Interactive Programs in Dependent Type Theory

Peter Hancock[1] and Anton Setzer[2]

[1] Dept. of Computing Science, University of Edinburgh, James Clerk Maxwell Building, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, fax: +44 131 667 7209, phone: +44 131 650 5129, pgh@dcs.ed.ac.uk.

[2] Dept. of Mathematics, Uppsala University, P.O. Box 480, S-751 06 Uppsala, Sweden, fax: +46 18 4713201, phone: +46 18 4713284, setzer@math.uu.se.

Abstract. We propose a representation of interactive systems in dependent type theory. This is meant as a basis for an execution environment for dependently typed programs, and for reasoning about their construction; there may be wider applications. The inspiration is the 'I/O-monad' of Haskell. The fundamental notion is an I/O-tree; its definition is parameterised over a general notion of dependently typed, command-response interactions called a world. I/O-trees represent strategies for one of the parties in a command/response interaction – the notion is not confined to functional programming. The definitions make essential use of the expressive strength of dependent typing. We present I/O-trees in two forms that we call 'non-normalising' and 'normalising'. The first form, which is simpler, is suitable for Turing-complete functional programming languages with general recursion. The second is definable within (ordinary) normalising type theory and we identify programs written in it as 'normalising I/O-programs'. We define new looping constructs (while and repeat), and a new refinement construct (redirect), which permits the implementation of libraries. We introduce a bisimulation relation between interactive programs, with respect to which we prove the monad laws and defining equations of while.

Keywords. Functional programming, reactive programming, interaction, dependent types, monadic I/O, repetition constructs, refinement.

1 I/O Concepts in Type Theory

Programming languages based on dependent types. A bit over 20 years ago Martin-Löf suggested that his type theory, originally developed as a framework for constructive mathematics, could be considered as a programming language [4]. Since that time, his suggestion has been taken up and explored in a number of ways (see for example in [6]). A question which so far seems to have received little attention is what form of input-output interface such programs should have. Indeed it is only in the last 10 years or so that this question has received a satisfactory answer in the context of conventional functional programming, through the efforts of Moggi [5], Wadler [10], and others.

Dependent types give us the ability to set up type systems for programming which can express with full precision any property we know how to define mathematically. For example, we can express the requirement for a function which maps lists to sorted lists using a dependent type [6]. Remarkably, with certain provisos of largely academic interest, we can still check the type of a program mechanically, and type-correctness carries full assurance that it satisfies its specification. The interest of dependent types for programming is therefore considerable. In the past few years, implementations of dependent type systems for functional programming have begun to appear [1].

Conventions, and plan of paper. In the following, we will work in standard dependent type theory (for example [6]) with the usual introduction and elimination rules and intensional equality, extended by some other rules. We will refer to it simply as 'type theory'. The notation we use, which is for the most part standard, is summarised in the appendix of the paper. Note that we sometimes omit indices and superscripts.

The plan of the paper is as follows. In the remainder of this section, we draw attention to the need for a model of interaction in type theory, and recall the approach taken by the designers of the functional programming language Haskell, using a monadic type form whose values are I/O-programs. In the next section, we present our extension of this notion, making use of type dependency. The third section introduces two repetition constructs (while and repeat), and a refinement construction. In the fourth, we point out that these can destroy normalisation, and develop an alternative formulation which preserves it. Finally there is a concluding section, followed by a summary of our notation.

The need for interactive programs in type theory. Traditional 'batch' programs may be written in type theory as functions from input values (given in advance) to output values. The output from such a program is obtained by evaluating the result of applying the function to its input. This batch model is adequate for a large class of programs, typically computationally intensive numerical search or optimisation programs. It is not however adequate for a program which runs, say, in the guidance system of an airplane.

The programs with which one is ordinarily confronted participate in interactions with their environment, i.e. us, while they are running. We give input via various devices like keyboard, mouse, or microphone and get output via devices like monitor, printer or loudspeaker, and this input-output cycle is repeated again and again. There may also be interaction with the file system and the network. Other kinds of program may be embedded in a physical system, in which there is continual interaction with physical sensors and actuators of some kind. So if we want to use type theory as a practical functional programming language, we have to consider how to use it to write interactive (or reactive) programs.

Some approaches to interactive programs in type theory. In conventional functional programming, several approaches to interaction have been pursued. A good survey of some of these approaches is made in [7]: dialogues (or lazy streams), continuations, and monadic I/O. (Mention should also be made of the 'uniqueness types' of the language Concurrent Clean, although a definitive exposition of this approach doesn't seem to have been published yet.[1] In this paper we follow the monadic approach, introduced by E. Moggi [5], upon which the input/output (I/O-) system of the language Haskell[2] has been erected. A monad (the concept comes from category theory) is a triple $(\mathrm{M}, *, \eta)$, whose components, written in dependent type theory, have types

$$\begin{aligned} \mathrm{M} &: \mathrm{Set} \to \mathrm{Set} \ , \\ * &: (A, B : \mathrm{Set}, p : \mathrm{M}\, A, q : A \to \mathrm{M}\, B) \to \mathrm{M}\, B \ , \\ \eta &: (A : \mathrm{Set}, a : A) \to \mathrm{M}\, A \ , \end{aligned}$$

such that the following laws hold with respect to suitable equality $=$ (which will not be definitional equality).

$$\begin{aligned} \eta^A_a *_{A,B} q &= q\, a \ , \\ p *_{A,B} \lambda x.\eta^A_x &= p \ , \\ (p *_{A,B} q) *_{B,C} r &= p *_{A,B} (\lambda x.q\, x *_{B,C} r) \ . \end{aligned}$$

A special case of a monad is the I/O-monad. When referring to the I/O-monad we write $(\mathrm{IO}\, A)$ instead of $(\mathrm{M}\, A)$. The interpretation of IO is as follows.

(a) For a given set A, $(\mathrm{IO}\, A)$ is the set of interactive programs which may or may not terminate, but terminate only with a result a of type A.
(b) $p * q$ is a program whose execution begins by executing p. If execution terminates with result a, then the program continues with $(q\, a)$. The result of the whole program is the result of $(q\, a)$.
(c) η_a is the program which has no interaction but terminates immediately, yielding result a.

In addition to these general components of any monad, there are functions specific to the kind of computations the programs are to evoke. For example, we can deal with programs that communicate by writing and reading strings (such as text lines): $\mathrm{write} : \mathrm{String} \to \mathrm{IO}\, \mathbf{1}$, $\mathrm{read} : \mathrm{IO}\, \mathrm{String}$. Here $(\mathrm{write}\, s)$ is the program which outputs s on some device and returns $\bullet$, and $\mathrm{read} : \mathrm{IO}\, \mathrm{String}$, is the program which reads a string and returns it.

Note that type checking does not require any interaction. What we really type check is the *text* of the program, rather than its execution. To execute or run a program means to interpret the program text as the specification of a certain I/O-behaviour.

For the I/O-monad, one sees that the laws mentioned above should hold with respect to an equality which identifies behaviourally equal programs.

Interactive programs are written in Haskell by using a form of the I/O-monad that gives access to most of the usual facilities of a traditional operating system, including files, graphics, time, as well as control features such as exception handling or multi-threading.

[1] `http://www.cs.kun.nl/~clean`
[2] `http://haskell.org`

The I/O-monad seems to be the most promising approach for the representation of interactive programs within dependent type theory. To add it to type theory as a new concept with new typing rules would however require extending a considerable body of metatheoretical research to the extended theory. Since we can define in type theory powerful data types, an easier approach is to define the I/O-monad directly in type theory. Thus the only feature we need over and above 'mathematical' type theory is that of actually running an I/O-program.

2 I/O-Trees

Worlds. Interactive programs are built from interactive commands. In dependent type theory we can define such a set of interactive commands in a very general way:

General assumption and definition 2.1. *A* world w *is a pair* $\langle C, R\rangle$ *such that* $C : \mathrm{Set}$ *and* $R : C \to \mathrm{Set}$. *In the following* w *is always a world* $\langle C, R\rangle$. *We will in most cases omit the parameter* w.

The idea is that C is the set of instructions or commands. These include commands to obtain input, commands to obtain output, and commands with a mixed effect. If $c : C$ is a command, then $(R\,c)$ is the type of responses or termination codes produced when such the command c has been performed. So we might have constructors

(a) $\mathrm{write} : \mathrm{String} \to C$ with $R(\mathrm{write}\,s) = \mathbf{1}$, and $(\mathrm{write}\,s)$ is the command to write s and return $\bullet : \mathbf{1}$ for success.
(b) $\mathrm{read} : C$, with $(R\,\mathrm{read} = \mathrm{String})$, where read stands for the command to read a string and returns it.

Of course in practice the commands would be more complex. For example, there might be an embellishment of write where $R(\mathrm{write}\,s) = \{\mathrm{success}, \mathrm{fail}\}$ and $(\mathrm{write}\,r)$ returns the information whether the output was performed successfully or not. More generally, there may be commands expressing interaction with file systems, network etc.

I/O-trees. We want to define the I/O-monad as a data type already in type theory. It seems particularly suitable to define it as an inductive data type, because using the elimination rule associated with such types we can then carry out program transformations. A naïve approach would be to take $*$, η and the additional primitive instructions such as read and $(\mathrm{write}\,a)$ as constructors for this type. However we need to verify the monad laws, and it turns out to that the naïve approach requires us to define a rather complicated equality relation. The situation is analogous to the definition of the set of natural numbers. We could define it from 0, 1 and $+$, which correspond in the I/O-monad to η, the primitive instructions and $*$. It is however much better to take 0 and successor S as constructors, and to define 1 and addition. What corresponds now to S in the IO-monad? This should be the operation which takes an instruction and a family of programs depending on the result of performing that instruction, and creates a new program that begins by performing this instruction and then continues with the program determined by its result. Instructions are given by C, where $w = \langle C, R \rangle$ is a world, and R provides the result type.The rules for $\mathrm{IO}_w\, A$ are:

$$\begin{aligned}
&\mathrm{IO}_w : \mathrm{Set} \to \mathrm{Set}, \textbf{ where } \mathrm{IO}_w\, A \textbf{ has constructors}\\
&\mathrm{leaf} : A \to \mathrm{IO}_w\, A\ \ ,\\
&\ \ \mathrm{do} : (c : C, p : R\, c \to \mathrm{IO}\, A) \to \mathrm{IO}\, A\ \ .
\end{aligned}$$

The constructor $(\mathrm{leaf}\, a)$ is what was written η_a^A before, and $(\mathrm{do}\, c\, p)$ denotes the program that first executes the command c, and depending on the result $r : R\, c$ of this execution continues with $(p\, r)$.

Note that $(\mathrm{IO}_w\, A)$ is now parametrised with respect to w, a feature which can be expressed only with dependent types.

$(\mathrm{IO}_w\, A)$ is the set of well-founded trees with leaves in A and inner nodes labelled by some $c : C$ having branching degree $(R\, c)$. If we compare $(\mathrm{IO}_w\, A)$ with standard type theory we see that we simply have a slight extension of the W-type. In fact we can even define $(\mathrm{IO}_w\, A)$ directly in terms of the W-type:

$$\begin{aligned}
\mathrm{IO}_w\, A = {} & \mathrm{W}x : (C + A).R'\, x\ \ ,\\
& \textbf{where } R'(\mathrm{inl}\, c) = R\, c\ \ ,\\
& \qquad\quad R'(\mathrm{inr}\, a) = \mathbf{0}\ \ .
\end{aligned}$$

We therefore call $(\mathrm{IO}_w\, A)$ the type of I/O-trees with leaves in A.

Execution of I/O-programs. Up to now we have defined an inductive data type of I/O-programs within constructive type theory, but there is still no way to actually *run* such a program. Execution is an *external* operation rather than a constant within type theory. Just as an implementation of type theory will provide an external operation or facility to compute (and display) the (head-) normal form of a term, so we propose to provide a second operation which executes a term which denotes an I/O-program.

More precisely, this works as follows. Let $w_0 = \langle C_0, R_0 \rangle$ be a world corresponding to the real commands, so that to every $c : C_0$ there corresponds a real I/O-command c having some value $r : R_0\, c$ as result. If we have derived $p : \mathrm{IO}_{w_0}\, A$ then the external operation `execute` can be performed upon p. The operation `execute` does the following. It reduces p to canonical form, i.e. to a term of constructor form. This form must be either $(\mathrm{leaf}\, a)$ or $(\mathrm{do}\, c\, q)$. If it is $(\mathrm{leaf}\, a)$, then $a : A$ and execution terminates, yielding as result a (which, when running the program from a command line will be displayed in a similar way as the result of the evaluation of an expression). If it is $(\mathrm{do}\, c\, q)$, then first the interactive command corresponding to c is performed obtaining a result $r : R\, c$, after which execution continues with $(q\, r)$.

Roughly speaking a program p is evaluated to normal form, as it were 'fetching' the next instruction. The instruction is 'executed', and the result used to select the next program to be evaluated. Another way of looking at it is that through successive interactions we trace out a descending chain through the tree p.

A first example. In the following example we assume commands readstr for reading a string, (writestr s) for writing a string s. Further let $=_{\text{String}}$ be a decidable (boolean valued) equality on strings. The following program prompts for the root-password. If the user types in the right one ("Wurzel"[3]) the program terminates successfully, but otherwise responds with "Login incorrect" and then fails. We use some syntactic sugar.

$$
\begin{aligned}
C &= \{\,\mathrm{readstr}\,\} \cup \{\,\mathrm{writestr}\, s \mid s : \mathrm{string}\,\} : \mathrm{Set} \ , \\
R &: C \to \mathrm{Set} \ , \\
R\,\mathrm{readstr} &= \mathrm{string} \ , \\
R\,(\mathrm{writestr}\, s) &= \mathbf{1} \ , \\
\mathrm{Wurzel} &= \mathrm{do}\ \mathrm{writestr}\ \text{“Password (root):”} \\
&\qquad\quad \lambda a.\,\mathrm{do}\ \mathrm{readstr} \\
&\qquad\qquad\quad \lambda s.\,\mathbf{if}\ s = \text{“Wurzel”} \\
&\qquad\qquad\qquad \mathbf{then}\ \mathrm{leaf}\,\mathrm{success} \\
&\qquad\qquad\qquad \mathbf{else}\ \mathrm{do}\ (\mathrm{writestr}\ \text{“Login incorrect”}) \\
&\qquad\qquad\qquad\qquad\quad \lambda a.\,\mathrm{leaf}\,\mathrm{fail} \\
&\ : \ \mathrm{IO}\,\{\,\mathrm{success}, \mathrm{fail}\,\} \ .
\end{aligned}
$$

η, $*$, **interact**. It is now easy to define the operations η and $*$:

$$
\begin{aligned}
\eta^A_a &= \mathrm{leaf}\, a \ , \\
\mathrm{leaf}\, a *_{A,B} q &= q\, a \ , \\
\mathrm{do}\, c\, p *_{A,B} q &= \mathrm{do}\, c\, (\lambda x.p\, x *_{A,B} q) \ .
\end{aligned}
$$

The monad laws are then provable for well-founded trees with respect to extensional equality.

Additionally we can define a function $\mathsf{interact} : (c : C) \to \mathrm{IO}\,(R\,c)$ by $\mathsf{interact}\, c = \mathrm{do}\, c\, (\lambda x.\mathrm{leaf}\, x)$. The program $(\mathsf{interact}\, c)$ simply executes the command c and returns its result.

3 Constructions for Defining I/O-trees

It should be possible to define interactive programs with infinitely many interactions. For instance, if we execute an editor and never terminate the program, the execution should go on for ever. So we need constructions for defining such programs. This will however destroy normalisation. We will see in Sect. 4, how to modify the concept in order to obtain a normalising type theory. The definitions of all constructions in this section are possible only in the presence of *dependent types*, which demonstrate their expressive power.

We will first look at the repeat-loop, which is very useful in examples.

[3] This really happened.

repeat. Assume $A, B : \mathrm{Set}$, $b : B$, $p : B \to \mathrm{IO}_w\,(B + A)$. We want to define a program $\mathsf{repeat}_A\,B\,b\,p : \mathrm{IO}_w\,A$, which, when executed, operates as follows. First program $(p\,b)$ is executed. If it terminates with result $(\mathrm{inl}\,b')$, then the program continues with $(\mathsf{repeat}_A\,B\,b'\,p)$. If it terminates with $(\mathrm{inr}\,a)$, the program terminates and returns a.

However, if $p\,b = \mathrm{leaf}\,a$ for some $b : B$, this might result in an expression that does not evaluate to constructor form – for instance $(\mathsf{repeat}_B\,b\lambda x.\mathrm{leaf}\,(\mathrm{inl}\,b))$. Therefore we have to restrict p, and the easiest solution is to replace $(\mathrm{IO}_w\,(B + A))$ by $(\mathrm{IO}^+_w\,(B + A))$. Here for $D : \mathrm{Set}$ let $\mathrm{IO}^+_w\,D = \Sigma c : C.R\,c \to \mathrm{IO}_w\,D$ be the set of I/O-programs with results in D, with one interaction (namely command c). Let $\mathrm{do}^+\,c\,p = \langle c, p\rangle : \mathrm{IO}_w\,D$ and, if $p : \mathrm{IO}^+_w\,D$, let $p^- : \mathrm{IO}_w\,D$ be defined by $(\mathrm{do}^+\,c\,p)^- = \mathrm{do}\,c\,p$.

The definition (which uses full recursion and so allows us to define non-well-founded trees and form non-normalising terms) of the repeat construction is as follows:

$$
\begin{aligned}
\mathsf{repeat}_w &: (A, B : \mathrm{Set}, b : B, p : B \to \mathrm{IO}^+_w\,(B + A)) \to \mathrm{IO}_w\,B \enspace , \\
\mathsf{repeat}_{w,A}\,B\,b\,p &= (p\,b)^- *_{w,B+A,A}\,q \enspace , \\
&\quad \textbf{where} \quad q\,(\mathrm{inl}\,b') = \mathsf{repeat}_{w,A}\,B\,b'\,p \enspace , \\
&\qquad\qquad\quad\; q\,(\mathrm{inr}\,a) = \mathrm{leaf}\,a \enspace .
\end{aligned}
$$

Example. As an example we define a rudimentary editor. The only command is readchar, which has as result either a character c typed in, cursorleft for the cursor-left-button or terminate for some key associated with termination. The program reads the text created by these keys and returns the result. ($(\mathrm{truncate}\,s)$ will be the result of deleting from string s the last letter, $(\mathrm{append}\,s\,c)$ an operation which appends at the end of string s character c and “” the empty string.)

$$
\begin{aligned}
C &= \{\mathrm{readchar}\} : \mathrm{Set} \enspace , \\
R &: C \to \mathrm{Set} \enspace , \\
R\,c &= \{\mathrm{ch}\,c \mid c : \mathrm{char}\} \cup \{\mathrm{cursorleft}, \mathrm{terminate}\} \enspace . \\
\mathrm{editor} &= \mathsf{repeat}_{\langle C,R\rangle,\mathrm{string}}\,\mathrm{string}\,\text{“”}\,\lambda s.\mathrm{do}^+\,\mathrm{readchar}\,q \enspace , \\
\textbf{where} &\quad q\,(\mathrm{ch}\,c) = \mathrm{leaf}\,(\mathrm{inl}\,(\mathrm{append}\,s\,c)) \enspace , \\
&\quad q\,\mathrm{cursorleft} = \mathrm{leaf}\,(\mathrm{inl}\,(\mathrm{truncate}\,s)) \enspace , \\
&\quad q\,\mathrm{terminate} = \mathrm{leaf}\,(\mathrm{inr}\,s) \enspace .
\end{aligned}
$$

While-loop. While loops are defined similarly to repeat loops.

$$
\mathsf{while}_w : (A, B : \mathrm{Set}, b : B, p : B \to (\mathrm{IO}^+_w\,B + \mathrm{IO}_w\,A)) \to \mathrm{IO}_w\,A \enspace .
$$

The definition proceeds by cases on the value of $(p\,b)$. If it is of the form $(\mathrm{inl}\,q)$, then q is executed, and, once it terminates with result b', the program continues with $(\mathsf{while}_{w,A}\,B\,b'\,p)$. If it is $(\mathrm{inr}\,q)$, q is executed and its result returned as final result. The definition, which uses again full recursion, is

$$\begin{aligned}\mathsf{while}_{w,A}\,B\,b\,p &= f\,(p\,b)\ , \\ \textbf{where}\quad f\,(\mathrm{inl}\,q) &= q^{-} *_{w,B,A} \lambda b'.\mathsf{while}_{w,A}\,B\,b'\,p\ , \\ f\,(\mathrm{inr}\,q) &= q\ .\end{aligned}$$

It is now an easy exercise to express while by repeat and vice versa.

Redirect. $*$ can be regarded as "horizontal composition" of programs. There is also a "vertical composition":
Assume worlds $w = \langle C, R\rangle$ and $w' = \langle C', R'\rangle$, $A : \mathrm{Set}$ and $p : \mathrm{IO}_w\,A$. We want to refine p to a program in world w', by replacing every command $c : C$ by a program $(q\,c)$ in world w' which has as result an answer $r : R\,c$. This suggest q to have type $(c : C) \to \mathrm{IO}_{w'}(R\,c)$. However, if we allow $(q\,c)$ to be a leaf and p has infinitely many commands, this will allow us to construct an expression that cannot be evaluated to constructor form. To avoid this we assume $q : (c : C) \to \mathrm{IO}^{+}_{w'}(R\,c)$. The construction that results is

$$\begin{aligned}&\mathsf{redirect}_{w,w'} : (A : \mathrm{Set}, p : \mathrm{IO}_w\,A, q : (c : C) \to \mathrm{IO}^{+}_{w'}(R\,c)) \to \mathrm{IO}_{w'}\,A\ , \\ &\quad\textbf{where}\quad \mathsf{redirect}_{w,w',A}\,(\mathrm{leaf}\,a)\,q = \mathrm{leaf}\,a\ , \\ &\qquad\qquad\quad \mathsf{redirect}_{w,w',A}\,(\mathrm{do}\,c\,p)\,q = (q\,c)^{-} *_{w',R\,c,A} \lambda r.\mathsf{redirect}_{w,w',A}\,(p\,r)\,q\ .\end{aligned}$$

Example. Let the high level world be $w_0 = \langle C_0, R_0\rangle$, with $C_0 = \{\mathrm{read}\} \cup \{\mathrm{write}\,s \mid s : \mathrm{string}\}$. read should be a command for reading a string, so $R_0\,\mathrm{read} = \mathrm{string}$, and $(\mathrm{write}\,s)$ should be an instruction for writing a string, $R_0\,(\mathrm{write}\,s) = \mathbf{1}$. Let the low level world w_1 have commands for reading a key, writing a symbol, and movements of the cursor left and right. Let $q : (c : C_0) \to \mathrm{IO}_{w_1}\,(R_0\,c)$, where $(q\,\mathrm{read})$ is some editor, which uses the keys to define a string and has as result that string, and $(q\,(\mathrm{write}\,s))$ be some output routine for strings. $(\mathsf{redirect}\,p\,q)$ generates a program using basic commands from a program p using the high level commands .

This illustrates that with $\mathsf{redirect}$ we have the possibility of building up *libraries*. In other words, we can define worlds with very complex and powerful I/O-commands and write in the same language programs for transforming these high level programs into those using very basic commands.

Equality. With $\mathsf{while}-$ and $\mathsf{repeat}-$loops we introduce non-well-founded I/O-trees. Even with extensional equality it seems that it is no longer possible to prove the monad laws. So extensional equality seems to be too weak for dealing with non-well-founded programs. Instead we use bisimulation as equality. In [3] I. Lindström has given a very elegant definition of such an equality. This equality, transferred to our setting, is defined as $\forall n.p \simeq'_{w,A,n} q$, where $p \simeq'_{w,A,n} q$ expresses that p and q coincide up to height n. In the following the world w will be a parameter in all definitions, which we omit for simplicity. We will refer to the equality-types $=_C$ and $=_A$ on C and A and only identify commands and leaves, which are equal with respect to these equalities. So we do not make use of any user-defined (setoid-)equality on C, A. In a followup to this article, we will consider the generalisation to user-defined equalities.

$$
\begin{aligned}
\simeq \;&:\; (A : \mathrm{Set}, p, q : \mathrm{IO}\, A) \to \mathrm{Set} \;\;,\\
\simeq' \;&:\; (A : \mathrm{Set}, n : \mathbb{N}, p, q : \mathrm{IO}\, A) \to \mathrm{Set} \;\;,\\
(p \simeq_A q) &= \forall n : \mathbb{N}. p \simeq'_{A,n} q,\\
(p \simeq_{A,0} q) &= \top \;\;,\\
(\mathrm{leaf} a \simeq'_{A,n+1} \mathrm{do}\, c\, p) &= (\mathrm{do}\, c\, p \simeq'_{A,n+1} \mathrm{leaf}\, a) = \bot \;\;,\\
(\mathrm{leaf}\, a \simeq'_{A,n+1} \mathrm{leaf}\, a') &= (a =_A a') \;\;,\\
(\mathrm{do}\, c\, p \simeq'_{A,n+1} \mathrm{do}\, c'\, p') &= \exists p : (c =_C c'). \forall r : R\, c. p\, r \simeq'_{A,n} p'\, (\mathrm{J}_C\, R\, c\, c'\, p\, r) \;\;.
\end{aligned}
$$

Definition 3.1. *(a) Let case-distinction for* IO *be the rule (under the assumptions that* $A : \mathrm{Set}$*,* $B : (p : \mathrm{IO}\, A) \to \mathrm{Set}$*):*

$$
\begin{aligned}
\mathrm{C}^{\mathrm{IO}}_{A,B} : (&(a : A) \to B\, (\mathrm{leaf}\, a), (c : C, q : R\, c \to \mathrm{IO}\, A) \to B\, (\mathrm{do}\, c\, q),\\
&p : \mathrm{IO}\, A) \to B\, p \;\;.
\end{aligned}
$$

(b) Let TT(IO) *be the extension of the standard type theory used in this article by the defining rules for* IO *and case-distinction for* IO*.*

Lemma 3.2. TT(IO) proves the following (under the assumptions that A, B : Set, and all other variables are of appropriate type)

(a) $\simeq_A$ is reflexive, symmetric and transitive.
(b) $p \simeq_A p' \to (\forall a : A. q\, a \simeq_B q'\, a) \to p *_{A,B} q \simeq_A p' *_{A,B} q'$.

Proof. (a): First we prove the lemma with $\simeq_A$ replaced by $\simeq'_{A,n}$ by induction on $n : \mathbb{N}$, using the elimination rules for equality. Then the assertion follows by the definition of $\simeq$. (b) Show that $p \simeq'_{A,n} p'$ and $\forall a : A. q\, a \simeq'_{B,m} q'\, a$ imply $\mathrm{do}\, p\, q \simeq'_{B,\min\{n,m\}} \mathrm{do}\, p'\, q'$ by induction on n. □

Theorem 3.3. TT(IO) proves the monad laws with respect to $\simeq_A$.

Proof. The first law holds definitionally and a fortiori with respect to $\simeq_A$, by its reflexivity. The second and third laws are proved first with $\simeq_A$ replaced by $\simeq'_{A,n}$ using induction on n and case distinction on p. Then the assertion follow from the definition of $\simeq_A$. □

I/O-trees as a general concept for command/response-interaction. It seems that the applications of I/O-trees, which are in general non-well-founded trees, are not limited only to functional programming languages. I/O-trees cover in a general way command/response-interaction with one agent (a program) having control over the commands. Every I/O-behaviour corresponds, up to the equality we have introduced above, to exactly one I/O-tree. Therefore I/O-trees are suitable models for this kind of interaction.

4 Normalising Version

Counterexample to normalization. If we take standard reduction rules corresponding to the equalities given above, the above definitions give non-normalizing programs. Let for instance $A = B = C = \mathbb{N}$, let $(R\,c)$ be arbitrary, and let $w = \langle C, R\rangle$, $f : \mathbb{N} \to \mathbb{N}$. We omit the parameter w.

$$\begin{aligned} p \;&:=\; \lambda n.\mathrm{do}^{+}\,(f\,n)\,\lambda x.\mathrm{leaf}\,(\mathrm{inl}\,(n+1)) : \mathbb{N} \to \mathrm{IO}^{+}\,(A+B)\ , \\ \mathsf{repeat}\,0\,p &\longrightarrow \mathrm{do}\,(f\,0)\,\lambda x.\mathsf{repeat}\,(\mathrm{S}\,0)\,p \\ &\longrightarrow \mathrm{do}\,(f\,0)\,\lambda x.\mathrm{do}\,(f\,(\mathrm{S}\,0))\,\lambda y.\mathsf{repeat}\,(\mathrm{S}\,(\mathrm{S}\,0))\,p \\ &\longrightarrow \mathrm{do}\,(f\,0)\,\lambda x.\mathrm{do}\,(f\,(\mathrm{S}\,0))\,\lambda y.\mathrm{do}\,(f\,(\mathrm{S}\,(\mathrm{S}\,0)))\,\lambda z.\mathsf{repeat}\,(\mathrm{S}\,(\mathrm{S}\,(\mathrm{S}\,0)))\,p \\ &\longrightarrow \cdots\ . \end{aligned}$$

We see that definitional equality is no longer decidable, since we cannot decide whether two functions $\mathbb{N} \to \mathbb{N}$ are extensionally equal. This implies the undecidability of type checking, since with an type checking algorithm we can decide definitional equality (for $a, b : A$, the term $\lambda B, f.f\,a$ is of type $(B : (x : A) \to \mathrm{Set}, f : (x : A) \to B\,x) \to B\,b$ if and only if a and b are definitionally equal). In the following we are going to develop a normalizing variant of the above.

How to regain normalisation. The main idea is to introduce either repeat or while as a constructor, where with while, the definition of $*$ and the proofs of equalities turn out to be easier. One problem is however that while (the same is the case with repeat) defines an element of $(\mathrm{IO}_w\,A)$ by referring to $(\mathrm{IO}_w\,B)$ for an arbitrary set B. To demand that $(\mathrm{IO}_w\,A)$ is a set means to define a set by referring negatively to all sets, which is a problematic definition. (The typing rules require that if A is a set, $(\mathrm{IO}_w\,A)$ is a type).

In order to control this we will restrict the sets referred to in while to elements of a universe, which we extend in order to get representatives for $\mathbf{1} + R\,c$.

General assumption and definition 4.1. *(a) Let $w = \langle C, R\rangle$ be a world.*
(b) Let $\mathrm{U} : \mathrm{Set}$, $\mathrm{T} : \mathrm{U} \to \mathrm{Set}$ be some fixed collection of sets (i.e. a universe).
(c) Let $\mathrm{set} := \mathrm{U} + C$, $\mathrm{el} : \mathrm{set} \to \mathrm{Set}$, $\mathrm{el}\,(\mathrm{inl}\,u) = \mathrm{T}\,u$, $\mathrm{el}\,(\mathrm{inr}\,c) = \mathbf{1} + R\,c$, R according to the world w. We write $(\widehat{\mathbf{1} + \mathrm{R}\,c})$ for $(\mathrm{inr}\,c)$.

All the following definitions depend on w, U, and T as parameters, which we however omit for simplicity.

We can now omit the constructor do, since it can be simulated by while.

Definition of $\mathrm{IO}\,A$.

$$\begin{aligned} \mathrm{IO} &: \mathrm{Set} \to \mathrm{Set}\ ,\ \textbf{where } (\mathrm{IO}\,A) \textbf{ has constructors} \\ \mathrm{leaf} &: A \to \mathrm{IO}\,A\ , \\ \mathsf{while} &: (u : \mathrm{set}, a : \mathrm{el}\,u, n : \mathrm{el}\,u \to (\mathrm{IO}^{+}\,(\mathrm{el}\,u) + \mathrm{IO}\,A)) \to \mathrm{IO}\,A\ , \\ \textbf{and } \mathrm{IO}^{+} &: \mathrm{Set} \to \mathrm{Set}\ , \\ \mathrm{IO}^{+}\,A &= \Sigma c : C.R\,c \to \mathrm{IO}\,A\ . \end{aligned}$$

Monad operations. Now we can define the Monad operations (where sets have to be replaced by elements of the universe)

$$
\begin{aligned}
\eta^A_a &:= \mathrm{leaf}\, a \ , \\
\mathrm{leaf}\, a *_{A,B} q \ &= \ q\, a \ , \\
\mathsf{while}\, u\, a\, p *_{A,B} q \ &= \ \mathsf{while}\, u\, a\, (p \circledast_{A,B,u} q) \ , \\
\textbf{where } \circledast \ &: \ (A, B : \mathrm{Set}, u : \mathrm{set}, p : \mathrm{el}\, u \to (\mathrm{IO}^+\, (\mathrm{el}\, u) + \mathrm{IO}\, A), \\
&\qquad q : A \to \mathrm{IO}\, B) \to \mathrm{el}\, u \to (\mathrm{IO}^+ (\mathrm{el}\, u) + \mathrm{IO}\, B) \ , \\
&\quad \text{if } p\, b = \mathrm{inl}\, p', \text{ then } (p \circledast_{A,B,u} q)\, b = \mathrm{inl}\, p' \ , \\
&\quad \text{if } p\, b = \mathrm{inr}\, p', \text{ then } (p \circledast_{A,B,u} q)\, b = \mathrm{inr}\, (p' *_{A,B} q) \ .
\end{aligned}
$$

Do. We define now the operation $(\mathrm{do}_A\, c\, p)$ (note that do is *not* a constructor):

$$
\begin{aligned}
&\mathrm{do}_A\, c\, p = \mathsf{while}\, (\widehat{\mathbf{1} + \mathrm{R}\, c})\, (\mathrm{inl}\, \bullet)\, q \ , \\
&\textbf{where} \quad q\, (\mathrm{inl}\, \bullet) = \mathrm{inl}\, \langle c, \lambda r.\mathrm{leaf}\, (\mathrm{inr}\, r)\rangle \ , \\
&\qquad\qquad\ \ q\, (\mathrm{inr}\, r) = \mathrm{inr}\, (p\, r) \ .
\end{aligned}
$$

Split. Elements of $(\mathrm{IO}\, A)$ can be regarded as representatives of non-wellfounded trees. We define a function split_A, which determines whether a value $p : \mathrm{IO}\, A$ is a leaf labelled by $a : A$ (then $\mathrm{split}_A\, p = \mathrm{inr}\, a$) or represents a program which executes command c and depending on the result $r : R\, c$ of it executes program $q\, r$ (then $\mathrm{split}_A\, p = \mathrm{inl}\, \langle c, q\rangle$). In the latter case p represents a tree with root labelled by c and rth subtree being the tree represented by $(q\, r)$.

$$
\begin{aligned}
&\mathrm{split} \ : \ (A : \mathrm{Set}, p : \mathrm{IO}\, A) \to (\mathrm{IO}^+\, A + A) \ , \\
&\mathrm{split}_A\, (\mathrm{leaf}\, a) = \mathrm{inr}\, a \ , \\
&\textbf{If } \ p\, a = \mathrm{inr}\, q, \qquad \textbf{then } \mathrm{split}_A\, (\mathsf{while}\, u\, a\, p) = \mathrm{split}_A\, q \ , \\
&\textbf{If } \ p\, a = \mathrm{inl}\, \langle c, q\rangle, \textbf{ then} \\
&\mathrm{split}_A\, (\mathsf{while}\, u\, a\, p) = \mathrm{inl}\, \langle c, \lambda r.q\, r *_{\mathrm{el}(u),A} \lambda x.\mathsf{while}\, u\, x\, p\rangle \ .
\end{aligned}
$$

Execution of I/O-programs. Assume a fixed world $w_0 = \langle C_0, R_0\rangle$ corresponding to real commands, as before. $\mathsf{execute}$, adapted to the new setting, operates as follows: Applied to a program $p : \mathrm{IO}_{C_0,R_0}\, A$ it evaluates $\mathrm{split}_A\, p$. If the result is $(\mathrm{inr}\, a)$, then $\mathsf{execute}$ stops with result a. If $\mathrm{split}_A\, p = \mathrm{inl}\, \langle c, q\rangle$, then c is executed, and depending on the result r, $\mathsf{execute}$ continues with $(q\, r)$.

Normalising I/O-programs. When using only inductive data types with their elimination rules, the type theory in question is strongly normalising. Therefore, when running execute, at the beginning and after the result of a command is obtained, split_A applied to the respective program will always reduce to an element of the form $(\mathrm{inr}\, a)$ or $(\mathrm{inl}\, \langle c, q\rangle)$. However, it might be that infinitely many commands are executed, which is to say that the tree is not necessarily well-founded. We call a program which at the beginning and after the result of a command is obtained, executes the next command after finite amount of time, or terminates, a *normalising I/O-program.* $(\mathrm{IO}\, A)$ (together with $\mathsf{execute}$) represents a class of normalising I/O-programs.

Equality. Under the same assumptions as in Sect. 3 we can define now an equality on elements of $\mathrm{IO}\,A$. However, we use split in order to get access to the corresponding tree-structure:

$$\begin{aligned}
\simeq\ &:\ (A : \mathrm{Set}, p, q : \mathrm{IO}\,A) \to \mathrm{Set}\ \ , \\
\simeq'\ &:\ (A : \mathrm{Set}, n : \mathbb{N}, p, q : \mathrm{IO}\,A) \to \mathrm{Set}\ \ , \\
(p \simeq_A q) &= \forall n : \mathbb{N}.p \simeq'_{A,n} q\ \ , \\
p \simeq'_{A,0} q &= \top\ \ , \\
p \simeq'_{A,n+1} q &= (\mathrm{split}_A\, p \simeq''_{A,n} \mathrm{split}_A\, q)\ \ , \ \textbf{where} \\
\simeq''\ &:\ (A : \mathrm{Set}, n : \mathbb{N}, p, q : \mathrm{IO}^+\,A + A) \to \mathrm{Set}\ \ , \\
(\mathrm{inr}\, a \simeq''_{A,n} \mathrm{inl}\,\langle c, p\rangle) &= (\mathrm{inl}\,\langle c, p\rangle \simeq''_{A,n} \mathrm{inr}\, a) = \perp\ \ , \\
(\mathrm{inr}\, a \simeq''_{A,n} \mathrm{inr}\, a') &= (a =_A a')\ \ , \\
(\mathrm{inl}\,\langle c, q\rangle \simeq''_{A,n} \mathrm{inl}\,\langle c', q'\rangle) &= \exists p : (c =_C c').\forall r : R\, c.q\, r \simeq'_{A,n} q'\, (\mathrm{J}\, C\, R\, c\, c'\, p\, r)\ \ .
\end{aligned}$$

Note that $\simeq'_{A,n}$ identifies programs which behave identically in the first n steps, and therefore $\simeq_A$ identifies exactly behaviourally equal programs. Note however that we only identify commands $c : C$, which are equal with respect to $=_C$.

Proof of the monad laws, definitional equalities for while *and other standard properties.* The following can be proved inside type theory. (Some indices or superscripts have been left implicit).

Lemma 4.2. (a) $\eta_a * p \simeq_A p\, a$.
(b) $\simeq_A$ and $\simeq'_{A,n}$ are reflexive, symmetric and transitive.
(c) If $p\, a =_{\mathrm{IO}\,(\mathrm{el}\, u)+\mathrm{IO}\,A} \mathrm{inr}\, q$, then $\mathsf{while}\, u\, a\, p \simeq_A q$.

Proof. (a) is trivial. (b) follows with $\simeq_A$ replaced by $\simeq'_{A,n}$ by induction on n – in case of symmetry and transitivity one uses additionally the elimination rules for $=_C$. From this the assertion follows.
(c) $\mathrm{split}\,(\mathsf{while}\, u\, a\, p) =_{\mathrm{IO}^+\,A+A} \mathrm{split}\, q$. □

For stating and proving the next lemmata we introduce an equality on the type of p in $(\mathsf{while}\, u\, a\, p)$, i.e. $(\mathrm{el}\, u) \to (\mathrm{IO}^+\,(\mathrm{el}\, u) + A)$:

Definition 4.3.

$$\begin{aligned}
\simeq^{\mathrm{w}}\ &:\ (A : \mathrm{Set}, u : \mathrm{set}, \\
&\qquad p, q : (\mathrm{el}\, u) \to (\mathrm{IO}^+\,(\mathrm{el}\, u) + \mathrm{IO}\,A)) \to \mathrm{Set}\ \ , \\
p \simeq^{\mathrm{w}}_{A,u} q &= \forall x : \mathrm{el}\, u.p\, x \simeq^{\mathrm{w,aux}}_{A,u} q\, x\ \ , \ \textbf{where} \\
\simeq^{\mathrm{w,aux}}\ &:\ (A : \mathrm{Set}, u : \mathrm{set}, p, q : \mathrm{IO}^+\,(\mathrm{el}\, u) + \mathrm{IO}\,A) \\
&\qquad \to \mathrm{Set}\ \ , \\
(\mathrm{inl}\, q \simeq^{\mathrm{w,aux}}_{A,u} \mathrm{inr}\, q) &= (\mathrm{inr}\, q \simeq^{\mathrm{w,aux}}_{A,u} \mathrm{inl}\, q) = \perp\ \ , \\
(\mathrm{inr}\, q \simeq^{\mathrm{w,aux}}_{A,u} \mathrm{inr}\, q) &= (q \simeq_A q')\ \ , \\
(\mathrm{inl}\,\langle c, q\rangle \simeq^{\mathrm{w,aux}}_{A,u} \mathrm{inl}\,\langle c', q'\rangle) &= \exists p : (c =_C c').\forall r : R\, c. \\
&\qquad q\, r \simeq_{\mathrm{el}\, u} q'\, (\mathrm{J}\, C\, R\, c\, c'\, p\, r)\ \ .
\end{aligned}$$

Similarly we define $\simeq^{\mathrm{w}\prime}$, $\simeq^{\mathrm{w,aux}\prime}$ *with an additional argument* $n : N$ *and refer to* $\simeq'_{A,n}$, $\simeq'_{\mathrm{el}\,u,n}$ *instead of* $\simeq_A$, $\simeq_{\mathrm{el}\,u}$.

Lemma 4.4. (a) $(p_0 \simeq_A p_1 \wedge \forall a : A.q_0\, a \simeq_B q_1\, a) \to (p_0 * q_0 \simeq_B p_1 * q_1)$.
(b) $p_0 \simeq^{\mathrm{w}}_{A,u} p_1 \to \mathsf{while}\, u\, a\, p_0 \simeq_A \mathsf{while}\, u\, a\, p_1$.
(c) For $p : \mathrm{IO}\, A$, $q : A \to \mathrm{IO}\, B$, $r : B \to \mathrm{IO}\, D$ it follows
$(p * q) * r \simeq_D p * \lambda x.((q\, x) * r)$.

Proof. We prove (a) – (c), with $\lambda x. \simeq_x$, $\lambda x. \simeq^{\mathrm{w}}_{x,u}$ replaced by $\lambda x. \simeq'_{x,n}$, $\lambda x. \simeq'^{\mathrm{w}}_{x,u,n}$, simultaneously by induction on n. The case $n = 0$ is trivial, so we assume the assertion has been proved for n and prove it for $n + 1$:
(a) Side-induction on p_0, side-side-induction on p_1:
If $p_0 = \mathrm{leaf}\, a_0$ and $p_1 = \mathrm{leaf}\, a_1$, then $a_0 =_A a_1$ and $q_0\, a_0 \simeq'_{B,n+1} q_1\, a_1$.
If $p_0 = \mathsf{while}\, u\, a\, \widetilde{p}_0$ with $\widetilde{p}_0\, a = \mathrm{inr}\, \widehat{p}_0$, then $p_0 \simeq_A \widehat{p}_0$, and by
$p_0 * q_0 = \mathsf{while}\, u\, a\, (\widetilde{p}_0 \circledast q_0)$, $(\widetilde{p}_0 \circledast q_0)\, a = \mathrm{inr}\, (\widehat{p}_0 * q_0)$ it follows
$p_0 * q_0 \simeq_B \widehat{p}_0 * q_0$, and the assertion follows by side-IH for $\widehat{p}_0$ instead of p_0.
Similarly the assertion follows if p_1 is of a similar form.
Otherwise $p_i = \mathsf{while}\, u_i\, a_i\, \widetilde{p}_i$, with $\widetilde{p}_i\, a_i = \mathrm{inl}\, \langle c_i, \widehat{p}_i \rangle$,
$\mathrm{split}\, p_i = \mathrm{inl}\, \langle c_i, \lambda r.\widehat{p}_i\, r * \lambda x.\mathsf{while}\, u_i\, x\, \widetilde{p}_i \rangle$.
By $p_i \simeq_{A,n+1} p'_i$ there exists $p_{c_0 c_1} : (c_0 =_C c_1)$, and for
$r_0 : R\, c_0$, $r_1 := \mathrm{J}\, C\, R\, c_0\, c_1\, p_{c_0 c_1}\, r_0$ there exist proofs of
$\widehat{p}_0\, r_0 * \lambda x.\mathsf{while}\, u_0\, x\, \widetilde{p}_0 \simeq'_{A,n} \widehat{p}_1\, r_1 * \lambda x.\mathsf{while}\, u_1\, x\, \widetilde{p}_1$.
$\mathrm{split}\, (p_i * q_i) = \mathrm{inl}\, \langle c_i, \lambda r.\widehat{p}_i\, r * \lambda x.\mathsf{while}\, u_i\, x\, (\widetilde{p}_i \circledast q_i) \rangle$
$= \mathrm{inl}\, \langle c_i, \lambda r.\widehat{p}_i\, r * \lambda x.\mathsf{while}\, u_i\, x\, \widetilde{p}_i * q_i \rangle$.
We have to show that for r_0, r_1 as above
$\widehat{p}_0\, r_0 * \lambda x.\mathsf{while}\, u_0\, x\, \widetilde{p}_0 * q_0 \simeq'_n \widehat{p}_1\, r_1 * \lambda x.\mathsf{while}\, u_1\, x\, \widetilde{p}_1 * q_1$.
By IH (c) and symmetry $\widehat{p}_i\, r_i {*} \lambda x.\mathsf{while}\, u_i\, x\, \widetilde{p}_i {*} q_i \simeq'_{B,n} (\widehat{p}_i\, r_i {*} \lambda x.\mathsf{while}\, u_i\, x\, \widetilde{p}_i) {*} q_i$, and by IH (a) $(\widehat{p}_0\, r_0 * \lambda x.\mathsf{while}\, u_0\, x\, \widetilde{p}_0) * q_0 \simeq'_{B,n} (\widehat{p}_1\, r_1 * \lambda x.\mathsf{while}\, u_1\, x\, \widetilde{p}_1) * q_1$.
The assertion follows now by transitivity and symmetry.

(b) If $p_i\, a_i = \mathrm{inr}\, q_i$, then $\mathsf{while}\, u\, a_i\, p_i = q_i$, $q_0 \simeq'_{A,n+1} q_1$.
Otherwise $p_i\, a_i = \mathrm{inl}\, \langle c_i, q_i \rangle$, $\mathrm{split}\, (\mathsf{while}\, u\, a_i\, p_i) = \mathrm{inl}\, \langle c_i, \lambda r.q_i\, r * \lambda x.\mathsf{while}\, u\, x\, p_i \rangle$.
By assumption there exists p_{c_0,c_1} as in (a) and for r_0 and r_1 as in (a) proofs of $q_0\, r_0 \simeq'_{\mathrm{el}\, u} q_1\, r_1$, and furthermore by IH (b) proofs of $\mathsf{while}\, u\, x\, p_0 \simeq'_{A,n} \mathsf{while}\, u\, x\, p_1$ for $x : \mathrm{el}(u)$. The assertion follows by IH (a).

(c) is proved by side-induction on p. If $p = \mathrm{leaf}\, a$ this follows by reflexivity.
Otherwise $p = \mathsf{while}\, u\, a\, \widetilde{p}$, $(p * q) * r = \mathsf{while}\, u\, a\, ((\widetilde{p} \circledast q) \circledast r)$, $p * \lambda x.q\, x * r = \mathsf{while}\, u\, a\, (\widetilde{p} \circledast \lambda x.q\, x \circledast r)$, by side-IH $(\widetilde{p} \circledast q) \circledast r \simeq^{\mathrm{w}\prime}_{D,u,n+1} \widetilde{p} \circledast \lambda x.q\, x \circledast r$, and by (b) for $n + 1$, as just proved, the assertion follows. □

Lemma 4.5. (a) $p * \lambda x.\eta_x \simeq_A p$.
(b) $\mathrm{split}\, (\mathrm{do}\, c\, p) =_{\mathrm{IO}^+\, A + A} \mathrm{inl}\, \langle c, p' \rangle$ for some p' s.t. $\forall r : R\, c.p\, r \simeq_A p'\, r$.
(c) If $p\, a =_{\mathrm{IO}^+\, (\mathrm{el}\, u) + \mathrm{IO}\, A} \mathrm{inl}\, \langle c, q \rangle$ then
$\mathsf{while}\, u\, a\, p \simeq_A \mathrm{do}\, c\, \lambda r.q\, r * \lambda x.\mathsf{while}\, u\, x\, p$.

Proof. (a) follows by straightforward induction on p and Lemma 4.4 (b).
(b) $\mathrm{split}\, (\mathrm{do}\, c\, p) = \langle c, \lambda r.(\mathrm{leaf}\, (\mathrm{inr}\, r)) * \lambda x.\mathsf{while}\, (\widehat{\mathbf{1} + \mathrm{R}\, c})\, x\, q \rangle$
$= \langle c, \lambda r.\mathsf{while}\, (\widehat{\mathbf{1} + \mathrm{R}\, c})\, (\mathrm{inr}\, r)\, q \rangle$,

where q is as in the definition of $(\mathrm{do}\, c\, p)$. For $r : R\, c$ we have by Lemma 4.2 (c) $\mathsf{while}\, (\widehat{\mathbf{1} + \mathrm{R}\, c})\, (\mathrm{inr}\, r)\, q \simeq_A p\, r$.
(c) follows by (b) and $\mathrm{split}\, (\mathsf{while}\, u\, a\, p) = \mathrm{inl}\, \langle c, \lambda r.q(r) * \lambda x.\mathsf{while}\, u\, x\, p\rangle$.

□

5 Conclusion

We have identified a need for a general and workable way of representing and reasoning about interactive programs in dependent type theory. We introduced in dependent type theory the notion of an I/O-tree, parameterised over a world, making essential use of type dependency. We gave it in two forms. The first breaks normalisation, but is conceptually simpler and suitable if one is tolerant of a programming language with 'bottom', or divergent programs. The second preserves normalisation. We called programs of this kind "normalising I/O-programs". We introduced an equality relation identifying behaviourally indistinguishable programs and showed that the monad laws hold, modulo this equality. (For the normalising version these are Lemma 4.2 (a), 4.5 (a) and 4.4 (c)). We introduced while-loops in both versions and repeat-loops and redirect in the first version (and leave it as an interesting exercise to extend the last two constructions to the normalising version). In the non-normalising version the characteristic equations for while and repeat are fulfilled by definition, whereas in the normalising version we have shown them for while (Lemma 4.2 (c) and 4.5 (c)). We have characterised do as well in the latter version (Lemma 4.5 (b)).

In a future paper we will show how to move from one universe to another in the normalising version, and will explore the case in which C is a setoid given with a specific equivalence relation. In addition we will introduce state-dependent I/O-programs, in which the set of commands available depends on the current state of knowledge about the world.

Appendix: notations

In the paper we do not distinguish between Σ and Π-type on the logical framework level and as set-constructions. The empty set is denoted by $\mathbf{0}$, the set containing one element by $\mathbf{1}$ (with element $\bullet$). The set of natural numbers is denoted by $\mathbb{N}$. The injections for the disjoint union $A + B$ of sets A and B are written $\mathrm{inl} : A \to (A + B)$, $\mathrm{inr} : B \to (A + B)$. The elements of $\Sigma x : A.B$ are denoted by $\langle a, b\rangle$. The dependent function type (sometimes written as $\Pi x : A.B$) is denoted by by $(x : A) \to B$, with abbreviations like $(x : A, y : B) \to C$ for $(x : A) \to (y : B) \to C$, $(x : A, B) \to C$ for $(x : A, y : B) \to C$ with y new, and $(x, y : A) \to B$ for $(x : A, y : A) \to B$. We use juxtaposition $(f\, a)$ for application, having a higher precedence than all other operators, so that for example $f\, a = g\, b$ means $(f\, a) = (g\, b)$. The scope of variable-binding operators $\lambda x.$, $\forall x.$, $\exists x.$, $\Sigma x.$ is maximal (so $\lambda x.f\, a =_A b$ stands for $\lambda x.((f\, a) =_A b)$). Some functions are represented as infix operators, writing some of the first few arguments as indices. (For instance we write $p *_{A,B} q$ for $(* \, A \, B \, p \, q)$.) Arguments which are written as indices are often omitted. We will omit the type in equality judgements, writing $r = s$ instead of $r = s : A$. An equation sign $=$ without indices denotes definitional equality, whereas we write $r =_A s$ (never omitting the A) for equality types (which are actually sets). The intensional equality has elimination rule $\mathrm{J} : (A : \mathrm{Set}, B : A \to \mathrm{Set}, a, a' : A, p : (a =_A a'), B\,a) \to B\,a'$. Note that with extensional equality J could trivially be defined as $\lambda A, B, a, a', p, b.\, b$.

References

1. L. Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.
2. P. Hancock and A. Setzer. The IO monad in dependent type theory. DTP'99, `http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99/proceedings.html`, 1999.
3. I. Lindström. A construction of non-well-founded sets within Martin-Löf's type theory. *Journal of Symbolic Logic*, 54(1):57–64, 1989.
4. P. Martin-Löf. Constructive mathematics and computer programming. In J. L. Cohen, J. Łoś, H. Pfeiffer, and K.-D. Podewski, editors, *Proceedings 6th Intl. Congress on Logic, Methodology and Philosophy of Science, Hannover, FRG, 22–29 Aug 1979*, pages 153–175. North Holland, Amsterdam, 1982.
5. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
6. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, Oxford, 1990.
7. 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.
8. P.N.Benton, G.M.Benton, and V.C.V.de Paiva. Computational types from a logical perspective. *J. Functional Programming*, 8(2):177–193, January 1993.
9. P. Wadler. The essence of functional programming. In *19'th Symposium on Principles of Programming Languages, Albuquerque*, volume 19. ACM Press, January 1992.
10. P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.