mid X \subseteq \phi(X)\}.$$

This is the greatest fixed point of ϕ.

[4] We introduce below a natural notion of equality between streams. This relation Eq is such that $\mathsf{Eq}(v(n), \mathsf{cons}(n, map(\mathsf{s}, v(n))))$. We will show also that, conversely, $\mathsf{Eq}(l, \mathsf{cons}(n, map(\mathsf{s}, l)))$ implies $\mathsf{Eq}(l, v(n))$. It can be proved that $v(n)$ and $\mathsf{cons}(n, map(\mathsf{s}, v(n)))$ are not convertible as expressions. Intuitively, any conversion derivation is finite, and any proof of equality of these two expressions has to be infinite.

We now give a purely inductive description of $K(\phi)$ in the case where Φ is **deterministic**, i.e. when $\Phi : X_1 \mapsto x$ and $\Phi : X_2 \mapsto x$ entail $X_1 = X_2$.

First, we define $S_\Phi(x)$ as the set of $y \in U$ such that there exists $\Phi : X \mapsto x$ with $y \in X$. Let $z \in S^*_\Phi(x)$ mean that $z = x$ or inductively that $z \in S^*_\Phi(y)$ for some $y \in S_\Phi(x)$. An element of $S_\Phi(x)$ is called a **direct component** of x, and an element of $S^*_\Phi(x)$ a **component** of x. Let $C(\phi) \subseteq U$ be the set of $x \in U$ such that there exists a rule of conclusion x. This defines the set of **canonical** elements. The alternative description of $K(\phi)$ is

$$K'(\phi) = \{x \in U \mid S^*_\Phi(x) \subseteq C(\phi)\},$$

that is, $K'(\phi)$ is the set of elements whose components are all canonical.

Theorem: $K(\phi) = K'(\phi)$.

Proof: If $A \subseteq \phi(A)$ and $x \in A$, then we have $A \subseteq C(\phi)$ and $S^*_\Phi(x) \subseteq A$, using the fact that Φ is deterministic, and hence all the components of x are canonical. This shows the inclusion $K(\phi) \subseteq K'(\phi)$. Conversely, the inclusion $K'(\phi) \subseteq \phi(K'(\phi))$ holds in general, and hence $K'(\phi) \subseteq K(\phi)$, without any hypothesis on Φ. **Q.E.D**

This theorem shows how it is possible to define the kernel of a rule set in a predicative way, namely as $K'(\phi)$, despite the fact that its usual definition as $K(\phi)$ is not predicative. (We take here "predicative" as defined for instance in [14]).

3 Simple examples of proofs and programs

3.1 Divergence

We introduce the following set of expressions

$$0 : \mathsf{Exp},\ \mathsf{s} : (\mathsf{Exp})\mathsf{Exp},\ \omega : \mathsf{Exp},$$

and the following inductively defined relation

$$\mathsf{e}_1 : \mathsf{Eval}(0, 0),\ \mathsf{e}_2 : (x : \mathsf{Exp})\mathsf{Eval}(\mathsf{s}(x), \mathsf{s}(x)),\ \mathsf{e}_3 : \mathsf{Eval}(\omega, \mathsf{s}(\omega)),$$

and the following predicate

$$\mathsf{inf} : (x, y : \mathsf{Exp})(\mathsf{Eval}(x, \mathsf{s}(y)))(\mathsf{Inf}(y))\mathsf{Inf}(x).$$

The term

$$p_\infty : \mathsf{Inf}(\omega)$$

is defined by the guarded equation

$$p_\infty = \mathsf{inf}(\omega, \omega, \mathsf{e}_3, p_\infty),$$

and is thus a lazy proof of $\mathsf{Inf}(\omega)$.

Though this example is quite simple, it illustrates one difference between the present proof system and proofs based on co-induction. A proof that ω is divergent using co-induction will consist in finding a predicate P, which holds for ω, such that $P(x)$ implies that there exists y such $\mathsf{Eval}(x, y)$ and $P(y)$. Thus, one has to find an "invariant" predicate. By contrast, the present approach does not involve the search of suitable predicates, but analyses the problem by looking at the introduction rule for the predicate Inf[5].

3.2 Abstract divergence

In general, if we start with a set A with a binary relation R, one can describe inductively the predicate of accessibility

$$\mathsf{acc} : (x : A)((y : A)(R(x, y))\mathsf{Acc}(y))\mathsf{Acc}(x),$$

of which we consider only well-founded elements, and the predicate of divergence

$$\mathsf{inf} : (x, y : A)(R(x, y))(\mathsf{Inf}(y))\mathsf{Inf}(x).$$

Classically, these subsets form a partition of A. In the present intuitionistic framework, one cannot expect in general to have a proof of

$$(x : A)[\mathsf{Acc}(x) + \mathsf{Inf}(x)].$$

In particular, we cannot derive in our system some results of [12], which establish the equivalence of two notions of divergence using the fact that an element either diverges or converges. It seems quite interesting to investigate this problem more in detail from an intuitionistic point of view (our guess is that this equivalence is not really used, and the non equivalence indicates only that the stronger notion of divergence is the correct intuitionistic notion).

It is however possible to show that these subsets are disjoint, by defining

$$\phi : (x : A)(\mathsf{Acc}(x))(\mathsf{Inf}(x))\ \bot$$

with the following equation

$$\phi(x, \mathsf{acc}(x, f), \mathsf{inf}(x, y, q, r)) = \phi(y, f(y, q), r),$$

which is well-founded because the recursive call of ϕ is smaller on its second argument, which is supposed to be well-founded.

---

[5] Of course, it may be that the proposition we try to prove cannot be proved by case analysis only, and we may have to find appropriate lemmas. We hope however that, both for well-founded and infinite objects, the process of finding these lemmas can be helped by such an analysis.

3.3 Representation of an unreliable medium

We want to build an element $m : \mathsf{P}$ that can be thought of as an unreliable medium: it asks first for an integer input, and either forgets it, or outputs it, and this recursively. For this, we introduce an infinite oracle set C with two constructors $0 : (\mathsf{C})\mathsf{C}$ and $1 : (\mathsf{C})\mathsf{C}$. An object of the set C can thus be thought of as an infinite stream of the form $0(1(0(0(\ldots))))$, and in this case, the computation tree of a term is similar to the binary development of a real number.

The following equations define a function $m : (\mathsf{C})\mathsf{P}$

$$m(0(x)) = \mathsf{in}([n]m(x)),\ \ m(1(x)) = \mathsf{in}([n]\mathsf{out}(n, m(x))),$$

since these equations satisfy the guarded condition.

What is important about this representation is that we will be able to define by a predicate on C when an element of C contains infinitely many ones, and hence to specify when an unreliable medium is fair.

3.4 Definition of co-recursion

We now show on one example how to translate co-induction and co-recursion in our proof system. We suppose given a map $f : (X)[X + X]$ and we want to build from this a map $corec(f) : (X)\mathsf{C}$ satisfying the usual co-recursive equations [22]. For this, we define first $\phi : (X + X)\mathsf{C}$ by the guarded equations

$$\phi(\mathsf{inl}(x)) = 0(\phi(f(x)))\ \ \phi(\mathsf{inr}(x)) = 1(\phi(f(x))).$$

One can then check that $corec(f)(x) = \phi(f(x))$ is such that $corec(f)(x) = 0(corec(f)(y))$ when $f(x)$ is of the form $\mathsf{inl}(y)$ and $corec(f)(x) = 1(corec(f)(y))$ when $f(x)$ is of the form $\mathsf{inr}(y)$. Hence, we have a representation of co-recursion over the set C.

This indicates how one can develop a realisability semantics of co-induction with streams (see [27] and [24]) in such a way that an element of a coinductive type is interpreted by a productive element.

3.5 Fairness

We introduce an inductively defined predicate Event_1 on C, such that $\mathsf{Event}_1(x, y)$ means that x is of the form $x = 0(0(\ldots 0(1(y))\ldots))$. We have two introduction rules

$$\mathsf{d}_1 : (x : \mathsf{C})\mathsf{Event}_1(1(x), x), \qquad \mathsf{e}_1 : (x, y : \mathsf{C})(\mathsf{Event}_1(x, y))\mathsf{Event}_1(0(x), y).$$

A well-founded proof of $\mathsf{Event}_1(x, y)$ has to be thought of as a finite term of the form

$$\mathsf{e}_1(x_1, y, \ldots, \mathsf{e}_1(x_{n-1}, y, \mathsf{d}_1(y))\ldots),$$

with $x = 0(x_1),\ x_1 = 0(x_2), \ldots, x_{n-1} = 1(y)$.

Using the inductively defined predicate Event_1, we can now introduce the predicate $\mathsf{Inf}_1(x)$ which means that x contains infinitely many ones in its development. It has only one introduction rule:

$$\mathsf{inf}_1 : (x, y : \mathsf{C})(\mathsf{Event}_1(x, y))(\mathsf{Inf}_1(y))\mathsf{Inf}_1(x),$$

and a proof of $\mathsf{Inf}_1(x_0)$ should be thought of as an infinite proof term of the form

$$\mathsf{inf}_1(x_0, x_1, p_1, \mathsf{inf}_1(x_1, x_2, p_2, \mathsf{inf}_1(\ldots)))$$

where p_n is a proof of $\mathsf{Event}_1(x_{n-1}, x_n)$. This corresponds closely to the intuition of what it means for such a stream to have infinitely many ones.

A fair unreliable medium will then be defined as a medium $m(x) : \mathsf{P}$, together with a proof of $\mathsf{Inf}_1(x)$.

It may be interesting to see how far such ideas can be adapted to the representation of proofs about a process system like CBS, which can be simulated in a simple way in a lazy functional language [23].

3.6 Proof about the list of iterates

This example is taken from [22]. We define first a relation on the set of stream of integers with the only constructor

$$\mathsf{eq} : (n : \mathsf{N})(l_1, l_2 : \mathsf{S})(\mathsf{Eq}(l_1, l_2))\mathsf{Eq}(\mathsf{cons}(n, l_1), \mathsf{cons}(n, l_2)).$$

As a parenthesis, let us illustrate further our proof principle by showing that Eq is transitive. For this, we declare

$$trans : (l_1, l_2, l_3 : \mathsf{S})(\mathsf{Eq}(l_1, l_2))(\mathsf{Eq}(l_2, l_3))\mathsf{Eq}(l_1, l_3),$$

and define it by the guarded equation

$$trans(\mathsf{cons}(n, l_1), \mathsf{cons}(n, l_2), \mathsf{cons}(n, l_3), \mathsf{eq}(n, l_1, l_2, p), \mathsf{eq}(n, l_2, l_3, q))$$
$$= \mathsf{eq}(n, l_1, l_3, trans(l_1, l_2, l_3, p, q)).$$

Notice finally that if we have a closed infinite proof of $\mathsf{Eq}(l_1, l_2)$, then the two infinite terms l_1 and l_2 have the same computation tree. This relation Eq is analogous to bisimulation equivalence [19].

We end this parenthesis, and present the problem: it is to show that, if we define $v : (\mathsf{N})\mathsf{S}$ by the guarded equation

$$v(n) = \mathsf{cons}(n, v(\mathsf{s}(n))),$$

and $map : ((\mathsf{N})\mathsf{N})(\mathsf{S})\mathsf{S}$ is defined by the guarded equation

$$map(f, \mathsf{cons}(n, l)) = \mathsf{cons}(f(n), map(f, l)),$$

then we have $\mathsf{Eq}(l_0, v(0))$ if $\mathsf{Eq}(l_0, \mathsf{cons}(0, map(\mathsf{s}, l_0)))$.

For this, we define a function

$$f : (n : \mathsf{N})(l : \mathsf{S})(\mathsf{Eq}(l, \mathsf{cons}(n, map(\mathsf{s}, l))))\mathsf{Eq}(l, v(n))$$

by the guarded equation

$$f(n, \mathsf{cons}(n, l), \mathsf{eq}(n, l, \mathsf{cons}(\mathsf{s}(n), map(\mathsf{s}, l)), h))$$

$$= \mathsf{eq}(n, l, v(\mathsf{s}(n)), f(\mathsf{s}(n), l, h)).$$

We have then

$$f(0, l_0, h) : \mathsf{Eq}(l_0, v(0)) \quad [h : \mathsf{Eq}(l_0, \mathsf{cons}(0, map(\mathsf{s}, l_0)))].$$

We can read this proof as a program that transforms a (lazy) proof tree which establishes $\mathsf{Eq}(l_0, v(0))$ to a proof tree that establishes $\mathsf{Eq}(l_0, map(\mathsf{s}, l_0))$. Both proof trees furthermore are infinite and built only with the introduction rule

$$\frac{\mathsf{Eq}(l_1, l_2)}{\mathsf{Eq}(\mathsf{cons}(n, l_1), \mathsf{cons}(n, l_2))}.$$

3.7 Soundness of a type inference system

As test examples, we have represented in a mechanized system the problem of soundness of a type inference system analysed in [18]. This corresponds to using the present version of type theory with possibly infinite objects instead of Peter Aczel's non-well-founded set theory [2].

We shall not describe the proof in detail, but only emphasize some points, using freely the notation of [18]. In our formalism, it is directly justified to introduce an object cl_∞ with the computation rule

$$cl_\infty = < x, exp, E + \{f \mapsto cl_\infty\} > .$$

The relation $v : \tau$ given by the rule (15) of the paper [18] is seen in our formalism as the introduction rule for a relation between expressions and types. Thus, in the case of recursion, rule 6, page 217 (which is the only case where our proof differs), we see the problem of proving $cl_\infty : \tau$ as the problem of building an infinite proof tree ending with $cl_\infty : \tau$. But this is direct, using

$$cl_\infty = < x, exp, E + \{f \mapsto cl_\infty\} >$$

and the fact that it is allowed/guarded to use recursively $cl_\infty : \tau$.

4 Mechanization

We now discuss briefly the mechanization of the present system, and how to use it in the design an interactive proof search. This section is still tentative, and only partial implementations of the ideas described here have been tried on machine so far.

The starting point is to consider the logical framework as described in [20] as a type system that refines the type system of a lazy programming language. Thus the first step is to have a lazy functional language (like haskell or lml) with dependent types. In particular, we can, like in a lazy functional language, introduce new data types with their constructors. The typing relation $x : A$ will mean that x is a lazy element of the data type A, and we have to use another notation, for instance $x \epsilon A$, for expressing that x is a well-founded element of the data type A. An alternative notation is to have only one typing judgement, and to have two kind of data types, ones that have only well-founded elements, and ones that may have productive elements. We can then associate to any data type of the second kind its well-founded part.

In general, of course, the definition of recursive programs/proofs can lead to inconsistent reasonings. We need to introduce the notion of correct environment. The present paper gives a sufficient syntactic condition, to be guarded, for an environment to be valid w.r.t. the semantic of terms as productive objects. This check ensures in particular the consistency of an environment seen as a logical theory, and is complementary to the check of structurally smaller recursive calls [3, 6] for well-founded arguments.

We believe that this system leads to an intuitive interactive proof system, well-suited for providing a mechanical help in the development of proofs in relational (or natural) semantics [18]. The user introduces new sets, predicates, relations defined by their introduction rule. We remark that, in practice and probably because it is clearer, in [12, 4], the relations are not given by their elimination rules, but by their introduction rules.

When one wants to prove a result, or builds a noncanonical function, one first gives to it a name and a type. The use of case expression corresponds to the analysis of the hypotheses. This analysis generates subgoals that can be further analysed until we can write a solution. The possibility of declaring and proving local lemmas (that can be themselves recursively defined) corresponds to the addition of a local let construct in our proof term language. Recursive reasoning is allowed, and the system points out when it may lead to an inconsistency.

Conclusion

We hope to have shown that the guarded proof induction principle is a quite natural way of reasoning about infinite objects. The duality between the guarded conditions for infinite objects and the structurally smaller conditions [6, 3] complements the categorical duality between initial and final objects that is the basis of the notion of co-inductive definition [2, 17].

One main point of this paper, which goes back to the work of Lars Hallnäs [10], is that the impredicative notions that seem necessary in dealing with infinite objects, typically the use of greatest fixed-point or infinite ordinals, can be avoided altogether by explicit consideration of proof objects. Though it was not originally conceived with this remark in mind, the proof system we present can be seen as a further illustration of this basic idea[6].

Acknowledgement

The initial idea of this paper came through a discussion with Eike Ritter and Peter Dybjer about proof systems for infinite objects, and was precised by several discussions with Peter Dybjer, Martin Hofmann, Andy Moran and Lars Hallnäs. Many thanks also to the referees for their comments.

References

1. P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, 739 - 782, (1977), Elsevier.
2. P. Aczel. *Non-Well-Founded Sets* CSLI Lecture Notes, Vol. 14 (LSCI, Stanford, 1988)
3. Th. Coquand. Pattern-Matching in Type Theory. Proceedings of the B.R.A. meeting on Proof and Types, (1992) Bastad.
4. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. POPL'91, (1991).
5. H. Curry and R. Feys. *Combinatory Logic, Vol. 1.* North-Holland Publishing Company.
6. O. Dahl. *Verifiable Programming.* Prentice Hall International, 1992.
7. M. Dummett. *Elements of Intuitionism.* Oxford University Press, 1977.
8. P. Dybjer. Inductive Families. To appear in Formal Aspects of Computing (1993).
9. J.Y. Girard. *Proof Theory and Logical Complexity.* Bibliopolis, 1988.
10. L. Hallnäs. An Intensional Characterization of the Largest Bisimulation. Theoretical Computer Science 53 (1987), 335 - 343.
11. L. Hallnäs. On the syntax of infinite objects: an extension of Martin-Löf's theory of expressions. LNCS 417, COLOG-88, P. Martin-Löf and G. Mints Eds., (1989), 94 - 103.
12. J. Hugues and A. Moran. A semantics for locally bottom-avoiding choice. Proceedings of the Glasogow Functional Programming Workshop'92, WICS (1992).
13. P. Lorenzen. Logical Reflection and Formalism. The Journal of Symbolic Logic, 23, 1958, 241 - 249.
14. P. Lorenzen and J. Myhill. Constructive Definition of Certain Sets of Numbers. The Journal of Symbolic Logic, 24, 1959, 37 - 49.
15. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[6] One can see Martin-Löf's constructive explanation of the addition of non-standard elements through the explicit consideration of non-standard proof-objects [16] as yet another example of this idea.

16. P. Martin-Löf. Mathematics of Infinity. LNCS 417, COLOG-88, P. Martin-Löf and G. Mints Eds., (1989), 146 - 197.
17. N.P. Mendler, P. Panangaden and R.L. Constable. Infinite Objects in Type Theory. Proceeding of the first Logic In Computer Science 1986, 249 - 255.
18. R. Milner, M. Tofte. Co-induction in Relational Semantics Theoretical Computer Science 87 (1991), 209 - 220.
19. R. Milner. *Communication and Concurrency* Prentice Hall International, 1989.
20. B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.
21. D. Park. Concurrency and automata on infinite sequences. in P. Deussen, editor, Proceedings of the 5th GI-conference on Theoretical Computer Science, LNCS 104, (1981), 167 - 183.
22. L. Paulson. Co-induction and Co-recursion in Higher-order Logic. Draft (1993), University of Cambridge.
23. K.V.S. Prasad. Programming with Broadcasts. CONCUR'93, LNCS 715, 173 - 187.
24. C. Raffalli. Fixed points and type systems (Abstract) proceeding of the third B.R.A. meeting on Proofs and Types (1992), Bastad, 309.
25. W. de Roever. On Backtracking and Greatest Fixpoints Formal Description of Programming Concepts, J. Neuhold (ed.), North-Holland, (1978), 621 - 639.
26. B.A. Sijtsma. On the productivity of recursive list functions. ACM Transactions on Programming Language and Systems, Vol. 11, No 4 (1989), 633 - 649.
27. M. Tatsuta. Realisability Interpretation of Coinductive Definitions and Program Synthesis with Streams. Proceedings of International Conference on Fifth Generation Computer Systems (1992) 666 - 673.

This article was processed using the LaTeX macro package with LLNCS style