Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 122 (2005) 127–146
www.elsevier.com/locate/entcs

State Dependent IO-Monads in Type Theory

Markus Michelbrink[1,2]

Department of Computer Science
University of Wales Swansea
Swansea, United Kingdom

Anton Setzer[3,4]

Department of Computer Science
University of Wales Swansea
Swansea, United Kingdom

Abstract

We introduce the notion of state dependent interactive programs for Martin-Löf Type Theory. These programs are elements of coalgebras of certain endofunctors on the presheaf category $S \to \mathsf{Set}$. 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. The correctness of the monad laws has been verified in the theorem prover Agda which is based on intensional type theory.

Keywords: Dependent type theory, Martin-Löf type theory, intensional type theory, monad, weakly final coalgebras, interactive programs.

[1] Supported by EPSRC grant GR/S30450/01.
[2] Email: m.michelbrink@swansea.ac.uk
[3] Supported by Nuffield Foundation, grant ref. NAL/00303/G and EPSRC grant GR/S30450/01.
[4] Email: a.g.setzer@swansea.ac.uk

doi:10.1016/j.entcs.2004.06.055

1 Introduction

Martin-Löf's type theory [9] 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 [4,13], Cornell [3], Cambridge [12], Edinburgh [8,14], 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 compute a result from its input. 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 [5,6]. We generalise the notion of interfaces (worlds) and IO-programs to state dependent interfaces and state dependent programs. In [6] a world is a pair (C, R), where $C : \mathsf{Set}$ and $R : C \to \mathsf{Set}$. $c : C$ 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 $IO\ A : \mathsf{Set}$ (we keep the world fixed) has constructors $\mathsf{leaf} : A \to IO\ A$ and $\mathsf{do} : (c : C, p : Rc \to IO\ A) \to IO\ A$. The program $\mathsf{leaf}\ a$ terminates and returns value a, whereas $\mathsf{do}\ (c, p)$ issues command 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 is better suited for real world applications. One of our key examples is a *window 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 $\mathbb{F}$ on the presheaf category $S \to \mathsf{Set}$. We show that this functor has a final coalgebra $\mathsf{elim} : IO \to \mathbb{F}(IO)$. We enrich type theory by rules for a weak version of this final coalgebra

(weak because we do not demand uniqueness of $\mu(\alpha)$, as seen below). The elimination rule corresponds to the morphism elim, the introduction rules to the requirement that there is a morphism $\mu(\alpha) : A \to IO$ for every coalgebra $\alpha : A \to \mathbb{F}A$, and the equality rule expresses that the associated diagram commutes. The formation rule simply reflects the fact that there is a coalgebra IO.

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 Section 6 are verified in Agda [4]. The code is available online at the URL http://www.cs.swan.ac.uk/~csmichel/. This is in contrast with the construction of non-well-founded sets by Ingrid Lindström [7], which is essentially the construction of a weakly final coalgebra. That construction makes heavily use of extensional equality.

Overview.

In Section 2 we motivate our basic definitions. In Section 3 we relate our basic ideas from Section 2 to an endofunctor on $S \to \mathsf{Set}$, and show that this functor has a final coalgebra. In Section 4 we introduce the new rules for IO-programs, and define bisimulation. In Section 5 we give some simple examples for IO-programs. In Section 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 Section 3 we work in a standard dependent type theory (e.g. [11]) with the usual formation, introduction, elimination, and equality rules, extended by our rules.

Acknowledgement

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.

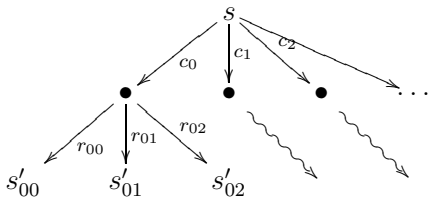
2 Interfaces, Programs

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

- $S : \mathsf{Set}$
- $C : S \to \mathsf{Set}$
- $R : \Pi s : S.C(s) \to \mathsf{Set}$
- $n : \Pi s : S.\Pi c : C(s).R(s, c) \to S$

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 a *window system* the *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, s') between states $s : S$ and $s' : S$ iff $c : C(s)$, $r : R(s, c)$ and $s' = n(s, c, r)$. The picture visualises a part of an interface:



There are two canonical ways to view an ordinary transition system as interface:

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

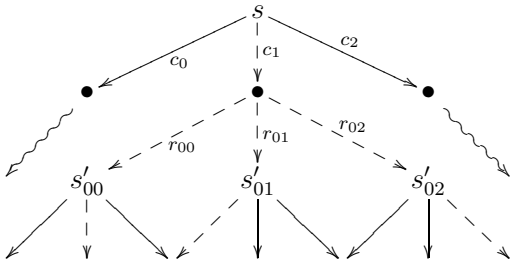
Let (S, C, R, n) be an interface. A *program* for this interface starting in state $s : S$ is a quadruple (A, c, next, a) s.t.

- $A : S \to \mathsf{Set}$
- $c : \Pi s : S.A(s) \to C(s)$
- $\mathsf{next} : \Pi s : S.\Pi a : A(s).\Pi r : R(s, c(s, a)).A(n(s, c(s, a), r))$
- $a : A(s)$

$A(s)$ is the set of programs starting in state s, $c(s, a)$ the command issued by the program $a : A(s)$, and $\mathsf{next}(s, a, r)$ 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 *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 $S \to \mathsf{Set}$ has objects $X : S \to \mathsf{Set}$ and morphism $f : \Pi s : S.X(s) \to Y(s)$, where $X, Y : S \to \mathsf{Set}$. The composition $g \circ f : \Pi s : S.X(s) \to Z(s)$ of two morphism $f : \Pi s : S.X(s) \to Y(s)$ and $g : \Pi s : S.Y(s) \to Z(s)$ is defined by

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

for $s : S$, $x : X(s)$. $\mathsf{id}_X : \Pi s : S.X(s) \to X(s)$ is given by $\mathsf{id}_X(s) = \mathsf{id}_{X(s)}$.

We look at the functor $\mathbb{F} : (S \to \mathsf{Set}) \to (S \to \mathsf{Set})$ defined by

- $\mathbb{F}X(s) = \Sigma c : C(s).\Pi r : R(s, c).X(n(s, c, r))$ for $X : S \to \mathsf{Set}$ and
- for $f : \Pi s : S.X(s) \to Y(s)$

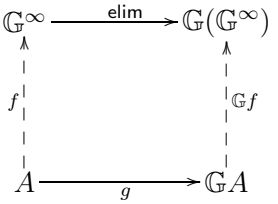
$$\mathbb{F}f(s) : \mathbb{F}X(s) \to \mathbb{F}Y(s),$$

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

One easily sees that $\mathbb{F}$ is a Functor.

A final coalgebra in a category $\mathbf{C}$ for an endofunctor $\mathbb{G} : \mathbf{C} \to \mathbf{C}$ is an object $\mathbb{G}^\infty$ together with a morphism $\mathsf{elim} : \mathbb{G}^\infty \to \mathbb{G}(\mathbb{G}^\infty)$ s.t. for any object A and morphism $g : A \to \mathbb{G}A$ there is exactly one morphism $f : A \to \mathbb{G}^\infty$

making the following diagram commute:



We will show in this section that the previous defined functor has a final coalgebra. This result can also be achieved by the observation that the functor is ω-continuous. So the final sequence of $\mathbb{F}$ stabilises at ω. The proof of ω-continuity makes heavily use of extensional equality. Therefore we prefer our proof, since it can be used for proofing the existence of weakly final coalgebras in Martin-Löf Type Theory. This will be done in a forthcoming paper. For simplicity, we argue in **ZF** for the rest of this section. To get the final coalgebra we first define by induction sets $CT_0(s)$ and functions $\mathsf{first}_S, \mathsf{last}_S : CT_0(s) \to S$, $\mathsf{first}_C, \mathsf{last}_C : CT_0(s) \to C(s)$, $\mathsf{length} : CT_0(s) \to N$ and $\mathsf{pd}(s)$ for $s : S$. In this section $*$ denotes the concatenation of two lists.

Definition 3.1 $CT_0(s)$ has as elements lists

$$(s_0, c_0, r_1, ..., r_n, s_n, c_n)$$

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

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

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

Definition 3.2 For $s \in S$ and $T \subseteq CT_0(s)$ let

(1) $\varphi(T,s) := (\exists! c \in C(s).(s,c) \in T)$
(2) $\quad \&(\forall l \in T. \forall r \in R(l). \exists! c \in C(n(l,r)). l * (r, n(l,r), c) \in T\ )$
(3) $\quad \&\forall l \in T. \mathsf{pd}(l) \in T$

We define $CT : S \to \mathsf{Set}$ by:

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

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 under predecessors. Note that

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

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

Lemma 3.3 *For $T, T' \in CT(s)$*

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

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

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

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

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

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

By the 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 $l \in T$. □

Definition 3.4 For $T \in CT(s)$ let

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

where for some $l \in T$

$$\begin{aligned} c &= \mathsf{first}_C(l) \\ h &: \Pi r : R(s,c) \to CT(n(s,c,r)) \\ h(r) &= \{l \in CT_0(n(s,c,r)) | (s,c,r) * l \in T\}. \end{aligned}$$

$h(r)$ is an element of $CT(n(s,c,r))$ and therefore the equations define a morphism $\mathsf{elim} : CT \to \mathbb{F}(CT)$. $h(r)$ gives us the subtree of T on position r.

Theorem 3.5 *The previous defined Functor* $\mathbb{F} : (S \to \mathsf{Set}) \to (S \to \mathsf{Set})$ *has a final coalgebra in the category* $S \to \mathsf{Set}$.

Proof. We claim that (CT, elim) is a final coalgebra for $\mathbb{F}$.

Let $g(s) : A(s) \to \mathbb{F}A(s)$ for $s : S$. We write $g = (g_0, g_1)$, where $g_0(s) = \pi_0(g(s)) \in C(s)$, and $g_1(s) = \pi_1(g(s)) \in \Pi r : R(s, g_0(s)).A(n(s, g_0(s), r))$.

We have to show that there is a unique morphism $T : A \to CT$ such that the diagram on page 6 with $\mathbb{G} = \mathbb{F}$, $\mathbb{G}^\infty = CT$ 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 $\mathsf{next}_S(l,r) \in S$, $\mathsf{next}_A(l,r) \in A(\mathsf{next}_S(l,r))$ for $l \in T(s,a), r \in R(l)$ by

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

It follows easily by induction on i that $T(s,a) \in CT(s)$ for $s \in S, a \in A(s)$ and 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 $i \in N$, $s \in S, a \in A(s), c = g_0(s,a), r \in R(s,c)$.

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

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

where the last equation follows by $(*)$.

It remains to show that T is unique. Let $T' : A \to CT$ be 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

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

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

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

and

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

and therefore $(s,c,r) * l \in T'(s,a)$.
The claim follows by lemma 3.3. □

4 Rules for IO-programs

We enrich our type theory by the following rules:

Formation Rule

$$\frac{S : \mathsf{Set} \qquad s : S}{IO(s) : \mathsf{Set}}$$

Elimination Rule

$$\frac{S : \mathsf{Set} \qquad s : S \qquad p : IO(s)}{\mathsf{elim}(s,p) : \underbrace{\Sigma c : C(s).\Pi r : R(s,c).IO(n(s,c,r))}_{\mathbb{F}(IO,s)}}$$

Introduction Rule

$$\frac{\begin{array}{c} S : \mathsf{Set} \\ A : S \to \mathsf{Set} \\ g : \Pi s : S.A(s) \to \mathbb{F}(A,s) \end{array}}{\mu(A,g) : \Pi s : S.A(s) \to IO(s)}$$

Equality Rule

$$\frac{\begin{array}{c} S : \mathsf{Set} \\ A : S \to \mathsf{Set} \\ g : \Pi s : S.A(s) \to \mathbb{F}(A,s) \\ s : S \\ a : A(s) \end{array}}{\mathsf{elim}(s, \mu(A,g)(s,a)) = \mathsf{onestep}(g(s,a)) : \mathbb{F}(IO,s)}$$

where

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

Furthermore, we define

$$\sim \ : \ (n : N, S : \mathsf{Set}, s : S, p, q : IO(s)) \to \mathsf{Set}$$
$$\approx := (S : \mathsf{Set}, s : S, p, q : IO(s)) \to \mathsf{Set}$$

by the following equations:

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

where

$$c := \pi_0(\mathsf{elim}(s,p)),\ c' := \pi_0(\mathsf{elim}(s,q)).$$

Note that the introduction rule for the IO-Sets looks more complicated than the elimination rule. As for inductively 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 $\Sigma c : C(s).\Pi r : R(s,c).IO(n(s,c,r))$. Note that we make use of extensional equality.

5 Examples

5.1 Mini Editor

We define two versions of a mini editor. Their interface is not state-dependent, so we ignore the arguments referring to S and don't need to define $\mathrm{n}(c,r)$. Interface:

- $\mathrm{C} = \{\mathtt{getchar}, \mathtt{writechar}(c : \mathrm{Char}), \mathtt{beep}\}$,
- $\mathrm{R}(n, \mathtt{getchar}) = \mathrm{Char}$,
- $\mathrm{R}(n+1, \mathtt{writechar}) = \mathrm{R}(n+1, \mathtt{beep}) = \{\mathtt{Ok}\}$,

Our first editor reads one character after the other from the keyboard and writes it to the screen. In "Haskell"-like pseudocode:

```
InOut : IO
InOut = getChar >>= writeChar >> InOut
```

In our Type Theory the program is defined by:

$$A = \{*\} \cup \mathrm{Char},\ g(*) = (\mathtt{getchar}, \lambda c.c),\ g(c) = (\mathtt{writechar}, \lambda \_ . *)$$

$$\mathtt{InOut} = \mu(A, g, *).$$

The second editor has an additional state parameter, namely the number of characters read. It reads a character. If it is a normal character, it proceeds as before. If it is the backspace character, then it deletes the last character, if there is one character left. If there is no character left, it signals a beep.
Pseudocode:

```
DelInOut : Int -> IO
DelInOut n = do c=getChar
                if c=backspace then
                   do if n==0 then
                         do beep
                            DelInOut n
                      else
                         do writeChar c
                            DelInOut (n-1)
                else
                   do writeChar c
                      DelInOut (n+1)
```

Type Theory:

$$
\begin{aligned}
A &= \mathbb{N} \times (\{*\} \cup \mathrm{Char}),\\
g((n,*)) &= (\texttt{getchar}, \lambda c.(n,c)),\\
g((0,\texttt{backspace})) &= (\texttt{beep}, \lambda\_.(0,*)),\\
g((n+1,\texttt{backspace})) &= (\texttt{writechar}(\texttt{backspace}), \lambda\_.(n,*))\\
g((n,c)) &= (\texttt{writechar}(c), \lambda\_.(n+1,*)) \text{ for } c \neq \texttt{backspace}\\
\texttt{DelInOut} &= \mu(A,g,(0,*)).
\end{aligned}
$$

5.2 Window System

A simple windowing interface is defined as follows:

- $S = \mathbb{N}$. The state denotes the number of windows currently open.
- $\mathrm{C}(n) = \{\texttt{open}, \texttt{writestring}(k :: \mathbf{n})(s :: \texttt{String}), \texttt{getchar}(c :: \texttt{Char})\}$. `open` opens a new window. $\texttt{writestring}(k,s)$ writes the string s into the window with number $k :: \mathbf{n} = \{1,\ldots,n-1\}$ and `getchar` reads a character.
- $\mathrm{R}(n,\texttt{open}) = \mathrm{R}(n,\texttt{writestring}(n,s)) = \{\texttt{Ok}\}$,
- $\mathrm{R}(n,\texttt{getchar}) = \texttt{Char}$,
- $\mathrm{n}(n,\texttt{open},\texttt{Ok}) = n+1$,

- $\mathrm{n}(n, c, r) = n$ otherwise.

The following program opens a window and types into it the string written so far.
Pseudocode:

```
Win::N->String->IO
Win n s = do open
             c <- getchar
             writestring n (s ++ c)
             Win n+1 (s ++ c)
```

Type Theory:

$$\begin{aligned}
A(0) &= \{*_1\} \times \texttt{String},\\
A(n+1) &= \{*_1, *_2, *_3\} \times \texttt{String},\\
g(n, (*_1, s)) &= (\texttt{open}, \lambda_.(*_2, s)),\\
g(n+1, (*_2, s)) &= (\texttt{getchar}, \lambda c.(*_3, s +\!\!+ c)),\\
g(n+1, (*_3, s)) &= (\texttt{writestring}(n, s), \lambda_.(*_1, s))\\
Win &= \mu(A, g, (0, ""))
\end{aligned}$$

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 $X : \Pi s : S.\mathsf{Set}$ and $A : \Pi s : S.\mathsf{Set}$, let $\mathbb{F}_A(X, s)$ be

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

$\mathsf{inl}\, a$ means that the program terminates with value a. $\mathsf{inr}\,(c, h)$ corresponds to (c, h) from Section 4.

Formation Rule

$$\frac{S : \mathsf{Set} \qquad s : S \qquad A : S \to \mathsf{Set}}{IO_A(s) : \mathsf{Set}}$$

Elimination Rule

$$\frac{S : \mathsf{Set} \qquad s : S \qquad p : IO_A(s)}{\mathsf{elim}_A(s, p) : \mathbb{F}_A(IO_A, s)}$$

Introduction Rule

$$\frac{\begin{array}{c} S : \mathsf{Set} \\ A, B : S \to \mathsf{Set} \\ g : \Pi s : S.B(s) \to \mathbb{F}_A(B, s) \end{array}}{\mu(B, g) : \Pi s : S.B(s) \to IO_A(s)}$$

Equality Rule

$$\frac{\begin{array}{c} S : \mathsf{Set} \\ A, B : S \to \mathsf{Set} \\ g : \Pi s : S.B(s) \to \mathbb{F}_A(B, s) \\ s : S \\ b : B(s) \end{array}}{\mathsf{elim}_A(s, \mu(B, g)(s, b)) = \mathsf{onestep}(g(s, b)) : F_A(IO_A, s)}$$

where

$$\mathsf{onestep}(\mathsf{inl}\ a) = \mathsf{inl}\ a,$$
$$\mathsf{onestep}(\mathsf{inr}\ (c, h)) = \mathsf{inr}\ (c, \lambda r.\mu(B, g)(n(s, c, r), h(r))).$$

Furthermore, we define

$$\sim : (n : N, S : \mathsf{Set}, s : S, p, q : IO(s)) \to \mathsf{Set}$$
$$\approx : (S : \mathsf{Set}, s : S, p, q : IO(s)) \to \mathsf{Set}$$

by the equations

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

We also write coit_g for $\mu(A, g)$, $p \rightsquigarrow a$ for $\mathsf{elim}(s, p) = \mathsf{inl}\ a$, $p \rightsquigarrow (c, h)$ for $\mathsf{elim}(s, p) = \mathsf{inr}\ (c, h)$, and sometimes omit indices and superscripts [5].

Note that $\approx$ gives us bisimulation since our programs are image finite processes in terms of process algebra: For every p there is at most one q s.t. $p \xrightarrow{r} q$, namely $q = h(r)$, if $p \rightsquigarrow (c, h)$.

The concept of a monad plays an important role in functional programming (e.g. [15]). There a monad is a triple $(\mathsf{M}, \eta, *)$ consisting of a type constructor M and a pair of polymorphic functions

[5] In functional programming literature the operation elim is also called out and the operation μ is called unfold.

$$\eta : A \to \mathsf{M}A \qquad * : \mathsf{M}A \to (A \to \mathsf{M}B) \to \mathsf{M}B$$

satisfying the following laws:

$$\eta(a) * k = k(a) \qquad m * \eta = m \qquad m * (\lambda a.k(a) * h) = (m * k) * h$$

We are going to define a monad operation

$$*_s : IO_A(s) \to (\Pi s : S.A(s) \to IO_B(s)) \to IO_B(s)$$

and show the monad laws with respect to bisimulation [6]. This proof has been carried out in intensional Type Theory. Note that this result can also be obtained by reformulation of an result of Lawrence Moss [10] that gives a Kleisli triple for a parametric corecursion system [7]. We don't know whether this result can be achieved in intensional Type Theory. Assume $p : IO_A(s)$ and $q : \Pi s : S.A(s) \to IO_B(s)$, 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 $\mathsf{can_l}$:

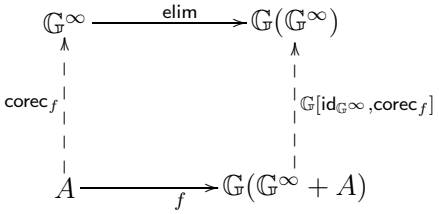
Definition 6.1 Let $X, Y, A : \Pi s : S.\mathsf{Set}$.
$\mathsf{can_l}(s) : \mathbb{F}_A(X, s) \to \mathbb{F}_A(X + Y, s)$ be given by

$$\mathsf{can_l} = \mathbb{F}_A(\mathsf{inl}),$$

i.e.

$$\begin{aligned} \mathsf{can_l}(s, \mathsf{inl}\, a) &= \mathsf{inl}\, a \\ \mathsf{can_l}(s, \mathsf{inr}\, (c, h)) &= \mathsf{inr}\, (c, \lambda r.\mathsf{inl}\, h(r)) \end{aligned}$$

In category theory, if $\mathsf{elim} : \mathbb{G}^\infty \to \mathbb{G}(\mathbb{G}^\infty)$ is a final coalgebra, then exists for every $f : A \to \mathbb{G}(\mathbb{G}^\infty + A)$ a unique arrow $\mathsf{corec_f}$ such that the following diagram commutes:



This motivates the following definitions in type theory:

[6] We retain the notations $*$, η instead of bindM and unitM to stay in accordance with [5,6].
[7] We would like to thank an anonymous referee for this advice.

Definition 6.2 For $g : \Pi s : S.A(s) \to C(s)$ and $h : \Pi s : S.B(s) \to C(s)$ we define

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

by

$$\begin{aligned}[g,h](s,o) := [g(s),h(s)](o) := \mathsf{Case}\ o\ of\\ \mathsf{inl}\, a : g(s,a)\\ \mathsf{inr}\, b : h(s,b)\end{aligned}$$

For $f : \Pi s : S.A(s) \to \mathbb{F}_B(IO_B + A, s)$ let

$$\begin{aligned}\overline{\mathsf{coit}}_f &:= \mathsf{coit}_{[\mathsf{can_I}\circ\mathsf{elim},f]} = \mu(IO_B + A, [\mathsf{can_I} \circ \mathsf{elim}, f])\\ &\in \Pi s : S.(IO_B(s) + A(s)) \to IO_B(s),\end{aligned}$$

and $\mathsf{corec}_f : \Pi s : S.A(s) \to IO_B(s)$ with

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

Definition 6.3 For $q : \Pi s : S.A(s) \to IO_B(s)$ let $q^* : \Pi s : S.IO_A(s) \to \mathbb{F}_B(IO_B + IO_A, s)$ be defined by

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

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

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

and

$$\eta_A := \mathsf{coit}_{\breve{\eta}} : \Pi s : S.A(s) \to IO_A(s),$$

where $\breve{\eta} : \Pi s : S.A(s) \to \mathbb{F}_A(A,s)$ with $\breve{\eta}(s,a) = \mathsf{inl}\, a$.

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

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

Lemma 6.4 *Let $o : IO_{A_1}(s)$, $p(s) : A_0(s) \to IO_{A_1}(s)$ for $s : S$. Then*

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

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

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

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

$$[\mathsf{can_I} \circ \mathsf{elim}, p^*](s, \mathsf{inl}\ o) = \mathsf{can_I}(s, \mathsf{elim}(s,o)) = \mathsf{inl}\ a,$$

and so by equality

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

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

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

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

so by equality

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

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

We are now able to prove the first monad law:

Theorem 6.5 *Let $p : IO_A(s)$ and $q : \Pi s : S.A(s) \to IO_B(s)$.*
If $\mathsf{elim}_A(s,p) = \mathsf{inl}\ a$, then

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

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

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

And by the equality rule

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

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

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

By the equality rule,

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

By Lemma 6.4 follows

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

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

Corollary 6.6 (First monad law) *If* $q : \Pi s : S.A(s) \to IO_B(s)$, *then*

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

Unless otherwise noted, let in the rest of the article for $s : S$

$$o : IO_{A_0}(s),\ \ p(s) : A_0(s) \to IO_{A_1}(s),\ \ q(s) : A_1(s) \to IO_{A_2}(s),\ \ \bar{p} = \overline{\mathsf{coit}}_{p^*}.$$

Lemma 6.7 *If* $o \rightsquigarrow (c,h)$, *then*

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

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

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

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

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

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

Theorem 6.8 (Second monad law) *If* $p : IO_A(s)$, *then* $p * \eta \approx p$.

Proof. We show $p * \eta \sim_n p$ by induction on n.
I. $\mathsf{elim}_A(s,p) = \mathsf{inl}\ a$. Then we have

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

II. $\mathsf{elim}_A(s,p) = \mathsf{inr}\,(c,h)$. By Lemma 6.7 we get

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

and by I.H. follows the claim. □

Lemma 6.9 *If* $o \rightsquigarrow (c,h)$, *then* $(o * p) * q \rightsquigarrow (c, (h * p) * q)$.

Proof. By Lemma 6.7. □

Lemma 6.10 *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 6.7. □

Lemma 6.11 *If $o \rightsquigarrow a_0$ and $p(s, a_0) \rightsquigarrow a_1$, then $o * p \rightsquigarrow a_1$.*

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

$$p^*(s, a_0) = \mathsf{can}_{\mathsf{l}}(s, \mathsf{elim}(s, p(s, a_0))) = \mathsf{can}_{\mathsf{l}}(s, \mathsf{inl}\ a_1) = \mathsf{inl}\ a_1.$$

Therefore, $[\mathsf{can}_{\mathsf{l}} \circ \mathsf{elim}, p^*](s, \mathsf{inr}\ o) = p^*(s, a_0) = \mathsf{inl}\ a_1$.
By equality we get

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

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

Lemma 6.12 *If $o \rightsquigarrow a$ and $p(s, a) \rightsquigarrow (c, h)$, then $o * p \rightsquigarrow (c, h')$. with $h'(r) = \bar{p}(n(s, c, r), \mathsf{inl}\ h(r))$.*

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

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

Therefore, $[\mathsf{can}_{\mathsf{l}} \circ \mathsf{elim}, p^*](s, \mathsf{inr}\ o) = p^*(s, a) = \mathsf{inr}\ (c, \lambda r.\mathsf{inl}\ h(r))$. By equality we get $\mathsf{elim}(s, o * p) = \mathsf{elim}(s, \bar{p}(s, \mathsf{inr}\ o)) = \mathsf{inr}\ (c, \lambda r.\bar{p}(n(s, c, r), \mathsf{inl}\ h(r)))$. □

Lemma 6.13 *Let $o' = \bar{p}(s, \mathsf{inl}\ o)$. Then $o * q \approx o' * q$.*

Proof. We show $o * q \sim_n o' * q$ by induction on n.
First case: $\mathsf{elim}(s, o) = \mathsf{inl}\ a$. We have $[\mathsf{can}_{\mathsf{l}} \circ \mathsf{elim}, p^*](s, \mathsf{inl}\ o) = \mathsf{inl}\ a$, and therefore $\mathsf{elim}(s, \bar{p}(s, \mathsf{inl}\ o)) = \mathsf{inl}\ a$.
First subcase: $\mathsf{elim}(s, q(s, a)) = \mathsf{inl}\ b$. By Lemma 6.11 we get $\mathsf{elim}(s, o' * q) = \mathsf{inl}\ b = \mathsf{elim}(s, o * q)$.

Second subcase: $\mathsf{elim}(s, q(s, a)) = \mathsf{inr}\ (c, h)$. By Lemma 6.12 we get $\mathsf{elim}(s, o * q) = \mathsf{inr}\ (c, h') = \mathsf{elim}(s, o' * q)$, where $h'(r) = \bar{q}(n(s, c, r), \mathsf{inl}\ h(r))$. Second case: $\mathsf{elim}(s, o) = \mathsf{inr}\ (c, h)$. By Lemma 6.7 we get

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

We have $[\mathsf{can}_{\mathsf{l}} \circ \mathsf{elim}, p^*](s, \mathsf{inl}\ o) = \mathsf{inr}\ (c, \lambda r.\mathsf{inl}\ h(r))$, and therefore $\mathsf{elim}(s, o') = \mathsf{inr}\ (c, h')$, where $h'(r) = \bar{p}(n(s, c, r), \mathsf{inl}\ h(r))$. By Lemma 6.7 we get

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

and by I.H. the claim. □

Lemma 6.14 *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 6.12 follows $\mathsf{elim}(s, o * p) = \mathsf{inr}\ (c, h')$, where $h'(r) = \bar{p}(n(s, c, r), \mathsf{inl}\ h(r))$. By Lemma 6.7 follows

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

By $\mathsf{elim}(s, p(s, a)) = \mathsf{inr}\ (c, h)$ and Lemma 6.7 we get

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

By Lemma 6.12 we get

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

where $h''(r) = \bar{f}(n(s, c, r), \mathsf{inl}\ h(r) * q)$, $\bar{f} = \overline{\mathsf{coit}}_{f^*}$, $f = \lambda s, a.p(s, a) * q$. By Lemma 6.4 follows $h''(r) \approx h(r) * q$, and by Lemma 6.13 $h(r) * q \approx h'(r) * q$.□

Lemma 6.15 *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 6.11 and Lemma 6.12 we get $\mathsf{elim}(s, o * p) = \mathsf{inl}\ a_1$, $\mathsf{elim}(s, (o * p) * q) = \mathsf{inr}\ (c, h')$, where $h'(r) = \bar{q}(n(s, c, r), \mathsf{inl}\ h(r))$.

Furthermore, by Lemma 6.12

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

and again

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

where $h''(r) = \bar{f}(n(s, c, r), \mathsf{inl}\ h'(r))$, $\bar{f} = \overline{\mathsf{coit}}_{f^*}$, $f = \lambda s, a.p(s, a) * q$.
By Lemma 6.4 follows $h''(r) \approx h'(r) \approx h(r)$. □

Theorem 6.16 (Third monad law)

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

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

Case I: $\mathsf{elim}(s, o) = \mathsf{inr}\ (c, h)$. Then by Lemma 6.9

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

and by Lemma 6.10

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

The claim follows by the I.H.

Case II: $\mathsf{elim}(s, o) = \mathsf{inl}\ a_0$. This case follows by Lemmata 6.11, 6.15, 6.14. □

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 $\mathsf{IO} : (S \to \mathsf{Set}) \to (S \to \mathsf{Set})$ to Martin-Löf type theory. Using these rules we have introduced the bisimulation relation $\approx$, and operations $*$, η, and have shown that $(\mathsf{IO}, *, \eta)$ is a state-dependent monad w.r.t. $\approx$.

References

[1] Andrea Asperti, Guiseppe 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] Catarina Coquand. *Agda.* http://www.cs.chalmers.se~catarina/agda/.

[5] 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.

[6] 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.

[7] Ingrid Lindström. *A Construction of non-well-founded Sets within Martin-Löf's Type Theory.* The Journal of Symbolic Logic, Volume 54, Number 1, 1989.

[8] Z. Luo. *Computation and reasoning.* Clarendon Press, Oxford, 1994.

[9] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[10] Lawrence S. Moss. *Parametric corecursion.* In: Theoretical Computer Science 260, 2001.

[11] Bengt Nordström, Kent Peterson, Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Clarendon Press, Oxford, 1990.

[12] Lawrence C. Paulson. *Natural Deduction Proof as Higher-Order Resolution.* Technical report 82, University of Cambridge Computer Laboratory, Cambridge, 1985.

[13] Kent Peterson. *A Programming System for Type Theory.* PMG Memo 21, Chalmers University of Technology, S-412 96 Göteborg, 1982.

[14] R. Pollack. *The theory of LEGO. A proof checker for the extended calculus of constructions.* PhD thesis, LFCS, Edinburgh, 1994.

[15] Philip Wadler. *The essence of functional programming.* In: 19th Symposium on Principles of Programming Languages, Albuquerque, volume 19. ACM Press, January 1992.