

State Dependent IO-Monads in Type Theory

Markus Michelbrink*
Anton Setzer†
Department of Computer Science
University of Wales Swansea
Singleton Park
Swansea
SA2 8PP
United Kingdom
{m.michelbrink,a.g.setzer}@swansea.ac.uk

May 18, 2004

Abstract

We introduce the notion of state dependent interactive programs for Martin-Löf Type Theory. These programs are elements of coalgebras of an endofunctor on the presheaf category $S \rightarrow \text{Set}$. We prove the existence of final coalgebras for these functors. This shows as well the consistency of type theory plus rules expressing the existence of weakly final coalgebras for these functors, which represents the type of interactive programs. We define in this type theory the bisimulation relation, and give some simple examples for interactive programs. A generalised monad operation is defined by corecursion on interactive programs with return value, and a generalised version of the monad laws for this operation is proved. All results have been verified in the theorem prover Agda which is based on intensional type theory.

1 Introduction

Martin-Löf's type theory [7] can be seen as a programming logic for a functional programming language. The judgement $a \in A$ can especially be read as:

1. a is a program with type A
2. a is a program which satisfies the specification A
3. a is an implementation of the abstract data type specification A .

The above relies on the identification of sets, proposition, and specifications. With this identification dependent type theory gives us the ability to express with full precision any extensional property of a program, which can be defined mathematically. We can check the type of a program mechanically, and type correctness carries full assurance that it satisfies its specification. Versions of type theory have been implemented e.g. in Göteborg [10], Cornell [3], Cambridge [9], Edinburgh [6, 11], and INRIA [2].

In type theory running a program means normalising an expression. Every program terminates, and there is no interaction with the environment. This model is adequate for a large class of programs which, when given a value, execute and give back another value. It is however not adequate for the whole class of programs, which interact with their environment and possibly never terminate.

In this article we continue work of Peter Hancock and Anton Setzer [4, 5]. We generalise

*Supported by EPSRC grant GR/S30450/01.

†Supported by Nuffield Foundation, grant ref. NAL/00303/G and EPSRC grant GR/S30450/01.

the notion of interfaces (worlds) and IO-programs to state dependent interfaces and state dependent programs. In [5] a world is a pair (C, R) , where $C : \text{Set}$ and $R : C \rightarrow \text{Set}$. $c : C$ is interpreted as a command, and Rc is the set of possible responses (from a user, a device or another program) to the command c . For every set A the set of programs $IO\ A : \text{Set}$ (we keep the world fixed) has constructors $\text{leaf} : A \rightarrow IO\ A$ and $\text{do} : (c : C, p : Rc \rightarrow IO\ A) \rightarrow IO\ A$. The program $\text{leaf } a$ terminates and returns value a , whereas $\text{do } (c, p)$ executes c , and after receiving response $r : Rc$ continues as $p\ r : IO\ A$. We generalise this by giving every program a state $s : S$. Now the set of executable commands, the responses, as well as the function giving us the next program depend on the state $s : S$. The resulting notion suits better to real world applications. One of our key examples is a *windowing system*. The client may request a server to open a window. The states now represent the open windows.

The generalisation leads us naturally to an endofunctor F on the presheaf category $S \rightarrow \text{Set}$. We show that this functor has a final coalgebra $\text{elim} : F^\infty \rightarrow F(F^\infty)$. We enrich type theory by rules for a weakly version of this final coalgebra (weakly because we do not demand uniqueness of $\mu(\alpha)$. See below.). The elimination rule corresponds to the morphism elim , the introduction rules to the requirement that there is a morphism $\mu(\alpha) : A \rightarrow F^\infty$ for every coalgebra $\alpha : A \rightarrow FA$, and the equality rule expresses that the associated diagram commutes. The formation rule simply reflects the fact that there is a coalgebra F^∞ .

We define bisimulation for interactive programs. After introducing rules for interactive programs with return value, we define a monad operation $*$ by corecursion, and show that the monad laws for this operation hold with respect to bisimulation.

We work in extensional Type Theory. However the results can be achieved in intensional Type Theory as well. Intensional versions of the results of Sect. 6 are verified in Agda [10]. The code is available under <http://www.cs.swan.ac.uk/~csmichel/>.

Overview. In Sect. 2 we motivate our settings. In Sect. 3 we relate our basic ideas from Sect. 2 to an endofunctor on $S \rightarrow \text{Set}$, and show that this functor has a final coalgebra. In Sect. 4 we introduce the new rules for IO-programs, and define bisimulation. In Sect. 5 we give some simple examples for IO-programs. In Sect. 6 we introduce the rules for IO-programs with return values, define a monad operation for this programs, and show the monad laws with respect to bisimulation.

Besides Sect. 3 we work in a standard dependent type theory (e.g. [8]) with the usual formation, introduction, elimination, and equality rules, extended by our rules.

Acknowledgements. Many ideas for this article are due to P. Hancock, Edinburgh. He could well have been a third author for this article, but preferred to publish his slightly different point of view separately.

2 Interfaces, Programs

An *interface* is a quadruple (S, C, R, n) s.t.

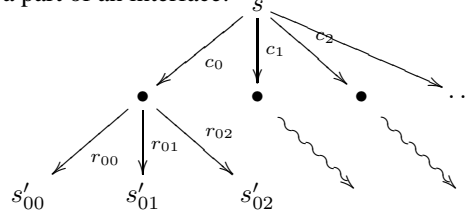
- $S : \text{Set}$
- $C : S \rightarrow \text{Set}$
- $R : \prod s : S. C(s) \rightarrow \text{Set}$
- $n : \prod s : S. \prod c : C(s). R(s, c) \rightarrow S$

S is the set of states, $C(s)$ the set of commands in state $s : S$, $R(s, c)$ the set of responses to a command $c : C(s)$ in state $s : S$, and $n(s, c, r)$ the next state of the system after this interaction. Continuing our example above, in X windows the X server performs the requests (commands) for its clients, and sends them back replies (responses). The possible requests depend on the state of the client, the replies depend on the state of the server and the state of the shared resources: the drawing area and the input channel.

We can view an interface as a generalised transition system, where we have a transition (s, c, r) between states $s : S$ and $s' : S$ iff $c : C(s)$, $r : R(s, c)$ and $s' = n(s, c, r)$. There are two canonical ways to view an ordinary transition system as interface:

- Take $C(s) = \{\text{Transition starting from } s\}$ and $R(s, t)$ as singletons.
- Take $C(s)$ as singletons and $R(s, *) = \{\text{Transition starting from } s\}$.

The picture visualises a part of an interface:

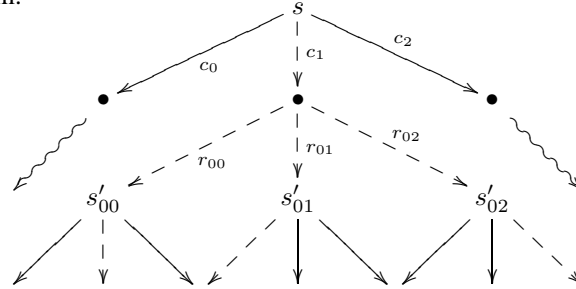


Let (S, C, R, n) be an interface. A *program* for this interface starting in state $s : S$ is a quadruple (A, c, next, a) s.t.

- $A : S \rightarrow \text{Set}$
- $c : \prod s : S. A(s) \rightarrow C(s)$
- $\text{next} : \prod s : S. \prod a : A(s). \prod r : R(s, c(s, a)). A(n(s, c(s, a), r))$
- $a : A(s)$

$A(s)$ is the set of programs starting in state s , $c(s, a)$ the command issued by the program $a : A(s)$, and $\text{next}(s, a, r)$ is the program that will be executed, after having obtained for command $c(s, a)$ the response $r : R(s, c(s, a))$. In the example the program would be an *X client*. It should be noted, that this is the client version of a program. If we interchange in the functor below products and sums, we get server side programs.

The picture visualises a part of a program in relation to its interface. Dashed lines belong to the program:



3 IO as Final Coalgebra

If we view the set S as a discrete category (with only arrows id_s for $s : S$), the presheaf category $S \rightarrow \text{Set}$ has objects $X : S \rightarrow \text{Set}$ and morphism $f : \prod s : S. X(s) \rightarrow Y(s)$, where $X, Y : S \rightarrow \text{Set}$. The composition $g \circ f : \prod s : S. X(s) \rightarrow Z(s)$ of two morphism $f : \prod s : S. X(s) \rightarrow Y(s)$ and $g : \prod s : S. Y(s) \rightarrow Z(s)$ is defined by

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

for $s : S, x : X(s)$.

$\text{id}_X : \prod s : S. X(s) \rightarrow X(s)$ is given by $\text{id}_X(s) = \text{id}_{X(s)}$.

We look at the functor $F : (S \rightarrow \text{Set}) \rightarrow (S \rightarrow \text{Set})$ defined by

- $F X(s) = \sum c : C(s). \prod r : R(s, c). X(n(s, c, r))$ for $X : S \rightarrow \text{Set}$ and
- for $f : \prod s : S. X(s) \rightarrow Y(s)$

$$F f(s) : F(X, s) \rightarrow F(Y, s) ,$$

$$F f(s)(c, g) = (c, \lambda r. f(n(s, c, r), g(r))) .$$

One easily sees that F is a Functor.

A final coalgebra in a category \mathbf{C} for an endofunctor $F : \mathbf{C} \rightarrow \mathbf{C}$ is an object F^∞ together with a morphism $\text{elim} : F^\infty \rightarrow F(F^\infty)$ s.t. for any object A and morphism $g : A \rightarrow FA$ there is exactly one morphism $f : A \rightarrow F^\infty$ making the following diagram commute:

$$\begin{array}{ccc} F^\infty & \xrightarrow{\text{elim}} & F(F^\infty) \\ \uparrow & & \uparrow \\ f \downarrow & & \downarrow Ff \\ A & \xrightarrow{g} & FA \end{array}$$

We will show in this section, that the previous defined functor has a final coalgebra. This is not surprising. However the proof gives a hint how to internalize the notions in Martin-Löf Type Theory. This will be done in a forthcoming paper. For simplicity, we argue in \mathbf{ZF} for the rest of this section. Essentially we just use induction on the natural numbers.

To get the final coalgebra we first define by induction sets $CT_0(s)$ and functions $\text{first}_S, \text{last}_S : CT_0(s) \rightarrow S$, $\text{first}_C, \text{last}_C : CT_0(s) \rightarrow C(s)$ and $\text{length} : CT_0(s) \rightarrow N$ for $s : S$. In this section $*$ denotes the concatenation of two lists.

Definition 1 $CT_0(s)$ has as elements lists

$$(s_0, c_0, r_1, \dots, r_n, s_n, c_n)$$

for $0 \leq n$ with $s_0 = s$, $c_i \in C(s_i)$, $r_{i+1} \in R(s_i, c_i)$ and $s_{i+1} = n(s_i, c_i, r_{i+1})$, and we define

$$\begin{aligned} \text{length}((s, c)) &:= 1 \\ \text{length}(l' * (r, s, c)) &:= \text{length}(l') + 1 \\ \text{first}_S((s_0, c_0, r_1, \dots, r_n, s_n, c_n)) &:= s_0 \\ \text{last}_S((s_0, c_0, r_1, \dots, r_n, s_n, c_n)) &:= s_n \\ \text{first}_C((s_0, c_0, r_1, \dots, r_n, s_n, c_n)) &:= c_0 \\ \text{last}_C((s_0, c_0, r_1, \dots, r_n, s_n, c_n)) &:= c_n \\ \text{pd}((s, c)) &:= (s, c) \\ \text{pd}(l' * (r, s, c)) &:= l' \end{aligned}$$

We write $R(l)$ for $R(\text{last}_S(l), \text{last}_C(l))$, $l \in CT_0(s)$, $n(l, r)$ for $n(\text{last}_S(l), \text{last}_C(l), r)$, $r \in R(l)$. We are now able to define the domain of the final coalgebra:

Definition 2 For $s \in S$ and $T \subseteq CT_0(s)$ let

$$\varphi(T, s) := \exists! c \in C(s). (s, c) \in T \ \& \tag{1}$$

$$\forall l \in T. \forall r \in R(l). \exists! c \in C(n(l, r)).$$

$$l * (r, n(l, r), c) \in T \ \& \tag{2}$$

$$\forall l \in T. \text{pd}(l) \in T \tag{3}$$

We define for $s \in S$:

$$CT(s) := \{T \subseteq CT_0(s) \mid \varphi(T, s)\}.$$

$CT : S \rightarrow \text{Set}$. That means CT is an object of $S \rightarrow \text{Set}$. We can interpret the elements of $CT(s)$ as computation trees for a program $p : IO(s)$. Part (1) of $\varphi(T, s)$ says that there is exactly one root (s, c) in each $T \in CT(s)$. Part (2) of $\varphi(T, s)$ ensures that for $l \in T$ and every $r \in R(l)$ there is exactly one successor $l * (r, s', c')$ in T and part (3) of $\varphi(T, s)$ says that T is closed against predecessors. Note that

$$\text{first}_C(l) = \text{first}_C(l') \text{ for } l, l' \in T \in CT(s) .$$

Sets $T, T' \in CT(s)$ have a nice property:

Lemma 1 For $T, T' \in CT(s)$

$$T \subseteq T' \Leftrightarrow T = T' .$$

Proof: Induction on $\text{length}(l)$. Let $l \in T'$.

If $l = (s', c')$, then $s' = \text{first}_S(l) = s$ because $T' \subseteq CT_0(s)$.

By Definition of $CT(s)$ there is exactly one $c \in C(s)$ with $(s, c) \in T \subseteq T'$. Again by Definition of $CT(s)$ follows $c = c'$ and therefore $l = (s, c) \in T$.

If $l = l' * (r', s', c')$, then $l' = \text{pd}(l) \in T'$.

By I.H. is $l' \in T$. Since $l' \in T \subseteq CT_0(s)$ is $r' \in R(l')$, $s' = n(l', r')$ and $c' \in C(s')$.

By Definition of $CT(s)$ there is again exactly one $c'' \in C(s')$ with $l' * (r', s', c'') \in T \subseteq T'$. $T' \in CT(s)$ implies $c' = c''$ and so $l \in T$. \square

Definition 3 For $T \in CT(s)$ let

$$\text{elim}(s, T) = (c, h) ,$$

where for some $l \in T$

$$\begin{aligned} c &= \text{first}_C(l) \\ h &: \prod r : R(s, c) \rightarrow C(n(s, c, r)) \\ h(r) &= \{l \in CT_0(n(s, c, r)) \mid (s, c, r) * l \in T\}. \end{aligned}$$

The equations define a morphism $\text{elim} : CT \rightarrow F(CT)$. $h(r)$ gives us the subtree of T on position r .

Theorem 1 The previous defined Functor $F : (S \rightarrow \text{Set}) \rightarrow (S \rightarrow \text{Set})$ has a final coalgebra in the category $S \rightarrow \text{Set}$.

Proof: We claim that (CT, elim) is a final coalgebra for F .

Let $g(s) : A(s) \rightarrow FA(s)$ for $s : S$. We write $g = (g_0, g_1)$, where $g_0(s) = \pi_0(g(s)) \in C(s)$, and $g_1(s) = \pi_1(g(s)) \in \prod r : R(s, g_0(s)).A(n(s, g_0(s), r))$.

We have to show that there is a unique morphism $T : A \rightarrow CT$ such that the diagram on page 4 with $F^\infty = CT$ and $f = T$ commutes.

For this purpose, we define simultaneously sets $T(s, a) \in CT(s)$ for $s \in S, a \in A(s)$ and elements $\text{next}_S(l, r) \in S, \text{next}_A(l, r) \in A(\text{next}_S(l, r))$ for $l \in T(s, a), r \in R(l)$ by

$$\begin{aligned} T^0(s, a) &:= \{(s, g_0(s, a))\} \\ \text{next}_S((s, g_0(s, a)), r) &:= n(s, g_0(s, a), r) \\ \text{next}_A((s, g_0(s, a)), r) &:= g_1(s, a, r) \\ \\ T^{i+1}(s, a) &:= \{l * (r, s', c') \mid l \in T^i(s, a) \\ &\quad \& r \in R(l) \& s' = n(l, r) \\ &\quad \& c' = g_0(s', \text{next}_A(l, r))\} \\ \text{next}_S(l * (r, s', c'), r') &:= n(s', c', r') \\ \text{next}_A(l * (r, s', c'), r') &:= g_1(s', \text{next}_A(l, r), r') \\ \\ T(s, a) &:= \bigcup_{i \in \mathbb{N}} T^i(s, a) \end{aligned}$$

We show by induction on i that

$$T^i(n(s, c, r), g_1(s, a, r)) = \{l \in CT_0(n(s, c, r)) | (s, c, r) * l \in T^{i+1}(s, a)\} \quad (*)$$

for $i \in N, s \in S, a \in A(s), c = g_0(s, a), r \in R(s, c)$:

Let $A_i := T^i(n(s, c, r), g_1(s, a, r))$ and

$B_i := \{l \in CT^0(n(s, c, r)) | (s, c, r) * l \in T^{i+1}(s, a)\}$.

$$\begin{aligned} (s', c') \in A_0 &\Rightarrow s' = n(s, c, r) \ \& \\ &c' = g_0(n(s, c, r), g_1(s, a, r)) \\ &= g_0(s', \text{next}_A((s, c), r)) \\ &\Rightarrow (s, c, r) * (s', c') \in T^1(s, a) \\ &\Rightarrow (s', c') \in B_0 \end{aligned}$$

$$\begin{aligned} l * (r', s', c') \in A_{i+1} &\Rightarrow l \in A_i \subseteq B_i \ \& \ r' \in R(l) \ \& \\ &s' = n(l, r') \ \& \\ &c' = g_0(s', \text{next}_A(l, r')) \\ &\Rightarrow (s, c, r) * l * (r', s', c') \\ &\in T^{i+2}(s, a) \\ &\Rightarrow l * (r', s', c') \in B_{i+1} \end{aligned}$$

It follows easily by induction on i that $T(s, a) \in CT(s)$ for $s \in S, a \in A(s)$.

$T : A \rightarrow CT$ makes the diagram commute:

$$\begin{aligned} \pi_0(\text{elim}(s, T(s, a))) &= g_0(s, a) =: c \\ \pi_1(\text{elim}(s, T(s, a)))(r) &= \{l \in CT_0(n(s, c, r)) | (s, c, r) * l \in T(s, a)\} \\ &= T(n(s, c, r), g_1(s, a, r)) \ , \end{aligned}$$

where the last equation follows by (*).

It remains to show that T is unique. Let $T' : A \rightarrow CT$ a morphism making the diagram commute. We show $T^i(s, a) \subseteq T'(s, a)$ for all $i \in N$ by induction:

$i = 0$: We have

$$\pi_0(\text{elim}(s, T'(s, a))) = \pi_0(g(s, a)) = g_0(s, a) \ ,$$

and so $T^0(s, a) \subseteq T'(s, a)$.

Let $T^i(s, a) \subseteq T'(s, a)$ for all $s \in S, a \in A(s)$ and $(s, c, r) * l \in T^{i+1}(s, a)$. Then

$$c = g_0(s, a) = \pi_0(\text{elim}(s, T'(s, a))) \ ,$$

and

$$\begin{aligned} l &\in T^i(n(s, c, r), g_1(s, a, r)) \\ &\subseteq T'(n(s, c, r), g_1(s, a, r)) \\ &= \pi_1(\text{elim}(s, T'(s, a)))(r) \\ &= \{l \in CT_0(n(s, c, r)) | (s, c, r) * l \in T'(s, a)\} \ , \end{aligned}$$

and therefore $(s, c, r) * l \in T'(s, a)$.

By the previous Lemma follows the claim. \square

4 Rules for IO-programs

Let $\text{elim} : \prod s : S. IO(s) \rightarrow F(IO)(s)$ be a final coalgebra for F in the category $S \rightarrow \text{Set}$. We can now define $c : \prod s : S. IO(s) \rightarrow C(s)$ and $\text{next} : \prod s : S. \prod c : IO(s). \prod r : R(s, c(s, p)). IO(n(s, c(s, p), r))$ by

$$\begin{aligned} c(s, p) &= \pi_0(\text{elim}(s, p)) \\ \text{next}(s, r) &= \pi_1(\text{elim}(s, p))(r). \end{aligned}$$

We enrich our type theory by the following rules:

Formation Rule

$$\frac{S : \text{Set} \quad s : S}{IO(s) : \text{Set}}$$

Elimination Rule

$$\frac{S : \text{Set} \quad s : S \quad p : IO(s)}{\text{elim}(s, p) : \underbrace{\Sigma c : C(s). \prod r : R(s, c). IO(n(s, c, r))}_{F(IO, s)}}$$

Introduction Rule

$$\frac{\begin{array}{l} S : \text{Set} \\ A : S \rightarrow \text{Set} \\ g : \prod s : S. A(s) \rightarrow F(A, s) \end{array}}{\mu(A, g) : \prod s : S. A(s) \rightarrow IO(s)}$$

Equality Rule

$$\frac{\begin{array}{l} S : \text{Set} \\ A : S \rightarrow \text{Set} \\ g : \prod s : S. A(s) \rightarrow F(A, s) \\ s : S \\ a : A(s) \end{array}}{\text{elim}(s, \mu(A, g)(s, a)) = \text{onestep}(g(s, a)) : F(IO, s)}$$

where

$$\text{onestep}((c, h)) = (c, \lambda r. \mu(A, g)(n(s, c, r), h(r))) .$$

Furthermore, we define

$$\begin{aligned} \sim & : (n : \mathbb{N}, S : \text{Set}, s : S, p, q : IO(s)) \rightarrow \text{Set} \\ \approx & := (S : \text{Set}, s : S, p, q : IO(s)) \rightarrow \text{Set} \end{aligned}$$

by the following equations:

$$\begin{aligned} p \sim_0 q & := \top \\ p \sim_{n+1} q & := \text{Id}(C(s), c, c') \\ & \quad \wedge \forall r \in R(s, c). \\ & \quad \pi_1(\text{elim}(s, p))(r) \sim_n \pi_1(\text{elim}(s, q))(r) \\ p \approx q & : \forall n \in \mathbb{N}. p \sim_n q , \end{aligned}$$

where

$$\begin{aligned} c & := \pi_0(\text{elim}(s, p)) \\ c' & := \pi_0(\text{elim}(s, q)) . \end{aligned}$$

Note that the introduction rule for the IO-Sets looks more complicated than the elimination rule. Like for inductive defined sets the introduction rule says what our canonical

elements are. However, whereas for inductive sets in the premises of the introduction rule only appear certain sets here we can have any family of sets to introduce a new element in $IO(s)$. Otherwise the elimination rules say how to define a function on these sets. However, whereas for inductive sets the range can be any set here it is the fixed set $\Sigma c : C(s). \prod r : R(s, c). IO(n(s, c, r))$.

5 Examples

Console I/O can be seen as state dependent IO.

- The states are $n : \mathbb{N}$ representing the number of characters written on console.
- The commands $C(n)$ are
 - $C(0) = \{\text{readchar}, \text{writechar}(n : \text{OutputChar})\}$ If we have not written anything, we can read a character from keyboard, or write a character to the console.
 - $C(Sn) = C(0) \cup \{\text{delete}\}$.
If we have written something, we can additionally delete the last character.
- The response sets are
 - $R(n, \text{readchar}) = \text{InputChar}$,
 - $R(n, \text{writechar } n) = \{0k\}$,
 - $R(Sn, \text{delete}) = \{0k\}$.

If we read a character, we obtain the character read. Otherwise we obtain a confirmation that the action was carried out.

- $n(n, \text{readchar}, c) = n$,
- $n(n, \text{writechar } n, 0k) = Sn$,
- $n(Sn, \text{delete}, 0k) = n$.

If we read a character, we do not do anything (the idea is that characters not automatically reflected on the console input). If we write a character, then the length of the output increases by one. If we delete the last character, the last character is deleted.

5.1 Example Programs

- ```
1 : readchar
 goto 1
```

$$A(s) = \{1\}, g(s, l) = (\text{readchar}, \lambda n.1),$$

$$p = \mu(A, g, (), 1).$$
- Assume  $\text{InputChar} = \text{OutputChar}$ 

```
char c
l0: readchar(c)
l1: writechar(c)
 goto l0
```

$$A(n) = \{l_0, l_1(k : N)\},$$

$$g(n, l_0) = (\text{readchar}, \lambda k.l_1(n)), g(n, l_1(n)) = (\text{writechar } n, \lambda_.l_0), p = \mu(A, g, (), l_0).$$
- Assume  $\text{InputChar} = \text{Char} \cup \{\text{delete}\}$ ,  $\text{OutputChar} = \text{Char} \cup \{\text{beep}\}$ . `delete` stands for the delete button, and `beep` means to signal a beep.

```

lengthOfInput : N
10 : readchar (c)
11 : if c == delete then{
 if lengthOfInput == 0 then {
 beep
 goto 10}
 else {
 backspace
 goto 10}
 else{
 writechar c
 goto 10}
A(0) = {l0, l1, l2},
A(S n) = {l0, l1(c : InputChar), l3},
g(n, l0) = (readchar, λc.l1 c), g(0, l1 delete) = (beep, λ_.l0), g(S n, l1 delete) =
(delete, λ_.l0), and for x ≠ delete, g(n, l1 x) = (writechar n, λ_.l0), p =
μ(A, g, (), l0).

```

## 5.2 Railway System

A railway control system is a state dependent interactive system.

- The states encode the segments of the railway system, which are blocked.
- Commands allow us to change the state of signals, but only in such a way that a green signal does not grant access to a blocked segment.
- In response to such a command, we get information about trains entering and leaving blocks.
- The next state is obtained from the response set.
- Any program written for this interface fulfils the safety requirement that one never sets a signal leading into a blocked segment to green.

## 6 IO Programs with Return Value

Until now the only way to terminate for our programs is that  $R(s, c)$  is empty for some  $s, c$ . If a program reaches this situation, there is never any response, and the program is locked up. We want our programs to terminate and to give back some value, which we can see as value for the function calculated by the program. Therefore, we give our programs the ability to terminate in a state  $s$  with a certain value  $a$  from a set  $A(s)$ .

For  $X : \prod s : S. \text{Set}$  and  $A : \prod s : S. \text{Set}$ , let  $F_A(X, s)$  be

$$A(s) + \sum c : C(s). \prod r : R(s, c). X(n(s, c, r)) .$$

**Formation Rule**

$$\frac{S : \text{Set} \quad s : S \quad A : S \rightarrow \text{Set}}{IO_A(s) : \text{Set}}$$

**Elimination Rule**

$$\frac{S : \text{Set} \quad s : S \quad p : IO_A(s)}{\text{elim}_A(s, p) : F_A(IO_A, s)}$$

**Introduction Rule**

$$\frac{\begin{array}{c} S : \text{Set} \\ A, B : S \rightarrow \text{Set} \\ g : \prod s : S. B(s) \rightarrow F_A(B, s) \end{array}}{\mu(B, g) : \prod s : S. B(s) \rightarrow IO_A(s)}$$

**Equality Rule**

$$\begin{array}{c}
S : \text{Set} \\
A, B : S \rightarrow \text{Set} \\
g : \prod s : S. B(s) \rightarrow F_A(B, s) \\
s : S \\
b : B(s) \\
\hline
\text{elim}_A(s, \mu(B, g)(s, b)) = \text{onestep}(g(s, b)) : F_A(IO_A, s)
\end{array}$$

where

$$\begin{aligned}
&\text{onestep}(\text{inl } a) = \text{inl } a \text{ ,} \\
&\text{onestep}(\text{inr } (c, h)) = \text{inr } (c, \lambda r. \mu(B, g)(n(s, c, r), h(r))) \text{ .}
\end{aligned}$$

Furthermore, we define

$$\begin{aligned}
\sim & : (n : N, S : \text{Set}, s : S, p, q : IO(s)) \rightarrow \text{Set} \\
\approx & : (S : \text{Set}, s : S, p, q : IO(s)) \rightarrow \text{Set}
\end{aligned}$$

by the equations

$$\begin{aligned}
p \sim_0 q & := \top \\
p \sim_{n+1} q & := \text{Case elim}(s, p) \text{ of} \\
& \text{inl } a \quad : \text{Case elim}(s, q) \text{ of} \\
& \quad \text{inl } b \quad : \text{ld}(A(s), a, b) \\
& \quad \text{inr } (c', h') : \perp \\
& \text{inr } (c, h) : \text{Case elim}(s, q) \text{ of} \\
& \quad \text{inl } b \quad : \perp \\
& \quad \text{inr } (c', h') : \text{ld}(C(s), c, c') \wedge \forall r \in R(s, c). h(r) \sim_n h'(r) \\
p \approx q & = \forall n \in N. p \sim_n q
\end{aligned}$$

We also write  $\text{coit}_g$  for  $\mu(A, g)$ ,  $p \rightsquigarrow a$  for  $\text{elim}(s, p) = \text{inl } a$ ,  $p \rightsquigarrow (c, h)$  for  $\text{elim}(s, p) = \text{inr } (c, h)$ , and sometimes omit indices and superscripts.

Note that  $\approx$  gives us bisimulation since our programs are image finite processes in terms of process algebra: if  $p \rightsquigarrow (c, h)$ , then  $p \xrightarrow{r} q$  exactly if  $q = h(r)$ .

The concept of a monad also originates from category theory, and generalises the notion of a monoid (see e.g. [1]). It plays an important role in functional programming (e.g. [12]). We are going to define a monad operation

$$*_s : IO_A(s) \rightarrow \left( \prod s : S. A(s) \rightarrow IO_B(s) \right) \rightarrow IO_B(s)$$

Assume  $p : IO_A(s)$  and  $q : \prod s : S. A(s) \rightarrow IO_B(s)$ , then (we suppress  $s$ )  $p * q : IO_B(s)$  is the program, which runs as  $p$ , until it terminates with a value  $a : A(s')$ , and then continues as  $q(a) : IO_B(s')$ . We start by defining a canonical translations  $\text{can}_1$ :

**Definition 4** Let  $X, Y, A : \prod s : S. \text{Set}$ .

$\text{can}_1(s) : F_A(X, s) \rightarrow F_A(X + Y, s)$  be given by

$$\text{can}_1 = F_A(\text{inl}) \text{ ,}$$

$$\begin{aligned}
i.e. \quad \text{can}_1(s, \text{inl } a) & = \text{inl } a \\
\text{can}_1(s, \text{inr } (c, h)) & = \text{inr } (c, \lambda r. \text{inl } h(r))
\end{aligned}$$

In category theory, if  $\text{elim} : F^\infty \rightarrow F F^\infty$  is a final coalgebra, then exists for every  $f : A \rightarrow F(F^\infty + A)$  a unique arrow  $\text{corec}_f$  such that the following diagram commutes:

$$\begin{array}{ccc}
F^\infty & \xrightarrow{\text{elim}} & F(F^\infty) \\
\uparrow & & \uparrow \\
\text{corec}_f \downarrow & & \downarrow F[\text{id}_{F^\infty}, \text{corec}_f] \\
A & \xrightarrow{f} & F(F^\infty + A)
\end{array}$$

This motivates the following definitions in type theory:

**Definition 5** For  $g : \prod s : S.A(s) \rightarrow C(s)$  and  $h : \prod s : S.B(s) \rightarrow C(s)$  we define

$$[g, h] : \prod s : S.(A(s) + B(s)) \rightarrow C(s)$$

by

$$[g, h](s, o) := [g(s), h(s)](o) := \text{when}(o, g(s), h(s)) .$$

For  $f : \prod s : S.A(s) \rightarrow F_B(IO_B + A, s)$  let

$$\begin{aligned} \overline{\text{coit}}_f &:= \text{coit}_{[\text{can}_1 \circ \text{elim}, f]} \\ &= \mu(IO_B + A, [\text{can}_1 \circ \text{elim}, f]) \\ &\in \prod s : S.(IO_B(s) + A(s)) \rightarrow IO_B(s) , \end{aligned}$$

and  $\text{corec}_f : \prod s : S.A(s) \rightarrow IO_B(s)$  with

$$\text{corec}_f(s, p) = \overline{\text{coit}}_f(s, \text{inr } p) .$$

**Definition 6** For  $q : \prod s : S.A(s) \rightarrow IO_B(s)$  let  $q^* : \prod s : S.IO_A(s) \rightarrow F_B(IO_B + IO_A, s)$  be defined by

$$\begin{aligned} q^*(s, p) &= \text{Case elim}(s, p) \text{ of} \\ &\quad \text{inl } a \quad : \text{can}_1(s, \text{elim}(s, q(s, a))) \\ &\quad \text{inr } (c, h) \quad : \text{inr } (c, \lambda r. \text{inr } h(r)) \end{aligned}$$

We define now  $*$  :  $IO_A(s) \rightarrow (\prod s : S.A(s) \rightarrow IO_B(s)) \rightarrow IO_B(s)$  by

$$p * q := *(p, q) := \text{corec}_{q^*}(s, p) ,$$

and

$$\eta_A := \text{coit}_{\check{\eta}} : \prod s : S.A(s) \rightarrow IO_A(s) ,$$

where  $\check{\eta} : \prod s : S.A(s) \rightarrow F_A(A, s)$  with  $\check{\eta}(s, a) = \text{inl } a$ .

If  $h : \prod r : R(s, c).IO_A(n(s, c, r))$  and  $q(s) : A(s) \rightarrow IO_B(s)$  for  $s : S$ , define

$$h * q = \lambda r. h(r) * q : \prod r : R(s, c).IO_B(n(s, c, r)) .$$

**Lemma 2** Let  $o : IO_{A_1}(s)$ ,  $p(s) : A_0(s) \rightarrow IO_{A_1}(s)$  for  $s : S$ . Then

$$\overline{\text{coit}}_{p^*}(s, \text{inl } o) \approx o .$$

Proof: Let  $\bar{p} = \overline{\text{coit}}_{p^*}$ . We show  $\bar{p}(s, \text{inl } o) \sim_n o$  by induction on  $n$ .

We have  $\bar{p}(s, \text{inl } o) \sim_0 o$ . Assume  $\bar{p}(s, \text{inl } o) \sim_n o$  for all  $o$ .

First case:  $\text{elim}(s, o) = \text{inl } a$ . Then we have

$$[\text{can}_1 \circ \text{elim}, p^*](s, \text{inl } o) = \text{can}_1(s, \text{elim}(s, o)) = \text{inl } a ,$$

and so by equality

$$\text{elim}(s, \bar{p}(s, \text{inl } o)) = \text{inl } a = \text{elim}(s, o) .$$

Therefore,  $\bar{p}(s, \text{inl } o) \sim_{n+1} o$ .

Second case:  $\text{elim}(s, o) = \text{inr } (c, h)$ . Then we have

$$\begin{aligned} [\text{can}_1 \circ \text{elim}, p^*](s, \text{inl } o) &= \text{can}_1(s, \text{elim}(s, o)) \\ &= \text{inr } (c, \lambda r. \text{inl } h(r)) , \end{aligned}$$

so by equality

$$\text{elim}(s, \bar{p}(s, \text{inl } o)) = \text{inr } (c, \lambda r. \text{inl } \bar{p}(n(s, c, r), \text{inl } h(r))) .$$

Then by I.H.  $\bar{p}(s, \text{inl } h(r)) \sim_n h(r)$  and the claim.  $\square$

We are now able to prove the first monad law:

**Theorem 2** Let  $p : IO_A(s)$  and  $q : \prod s : S.A(s) \rightarrow IO_B(s)$ . Then by  $\text{elim}_A(s, p) = \text{inl } a$  follows

$$p * q \approx q(s, a) .$$

Proof: I.  $\text{elim}_B(s, q(s, a)) = \text{inl } b$ . Then we get  $\text{can}_1(s, \text{elim}(s, q(s, a))) = \text{inl } b$ , and therefore by  $\text{elim}_A(s, p) = \text{inl } a$

$$[\text{can}_1 \circ \text{elim}, q^*](s, \text{inr } p) = q^*(s, p) = \text{inl } b .$$

And by the equality rule

$$\text{elim}_B(s, \underbrace{\overline{\text{coit}}_{q^*}(s, \text{inr } p)}_{=p*q}) = \text{inl } b = \text{elim}_B(s, q(s, a)) .$$

II.  $\text{elim}_B(s, q(s, a)) = \text{inr } (c, h)$ . Then we get  $\text{can}_1(s, \text{elim}(s, q(s, a))) = \text{inr } (c, \lambda r. \text{inl } h(r))$ , and therefore by  $\text{elim}_A(s, p) = \text{inl } a$

$$[\text{can}_1 \circ \text{elim}, q^*](s, \text{inr } p) = q^*(s, p) = \text{inr } (c, \lambda r. \text{inl } h(r)) .$$

By the equality rule,

$$\text{elim}_B(s, \underbrace{\overline{\text{coit}}_{q^*}(s, \text{inr } p)}_{=p*q}) = \text{inr } (c, \lambda r. \overline{\text{coit}}_{q^*}(n(s, c, r), \text{inl } h(r))) .$$

By Lemma 2 follows

$$h(r) \approx \overline{\text{coit}}_{q^*}(n(s, c, r), \text{inl } h(r))$$

for  $r : R(s, c)$ , and therefore  $p * q \approx q(s, a)$  . □

**Corollary 1** If  $q : \prod s : S.A(s) \rightarrow IO_B(s)$ , then

$$\eta(s, a) * q \approx q(s, a) .$$

Unless otherwise noted, let

$$\begin{aligned} o & : IO_{A_0}(s) \\ p(s) & : A_0(s) \rightarrow IO_{A_1}(s) \\ q(s) & : A_1(s) \rightarrow IO_{A_2}(s) \\ \bar{p} & = \overline{\text{coit}}_{p^*} \end{aligned}$$

for  $s : S$  for the rest of the article.

**Lemma 3** If  $o \rightsquigarrow (c, h)$ , then

$$o * p \rightsquigarrow (c, h * p) .$$

Proof: By  $\text{elim}(s, o) = \text{inr } (c, h)$  follows  $p^*(s, o) = \text{inr } (c, \lambda r. \text{inr } h(r))$ . By equality, we get

$$\text{elim}(s, \bar{p}(s, \text{inr } o)) = \text{inr}(c, \lambda r. \bar{p}(n(s, c, r), \text{inr } h(r))) .$$

We have  $o * p = \text{corec}_{p^*}(s, o) = \bar{p}(s, \text{inr } o)$ , and

$$h(r) * p = \text{corec}_{p^*}(n(s, c, r), h(r)) = \bar{p}(n(s, c, r), \text{inr } h(r))$$

for  $r : R(s, c)$ . Therefore,  $\text{elim}(s, o * p) = \text{inr}(c, \lambda r. h(r) * p)$  . □

**Theorem 3** If  $p : IO_A(s)$ , then  $p * \eta \approx p$  .

Proof: We show  $p * \eta \sim_n p$  by induction on  $n$ .

I.  $\text{elim}_A(s, p) = \text{inl } a$ . Then we have

$$\text{elim}_A(s, p * \eta) = \text{elim}_A(s, \eta(s, a)) = \text{inl } a = \text{elim}_A(s, p)$$

II.  $\text{elim}_A(s, p) = \text{inr } (c, h)$ . By Lemma 3 we get

$$\text{elim}_A(s, p * \eta) = \text{inr } (c, \lambda r. h(r) * \eta) ,$$

and by I.H. follows the claim. □

**Lemma 4** *If  $o \rightsquigarrow (c, h)$ , then  $(o * p) * q \rightsquigarrow (c, (h * p) * q)$  .*

Proof: By Lemma 3. □

**Lemma 5** *If  $o \rightsquigarrow (c, h)$ , then  $o * (\lambda s, a.p(s, a) * q) \rightsquigarrow (c, h * (\lambda s, a.p(s, a) * q))$  .*

Proof: By Lemma 3. □

**Lemma 6** *If  $o \rightsquigarrow a_0$  and  $p(s, a_0) \rightsquigarrow a_1$ , then  $o * p \rightsquigarrow a_1$  .*

Proof: By  $\text{elim}(s, o) = \text{inl } a_0$  follows

$$p^*(s, a_0) = \text{can}_1(s, \text{elim}(s, p(s, a_0))) = \text{can}_1(s, \text{inl } a_1) = \text{inl } a_1.$$

Therefore,  $[\text{can}_1 \circ \text{elim}, p^*](s, \text{inr } o) = p^*(s, a_0) = \text{inl } a_1$  .

By equality we get

$$\text{elim}(s, \bar{p}(s, \text{inr } o)) = \text{inl } a_1 .$$

$o * p = \text{corec}_{p^*}(s, o) = \bar{p}(s, \text{inr } o)$ , and therefore  $\text{elim}(s, o * p) = \text{inl } a_1$  . □

**Lemma 7** *If  $o \rightsquigarrow a$  and  $p(s, a) \rightsquigarrow (c, h)$ , then  $o * p \rightsquigarrow (c, h')$  .*

*with  $h'(r) = \bar{p}(n(s, c, r), \text{inl } h(r))$ .*

Proof: By  $\text{elim}(s, o) = \text{inl } a$  follows

$$p^*(s, o) = \text{can}_1(s, \text{elim}(s, p(s, a))) = \text{can}_1(s, \text{inr } (c, h)) = \text{inr } (c, \lambda r. \text{inl } h(r)).$$

Therefore,  $[\text{can}_1 \circ \text{elim}, p^*](s, \text{inr } o) = p^*(s, a) = \text{inr } (c, \lambda r. \text{inl } h(r))$  .

By equality we get  $\text{elim}(s, o * p) = \text{elim}(s, \bar{p}(s, \text{inr } o)) = \text{inr } (c, \lambda r. \bar{p}(n(s, c, r), \text{inl } h(r)))$ .

□

**Lemma 8** *Let  $o' = \bar{p}(s, \text{inl } o)$ . Then  $o * q \approx_n o' * q$  .*

Proof: We show  $o * q \sim_n o' * q$  by induction on  $n$ .

First case:  $\text{elim}(s, o) = \text{inl } a$

First subcase:  $\text{elim}(s, g(s, a)) = \text{inl } b$ . We have  $[\text{can}_1 \circ \text{elim}, p^*](s, \text{inl } o) = \text{inl } a$  , and therefore  $\text{elim}(s, \bar{p}(s, \text{inl } o)) = \text{inl } a$  .

By Lemma 6 we get  $\text{elim}(s, o' * q) = \text{inl } b = \text{elim}(s, o * q)$  .

Second subcase:  $\text{elim}(s, g(s, a)) = \text{inr } (c, h)$

By Lemma 7 we get  $\text{elim}(s, o * q) = \text{inr } (c, h') = \text{elim}(s, o' * q)$  , where  $h'(r) = \bar{p}(n(s, c, r), \text{inl } h(r))$ .

Second case:  $\text{elim}(s, o) = \text{inr } (c, h)$ .

By Lemma 3 we get

$$\text{elim}(s, o * q) = \text{inr } (c, \lambda r. h(r) * q) .$$

We have  $[\text{can}_1 \circ \text{elim}, p^*](s, \text{inl } o) = \text{inr } (c, \lambda r. \text{inl } h(r))$  , and therefore  $\text{elim}(s, o') = \text{inr } (c, h')$  , where  $h'(r) = \bar{p}(n(s, c, r), \text{inl } h(r))$ .

By Lemma 3 we get

$$\text{elim}(s, o' * q) = \text{inr } (c, \lambda r. h'(r) * q) ,$$

and by I.H. the claim. □

**Lemma 9** *If  $o \rightsquigarrow a$  and  $p(s, a) \rightsquigarrow (c, h)$ , then*

$$(o * p) * q \approx o * (\lambda s, a.p(s, a) * q) .$$

Proof: By Lemma 7 follows  $\text{elim}(s, o * p) = \text{inr}(c, h')$  , where  $h'(r) = \bar{p}(n(s, c, r), \text{inl } h(r))$ .  
By Lemma 3 follows

$$\text{elim}(s, (o * p) * q) = \text{inr}(c, \lambda r. h'(r) * q) .$$

By  $\text{elim}(s, p(s, a)) = \text{inr}(c, h)$  and Lemma 3 we get

$$\text{elim}(s, p(s, a) * q) = \text{inr}(c, \lambda r. h(r) * q) .$$

By Lemma 7 we get

$$\text{elim}(s, o * (\lambda s, a. p(s, a) * q)) = \text{inr}(c, \lambda r. h''(r)) ,$$

where  $h''(r) = \bar{f}(n(s, c, r), \text{inl } h(r) * q)$ ,  $\bar{f} = \overline{\text{coit}}_{f^*}$ ,  $f = \lambda s, a. p(s, a) * q$ . By Lemma 2 follows  $h''(r) \approx h(r) * q$  , and by Lemma 8  $h(r) * q \approx h'(r) * q$  .  $\square$

**Lemma 10** *If  $o \rightsquigarrow a_0$ ,  $p(s, a_0) \rightsquigarrow a_1$  and  $q(s, a_1) \rightsquigarrow (c, h)$ , then*

$$(o * p) * q \approx o * (\lambda s, a. p(s, a) * q) .$$

Proof: By Lemma 6 and Lemma 7 we get  $\text{elim}(s, o * p) = \text{inl } a_1$  ,  $\text{elim}(s, (o * p) * q) = \text{inr}(c, h')$  , where  $h'(r) = \bar{q}(n(s, c, r), \text{inl } h(r))$ .  
Furthermore, by Lemma 7

$$\text{elim}(s, p(s, a_0) * q) = \text{inr}(c, h') ,$$

and again

$$\text{elim}(s, o * (\lambda s, a. p(s, a) * q)) = \text{inr}(c, h'') ,$$

where  $h''(r) = \bar{f}(n(s, c, r), \text{inl } h'(r))$ ,  $\bar{f} = \overline{\text{coit}}_{f^*}$ ,  $f = \lambda s, a. p(s, a) * q$ .  
By Lemma 2 follows  $h''(r) \approx h'(r) \approx h(r)$  .  $\square$

**Theorem 4**

$$(o * p) * q \approx o * (\lambda s, a. p(s, a) * q) .$$

Proof: We show  $(o * p) * q \sim_n o * (\lambda s, a. p(s, a) * q)$  by induction on  $n$ .

Case I:  $\text{elim}(s, o) = \text{inr}(c, h)$ . Then by Lemma 4

$$\text{elim}(s, (o * p) * q) = \text{inr}(c, \lambda r. (h(r) * p) * q) ,$$

and by Lemma 5

$$\text{elim}(s, o * (\lambda s, a. p(s, a) * q)) = \text{inr}(c, \lambda r. h(r) * (\lambda s, a. p(s, a) * q)) .$$

By I.H. follows the claim.

Case II:  $\text{elim}(s, o) = \text{inl } a_0$  . This case follows by Lemmata 6, 10, and 9.  $\square$

## 7 Conclusion

We have introduced state dependent interactive programs in Martin-Löf type theory. We have given a model of the corresponding final coalgebras in set theory, and added corresponding rules introducing operations  $\text{IO} : (S \rightarrow \text{Set}) \rightarrow (S \rightarrow \text{Set})$  to Martin-Löf type theory. Using these rules we have introduced the bisimulation relation  $\approx$ , and operations  $*$ ,  $\eta$ , and have shown that  $(\text{IO}, *, \eta)$  is a state-dependent monad w.r.t.  $\approx$ .

## References

- [1] Andrea Asperti, Guisepppe Longo. *Categories, Types and Structures. An Introduction to Category Theory for the working computer scientist*. Foundations of Computing Series. M.I.T. Press, 1991.
- [2] C. D. Team. *The Coq proof assistant. reference manual*. Available from <http://coq.inria.fr/doc/main.html>, 2003.
- [3] Robert L. Constable et. al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [4] 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*, 2000. Available via <http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html>.
- [5] Peter Hancock, Anton Setzer. *Interactive programs in dependent type theory*. In: P. Clote, H. Schwichtenberg: *Computer Science Logic. 14th international workshop, CSL 2000*. Springer Lecture Notes in Computer Science, Vol. 1862, pp. 317 - 331, 2000.
- [6] Z. Luo. *Computation and reasoning*. Clarendon Press, Oxford, 1994.
- [7] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
- [8] Bengt Nordström, Kent Peterson, Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, Oxford, 1990.
- [9] Lawrence C. Paulson. *Natural Deduction Proof as Higher-Order Resolution*. technical report 82, University of Cambridge Computer Laboratory, Cambridge, 1985.
- [10] Kent Peterson. *A Programming System for Type Theory*. PMG Memo 21, Chalmers University of Technology, S-412 96 Göteborg, 1982.
- [11] R. Pollack. *The theory of LEGO. A proof checker for the extended calculus of constructions*. PhD thesis, LFCS, Edinburgh, 1994.
- [12] Philip Wadler. *The essence of functional programming*. In: 19'th Symposium on Principles of Programming Languages, Albuquerque, volume 19. ACM Press, January 1992.