Programming with Dependent Types – Interactive programs and Coalgebras

Anton Setzer
Swansea University,
Swansea, UK

14 August 2012

1. A Brief Introduction into ML Type Theory

- Martin-Löf type theory = version of predicative dependent type theory.
- As in simple type theory we have judgements of the form

$$s : A$$

"s is of type A".

- Additionally we have judgements of the form

$$A : \mathrm{type}$$

and judgements expressing on the term and type level having α-equivalent normal form w.r.t. reductions.

$$\begin{array}{rcl} s = t & : & A \\ A = B & : & \mathrm{type} \end{array}$$

Logical Framework

- We have a collection of small types:

$$\mathrm{Set} : \mathrm{type}$$

- If $A : \mathrm{Set}$ then $A : \mathrm{type}$.
- If $A = B : \mathrm{Set}$ then $A = B : \mathrm{type}$
- All types used in the following will be elements of Set, except for Set itself and function types which refer to Set.
  - E.g. $A \to \mathrm{Set} : \mathrm{type}$.
- Types will be used for expressiveness (and that's what Martin-Löf intended):
  - Instead of "B is a set depending on $x : A$" we write "$B : A \to \mathrm{Set}$".

Judgements

- E.g.

$$\lambda x.x : \mathbb{N} \rightarrow \mathbb{N}$$

  where $\mathbb{N}$ is the type of natural numbers.
- Because of the higher complexity of the type theory, one doesn't define the valid judgements inductively, but introduces rules for deriving valid judgements.
  - Similar to derivations of propositions.
  - For the main version of ML type theory however, whether $s : A$ is decidable.

Dependent Types

- In ML type theory we have dependent types.
- Simplest example are the type of $n \times m$ matrices $\mathrm{Mat}\ n\ m$.
  - Depends on $n, m : \mathbb{N}$.
- In ordinary programming languages, matrix multiplication can in general not be typed correctly.
  - All we can do is say that it takes two matrices and returns a 3rd matrix.
  - We cannot enforce that the dimensions of the inputs are correct.
- In dependent type theory it can be typed as follows:

$$\mathrm{matmult} : (n, m, k : \mathbb{N}) \to \mathrm{Mat}\ n\ m \to \mathrm{Mat}\ m\ k \to \mathrm{Mat}\ n\ k$$

- Example of dependent function type.

Propositions as Types

- Using the Brouwer-Heyting-Kolmogorov interpretation of the intuitionistic propositions one can now define propositions as types.
- Done in such a way such that ψ is intuitionistically provable iff there exists $p : \psi$.
- For instance, we can define

$$\phi \wedge \psi := \varphi \times \psi$$

- $\varphi \times \psi$ is the product of φ and ψ.
- A proof of $\varphi \wedge \psi$ is a pair $\langle p, q \rangle$ consisting of
  - An element $p : \varphi$, i.e. a proof of φ
  - and an element $q : \psi$, i.e. a proof of ψ.

$\vee$, $\rightarrow$, $\top$, $\neg$

- We can define

$$\phi \vee \psi := \varphi + \psi$$

  - $\varphi + \psi$ is the disjoint union of φ and ψ.
  - A proof of $\varphi \vee \psi$ is
    - $\mathrm{inl}\ p$ for $p : \varphi$ or
    - $\mathrm{inr}\ q$ for $q : \varphi$
  - $\varphi \rightarrow \psi$ is the function type, which maps a proof of φ to a proof of ψ.
  - $\perp$ is the false formula, which has no proof, and we can define

$$\perp := \emptyset$$

  - $\top$ is the true formula, which has exactly one proof, and we can interpret it as the one element set

$$\begin{array}{l} \text{data } \top : \text{Set where} \\ \quad \text{triv} : \top \end{array}$$

  - $\neg\varphi := \varphi \rightarrow \perp$.

Propositions as Types

- We can define

$$\forall x : A.\varphi := (x : A) \to \varphi$$

  - The type of functions, mapping any element $a : A$ to a proof of $\varphi[x := a]$
- We can define

$$\exists x : A.\varphi := (x : A) \times \varphi$$

  - The type of pairs $\langle a, p \rangle$, consisting of an $a : A$ and a $p : \varphi[x := a]$.

Sorting functions

- We can now define, depending on $l : \mathrm{List}\ \mathbb{N}$ the proposition

$$\mathrm{Sorted}\ l$$

- Now we can define

$$\mathrm{sort} : \mathrm{List}\ \mathbb{N} \to (l : \mathrm{List}\ \mathbb{N}) \times \mathrm{Sorted}\ l$$

  which maps lists to sorted lists.
- We can define as well $\mathrm{Eq}\ l\ l'$ expressing that l and l' are lists having the same elements and define even better

$$\mathrm{sort} : (l : \mathrm{List}\ \mathbb{N}) \to (l' : \mathrm{List}\ \mathbb{N}) \times \mathrm{Sorted}\ l \times \mathrm{Eq}\ l\ l'$$

Verified programs

- This allows to define verified programs.
- Usage in critical systems.
- Example, verification of railway interlocking systems (including underground lines).
  - Automatic theorem proving used for proving that concrete interlocking system fulfils signalling principles.
  - Interactive theorem proving used to show that signalling principle imply formalised safety.
  - Interlocking can be run inside Agda without change of language.

Normalisation

- **However**, we need some degree of normalisation, in order to guarantee that $p : \varphi$ implies φ is true.
  - By using full recursion, one can define $p : \varphi$ recursively by defining:

$$\begin{array}{rcl} p & : & \varphi \\ & = & p \end{array}$$

- Therefore most types (except for the dependent function type) in standard ML-type theory correspond to essentially inductive-recursive definitions (an extension of inductive data types).
  - Therefore all data types are well-founded.
- Causes problems since interactive programs correspond to non-well-founded data types.

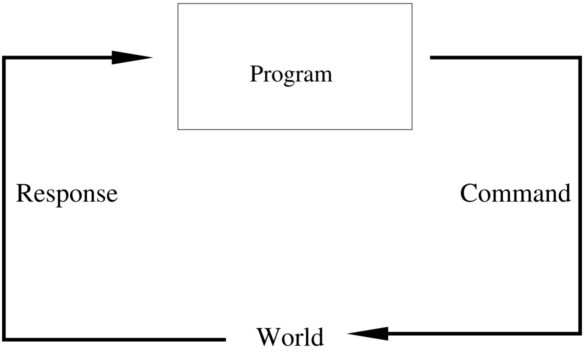
2. Interactive Programs

- Functional programming based on **reduction of expressions**.
- Program is given by an expression which is applied to finitely many arguments.
  The normal form obtained is the result.
- Allows only **non-interactive batch programs** with a fixed number of inputs.
- In order to have interactive programs, something needs to be added to functional programming (constants with side effects, monads, streams, ...).
- We want a solution which exploits the flexibility of dependent types.

Interfaces

- We consider programs which interact with the real world:
  - They issue a command . . .
    (e.g.
    (1) get last key pressed;
    (2) write character to terminal;
    (3) set traffic light to red)
  - . . . and obtain a response, depending on the command . . .
    (e.g.
    - in (1) the key pressed
    - in (2), (3) a trivial element indicating that this was done, or a message indicating success or an error element).

Interactive Programs



Dependent Interfaces

- The set of commands might **vary after interactions.** E.g.
  - after switching on the printer, we can print;
  - after opening a new window, we can communicate with it;
  - if we have tested whether the printer is on, and got a positive answer, we can print on it (increase of knowledge).
- States indicate
  - **principal possibilities of interaction**
    (we can only communicate with an existing window),
  - **objective knowledge**
    (e.g. about which printers are switched on).

Interfaces (Cont.)

- An **interface** is a quadruple $(\mathrm{S}, \mathrm{C}, \mathrm{R}, \mathrm{n})$ s.t.
  - $\mathrm{S} : \mathrm{Set}$.
    - $\mathrm{S} =$ set of **states** which determine the interactions possible.
  - $\mathrm{C} : \mathrm{S} \to \mathrm{Set}$.
    - $\mathrm{C}\ s =$ set of **commands** the program can issue when in state $s : S$.
  - $\mathrm{R} : (s : \mathrm{S}) \to (\mathrm{C}\ s) \to \mathrm{Set}$.
    - $\mathrm{R}\ s\ c =$ set of **responses** the program can obtain from the real world, when having issued command c.
  - $\mathrm{n} : (s : \mathrm{S}) \to (c : \mathrm{C}\ s) \to (r : \mathrm{R}\ s\ c) \to \mathrm{S}$.
    - $\mathrm{n}\ s\ c\ r$ is the **next state** the system is in after having issued command c and received response $r : \mathrm{R}\ s\ c$.

Expl. 1: Interact. with 1 Window

- $\mathrm{S} = \{*\}$.
  - Only one state, no state-dependency.
- $\mathrm{C}\ * = \{\mathrm{getchar}\} \cup \{\mathrm{writechar}\ c \mid c \in \mathrm{Char}\}$.
  - $\mathrm{getchar}$ means: get next character from the keyboard.
  - $\mathrm{writechar}\ c$ means: write character on the window.
- $\mathrm{R}\ *\ \mathrm{getchar} = \mathrm{Char}$.
  - Response of the real world to $\mathrm{getchar}$ is the character code for the key pressed.
- $\mathrm{R}\ *\ (\mathrm{writechar}\ c) = \{*\}$.
  - Response to the request to writing a character is a success message.
- $\mathrm{n}\ *\ c\ r = *$

Ex. 2: Interact. with many Windows

- $\mathrm{S} = \mathbb{N}$.
  - $n : \mathbb{N}$ = number of windows open.
  - Let $\mathrm{Fin}_n := \{0, \ldots, n-1\}$.
- $\mathrm{C}\ n = \{\mathrm{getchar}\}$.

$$\cup\{\mathrm{getselection} \mid n > 0\}$$
$$\cup\{\mathrm{writestring}\ k\ s \mid k \in \mathrm{Fin}_n \wedge s \in \mathsf{String}\}$$
$$\cup\{\mathrm{open}\}$$
$$\cup\{\mathrm{close}\ k \mid k \in \mathrm{Fin}_n\}$$

  - $\mathrm{writestring}\ k\ s$ means: output string s on window k.
  - $\mathrm{getselection}$ means: get the window selected.
  - open means: open a new window.
  - $\mathrm{close}\ k$ means: close the kth window.

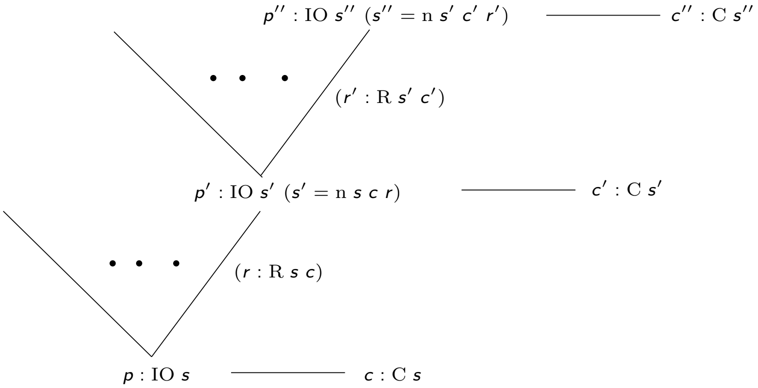
Example 2 (Cont.)

- $$\begin{array}{lcl} \mathrm{R}\ n\ \mathrm{getchar} & = & \mathrm{Char} \\ \mathrm{R}\ n\ \mathrm{getselection} & = & \mathrm{Fin}_n \\ \mathrm{R}\ n\ c & = & \{*\}\ \text{ otherwise} \end{array} .$$
- $$\begin{array}{lcl} \mathrm{n}\ n\ \mathrm{open}\ * & = & n+1 \\ \mathrm{n}\ n\ (\mathrm{close}\ k)\ * & = & n-1 \\ \mathrm{n}\ n\ c\ r & = & n\ \text{ otherwise} \end{array} .$$

3. Weakly Final Coalgebras

- The **interactive programs** for such an interface is given by
  - a family of sets $\mathrm{IO} : \mathrm{S} \to \mathrm{Set}$
    - $\mathrm{IO}\ s =$ set of **interactive programs**, starting in state s;
  - a function $\mathrm{c} : (s : \mathrm{S}) \to \mathrm{IO}\ s \to \mathrm{C}\ s$
    - $\mathrm{c}\ s\ p =$ **command** issued by program p;
  - and a function
    $\mathrm{next} : (s : \mathrm{S}) \to (p : \mathrm{IO}\ s) \to, (r : \mathrm{R}\ s\ (\mathrm{c}\ s\ p)) \to \mathrm{IO}\ (\mathrm{n}\ s\ (\mathrm{c}\ s\ p)\ r)$
    - $\mathrm{next}(s, p, r) =$ **program we execute**, after having obtained for command $\mathrm{c}\ s\ p$ response r.

IO-Trees


$p'' : \mathrm{IO}\ s''\ (s'' = \mathrm{n}\ s'\ c'\ r')$
$c'' : \mathrm{C}\ s''$
$(r' : \mathrm{R}\ s'\ c')$
$p' : \mathrm{IO}\ s'\ (s' = \mathrm{n}\ s\ c\ r)$
$c' : \mathrm{C}\ s'$
$(r : \mathrm{R}\ s\ c)$
$p : \mathrm{IO}\ s$
$c : \mathrm{C}\ s$

Need for Coalgebraic Types

$$\mathrm{IO} : \mathrm{S} \to \mathrm{Set}$$
$$\mathrm{c} : (s : \mathrm{S}) \to \mathrm{IO}\ s \to \mathrm{C}\ s$$
$$\mathrm{next} : (s : \mathrm{S}) \to (p : \mathrm{IO}\ s) \to, (r : \mathrm{R}\ s\ (\mathrm{c}\ s\ p)) \to \mathrm{IO}\ (\mathrm{n}\ s\ (\mathrm{c}\ s\ p)\ r)$$

- We might think we can define $\mathrm{IO}\ s$ as

$$\begin{aligned} &\mathrm{data\ IO} : \mathrm{S} \to \mathrm{Set\ where} \\ &\quad \mathrm{do} : (s : \mathrm{S}) \\ &\qquad \to (c : \mathrm{C}\ s) \\ &\qquad \to ((r : \mathrm{R}\ s\ c) \to \mathrm{IO}\ (\mathrm{n}\ s\ c\ r)) \\ &\qquad \to \mathrm{IO}\ s \end{aligned}$$

- However this is the type of well-founded IO-trees, programs which always terminate.
- Artificial to force programs to always terminate.

Coalgebras

- Instead we use the type of non-well-founded trees, as given by coalgebras.
- We consider first non-state dependent programs.
- So we have as interfaces
  - $\mathrm{C} : \mathrm{Set}$,
  - $\mathrm{R} : \mathrm{C} \to \mathrm{Set}$
- The type of programs for this interface requires
  - $\mathrm{IO} : \mathrm{Set}$,
  - $\mathrm{c} : \mathrm{IO} \to \mathrm{C}$
  - $\mathrm{next} : (p : \mathrm{IO}) \to (r : \mathrm{R}\ (\mathrm{c}\ p)) \to \mathrm{IO}$.

Coalgebras

- Can be combined into
  - $\mathrm{IO} : \mathrm{Set}$.
  - $\mathrm{evolve} : \mathrm{IO} \to (c : \mathrm{C}) \times (\mathrm{R}\ c \to \mathrm{IO})$
- Let $F\ X := (c : \mathrm{C}) \times (\mathrm{R}\ c \to X)$.
- Then need to define $\mathrm{evolveIO} \to F\ \mathrm{IO}$, i.e. an F-coalgebra IO.

Weakly Final Coalgebras

- Having non-terminating programs can be expressed as having a weakly final F-coalgebra:

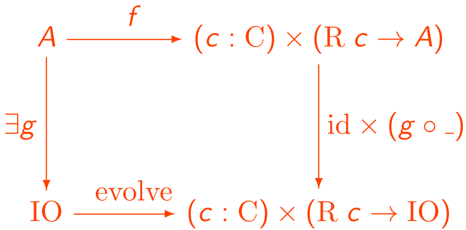
$$
\begin{array}{ccc}
A & \xrightarrow{\quad f \quad} & F\ A \\
{\scriptstyle \exists g}\Big\downarrow & & \Big\downarrow{\scriptstyle F\ g} \\
\mathrm{IO} & \xrightarrow{\quad \mathrm{evolve} \quad} & F\ (\mathrm{IO}\ A)
\end{array}
$$

Weakly Final Coalgebras

- In our example we have

$$\begin{array}{ccc}
A & \xrightarrow{\quad f \quad} & (c : \mathrm{C}) \times (\mathrm{R}\ c \to A) \\
\Big\downarrow{\scriptstyle \exists g} & & \Big\downarrow{\scriptstyle \mathrm{id} \times (g \circ _)} \\
\mathrm{IO} & \xrightarrow{\quad \mathrm{evolve} \quad} & (c : \mathrm{C}) \times (\mathrm{R}\ c \to \mathrm{IO})
\end{array}$$

Guarded Recursion



- If we split f into two functions:

$$\begin{aligned} f_0 &: A \to \mathrm{C} \\ f_1 &: (a : A) \to \mathrm{R}\ (f_0\ a) \to A \end{aligned}$$

and evolve back into

$$\begin{aligned} \mathrm{c} &: \mathrm{IO} \to \mathrm{C} \\ \mathrm{next} &: (p : \mathrm{IO}) \to \mathrm{R} :\ (\mathrm{c}\ a) \to \mathrm{IO} \end{aligned}$$

Guarded Recursion

- we obtain that we can define $g : A \to \mathrm{IO}$ s.t.

$$\begin{array}{lllll} \mathrm{c} & (g\ a) & = & f_0\ a \\ \mathrm{next} & (g\ a) & = & g\ (f_1\ a) \end{array}$$

- Simple form of guarded recursion.

Generalisation

- In case of final coalgebras we can get a more general principle

$$\begin{array}{lll} \mathrm{c} & (g\ a) & = & \text{some } c : \mathrm{C} \text{ depending on a} \\ \mathrm{next} & (g\ a) & = & \left\{ \begin{array}{l} g\ a' \text{ for some } a' \text{ depending on } a \\ \text{or some } p : \mathrm{IO} \text{ depending on } a \end{array} \right. \end{array}$$

- We can't have final coalgebras (uniqueness of g above), since this would result in undecidability of type checking.
- However we can add rules for this and other extended principles.

Desired Notations in Agda

$$
\begin{array}{l}
\text{record IO : Set where} \\
\quad \begin{array}{lcl}
\text{c} & : & \text{IO} \rightarrow \text{C} \\
\text{next} & : & (p : \text{IO}) \rightarrow \text{R}\ (\text{c}\ p) \rightarrow \text{IO}
\end{array}
\end{array}
$$

Example Program

- Assume interface
  - $\mathrm{C} = \{\mathrm{getchar}\} \cup \{\mathrm{writechar}\ c \mid c \in \mathrm{Char}\}$
  - $\mathrm{R}\ \mathrm{getchar} = \mathrm{Char}$,
  - $\mathrm{R}\ (\mathrm{writechar}\ c) = \{*\}$

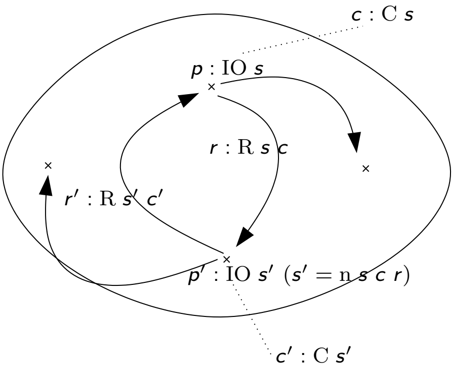
$$
\begin{array}{l}
\mathit{read} : \mathrm{IO} \\
\begin{array}{lll}
\mathrm{c} & \mathit{read} & = & \mathrm{getchar} \\
\mathrm{next} & \mathit{read}\ c & = & \mathit{write}\ c
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{write} : \mathrm{Char} \to \mathrm{IO} \\
\begin{array}{llll}
\mathrm{c} & (\mathit{write}\ c) & = & \mathrm{writechar}\ c \\
\mathrm{next} & (\mathit{write}\ c)\ * & = & \mathit{read}
\end{array}
\end{array}
$$

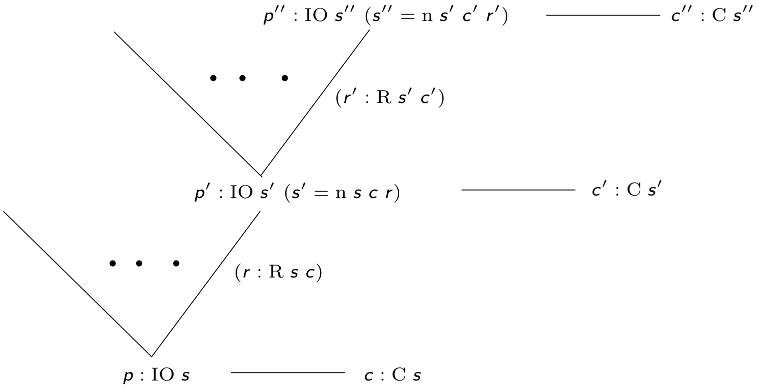
Difference to codata

- Note that in this setting, coalgebras are defined by their elimination rules.
- So they are not introduced by some constructor,
  - "Constructor" can be defined using guarded recursion
- Elements of coalgebras are not infinite objects, but objects which evolves into something infinite.
- No problem of subject reduction problem as it occurs in Coq and in Agda if allowing dependent pattern matching on coalgebras.
- Maybe a more accurate picture are IO graphs which unfold to IO trees.

IO Graphs


$c : \mathrm{C}\ s$
$p : \mathrm{IO}\ s$
$r : \mathrm{R}\ s\ c$
$r' : \mathrm{R}\ s'\ c'$
$p' : \mathrm{IO}\ s'\ (s' = \mathrm{n}\ s\ c\ r)$
$c' : \mathrm{C}\ s'$

IO-Trees



Dependent Coalgebras

- Generalisation to dependent weakly final coalgebras.
  - $\mathrm{S} : \mathrm{Set}$,
  - $\mathrm{C} : \mathrm{S} \to \mathrm{Set}$,
  - $\mathrm{R} : (s : \mathrm{S}) \to \mathrm{C}\ s \to \mathrm{Set}$,
  - $\mathrm{n} : (s : \mathrm{S}) \to (c : \mathrm{C}\ s) \to \mathrm{R}\ s\ c \to \mathrm{S}$,

$$
\begin{array}{l}
\mathrm{record\ IO} : \mathrm{S} \to \mathrm{Set\ \ where} \\
\quad \begin{array}{lcl}
\mathrm{c} & : & \mathrm{IO}\ s \to \mathrm{C}\ s \\
\mathrm{next} & : & (p : \mathrm{IO}\ s) \to (r : \mathrm{R}\ s\ (\mathrm{c}\ s\ p)) \to \mathrm{IO}\ (\mathrm{n}\ s\ (\mathrm{c}\ s\ p)\ r)
\end{array}
\end{array}
$$

Example

- Assume the interface interacting with arbitrarily many windows.
- We can define a function, which
  - when the user presses key 'o' will open a window,
  - when the user presses key 'c' and has at least two open windows, get the selection of a window by the user and will close it

$$\mathit{start} : (n : \mathbb{N}) \to \mathrm{IO}\ n$$

$$\begin{array}{lll} \mathrm{c}\ \ (\mathit{start}\ n) & = & \mathrm{getchar} \\ \mathrm{next}\ (\mathit{start}\ (n+2))\ '\mathrm{c}' & = & \mathit{close}\ n \\ \mathrm{next}\ (\mathit{start}\ n)\ '\mathrm{o}' & = & \mathit{open}\ n \\ \mathrm{next}\ (\mathit{start}\ n)\ x & = & \mathit{start}\ n \end{array}$$

$$\mathit{close} : (n : \mathbb{N}) \to \mathrm{IO}\ (n+2)$$

$$\begin{array}{lll} \mathrm{c}\ \ (\mathit{close}\ n) & = & \mathrm{getselection} \\ \mathrm{next}\ (\mathit{close}\ n)\ k & = & \mathit{close}'\ n\ k \end{array}$$

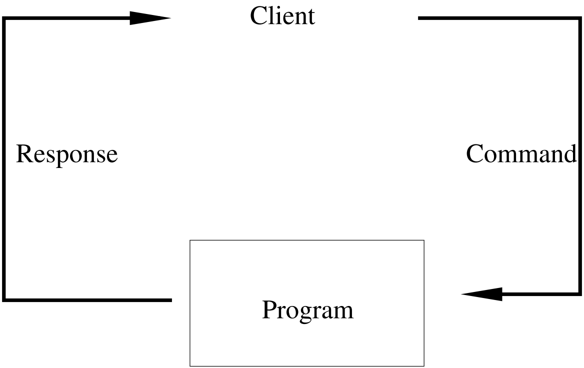
$$\mathit{close}' : (n : \mathbb{N}) \to (k : \mathrm{Fin}_n) \to \mathrm{IO}\ (n+2)$$

$$\begin{array}{lll} \mathrm{c}\ \ (\mathit{close}'\ n\ k) & = & \mathrm{close}\ k \\ \mathrm{next}\ (\mathit{close}'\ n\ k)\ k & = & \mathit{start}\ (n+1) \end{array}$$

$$\mathit{open} : (n : \mathbb{N}) \to \mathrm{IO}\ n$$

$$\begin{array}{lll} \mathrm{c}\ \ (\mathit{open}\ n) & = & \mathrm{open} \\ \mathrm{next}\ (\mathit{open}\ n)\ * & = & \mathit{start}\ (n+1) \end{array}$$

Server-Side Programs



GUIs

- In object-oriented programming, GUIs are treated in server-side style:
  - With each event (e.g. button click) an event handler is associated.
  - When the event occurs, the corresponding event handler is activated, which carries out some calculations, possibly modifies the interface and then waits for the next event.
  - So $\mathrm{C}\ s$ = set of events in state s.
  - $\mathrm{R}\ s\ c$ = possible modifications of the GUI the program can execute.
  - $\mathrm{n}\ s\ c\ r$ = next state of the GUI after this interaction.

IO-Monad

- By adding leaves labelled by A to IO-trees we can define the IO-monad

$$\mathrm{IO}\ (A : \mathrm{Set}) : \mathrm{Set}$$

together with operations

$$\begin{array}{lcl} \eta & : & A \to \mathrm{IO}\ A \\ * & : & \mathrm{IO}\ A \to (A \to \mathrm{IO}\ B) \to \mathrm{IO}\ B \end{array}$$

Compilation

- We can define a horizontal transformation from a $(\mathrm{S}, \mathrm{C}, \mathrm{R}, \mathrm{n})$-program into a $(\mathrm{S}', \mathrm{C}', \mathrm{R}', \mathrm{n}')$-program:
  - Assume

$$\mathit{translatec} : (s : \mathrm{S}) \to (c : \mathrm{C}\ s) \to \mathrm{IO}_{\mathrm{S}',\mathrm{C}',\mathrm{R}',\mathrm{n}'}\ s\ (\mathrm{R}\ c)$$

Then we can define

$$\mathit{translate} : \mathrm{IO}_{\mathrm{S},\mathrm{C},\mathrm{R},\mathrm{n}} \to \mathrm{IO}_{\mathrm{S}',\mathrm{C}',\mathrm{R}',\mathrm{n}'}$$

by replacing $c : \mathrm{C}$ by an execution of *translatec* c.

4. Coalgebras and Bisimulation

- For simplicity consider non-state dependent IO-trees.
- Two IO-trees are the same, if their commands are the same and for every response to it, the resulting IO-trees are bisimilar.
- Because of non-well-foundedness of trees we need non-well-foundedness of bisimulation.

Definition of Bisimulation

$$
\begin{array}{l}
\text{mutual} \\
\quad \text{record } \_ \sim \_ : \mathrm{IO} \to \mathrm{IO} \to \mathrm{Set} \text{ where} \\
\quad\quad \text{toproof} : (p, p' : \mathrm{IO}) \\
\quad\quad\quad \to p \sim p' \\
\quad\quad\quad \to \mathrm{Bisimaux}\ (\mathrm{c}\ p)\ (\mathrm{next}\ p)\ (\mathrm{c}\ p')\ (\mathrm{next}\ p') \\
\\
\quad \text{data Bisimaux} : (c : \mathrm{C}) \\
\quad\quad \to (next : \mathrm{R}\ c \to \mathrm{IO}) \\
\quad\quad \to (c' : \mathrm{C}) \\
\quad\quad \to (next' : \mathrm{R}\ c' \to \mathrm{IO}) \\
\quad\quad \to \mathrm{Set} \quad\quad\quad\quad \text{where} \\
\quad\quad \text{eq} : (c : \mathrm{C}) \\
\quad\quad\quad \to (next, next' : \mathrm{R}\ c \to \mathrm{IO}) \\
\quad\quad\quad \to (p : (r : \mathrm{R}\ c) \to (next\ r) \sim (next'\ r)) \\
\quad\quad\quad \to \mathrm{Bisimaux}\ c\ next\ c\ next'
\end{array}
$$

Conclusion

- Introduction of state-dependent interactive programs.
- Coalgebras defined by their elimination rules.
- Categorical diagram corresponds exactly to guarded recursion.
- IO-monad definable.
- Compilation.
- Bisimilarity as a state-dependent weakly final coalgebra.