# Interactive Programs and Weakly Final Coalgebras in Dependent Type Theory

Peter Hancock* Anton Setzer†

November 28, 2004

**Abstract**

We reconsider the representation of interactive programs in dependent type theory, proposed by the authors in earlier papers. The basis of this approach is monadic I/O as used in Haskell. We consider two versions: in the first the interface with the real world is fixed, while in the second the potential interactions can depend on the history of previous interactions. We consider also both client and server programs, that run on opposite sides of an interface. Whereas in previous versions the type of interactive programs was introduced in an ad hoc way, it is here defined as a weakly final coalgebra for a general form of polynomial functor. We give formation/introduction/elimination/equality rules for these coalgebras. Finally we study the relationship of the corresponding rules with guarded induction. We show that the introduction rules are nothing but a slightly restricted form of guarded induction. However, the form in which we write guarded induction is not recursive equations (which would break normalisation – we show that type checking becomes undecidable), but instead involves an elimination operator in a crucial way.

## 1 Introduction

Naïvely conceived, programs developed in dependent type theory are not interactive. They are functions that receive one or more arguments as input, and return one value as output or result. This view of program execution as consisting of a single step of interaction is perhaps appropriate for batch programming, prevalent in the 60's and 70's. At that time a job was submitted to the computer, typically consisting of some numerical computation on prepared data, and the results printed or stored in a file.

Nowadays one expects programs to be interactive. A running program should receive input from external devices (*e.g.* keyboard, mouse, network or sensors), and in response send output to external devices (*e.g.* display, sound card, network, actuators), and this cycle should repeat over and over again, perhaps forever.

The chief interest of dependent type theory for programming is not merely that it is a programming language, but rather that it is a framework for specifying and reasoning about programs. It is therefore necessary to understand how to develop interactive programs in dependent type theory. We hope to use it to develop verified interactive programs.

In this article we explore one approach to the representation of interactive programs in dependent type theory. This approach takes as its basis the concept of "monadic IO" ([21]) used in functional programming. We shall see that in dependent type theory, besides non-dependent interactive programs in which the interface between the user and the real world is fixed as in ordinary functional

*7 Cluny Avenue, Edinburgh EH10 4RN, Email: `hancock@spamcop.net`

†Supported by Nuffield Foundation, grant ref. NAL/00303/G and EPSRC grant GR/S30450/01. Department of Computer Science, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK. Email: `a.g.setzer@swan.ac.uk`.

programming, there is a natural notion of state-dependent interactive programs, in which the interface changes over time. The representation is based on a structure identified by Petersson and Synek in [16].

We shall see that the concept of an interactive program is closely connected with the representation of weakly final coalgebras for certain specific functors. This notion will then be generalised to coalgebras for general polynomial functors.

We will then suggest an extension of Martin-Löf type theory by rules for weakly final coalgebras. There is work on encoding weakly final coalgebras in standard intensional Martin-Löf type theory. However, because it seems that reasoning about final coalgebras will be important in the future, we believe that it is natural to have them as "first class", directly-represented objects, as given by our rules. We shall see that a restricted form of guarded induction (where there is exactly one constructor on the right hand side and where reference can be made only to the function one is defining and not to elements of the coalgebra defined previously) is in exact correspondence with coiteration. Finally we will show that bisimulation is a state-dependent weakly final coalgebra.

**Other approaches to interactive programs in dependent type theory.**

As pointed out to us by Peter Dybjer, in a certain sense one can use an expression with an algebraic data type as an interactive program. First one brings the expression to constructor form. (The reductions considered by Martin-Löf reduce a term only to weak head normal form.) Then one can "peel away" the constructor, choose one of its operands, and reduce it further. To use the expression as an interactive program, one associates with each constructor some action upon the world, and with each response or output forthcoming from that action a selector, that chooses an operand of the constructor. For instance, $2 + 3$ reduces to $\mathrm{S}(2 + 2)$, and one can then decide to investigate the argument $2 + 2$ further and find that $2 + 2$ reduces to $\mathrm{S}(2 + 1)$ *etc.* These successive reductions gives rise to a sequence of (trivial) interactions: handing over a coin to a shop-keeper for example, and receiving an acknowledgement. Or, if one defines $B : \{0, 1\} \to \mathrm{Set}$, $B\ 0 = \emptyset$, $B\ 1 = \mathbb{N}$, $C := \mathrm{W}x : \{0, 1\}.B\ x$, and starts with an element $c : C$, then $c$ reduces to the form $\sup\ a\ f$. In the case $a = 1$ one can apply $f$ to an externally given natural number in order to obtain another element of $C$, and so on; in the case $a = 0$ no response is possible, and the process comes to an end. Note that the process of interpreting a constructor as a command or action, peeling them away and using the response to select an operand with which to continue is not an operation *within* type theory, but an extra-mathematical application *of* type theory. However, in order to obtain strong normalisation and therefore decidable type checking one usually requires that types are well-founded, entailing that such a sequence of interactions will necessarily terminate eventually with some constructor without operands. So nonterminating sequences of interactions are impossible. For this reason, if we are not content merely to model terminating interactive programs, we need to consider coalgebras rather than algebras.

**Related work.** H. Geuvers has introduced in [6] rules for inductive and coinductive type corresponding to corecursion in the context of the simply typed $\lambda$-calculus and in the context of system F. He showed that the resulting systems are strongly normalising. E. Gimenéz ([7]; see as well the book on Coq [2], chapter 13, for an exposition of the coalgebraic data types in Coq, which are based on the work by Gimenéz) has studied guarded recursion for weakly final coalgebras and a corresponding general recursive scheme for initial algebras in the context of Coq. He showed that the definable functions are extensionally the same as those definable by the rules given by Geuvers. However, interactive programs are not studied in their work, nor do they investigate in depth the formation/introduction/elimination/equality rules in the context of Martin-Löf type theory. What is not obvious in the work by Gimenéz is that not only can guarded induction be interpreted using the rules for weakly final coalgebras, but in fact the rules for weakly final coalgebras are exactly those arising from a slightly restricted form of guarded induction. Furthermore, the syntax used by Gimenéz seems to suggest that when one introduces recursive functions by guarded induction, it is only lazy evaluation which prevents their complete reduction. On the other hand, when looking at the rules one realises that this is not the case, and the evaluation of these functions is driven by applying case distinction to an element of the coalgebra, corresponding to our elim-function discussed below.

The problem of representing final coalgebras in type theory was addressed in the special case of of Aczel's non-well-founded sets by Lindström in [17], who gave a representation using an inverse-limit construction that requires an extensional form of type theory. Markus Michelbrink is working on an encoding of weakly final coalgebras in standard intensional Martin-Löf type theory, i.e. on introducing sets representing the weakly final coalgebras and functions corresponding to those given by the introduction and elimination rules such that the equalities given by the rules hold w.r.t. bisimilarity rather than definitional equality.

**Notations and type theory used.** In this article we work in standard Martin-Löf type theory, based on the logical framework with both dependent products and function types. Apart from sets introduced by rules added to type theory in this article, we use the constructs of the logical framework including Set, finite sets, the sets of natural numbers, the disjoint union of sets and the identity type for forming new types.

We write $(x : A) \to B$ for the type of dependent functions $f$, where $f$ takes for its argument an $a : A$ and returns an element $f(a)$ of type $B[x := a]$. This type is the logical framework version of the dependent function type denoted by $\Pi x : A.B$ – the difference is that for $(x : A) \to B$ the $\eta$-rule is postulated at the level of judgemental equality, whereas for $\Pi x : A.B$ it holds at the level of propositional equality. We write $\lambda x.s$ for the function $f$ taking argument $a$ and returning $s[x := a]$. If for $x : A$ we have $s : B$, then $\lambda x.s : ((x : A) \to B)$. We write $\lambda x, y.s$ for $\lambda x.\lambda y.s$. We write $f(a)$ for application of $f$ to $a$, $f(a, b)$ for the application of $f$ to $a$ and $b$, and similarly for longer sequences of applications.

We write $(x : A) \times B$ for the dependent product. The elements of this type are pairs $\langle a, b\rangle$ where $a : A$ and $b : B[x := a]$. We write $\pi_0(a)$ and $\pi_1(a)$ for the first and second projection of an element of this type. $(x : A) \times B$ is the logical framework version of the type $\Sigma x : A.B$ – again the difference is that with $(x : A) \times B$ we postulate the $\eta$-rule at the level of judgemental equality, whereas with $\Sigma x : A.B$ it holds rather at the level of propositional equality.

We write $(x : A, y : B) \to C$ for $(x : A) \to ((y : B) \to C)$, and $(x : A) \times (y : B) \times C$ for $(x : A) \times ((y : B) \times C))$, similarly for longer chains of types. Sometimes we assign a variable to the last set in a product, *e.g.* $(x : A) \times (y : B) \times (z : C)$ although $z$ is never used.

We omit from products and function types variables which are not used (*e.g.* $(x : A, B, z : C) \to D$ instead of $(x : A, y : B, z : C) \to D$, if $C$, $D$ don't depend on $y$), and write $A \to B$ and $A \times B$ instead of $(x : A) \to B$ and $(x : A) \times B$, respectively, where $B$ does not depend on $x$. Furthermore we write $\langle a, b, c\rangle$ for $\langle a, \langle b, c\rangle\rangle$, and similarly for longer sequences.

If $A, B$ : Set then $A+B$ : Set is the disjoint union of $A$ and $B$ with constructors $\mathrm{inl} : A \to (A+B)$ and $\mathrm{inr} : B \to (A+B)$. If $f : A \to C$ and $g : B \to C$, then we define $[f, g] : (A+B) \to C$ as the function such that $[f, g](\mathrm{inl}(a)) = f(a)$, $[f, g](\mathrm{inr}(b)) = g(b)$. Further we usually write $A_0 + A_1 + \cdots + A_m$ instead of $A_0 + (A_1 + \cdots + (A_{m-1} + A_m))$. When referring to this type, we write $\mathrm{in}^m_i$ for the injection from $A_i$ into $A_0 + \cdots + A_m$.

We have the set $\mathbf{1}$ : Set, which has sole element $* : \mathbf{1}$, and assume the $\eta$-rule for $\mathbf{1}$, so we have that if $x : \mathbf{1}$, $x = * : \mathbf{1}$. Furthermore we have the empty set $\emptyset$ : Set, with no elements and elimination rule $\mathrm{efq}_A : (x : \emptyset) \to (A\ x)$ for any function $A : \emptyset \to \mathrm{Set}$. We usually omit the index $A$ of efq.

We frequently refer to the set $\mathbf{1} + C$ (in Haskell called Maybe($C$)), and in connection with this set, write inl instead of inl($*$).

When carrying out proofs, for convenience we usually work in extensional type theory, although many proofs can be carried out in intensional type theory. We write $\mathrm{Id}(A, a, b)$ for the equality type expressing equality of $a : A$ and $b : A$. The canonical element of $\mathrm{Id}(A, a, a)$ will be called $\mathrm{refl}_A(a)$. When the overhead is not too great, we make basic definitions in intensional type theory. Then we use J for the transfer principle derived from the elimination rule, where $\mathrm{J} : (C : A \to \mathrm{Set}, a : A, b : A, x : \mathrm{Id}(A, a, b), C(a)) \to C(b)$, and $\mathrm{J}(C, a, a, \mathrm{refl}_A(a), c) = c$.

We denote the set with two elements $*_0$ and $*_1$ by $\mathbf{2}$.

Apart from the type constructions above, we have one additional type, the type of small types called Set. Elements of Set are types. Set will be closed under all type constructions mentioned in this section (including the function type and product), except for Set itself.

## 2 Non-dependent Interactive Programs

We have studied two main approaches taken in functional programming languages that allow interactive programs to be written[1]

- Constants whose evaluation has side effects.
- The IO-monad.

Constants with side effects are used for instance in ML and Lisp. One cannot use this approach in dependent type theory without imposing restrictions on the language, because in dependent type theory expressions are evaluated during type checking. For example, if $a, a' : A$, then the term $\lambda B, x.x$ is of type $(B : A \to \mathrm{Set}) \to B\ a \to B\ a'$ if and only if $a$ and $a'$ are equal elements of type $A$, which is to say that $a$ and $a'$ evaluate to the same normal form. If there were constants with side effects, the evaluation of $a$ might trigger interactions with the real world. The type correctness of the program might (bizarrely) depend on the results of these interactions.

The idea underlying the IO-monad, as it is used in Haskell, is that a program is a static, mathematical structure that can be used to determine the next interaction, on the basis of previous interactions. Performing an interaction is an external, or extra-mathematical operation, carried out in a loop. operation, which is performed in a loop. Suppose that zero or more interactions have already been performed, and responses to those interactions have been received. Then, using this structure, the next interaction is calculated and performed in the real world. Once a response is obtained the loop is repeated. This was the approach taken in our articles [12, 13, 14], and we repeat the key ideas in the following, following mainly [13].

In [13], an atomic interaction starts with the interactive program issuing a command in the real world (*e.g.* to write a character to the screen, or to return a code of the next key pressed by the user). In response to a command the real world returns an answer. For example if the command was to write a character on the screen, the answer is an acknowledgement message; if the command was to get a key pressed, the answer is a code of the key. Once the answer is obtained, the atomic interaction is finished, and the program continues with the next atomic interaction.

In type theory, we can represent the set of commands as a set $\mathrm{C} : \mathrm{Set}$. The set of responses that can be returned to a command $c : \mathrm{C}$ is represented as a set $\mathrm{R}(c) : \mathrm{Set}$. The interface of an interactive program with the real world is therefore represented by a pair $\langle \mathrm{C}, \mathrm{R} \rangle$, which is an element of

$$\mathrm{Interface}^{\mathrm{nondep}} := (C : \mathrm{Set}) \times (R : C \to \mathrm{Set})$$

(In [13] we used the terminology "world" instead of "interface". We now think that the new terminology is more appropriate.) In the following, when referring to non-dependent programs, we assume a fixed interface $\langle \mathrm{C}, \mathrm{R} \rangle$.

**The set** IO. To run an interactive program $p$ appropriate to the interface $\langle \mathrm{C}, \mathrm{R} \rangle$, we need the following ingredients.

- We need to determine from $p$ the command $c : \mathrm{C}$ to be issued next, by calculating the normal form of $p$.
- For every possible response $r : \mathrm{R}(c)$ to $c$ we need to determine a continuation program $q$. When the atomic interaction initiated by issuing the command $c$ is complete, the interactive program should continue with the interactive program $q$.

Let $\mathrm{IO} : \mathrm{Set}$ be the set of interactive programs – we will see below how to actually introduce this set and the associated functions elim, Coiter in type theory. Note that we suppress here the dependence of IO on the interface. (The same will apply to other operations like elim.) Then we need a function

[1] For an overview over I/O in functional programming, see [8].

$c : \mathrm{IO} \to \mathrm{C}$ determining the command to issue and a function $\mathrm{next} : (p : \mathrm{IO}, \mathrm{R}(c(p))) \to \mathrm{IO}$ that determines the next program from the response. We can combine both ingredients into one function

$$\mathrm{elim} : \mathrm{IO} \to ((c : \mathrm{C}) \times (\mathrm{R}(c) \to \mathrm{IO})) \ .$$

Define

$$\mathrm{F} : \mathrm{Set} \to \mathrm{Set} \ , \qquad \mathrm{F}(X) := (c : \mathrm{C}) \times (\mathrm{R}(c) \to X) \ .$$

Then

$$\mathrm{elim} : \mathrm{IO} \to \mathrm{F}(\mathrm{IO}) \ .$$

If we have $p : \mathrm{IO}$, then execution of the program proceeds as follows. First, we compute $\mathrm{elim}(p) = \langle c, f \rangle$, and issue the command $c$. When we have obtained a response $r : \mathrm{R}(c)$ from the real world, we compute the new program $f(r) : \mathrm{IO}$. This cycle with its two phases of computation and interaction is repeated with $f(r)$, *i.e.* we compute $\mathrm{elim}(f(r))$, issue the relevant command, receive a response which we use to determine the next element of IO to be performed, and so forth.

The process terminates if and when one reaches an element $p : \mathrm{IO}$ which has associated with it a command $c$ such that $\mathrm{R}(c) = \emptyset$. This means that no response by the real world is possible. It can also happen that the program "hangs", or waits forever for a response because the real world never provides a response to a command, although there are possible responses.

Note that execution of interactive programs need not terminate. Consider for instance an editor or word-processor. There is no a priori bound on the length of an editing session. On the other hand, there are situations in which one wants to enforce termination of interactive programs. Consider for instance a program that writes a file to a disk. There will be several interactions with the disk, during which blocks of data are written to different sectors on the disk and information about their location is stored in the directory structure of the disk. In this case one expects that this process terminates after a certain amount of time, so it is natural to demand that only finitely many interactions are possible. In general many functions of an operating system, especially those controlling interactions with hardware, are of this kind.

**Introduction of Elements of IO.** One could describe an interactive program as a labelled tree: the nodes are labelled by commands $c : \mathrm{C}$ and a node with label $c$ has immediate subtrees indexed by $r : \mathrm{R}(c)$. When performing the corresponding program, one would start by issuing the command at the root. Then one would, depending on the response of the real world $r$, move to the subtree with index $r$, issue the command which is given as label of that node, and having received response $r'$, move to the $r'$th subtree and so forth.

In type theory, it turns out to be technically simpler to omit two properties of trees, firstly that each node is reached at most once, and secondly that each node is reached at least once. If one omits these two conditions, then an interactive program is introduced by

- an $X : \mathrm{Set}$, corresponding to the nodes of the tree,
- a function which associates with each node $x : X$ the command $c : \mathrm{C}$ to be issued when control has reached that node and for every $r : \mathrm{R}(c)$ the next node, from which the program should continue having received the response $r$,
- the initial node of the tree $x : X$, with which the program starts.

This means that elements are introduced by a triple $\langle X, f, x \rangle$ where $X : \mathrm{Set}$, $f : X \to ((c : \mathrm{C}) \times (\mathrm{R}(c) \to X))$ and $x : X$. Note that $f : X \to \mathrm{F}(X)$. So the introduction rule for IO is that we have a constructor

$$\mathrm{Coiter} : (X : \mathrm{Set}, f : X \to \mathrm{F}(X)) \to X \to \mathrm{IO}$$

(The name Coiter, which stands for coiteration, will be explained later. The principle of coiteration is well-known in the area of coalgebra theory.)

**Bisimilarity.** Two programs $p, q : \mathrm{IO}$ behave in the same way, if firstly they issue the same command, and secondly when supplied with the same response, they continue with programs which again issue the same command, and so on. The equivalence relation which holds between programs that behave in the same way is (as is well-known) bisimilarity.

In our setting, bisimilarity can be defined as follows: A bisimulation relation is a relation $B \subseteq \mathrm{IO} \times \mathrm{IO}$, such that for every $p, p' : \mathrm{IO}$, if $B(p, p')$ holds and $\mathrm{elim}(p) = \langle c, n \rangle$ and $\mathrm{elim}(p') = \langle c', n' \rangle$, then there exists a proof $cc' : \mathrm{Id}(\mathrm{C}, c, c')$ and for $r : \mathrm{R}(c)$ we have $B(n(r), n'(r'))$, where $r' : \mathrm{R}(c')$ is obtained from $r : \mathrm{R}(c)$ using the transfer principle, *i.e.* $r' = \mathrm{J}(\lambda d.\mathrm{R}(d), c, c', cc', r)$.

If there is a bisimulation relation between $p$ and $p'$, then $p$ and $p'$ obviously exhibit the same behaviour. Conversely, if $p$ and $p'$ behave in the same way, then one can obtain a bisimulation relation, namely the one which identifies $q$ and $q'$ if and only if $q$ is a descendent of $p$ and $q'$ is the corresponding descendant of $p'$. Therefore two interactive programs $p$ and $p'$ behave in the same way if and only if there exists a bisimulation relation $B$ such that $B(p, p')$ holds.

Let B be the union of all bisimulation relations. Then B is called bisimilarity. It is a bisimulation relation, and moreover it is the largest one, since it contains any other bisimulation relation. We write $p \approx p'$ for $\mathrm{B}(p, p')$, and will show below how to define $\approx$ in type theory.

**Equalities and weakly final coalgebras.** When we introduce IO in type theory, we want the following equality to hold. Assume $X : \mathrm{Set}$, $f : X \to \mathrm{F}(X)$ and $x : X$. Assume $f(x) = \langle c, g \rangle$ where $g : \mathrm{R}(c) \to X$. Then $\mathrm{elim}(\mathrm{Coiter}(X, f, x)) = \langle c, \lambda r.\mathrm{Coiter}(X, f, g(r)) \rangle$. In other words, if an element $x$ in $X$ has associated with it a command $c$ and a function that for any $r : \mathrm{R}(c)$ returns $x_r$, then the corresponding IO-program should have associated with it the same command $c$ and, depending on $r$ should return the program associated with the next node $x_r : X$, which is $\mathrm{Coiter}(X, f, x_r)$.

We can extend F to a functor $\mathrm{Set} \to \mathrm{Set}$, whose action on morphisms $f : X \to Y$ gives function $\mathrm{F}(f) : \mathrm{F}(X) \to \mathrm{F}(Y)$, where $\mathrm{F}(f, \langle c, g \rangle) := \langle c, \lambda r.f(g(r)) \rangle$. The functor laws will however only be provable using extensional equality.

With this extension, we can see that IO together with elim will be a weakly final coalgebra for $\mathrm{F} : \mathrm{Set} \to \mathrm{Set}$[2]

That $(\mathrm{IO}, \mathrm{elim})$ is a coalgebra means that $\mathrm{elim} : \mathrm{IO} \to \mathrm{F}(\mathrm{IO})$. That it is weakly final means that for every other coalgebra $(X, f)$, where $X : \mathrm{Set}$ and $f : X \to \mathrm{F}(X)$, there exists an arrow $\mathrm{Coiter}(X, f) : X \to \mathrm{IO}$ such that $\mathrm{elim} \circ \mathrm{Coiter}(X, f) = \mathrm{F}(\mathrm{Coiter}(X, f)) \circ f$:

$$
\begin{array}{ccc}
X & \xrightarrow{\quad f \quad} & \mathrm{F}(X) \\
{\scriptstyle \mathrm{Coiter}(X,f)}\downarrow & & \downarrow{\scriptstyle \mathrm{F}(\mathrm{Coiter}(X,f))} \\
\mathrm{IO} & \xrightarrow[\mathrm{elim}]{\qquad} & \mathrm{F}(\mathrm{IO})
\end{array}
$$

We do not demand uniqueness of the arrow $\mathrm{Coiter}(X, f)$. If we had uniqueness of this arrow, then $(\mathrm{IO}, \mathrm{elim})$ would be a final coalgebra for F. We don't know, whether there are rules, which can be considered as a formulation of the existence of final coalgebras in intensional dependent type theory – the usual principles imply that bisimilarity is equality, which implies extensionality of the equality on $\mathrm{N} \to \mathrm{N}$.

[2] Note that IO emerges here as a coalgebra rather than an algebra. This is natural, since what we actually need for running such programs is the function elim. One could introduce instead IO as an F-algebra (which can be introduced by the Petersson-Synek trees [16]) and use the fact that under some weak initiality condition, an F-algebra is as well an F-coalgebra. Then IO is an inductive-recursive definition and therefore part of a standard extension of Martin-Löf type theory. However, unless one uses non-well-founded type theory, one would not be able to introduce non-well-founded elements of IO.

Continuation-passing I/O (see [8], Sect. 7.6 for an excellent description) represents IO as an algebraic type. If one distinguishes between algebras and coalgebras, this could be considered to be a coalgebra. That type is very close to our definition of IO.

## 3 Dependent Interactive Programs

**Dependent Interactive Programs.** In the preceding section we haven't fully exploited the power of dependent types. With dependent types, it is possible to vary the set of commands available at different times. A typical example would be a program interacting through several windows. Once the program has opened a new window, it can interact with it (*e.g.* read input the user input to this window, or write to that window). After closing the window, such interaction is no longer possible. Another example might be the switching on and off of a printer. After switching it on we can print, whereas when the printer is switched off we can no longer print. Sometimes, the commands available depend on responses of the environment to our commands. For instance, if we try to open a network connection, we either get a success message – then we can communicate via the new channel created – or a failure message – then we can't communicate.

A very general situation can be modelled by having a set $\mathrm{S} : \mathrm{Set}$ of states of the system. Depending on $s : \mathrm{S}$ we have a set of commands $\mathrm{C}(s) : \mathrm{Set}$. For every $s : \mathrm{S}$ and $c : \mathrm{C}(s)$ we have a set of responses $\mathrm{R}(s, c)$ to this command. After a response to a command is received the system reaches a new state, so we have, depending on $s : \mathrm{S}, c : \mathrm{C}(s)$ and $r : \mathrm{R}(s, c)$ a next state $\mathrm{n}(s, c, r)$ of the system. A dependent interface consists of these four components,

$$\begin{array}{rcl} \mathrm{S} & : & \mathrm{Set} \\ \mathrm{C} & : & \mathrm{S} \to \mathrm{Set} \\ \mathrm{R} & : & (s : \mathrm{S}, c : \mathrm{C}(s)) \to \mathrm{Set} \\ \mathrm{n} & : & (s : \mathrm{S}, c : \mathrm{C}(s), r : \mathrm{R}(s, c)) \to \mathrm{S} \end{array}$$

So the set of dependent interfaces is

$$\begin{array}{rl} \mathrm{Interface}^{\mathrm{dep}} := & (S : \mathrm{Set}) \\ & \times (C : S \to \mathrm{Set}) \\ & \times (R : (s : S, c : C(s)) \to \mathrm{Set}) \\ & \times ((s : S, c : C(s), r : R(s, c)) \to S) \end{array}$$

**Programs for dependent interfaces** As with non-dependent interfaces, we require for $\langle \mathrm{S}, \mathrm{C}, \mathrm{R}, \mathrm{n} \rangle : \mathrm{Interface}$, that we have a set of interactive programs $\mathrm{IO}(s) : \mathrm{Set}$ for every $s : \mathrm{S}$. $\mathrm{IO}(s)$ should be the set of interactive programs starting in state $s$. In order to be able to perform an interactive program $p : \mathrm{IO}(s)$, we need to determine the command $c : \mathrm{C}(s)$ to be issued, and a function which for every $r : \mathrm{R}(s, c)$ returns a program to be performed after this response, starting in state $\mathrm{n}(s, c, r)$. That program is therefore an element of $\mathrm{IO}(\mathrm{n}(s, c, r))$. Let elim be the function which determines $c$ and the next program. If we define

$$\mathrm{F} : (S \to \mathrm{Set}) \to (S \to \mathrm{Set}) \qquad \mathrm{F}(X, s) := (c : \mathrm{C}(s)) \times ((r : \mathrm{R}(s, c)) \to X(\mathrm{n}(s, c, r)))$$

then we obtain

$$\begin{array}{rcl} \mathrm{IO} & : & S \to \mathrm{Set} \ , \\ \mathrm{elim} & : & (s : \mathrm{S}) \to \mathrm{IO}(s) \to \mathrm{F}(\mathrm{IO}, s) \ . \end{array}$$

The introduction of elements of $\mathrm{IO}(s)$ is similar to the case of non-dependent interfaces. Instead of one set $X$ of nodes, as in the non-dependent case, the introduction of an interactive program now requires for every state $s$ a set of nodes $X(s)$, *i.e.* an S-indexed set $X : \mathrm{S} \to \mathrm{Set}$. For every $s : \mathrm{S}$ and $x : X(s)$ we need to determine from $p : \mathrm{IO}(s)$ the command $c : \mathrm{C}(s)$ to be issued and for $r : \mathrm{R}(s, c)$ the next node of type $X(\mathrm{n}(s, c, r))$ with which the program continues. As before, these two functions can be summarised by one function $f : (s : \mathrm{S}) \to X(s) \to \mathrm{F}(X, s)$. Further we need an initial node $x_0 : X(s)$. So we have the following introduction rule for IO:

$$\mathrm{Coiter} : (X : \mathrm{S} \to \mathrm{Set}, f : (s : \mathrm{S}) \to X(s) \to \mathrm{F}(X, s), s : \mathrm{S}) \to X(s) \to \mathrm{IO}(s) \ .$$

**Weakly final coalgebras on** $\mathrm{S} \to \mathrm{Set}$**.** As in the case of non-dependent interactive programs, we require an equality rule to hold. Assume $X : \mathrm{S} \to \mathrm{Set}$, $f : (s : \mathrm{S}) \to X(s) \to \mathrm{F}(X, s)$, $s : \mathrm{S}$, and $x : X(s)$. Assume $f(s, x) = \langle c, g \rangle$ where $g : \mathrm{R}(s, c) \to X(\mathrm{n}(s, c, r))$. Then

$$\mathrm{elim}(s, \mathrm{Coiter}(X, f, s, x)) = \langle c, \lambda r.\mathrm{Coiter}(X, f, \mathrm{n}(s, c, r), g(r)) \rangle$$

Again we can extend F to an endofunctor on the presheaf category $\mathrm{S} \to \mathrm{Set}$, by having as morphism part for $X, Y : \mathrm{S} \to \mathrm{Set}$ and $f : (s : \mathrm{S}) \to X(s) \to Y(s)$ the function $\mathrm{F}(f) : (s : \mathrm{S}) \to \mathrm{F}(X, s) \to \mathrm{F}(Y, s)$, where $\mathrm{F}(f, s, \langle c, g \rangle) := \langle c, \lambda r.f(\mathrm{n}(s, c, r), g(r)) \rangle$. As before the functor laws can be proved only with respect to extensional equality.

With this extension, we can see that IO will be a weakly final coalgebra for F: We have $\mathrm{elim} : (s : \mathrm{S}) \to \mathrm{IO}(s) \to \mathrm{F}(\mathrm{IO}, s)$ and for every $X : \mathrm{S} \to \mathrm{Set}$ and $f : (s : \mathrm{S}) \to X(s) \to \mathrm{F}(X, s)$ there exists an arrow $\mathrm{Coiter}(X, f) : (s : \mathrm{S}) \to X(s) \to \mathrm{IO}(s)$ such that $\mathrm{elim} \circ \mathrm{Coiter}(X, f) = \mathrm{F}(\mathrm{Coiter}(X, f)) \circ f$ holds with respect to composition in the presheaf category (where $f \circ g := \lambda s, x.f(s, g(s, x))$):

$$\begin{array}{ccc} X & \xrightarrow{\ f\ } & \mathrm{F}(X) \\ {\scriptstyle \mathrm{Coiter}(X,f)}\downarrow & & \downarrow{\scriptstyle \mathrm{F}(\mathrm{Coiter}(X,f))} \\ \mathrm{IO} & \xrightarrow[\mathrm{elim}]{} & \mathrm{F}(\mathrm{IO}) \end{array}$$

**Bisimilarity.** The definition of bisimilarity between non-state-dependent interactive programs extends directly to state-dependent interactive programs. A relation $B' : (s : \mathrm{S}) \to \mathrm{IO}(s) \to \mathrm{IO}(s) \to \mathrm{Set}$ is a bisimulation relation, if for $s : \mathrm{S}$ and $p, p' : \mathrm{IO}(s)$ such that $B'(s, p, p')$ we have that if $\mathrm{elim}(p) = \langle c, g \rangle$ and $\mathrm{elim}(p') = \langle c', g' \rangle$, then there exists a $cc' : \mathrm{Id}(\mathrm{C}(s), c, c')$ and for $r : \mathrm{R}(s, c)$ we have that $B'(\mathrm{n}(s, c, r), g(r), g''(r))$ holds, where $g'' : (r : \mathrm{R}(s, c)) \to \mathrm{IO}(\mathrm{n}(s, c, r))$ is obtained from $g' : (r : \mathrm{R}(s, c')) \to \mathrm{IO}(\mathrm{n}(s, c', r))$ by using the transfer principle and $cc'$ (this short definition, which can be used with intensional equality, is due to M. Michelbrink). Bisimilarity B is now the largest such relation, *i.e.* the union of all bisimulation relations, and one can easily see that B is in fact a bisimulation relation. As before one can easily see that two programs $p, q : \mathrm{IO}(s)$ behave in the same way if and only if $\mathrm{B}(s, p, q)$ holds. We write $p \approx_s q$ for $\mathrm{B}(s, p, q)$.

# 4 Server-Side Programs and Generalisation to Polynomial Functors

**Server-side Programs.** What we have described above are in a sense "client-side" programs: the program issues a command and gets back a response from the other side of the interface. There are as well server-side programs, in which the program receives commands to which it returns responses.

An example is a user interface. Currently, the standard way for defining user interface is that one first places components like buttons, text boxes and labels in the screen. Then one associates with certain components event handlers, which are functions that take as argument an event (*e.g.* the event of clicking the mouse on a button – the event will encode certain data about this, *e.g.* the coordinates of the mouse click, or flags indicating whether it was a single or double click). The event handler usually doesn't return an answer, but when it is executed, a side effect will take place.

This model of a GUI corresponds to a server side program: the program waits for a command, *e.g.* a mouse click event associated with a button. Depending on that event, it performs one or more interactions with the window manager, the database and possibly other systems. Once these interactions are finished the program is waiting for the next event.

Server side programs correspond to the same definition of the set of interactive programs as above, but with respect to a different functor F, namely in case of non-dependent interfaces

$$\mathrm{F}(X) = (c : \mathrm{C}) \to ((r : \mathrm{R}(c)) \times X)$$

and in case of dependent interfaces

$$\mathrm{F}(X,s) = (c : \mathrm{C}(s)) \to ((r : \mathrm{R}(s,c)) \times X(\mathrm{n}(s,c,r)))$$

Let us write in the following $\mathrm{F}^\infty$ for the set of interactive programs corresponding to functor F. In the non-dependent case we need $\mathrm{F}^\infty : \mathrm{Set}$ and a function $\mathrm{elim} : \mathrm{F}^\infty \to \mathrm{F}(\mathrm{F}^\infty)$, which, depending on a program and a command $c : \mathrm{C}$ from the outside world, determines the response of the interactive program to it and the next interactive program to be performed. In the dependent version, we need an S-indexed set $\mathrm{F}^\infty : \mathrm{S} \to \mathrm{Set}$ of interactive programs, and a function $\mathrm{elim} : (s : \mathrm{S}) \to \mathrm{F}^\infty(s) \to \mathrm{F}(\mathrm{F}^\infty, s)$, which determines for every $s : \mathrm{S}$, $p : \mathrm{F}^\infty(s)$ and every command $c : \mathrm{C}(s)$ from the outside world a response $r : \mathrm{R}(s,c)$ and a program $p' : \mathrm{F}^\infty(\mathrm{n}(s,c,r))$, the program continues with.

**Generalisation: Polynomial functors on families of sets.** We have seen the need to introduce sets $F^\infty : \mathrm{S} \to \mathrm{Set}$ such that there exists $\mathrm{elim} : (s : \mathrm{S}) \to F^\infty(s) \to F(F^\infty, s)$ and constructors

$$\mathrm{Coiter} : (X : \mathrm{S} \to \mathrm{Set}) \to ((s : \mathrm{S}) \to X(s) \to F(X,s)) \to (s : \mathrm{S}) \to X(s) \to F^\infty(s)$$

where we used two kinds of endofunctors on $\mathrm{S} \to \mathrm{Set}$, namely $F = \lambda X, s.(c : \mathrm{C}(s)) \times (\mathrm{R}(s,c) \to X(\mathrm{n}(s,c,r)))$ and $F = \lambda X, s.(c : \mathrm{C}(s)) \to (\mathrm{R}(s,c) \times X(\mathrm{n}(s,c,r)))$. Note that endofunctor of the first kind are more general than those of the second kind: functors of the second kind are by the axiom of choice equivalent to functors of the first kind, whereas the other direction does not always hold.

We can generalise the above to general polynomial functors $F : (\mathrm{S} \to \mathrm{Set}) \to (\mathrm{S} \to \mathrm{Set})$, which are essentially of the form $\lambda X.(c : \mathrm{C}(s)) \to ((r : \mathrm{R}(s,c)) \times ((d : \mathrm{D}(s,r,c)) \to \cdots$. All these functors are strictly positive, and we will in a later article extend the set of polynomial functors to a set of strictly positive ones.

The definition of the set of polynomial functors (which is a dependent form of the usual definition of polynomial functors and extends for instance [4]) is as follows: First we define inductively the set of polynomial functors $F : (\mathrm{S} \to \mathrm{Set}) \to \mathrm{Set}$:

- (*Projection.*) For $s : \mathrm{S}$, $\lambda X.X(s)$ is a polynomial functor.
- (*The constant functor.*) If $A : \mathrm{Set}$, then $\lambda X.A$ is a polynomial functor.
- If $A : \mathrm{Set}$ and for $a : A$ $F_a : (\mathrm{S} \to \mathrm{Set}) \to \mathrm{Set}$ is a polynomial functor, so is $\lambda X.(a : A) \to F_a(X)$.
- If $A : \mathrm{Set}$ and for $a : A$ $F_a : (\mathrm{S} \to \mathrm{Set}) \to \mathrm{Set}$ is a polynomial functor, then $\lambda X.(a : A) \times F_a(X)$ is a polynomial functor.
- If $F, F'$ are polynomial functors $(\mathrm{S} \to \mathrm{Set}) \to \mathrm{Set}$, so is $\lambda X, s.F(X,s) + F'(X,s)$.

(The last case could be reduced to the previous cases by defining it as $\lambda X.s.(x : \mathbf{2}) \times G_x(X,s)$ where $G_{*_0}(X,s) = F(X,s)$, $G_{*_1}(X,s) = F'(X,s)$.)

If for $s : \mathrm{S}$, $F_s : (\mathrm{S} \to \mathrm{Set}) \to \mathrm{Set}$ is a polynomial functor, then $\lambda X, s.F_s(X)$ is a polynomial functor $(\mathrm{S} \to \mathrm{Set}) \to \mathrm{S} \to \mathrm{Set}$.

Polynomial functors $F : \mathrm{Set} \to \mathrm{Set}$ are inductively defined in the same way as polynomial functors $F : (\mathrm{S} \to \mathrm{Set}) \to \mathrm{Set}$, except the clause for projection is replaced by the following:

- (*Identity*). $\lambda X.X$ is a polynomial functor.

It is an easy exercise to introduce for polynomial functors $F$ the morphism part, *i.e.* for $f : (s : \mathrm{S}) \to X(s) \to Y(s)$ a function $F(f) : (s : \mathrm{S}) \to F(X,s) \to F(Y,s)$. However to show that the functor-laws hold on the category of pre-sheaves $\mathrm{S} \to \mathrm{Set}$ requires extensional equality.

**Equivalents of polynomial functors.** In the presence of extensional type theory, one can show that each polynomial functor $F : (\mathrm{S} \to \mathrm{Set}) \to (\mathrm{S} \to \mathrm{Set})$ is equivalent to a functor $G$ of the form $G(X, s) = (c : \mathrm{C}(s)) \times ((r : \mathrm{R}(s, c)) \to X(\mathrm{n}(s, c, r)))$ for some $\mathrm{C}, \mathrm{R}, \mathrm{n}$, *i.e.* there exists a natural equivalence $f : F \dot{\to} G$. (A similar result for a weaker (non-dependent) version of polynomial functors was shown in [4].) We show first that every polynomial functor $F : (\mathrm{S} \to \mathrm{Set}) \to \mathrm{Set}$ is equivalent to a functor $\lambda X.(c : \mathrm{C}) \times ((r : \mathrm{R}(c)) \to X(\mathrm{n}(c, r)))$ for some $\mathrm{C} : \mathrm{Set}$, $\mathrm{R} : \mathrm{C} \to \mathrm{Set}$ and $\mathrm{n} : (c : \mathrm{C}) \to \mathrm{R}(c) \to \mathrm{S}$:

- In case of $F(X) = X(s)$ we define $\mathrm{C} := \mathbf{1}$, $\mathrm{R}(c) := \mathbf{1}$, $\mathrm{n}(c, r) := s$. The corresponding functor $G = \lambda X.\mathbf{1} \times (\mathbf{1} \to X(s))$ is easily seen to be equivalent to $F$.
- In case of $F(X) = A$ for $A : \mathrm{Set}$ we define $\mathrm{C} := A$, $\mathrm{R}(c) := \emptyset$, $\mathrm{n}(c, r) := \mathrm{efq}(r)$. Using extensional equality, one can easily see that the corresponding functor $G = \lambda X.A \times ((r : \emptyset) \to X(\mathrm{efq}(r)))$ is equivalent to $F$.
- In case of $F(X) = (a : A) \to F_a(X)$, and $F_a(X)$ being equivalent to $(c : C'(a)) \times ((r : R'(a, c)) \to X(n'(a, c, r)))$, let $\mathrm{C} := ((a : A) \to C'(a))$, $\mathrm{R}(c) := (a : A) \times R'(a, c(a))$, $\mathrm{n}(c, \langle a, r\rangle) := n'(a, c(a), r)$. The corresponding functor $G = \lambda X.(c : (a : A) \to C'(a)) \times ((r' : (a : A) \times (r : R'(a, c(a)))) \to X(\mathrm{n}(c, r')))$ is equivalent to $\lambda X.(c : (a : A) \to C'(a)) \times ((a : A) \to (r : R'(a, c(a))) \to X(n'(a, c(a), r)))$, which by the axiom of choice is equivalent to $\lambda X.(a : A) \to ((c : C'(a)) \times ((r : R'(a, c(a))) \to X(n'(a, c, r))))$, which is equivalent to $F$.
- In case of $F(X) = (a : A) \times F_a(X)$ and $F_a(X)$ being equivalent to $(c : C'(a)) \times ((r : R'(a, c)) \to X(n'(a, c, r)))$, let $\mathrm{C} := (a : A) \times C'(a)$, $\mathrm{R}(\langle a, c\rangle) := R'(a, c)$ and $\mathrm{n}(\langle a, c\rangle, r) := n'(a, c, r)$. The corresponding functor $G = \lambda X.(b : ((a : A) \times C'(a))) \times ((r : \mathrm{R}(b)) \to X(\mathrm{n}(b, r)))$ is equivalent to $\lambda X.(a : A) \times ((c : C'(a)) \times ((r : R'(a, c)) \to X(n'(a, c, r))))$, which is equivalent to $F$.
- In case of $F(X) = F_0(X) + F_1(X)$ and $F_i(X)$ being equivalent to $(c : C_i) \times ((r : R_i(c)) \to X(n_i(c, r)))$, let $\mathrm{C} := C_0 + C_1$ and $\mathrm{R}(\mathrm{inl}(c)) := R_0(c)$, $\mathrm{R}(\mathrm{inr}(c)) := R_1(c)$, $\mathrm{n}(\mathrm{inl}(c), r) := n_0(c, r)$, $\mathrm{n}(\mathrm{inr}(c), r) := n_1(c, r)$. The corresponding functor $G = \lambda X.(c : \mathrm{C}) \times ((r : \mathrm{R}(c)) \to X(\mathrm{n}(c, r)))$ is equivalent to $\lambda X.((c : C_0) \times ((r : R_0(c)) \to X(n_0(c, r)))) + ((c : C_1) \times ((r : R_1(c)) \to X(n_1(c, r))))$, which is equivalent to $F$.

If now $F : (\mathrm{S} \to \mathrm{Set}) \to (\mathrm{S} \to \mathrm{Set})$ is polynomial and $F(X, s) = F_s(X)$ where $F_s$ is equivalent to $G_s := \lambda X.(c : \mathrm{C}(s)) \times ((r : \mathrm{R}(s, c) \to X(\mathrm{n}(s, c, r)))$, then $F$ is equivalent to $\lambda X, s.G_s(X)$, which is of the desired form.

**The natural numbers, co-natural numbers, iteration and coiteration.** The natural numbers can be introduced as the initial algebra of the polynomial functor $\mathrm{F} : \mathrm{Set} \to \mathrm{Set}$, where $\mathrm{F}(X) := \mathbf{1} + X$. That N is an F-algebra means that we have a constructor $\mathrm{intro} : \mathrm{F}(\mathrm{N}) \to \mathrm{N}$. The relationship to the usual constructors 0 and S of the natural numbers is that $\mathrm{intro}(\mathrm{inl}) = 0$, $\mathrm{intro}(\mathrm{inr}(n)) = \mathrm{S}(n)$. That N is a weakly initial algebra with respect to F means that if $A : \mathrm{Set}$ and $f : A \to (\mathbf{1} + A)$, then there exists a function $\mathrm{Iter}'(A, f) : \mathrm{N} \to A$ such that $\mathrm{Iter}'(A, f) \circ \mathrm{intro} = f \circ \mathrm{F}(\mathrm{Iter}'(A, f))$. If one specialises the equalities for $\mathrm{Iter}'(A, f)$ to $\mathrm{inl} = 0$ and $\mathrm{inr}(n) = \mathrm{S}(n)$ one obtains $\mathrm{Iter}'(A, f, 0) = f(\mathrm{inl})$ and $\mathrm{Iter}'(A, f, \mathrm{S}(n)) = f(\mathrm{inr}(\mathrm{Iter}'(A, f, n)))$. If one replaces the argument $f$ in $\mathrm{Iter}'$ by $n := f(\mathrm{inl})$ and $g := f \circ \mathrm{inr}$, one obtains a function $\mathrm{Iter} : (A : \mathrm{Set}) \to A \to (A \to A) \to \mathrm{N} \to A$ such that $\mathrm{Iter}(A, a, g, 0) = a$ and $\mathrm{Iter}(A, a, g, \mathrm{S}(n)) = g(\mathrm{Iter}(A, a, g, n))$, *i.e.* $\mathrm{Iter}(A, a, g, n) = g^n(a)$. Therefore Iter is the principle of *iteration*. N is not only a weakly initial algebra but an initial algebra, which means that $\mathrm{Iter}'(a, f)$ (or equivalently $\mathrm{Iter}(a, f)$) is the only function fulfilling the above mentioned equation. This is not guaranteed by the principle of iteration alone. In type theory it *is* guaranteed by the principle of induction – using induction one can show that if $g$ is any other function s.t. $g \circ \mathrm{intro} = f \circ \mathrm{F}(g)$, then $(n : \mathrm{N}) \to \mathrm{Id}(A, g(n), \mathrm{Iter}'(a, f, n))$.

The weakly final coalgebra for F introduces the co-natural numbers $\mathrm{N}^\infty$. So we have $\mathrm{elim} : \mathrm{N}^\infty \to (\mathbf{1} + \mathrm{N}^\infty)$. Let us write $0_A$ for $\mathrm{inl} : \mathbf{1} + A$, $\mathrm{S}_A(a)$ for $\mathrm{inr}(a) : \mathbf{1} + A$, where $a : A$, and let us omit the subscript $A$ in case $A = \mathrm{N}^\infty$. Then the elimination rule means that for every element of $n : \mathrm{N}^\infty$ we have that $\mathrm{elim}(n) = 0$ or $\mathrm{elim}(n) = \mathrm{S}(m)$ for some $m : \mathrm{N}^\infty$.

The existence of Coiter means that, if we have $A : \mathrm{Set}$ and $g : A \to (\mathbf{1} + A)$, then there exists a function $\mathrm{Coiter}(A, g) : A \to \mathrm{N}^\infty$ such that $\mathrm{elim} \circ \mathrm{Coiter}(A, g) = \mathrm{F}(\mathrm{Coiter}(A, g)) \circ g$. This means: If $a : A$ and $g(a) = 0_A$ then $\mathrm{elim}(\mathrm{Coiter}(A, g, a)) = 0$. If $g(a) = \mathrm{S}_A(a')$ then $\mathrm{elim}(\mathrm{Coiter}(A, g, a)) = \mathrm{S}(\mathrm{Coiter}(A, g, a'))$. Coiter is the dual of Iter, and here called *co-iteration*. Note that if one defines $n := \mathrm{Coiter}(\mathbf{1}, \lambda x.\mathrm{inr}(*))$, we get $\mathrm{elim}(n) = \mathrm{S}(n)$, so $\mathrm{N}^\infty$ contains infinite co-natural numbers.

## 5 Coiteration in Dependent Type Theory

The standard rules for dependent type theory allow us to introduce inductively defined sets, which correspond to (weakly) initial algebras. Coalgebraic types are not represented in a direct way. Markus Michelbrink is working on modelling state-dependent coalgebras in intensional type theory. At the time of writing this article it seems that he has succeeded, although the proof is complex, and has yet to be verified. Even when his approach is finally accepted, it will still be rather complicated to carry out proofs about coalgebras in this way. Furthermore, if one models interactive programs in this way, it would probably be rather inefficient to actually execute such programs.

The usual approach in dependent type theory is to introduce new types directly as first class citizens rather than reducing them using complicated methods to already existing types. That's what one needs in programming in general: a rich type structure rather than a minimal one, that allows one to program without having to carry out a complicated encoding.

In the same way we think it is the right approach to extend type theory by new rules for weakly final coalgebras. Of course one needs to show that such an extension is consistent, and we will do so in a future article by developing a PER module (In [19] a set theoretic model for final coalgebras was developed).

All rules will depend on a polynomial function $\mathrm{F} : \mathrm{S} \to \mathrm{Set}$, which we suppress. In fact, it requires a derivation to show that F is a polynomial functor, which means that all rules have additional premises which derive that F is a polynomial functor. A complete set of rules for deriving polynomial functor would require the introduction of a data type of such functors analogous to the data type of inductive-recursive definitions introduced in [5]. However, such a theory would lie outside the scope of the present article. For the moment it suffices to restrict the theory to functors of the form $\mathrm{F} = \lambda X, s.(c : \mathrm{C}(s)) \times ((r : \mathrm{R}(s, c)) \to X(\mathrm{n}(s, c, r)))$. That F is a polynomial functor is guaranteed by $\langle \mathrm{S}, \mathrm{C}, \mathrm{R}, \mathrm{n} \rangle : \mathrm{Interface}$.

From the considerations in the previous section we obtain the following rules for weakly final coalgebras (these rules are well-known in the area of coalgebra theory, but have to our knowledge not yet been discussed in the context of Martin-Löf type theory):

**Formation Rule:**

$$\frac{s : \mathrm{S}}{\mathrm{F}^\infty(s) : \mathrm{Set}}$$

**Introduction Rule:**

$$\frac{A : \mathrm{S} \to \mathrm{Set} \qquad f : (s : \mathrm{S}) \to A(s) \to \mathrm{F}(A, s) \qquad s : \mathrm{S} \qquad x : A(s)}{\mathrm{Coiter}(A, f, s, x) : \mathrm{F}^\infty(s)}$$

**Elimination Rule:**

$$\frac{s : \mathrm{S} \qquad p : \mathrm{F}^\infty(s)}{\mathrm{elim}(s, p) : \mathrm{F}(\mathrm{F}^\infty, s)}$$

**Equality Rule:**

$$\mathrm{elim}(s, \mathrm{Coiter}(A, f, s, x)) = \mathrm{F}(\mathrm{Coiter}(A, f), s, f(x))$$

Note that, in case $\mathrm{F}(X, s) = (c : \mathrm{C}(s)) \times ((r : \mathrm{R}(s, c)) \to X(\mathrm{n}(s, c, r)))$ we have that if $f(x) = \langle c, g \rangle$ then $\mathrm{elim}(s, \mathrm{Coiter}(A, f, s, x)) = \langle c, \lambda r.\mathrm{Coiter}(A, f, s, g(r)) \rangle$. So we can write the equality in this case as follows.

$$\text{elim}(s, \text{Coiter}(A, f, s, x)) = \text{case } (f(x)) \text{ of}$$
$$\langle c, g\rangle \quad \to \quad \langle c, \lambda r.\text{Coiter}(A, f, s, g(r))\rangle$$

**Non-dependent version.** In the case $\text{S} = \mathbf{1}$ we have by the $\eta$-rule that $\text{S} \to \text{Set}$ and Set are isomorphic. Instead of using this isomorphism, it is more convenient to add special rules for non-dependent polynomial functors $\text{F} : \text{Set} \to \text{Set}$, which are as follows:

**Formation Rule:**

$$\text{F}^\infty : \text{Set}$$

**Introduction Rule:**

$$\frac{A : \text{Set} \qquad f : A \to \text{F}(A) \qquad x : A}{\text{Coiter}(A, f, x) : \text{F}^\infty}$$

**Elimination Rule:**

$$\frac{p : \text{F}^\infty}{\text{elim}(p) : \text{F}(\text{F}^\infty)}$$

**Equality Rule:**

$$\text{elim}(\text{Coiter}(A, f, x)) = \text{F}(\text{Coiter}(A, f), f(x))$$

**Inductive data types vs. coalgebras.** If we compare the above rules with the rules for inductive data types like the natural numbers or the W-type, we observe the following:

- With inductive data types, the introduction rules are "simple": they don't refer to all sets. On the other hand, the elimination rules are complex and refer universally to all sets (*e.g.* induction on N can have any set as result type).
- For coalgebras, the elimination rules are "simple", and don't refer to arbitrary sets. However the introduction rules are complex, and can refer existentially to arbitrary sets.

This duality is in the nature of things. In the case of inductive data types, we form the least set closed under various operations. What 'closed' means is given by introduction rules, but is described in a simple way. The real power of these types lies in the stipulation that we have the *least* such closed set, and this requires an induction principle referring to all sets.

In case of coalgebras, we form the largest set which fulfils a certain elimination principle. The elimination principle corresponds to a simple elimination rule. The strength comes from the fact that we have the *largest* such set and that requires reference to arbitrary sets.

# 6 Guarded Induction

**Coiter and guarded induction.** Coiteration can be read as a recursion principle. In order to make clear what we mean by this, let us consider the weakly final F-coalgebra IO for the functor $\text{F} = \lambda X.(c : \text{C}) \times (\text{R}(c) \to X) : \text{Set} \to \text{Set}$. A function $f : X \to (c : \text{C}) \times (\text{R}(c) \to X)$ can be split into two function $c : X \to \text{C}$ and $\text{next} : (x : X) \to \text{R}(c(x)) \to X$. Then the rules for $\text{Coiter}(X, f)$ express that for functions $c$ and next as above there exists a function $g : X \to \text{IO}$ (defined as $\text{Coiter}(X, f)$) such that for $x : X$ we have

$$\text{elim}(g(x)) = \langle c(x), \lambda r.g(\text{next}(x, r))\rangle \ .$$

If one thinks of C, R as an interface of an interactive system, this can be read as: the interactive program $g(x)$ is defined recursively by determining for every $x : X$ a command $c(x)$ and then the continuation function that handles a response to this command, defined in terms of $g$ itself.

In coalgebra theory one often discusses the introduction of coalgebras $A$ by definitions of the form

$$A = \text{codata } C_0(\cdots) \mid \cdots \mid C_n(\cdots) \ ,$$

where $C_i$ are constructors, and the arguments of the constructors may refer to $A$ itself at strictly positive positions, as well as to previously defined sets and set constructors. For example the co-natural numbers $\mathrm{N}^\infty$, the set of streams of values of type $A$, and the set of interactive programs can be introduced by the definitions

$$\begin{aligned} \mathrm{N}^\infty &= \mathrm{codata}\ 0 \mid \mathrm{S}(n : \mathrm{N}^\infty) \ , \\ \mathrm{Stream}(A) &= \mathrm{codata}\ \mathrm{cons}(a : A, l : \mathrm{Stream}(A)) \ , \\ \mathrm{IO} &= \mathrm{codata}\ \mathrm{do}(c : \mathrm{C}, \mathrm{next} : \mathrm{R}(c) \to \mathrm{IO}) \ . \end{aligned}$$

With this point of view one might be tempted to reread the above recursion equation for $g$ as

$$g(x) = \mathrm{do}(c(x), \lambda r.g(\mathrm{next}(x, r))) \ .$$

However, were we to allow definitions of that kind, we would immediately get non-terminating programs (define for instance $g : \mathbf{1} \to \mathrm{IO}$ by $g(x) = \mathrm{do}(c, \lambda r.g(*))$). With the original equation $\mathrm{elim}(g(x)) = \langle c(x), \lambda r.g(\mathrm{next}(x, r))\rangle$, termination is maintained because the elimination constant elim must be applied to $g(x)$ to obtain a reducible expression. (The example just given reads $\mathrm{elim}(g(x)) = \langle c, \lambda r.g(*)\rangle$, which is unproblematic).

The principle for defining $g(x) = \mathrm{do}(c(x), \lambda r.g(\mathrm{next}(x, r)))$ is a simple case of guarded induction [3] (see as well the work [7] by Gimenéz; its relationship to the current work was discussed in the introduction). The idea of guarded induction is that one can define elements of such a codata set recursively, as long as every reference in the right hand side of the definition to the function we are defining recursively is "guarded" by at least one constructor. So one can define $f : \mathbf{1} \to \mathrm{N}^\infty$ ('infinity') by $f(x) = \mathrm{S}(f(x))$, one can define $f : \mathrm{N}^\infty \to \mathrm{N}^\infty$ (the successor of a co-natural) by $f(x) = \mathrm{S}(x)$ (without recursion), and one can define $f : \mathrm{N} \to \mathrm{Stream}(\mathrm{N})$ by $f(n) = \mathrm{cons}(n, f(n + 1))$. However, one cannot define $f : \mathbf{1} \to \mathrm{N}^\infty$ by $f(x) = f(x)$, since in this equation the recursion is not guarded.

Guarded induction is unproblematic in the context of lazy functional programming, as long as one doesn't need to test for equality, since there reduction to weak head normal form suffices. In dependent type theory, type checking depends on the decidability of equality of terms (see the example at the beginning of Sect. 2), and we have the following theorem:

**Theorem** *In intensional Martin-Löf type theory extended by the principle of guarded recursion for streams in its original form equality of terms and therefore type checking is undecidable.*

**Proof:** Define, depending on $g_i : \mathbb{N} \to \mathbb{N}$, the functions $f_i : \mathbb{N} \to \mathrm{Stream}$ $(i = 0, 1)$ by guarded recursion as $f_i(n) = \mathrm{cons}(g_i(n), f_i(n{+}1))$. Then $f_0(0) = f_1(0)$ if and only if $g_0$ and $g_1$ are extensionally equal, which is undecidable.

If one instead reads guarded induction as a definition of $\mathrm{elim}(g(x)) = \cdots$ rather than $g(x) = \cdots$, we obtain an unproblematic principle, and one can see that a restricted form of guarded induction is, when viewed in this form, equivalent to coiteration:

The right-hand side of the codata definition

$$A = \mathrm{codata}\ C_0(x_0 : A_{0,0}, \ldots, x_{m_0} : A_{0,m_0}) \mid \cdots \mid C_l(x_l : A_{l,0}, \ldots, x_{m_l} : A_{l,m_l})$$

can be read as a polynomial functor $\mathrm{F} : \mathrm{Set} \to \mathrm{Set}$,

$$\mathrm{F}(X) = \mathrm{F}_0(X) + \cdots + \mathrm{F}_l(X)$$

where

$$\mathrm{F}_i(X) = ((x_0 : A'_{i,0}(X)) \times \cdots \times (x_{m_0} : A'_{i,m_0}(X)))$$

and $A'_{i,j}(X)$ is obtained from $A_{i,j}$ by replacing $A$ by $X$. Note that $A_{i,j}$ either does not depend on $X$, so the functor $A'_{i,j}$ is constant, or $A_{i,j}$ is of the form $(y_0 : D_0) \to \cdots \to (y_k : D_k) \to A$, in which case

$$A'_{i,j} = \lambda X.(y_0 : D_0) \to \cdots \to (y_k : D_k) \to X \ ,$$

which is a polynomial functor.

Let $C_i$ be the injection from $\mathrm{F}_i(\mathrm{F}^\infty)$ into $\mathrm{F}(\mathrm{F}^\infty)$. Then for every element of $\mathrm{F}^\infty$ we have $\mathrm{elim}(a) = C_i(a)$ for some $i$ and $a : \mathrm{F}_i(\mathrm{F}^\infty)$.

Now the principle of coiteration means that if we have a set $A$ and a function $f : A \to \mathrm{F}(A)$ then we get a function $g : A \to \mathrm{F}^\infty$ such that $\mathrm{elim}(g(a)) = \mathrm{F}(g)(f(a))$. This reads: if $f(a) = C_i(\langle a_0, \ldots, a_k\rangle)$, then $\mathrm{elim}(g(a)) = C_i(\langle a'_0, \ldots, a'_k\rangle)$, where $a'_i = a_i$, in case $A_{i,j}(X)$ does not depend on $X$, and $a'_i = \lambda y_0, \ldots, y_k.g(a_i(y_0, \ldots, y_k))$, in case $A_{i,j}(X) = (y_0 : D_0) \to \cdots \to (y_k : D_k) \to X$.

We can reread this as follows. We can define a function $f : A \to \mathrm{F}^\infty$ by defining $f(a)$ for $a : A$ as some $C_i(\langle b_0, \ldots, b_k\rangle)$ where $b_k$ refer, when an element of $\mathrm{F}^\infty$ is needed, to $f$ applied to any other element of $A$. This corresponds to a restricted form of guarded induction, where the right hand side of the recursion has exactly one constructor, and one never refers to $\mathrm{F}^\infty$ but only to $f$ applied to some other arguments.

Let us consider now the definition of an **indexed codata** definition, *i.e.*

$$\begin{array}{rcl} A_0(x : B_0) & = & \text{codata } C_{0,0}(\cdots) \mid \cdots \mid C_{0,m_0}(\cdots) \\ & & \cdots \\ A_l(x : B_l) & = & \text{codata } C_{l,0}(\cdots) \mid \cdots \mid C_{l,m_l}(\cdots) \ , \end{array}$$

where the arguments of $C_{i,j}$ refer to $A_i$ at strictly positive positions. Let $B := B_0 + \cdots + B_l$. Then the above can be read as the definition of a $B$-indexed weakly final coalgebra $A = \mathrm{F}^\infty : B \to \mathrm{Set}$ for a suitable polynomial functor F, which is introduced in a similar way as before. The analogy between guarded induction and iteration is as before, except that one defines now a function $f : (b : B, C(b)) \to \mathrm{F}^\infty(b)$ recursively by defining, in case $b = \mathrm{in}_i^l(b')$ $\mathrm{elim}(b', f(b', c)) = C_{i,j}(\langle c_0, \cdots, c_k\rangle)$, where $c_k$ can refer (and has to refer), in case an element of $\mathrm{F}^\infty(b'')$ is needed, to an element $f(b'', c')$ for some $c'$. So dependent coalgebras correspond to indexed codata definitions.

**Bisimilarity as a state-dependent coalgebra.** Bisimilarity in case of the functor $\mathrm{F}(X, s) = (c : \mathrm{C}(s)) \times (\mathrm{R}(s, c) \to X(\mathrm{n}(s, c, r)))$ can be considered as a weakly final coalgebra over the index set

$$(s : \mathrm{S}) \times \mathrm{F}^\infty(s) \times \mathrm{F}^\infty(s)$$

The condition for a bisimulation relation $B$ as introduced above is that, if $B(s, p, p')$ holds, and $\mathrm{elim}(p) = \langle c, g\rangle$ and $\mathrm{elim}(p') = \langle c', g'\rangle$, then we have $cc' : \mathrm{Id}(\mathrm{C}(s), c(s), c'(s))$ and for $r : \mathrm{R}(s, c(s))$ we have $B(\mathrm{n}(s, c, r), g(r), g''(r))$, where $g''$ was obtained from $g'$ using $cc'$. We can now define the polynomial functor (we curry the arguments for convenience)

$$\begin{array}{l} \mathrm{G} : ((s : \mathrm{S}) \to \mathrm{F}^\infty(s) \to \mathrm{F}^\infty(s) \to \mathrm{Set}) \to (s : \mathrm{S}) \to \mathrm{F}^\infty(s) \to \mathrm{F}^\infty(s) \to \mathrm{Set} \\ \mathrm{G}(X, s, p, p') = \text{case } \mathrm{elim}(p) \text{ of} \\ \qquad \langle c, g\rangle \ \to \ \text{case } \mathrm{elim}(p') \text{ of} \\ \qquad\qquad \langle c', g'\rangle \ \to \ (cc' : \mathrm{Id}(\mathrm{C}(s), c, c'))\times \\ \qquad\qquad\qquad ((r : \mathrm{R}(s, c)) \to X(\mathrm{n}(s, c, r), g(r), g''(r))) \end{array}$$

with $g''$ defined as above (using $cc'$). Then

$$\mathrm{elim} : (s : \mathrm{S}) \to (p, p' : \mathrm{F}^\infty(s)) \to \mathrm{G}^\infty(s, p, p') \to \mathrm{G}(\mathrm{G}^\infty, s, p, p')$$

expresses that $\mathrm{G}^\infty(s, p, p')$ is a bisimulation relation. Further the principle of coiteration means that if we have

$$B : (s : \mathrm{S}) \to \mathrm{F}^\infty(s) \to \mathrm{F}^\infty(s) \to \mathrm{Set} \ ,$$

and

$$f : (s : \mathrm{S}, p, p' : \mathrm{F}^\infty(s)) \to B(s, p, p') \to \mathrm{G}(B, s, p, p') \ ,$$

then

$$\mathrm{Coiter}(B, f) : (s : \mathrm{S}, p, p' : \mathrm{F}^\infty(s)) \to B(s, p, p') \to \mathrm{G}^\infty(s, p, p')$$

The existence of $f$ means that $B$ is a bisimulation relation, and $\mathrm{Coiter}(B, f)$ means that $B$ is contained in $\mathrm{G}^\infty$. So Coiter expresses that $\mathrm{G}^\infty$ contains any bisimulation relation. The introduction and equality rules together express therefore that $\mathrm{G}^\infty$ is the largest bisimulation relation, *i.e.* bisimilarity. So the above rules allow to introduce bisimulation in type theory in a direct way, and one can use guarded induction as a proof principle for carrying out proofs about properties of bisimulation.

**Normalisation.** It seems that the normalisation proof by Geuvers [6] carries over to the intensional version of type theory used in this paper, and that therefore intensional type theory with the rules for state-dependent coalgebras is normalising. If one had guarded induction, normalisation would fail. A counter example is $f : \mathbf{1} \to \mathrm{N}^\infty$, $f(*) = \mathrm{S}(f(*))$. Translating the guarded induction principle used here back into our rules, we obtain a function $f := \mathrm{Coiter}(\mathbf{1}, g)$, where $g := \lambda x.\mathrm{inr}(x) : \mathbf{1} \to (\mathbf{1} + \mathbf{1})$. Note that $f(*)$ is already in normal form. The recursion is carried out only when one applies elim to $f(*)$, and we then obtain $\mathrm{elim}(f(*)) = \mathrm{S}(f(*))$, where the right hand side is again in normal form. So evaluation of full recursion is inhibited, since one needs to supply one application of elim in order to trigger a one step reduction of $f$.

# 7 Conclusion

We have introduced one approach to the representation of interactive programs in dependent type theory, and seen that it gives rise to weakly final coalgebras for polynomial functors. We have investigated rules for final coalgebras that correspond to coiteration, and shown why they correspond to a certain form of guarded induction, namely the definition of functions $g : A \to \mathrm{F}^\infty$ by equations $\mathrm{elim}(g(x)) = C_i(t_0, \ldots, t_k)$ where the terms $t_i$ can (and in fact have to) refer, if an element of $\mathrm{F}^\infty$ is required, to $g$ itself.

The story is far from complete. The next step would be to introduce rules which correspond to a more general form of guarded induction in which we can refer directly to previously introduced elements of $\mathrm{F}^\infty$ in the terms $t_i$, where an element of $\mathrm{F}^\infty$ is required. Those rules will express the principle of corecursion, rather than merely coiteration. A further extension that remains to be formulated is a principle which, when considered as guarded induction, allows further uses of the constructors for the coalgebra in the terms $t_i$. This would allow for instance the definition of a function from N into streams of natural numbers such that $\mathrm{elim}(f(n)) = \mathrm{cons}(n, \mathrm{cons}(n, f(n + 1)))$.

We haven't yet given a PER model for the rules (however, in [19] a set theoretic model is given). We haven't explored in full the relationship between guarded induction and the monad (in [19] it is shown that one obtains a monad up to bisimulation). Further we haven't yet interpreted non-state-dependent coalgebras in ordinary type theory. All this will be presented in a future article.

# References


[1] T. Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 8 – 21, 2001.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[3] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1994.

[4] P. Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoret. Comput. Sci.*, 176(1-2):329–335, 1997.

[5] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1 – 47, 2003.

[6] H. Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Informal proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden*, pages 183 – 207, 1992.

[7] E. Gimenéz. Codifying guarded definitions with recursive schemes. In *Proceedings of the 1994 Workshop on Types for Proofs and Programs*, pages 39–59. LNCS No. 996, 1994.

[8] A. Gordon. *Functional programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[9] L. Hallnäs. An intensional characterization of the largest bisimulation. *Theoretical Computer Science*, 53:335–343, 1987.

[10] P. Hancock. *Ordinals and interactive programs*. PhD thesis, LFCS, University of Edinburgh, 2000.

[11] P. Hancock and P. Hyvernat. Programming as applied basic topology. Submitted, available via http://homepages.inf.ed.ac.uk/v1phanc1/chat.html, 2004.

[12] P. Hancock and A. Setzer. The IO monad in dependent type theory. In *Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999*, 1999. Available via http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html.

[13] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic. 14th international workshop, CSL 2000.*, volume 1862 of *Springer lecture notes in computer science*, pages 317–331, 2000.

[14] P. Hancock and A. Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000*, 2000. Electronic proceedings, available via http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

[15] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141, North Holland, Elsevier, 1999.

[16] K. Kent Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Category theory and computer science (Manchester, 1989)*, pages 128 – 140. LNCS 389, Springer, 1989.

[17] I. Lindström. A construction of non-well-founded sets within Martin-Löf's type theory. *Journal of Symbolic Logic*, 54(1):57–64, 1989.

[18] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory: Lecture Notes*. Bibliopolis, Napoli, 1984.

[19] M. Michelbrink and A. Setzer. State dependent IO-monads in type theory. To appear in the proceedings of the CTCS'04, Electronic Notes in Theoretical Computer Science, Elsevier, 2004.

[20] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[21] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Logic in Computer Science Conference*, 1989.

[22] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[23] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, Oxford, 1990.

[24] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *20'th ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, January 1993.

[25] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI series, Series F: Computer and System Sciences*. Springer Verlag, 1994.