Programming with Dependent Types – Interactive programs and Coalgebras

Anton Setzer
Swansea University,
Swansea, UK

14 August 2012
A Brief Introduction into ML Type Theory

Interactive Programs in Dependent Type Theory

Weakly Final Coalgebras

More on IO

Coalgebras and Bisimulation
A Brief Introduction into ML Type Theory

Interactive Programs in Dependent Type Theory

Weakly Final Coalgebras

More on IO

Coalgebras and Bisimulation
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
  \[ A : \text{type} \]
  and judgements expressing on the term and type level having $\alpha$-equivalent normal form w.r.t. reductions.
  \[ s = t : A \]
  \[ A = B : \text{type} \]
Logical Framework

- We have a collection of small types:
  
  \[ \text{Set} : \text{type} \]

- If \( A : \text{Set} \) then \( A : \text{type} \).

- If \( A = B : \text{Set} \) then \( A = B : \text{type} \).

- All types used in the following will be elements of \( \text{Set} \), except for \( \text{Set} \) itself and function types which refer to \( \text{Set} \).
  
  - E.g. \( A \rightarrow \text{Set} : \text{type} \).

- 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 “\( B : A \rightarrow \text{Set} \)”.
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 $\text{Mat} \ n \ m$.
  - 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:

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

- 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 $\psi$ is intuitionistically provable iff there exists $p : \psi$.
- For instance, we can define

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

- $\varphi \times \psi$ is the product of $\varphi$ and $\psi$.
- A proof of $\varphi \land \psi$ is a pair $\langle p, q \rangle$ consisting of
  - An element $p : \varphi$, i.e. a proof of $\varphi$
  - and an element $q : \psi$, i.e. a proof of $\psi$. 
\textbf{∨, →, ⊤, ¬}

- We can define
  \[
  \phi \lor \psi := \phi + \psi
  \]

- \(\phi + \psi\) is the disjoint union of \(\phi\) and \(\psi\).
- A proof of \(\phi \lor \psi\) is
  - \(\text{inl} \ p\) for \(p : \phi\) or
  - \(\text{inr} \ q\) for \(q : \phi\)
- \(\phi \to \psi\) is the function type, which maps a proof of \(\phi\) to a proof of \(\psi\).
- \(\bot\) is the false formula, which has no proof, and we can define
  \[
  \bot := \emptyset
  \]
- \(\top\) is the true formula, which has exactly one proof, and we can interpret it as the one element set
  
  ```
  
  data \top : \text{Set} \text{where}
  
  triv : \top
  ```
- \(\neg \phi := \phi \to \bot\).
Propositions as Types

- We can define
  \[ \forall x : A. \varphi := (x : A) \rightarrow \varphi \]

  - 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 \( l : \text{List} \mathbb{N} \) the proposition
  \[
  \text{Sorted } l
  \]

- Now we can define
  
  \[
  \text{sort} : \text{List} \mathbb{N} \rightarrow (l : \text{List} \mathbb{N}) \times \text{Sorted } l
  \]
  
  which maps lists to sorted lists.

- We can define as well \( \text{Eq } l \ l' \) expressing that \( l \) and \( l' \) are lists having the same elements and define even better

  \[
  \text{sort} : (l : \text{List} \mathbb{N}) \rightarrow (l' : \text{List} \mathbb{N}) \times \text{Sorted } l \times \text{Eq } l \ l'
  \]
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.
However, we need some degree of normalisation, in order to guarantee that $p : \varphi$ implies $\varphi$ is true.

- By using full recursion, one can define $p : \varphi$ recursively by defining:

$$p : \varphi = p$$

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.
A Brief Introduction into ML Type Theory

Interactive Programs in Dependent Type Theory

Weakly Final Coalgebras

More on IO

Coalgebras and Bisimulation
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.
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

Program

Response

Command

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

- **\(S : \text{Set.}\)**
  - \(S\) = set of **states** which determine the interactions possible.
- **\(C : S \rightarrow \text{Set.}\)**
  - \(C s\) = set of **commands** the program can issue when in state \(s : S\).
- **\(R : (s : S) \rightarrow (C s) \rightarrow \text{Set.}\)**
  - \(R s c\) = set of **responses** the program can obtain from the real world, when having issued command \(c\).
- **\(n : (s : S) \rightarrow (c : C s) \rightarrow (r : R s c) \rightarrow S.\)**
  - \(n s c r\) is the **next state** the system is in after having issued command \(c\) and received response \(r : R s c\).
Expl. 1: Interact. with 1 Window

- $S = \{\ast\}$. Only one state, no state-dependency.
- $C \ast = \{\text{getchar}\} \cup \{\text{writechar } c \mid c \in \text{Char}\}$. 
  - getchar means: get next character from the keyboard.
  - writechar $c$ means: write character on the window.
- $R \ast \text{ getchar } = \text{Char}$. Response of the real world to getchar is the character code for the key pressed.
- $R \ast (\text{writechar } c) = \{\ast\}$. Response to the request to writing a character is a success message.
- $n \ast c \; r = \ast$
Ex. 2: Interact. with many Windows

- \( S = \mathbb{N} \).
  - \( n : \mathbb{N} \) = number of windows open.
  - Let \( \text{Fin}_n := \{0, \ldots, n-1\} \).
- \( C \ n = \{\text{getchar}\} \)
  - \( \cup \{\text{getselection} \mid n > 0\} \)
  - \( \cup \{\text{writestring} \ k \ s \mid k \in \text{Fin}_n \land s \in \text{String}\} \)
  - \( \cup \{\text{open}\} \)
  - \( \cup \{\text{close} \ k \mid k \in \text{Fin}_n\} \)
- \( \text{writestring} \ k \ s \) means: output string \( s \) on window \( k \).
- \( \text{getselection} \) means: get the window selected.
- \( \text{open} \) means: open a new window.
- \( \text{close} \ k \) means: close the \( k \)th window.
Example 2 (Cont.)

\[
\begin{align*}
\text{R } n \ & \text{ getchar} \quad = \quad \text{Char} \\
\text{R } n \ & \text{ getselection} \quad = \quad \text{Fin}_n \\
\text{R } n \ & \text{ c} \quad = \quad \{\ast\} \quad \text{otherwise} \\
\text{n } n \ & \text{ open } \ast \quad = \quad n + 1 \\
\text{n } n \ & (\text{close } k) \ast \quad = \quad n - 1 \\
\text{n } n \ & \text{ c } r \quad = \quad n \quad \text{otherwise}
\end{align*}
\]
A Brief Introduction into ML Type Theory

Interactive Programs in Dependent Type Theory

Weakly Final Coalgebras

More on IO

Coalgebras and Bisimulation
3. Weakly Final Coalgebras

The interactive programs for such an interface is given by

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

**IO-Trees**

\[ p'' : \text{IO} \ s'' \ (s'' = n \ s' \ c' \ r') \quad \quad \quad c'' : \text{C} \ s'' \]

\[ p' : \text{IO} \ s' \ (s' = n \ s \ c \ r) \quad \quad \quad c' : \text{C} \ s' \]

\[ p : \text{IO} \ s \quad \quad \quad c : \text{C} \ s \]

\[ (r' : \text{R} \ s' c') \]

\[ (r : \text{R} \ s \ c) \]
Need for Coalgebraic Types

\[ \text{IO} : S \rightarrow \text{Set} \]
\[ c : (s : S) \rightarrow \text{IO } s \rightarrow C \ s \]
\[ \text{next} : (s : S) \rightarrow (p : \text{IO } s) \rightarrow, (r : R \ s \ (c \ s \ p)) \rightarrow \text{IO } (n \ s \ (c \ s \ p) \ r) \]

- We might think we can define IO s as
  
  \[
  \text{data IO : S \rightarrow Set where}
  \]
  
  \[
  \text{do : } (s : S) \\
  \rightarrow (c : C \ s) \\
  \rightarrow ((r : R \ s \ c) \rightarrow \text{IO } (n \ s \ c \ r)) \\
  \rightarrow \text{IO } s
  \]

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

- Artificial to force programs to always terminate.
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

- \( C : \text{Set}, \)
- \( R : C \to \text{Set} \)

The type of programs for this interface requires

- \( IO : \text{Set}, \)
- \( c : IO \to C \)
- \( \text{next} : (p : IO) \to (r : R \, (c \, p)) \to IO. \)
Can be combined into

- $\text{IO} : \text{Set}$.
- $\text{evolve} : \text{IO} \to (c : C) \times (R \ c \to \text{IO})$

Let $F \ X := (c : C) \times (R \ c \to X)$.

Then need to define $\text{evolveIO} \to F \ \text{IO}$, i.e. an $F$-coalgebra $\text{IO}$. 
Having non-terminating programs can be expressed as having a weakly final $F$-coalgebra:

\[
\exists g \quad \text{evolve} \quad F (\text{IO } A)
\]
In our example we have

\[ A \xrightarrow{f} (c : C') \times (R c \to A) \]

\[ \exists g \]

\[ \xrightarrow{\text{evolve}} \]

\[ \text{IO} \]

\[ \xrightarrow{\text{id} \times (g \circ \_)} \]

\[ (c : C) \times (R c \to \text{IO}) \]
Guarded Recursion

If we split $f$ into two functions:

$$f_0 : A \rightarrow C$$
$$f_1 : (a : A) \rightarrow R (f_0 a) \rightarrow A$$

and evolve back into

$$c : IO \rightarrow C$$
$$next : (p : IO) \rightarrow R : (c a) \rightarrow IO$$
we obtain that we can define $g : A \to \text{IO}$ s.t.

\[
\begin{align*}
c (g \ a) &= f_0 \ a \\
\text{next} (g \ a) &= g (f_1 \ a)
\end{align*}
\]

Simple form of guarded recursion.
In case of final coalgebras we can get a more general principle

\[
\begin{align*}
c(g\ a) & = \text{some } c : C \text{ depending on } a \\
\text{next } (g\ a) & = \begin{cases} 
g\ a' \text{ for some } a' \text{ depending on } a \\
\text{or some } p : \text{IO} \text{ depending on } a
\end{cases}
\end{align*}
\]

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.
record IO : Set where
  c      : IO → C
  next   : (p : IO) → R (c p) → IO
Example Program

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

\[
\text{read} : \text{IO} \\
c \quad \text{read} \quad = \quad \text{getchar} \\
\text{next read } c \quad = \quad \text{write } c
\]

\[
\text{write} : \text{Char} \to \text{IO} \\
c \quad (\text{write } c) \quad = \quad \text{writechar } c \\
\text{next } (\text{write } c)\ast \quad = \quad \text{read}
\]
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
IO-Trees

\[ p'' : \text{IO } s'' \quad (s'' = n \ s' \ c' \ r') \]

\[ c'' : \text{C } s'' \]

\[ (r' : \text{R} \ s' \ c') \]

\[ p' : \text{IO } s' \quad (s' = n \ s \ c \ r) \]

\[ c' : \text{C } s' \]

\[ (r : \text{R} \ s \ c) \]

\[ p : \text{IO } s \quad c : \text{C } s \]
Dependent Coalgebras

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

  record IO : S → Set where
  c : IO s → C s
  next : (p : IO s) → (r : R s (c s p)) → IO (n s (c s p) r)
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
\begin{align*}
\text{start} : (n : \mathbb{N}) &\rightarrow \text{IO } n \\
c \ (\text{start } n) &\quad = \quad \text{getchar} \\
\text{next} \ (\text{start } (n + 2)) \ 'c' &\quad = \quad \text{close } n \\
\text{next} \ (\text{start } n) \ 'o' &\quad = \quad \text{open } n \\
\text{next} \ (\text{start } n) \ x &\quad = \quad \text{start } n \\
\text{close} : (n : \mathbb{N}) &\rightarrow \text{IO } (n + 2) \\
c \ (\text{close } n) &\quad = \quad \text{getselection} \\
\text{next} \ (\text{close } n) \ k &\quad = \quad \text{close}' \ n \ k \\
\text{close}' : (n : \mathbb{N}) &\rightarrow (k : \text{Fin}_n) \rightarrow \text{IO } (n + 2) \\
c \ (\text{close}' \ n \ k) &\quad = \quad \text{close } k \\
\text{next} \ (\text{close}' \ n \ k) \ k &\quad = \quad \text{start } (n + 1) \\
\text{open} : (n : \mathbb{N}) &\rightarrow \text{IO } n \\
c \ (\text{open } n) &\quad = \quad \text{open} \\
\text{next} \ (\text{open } n) \ * &\quad = \quad \text{start } (n + 1)
\end{align*}
A Brief Introduction into ML Type Theory

Interactive Programs in Dependent Type Theory

Weakly Final Coalgebras

More on IO

Coalgebras and Bisimulation
Server-Side Programs

Response

Client

Program

Command
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 \( C_s \) = set of events in state \( s \).
- \( R_s c \) = possible modifications of the GUI the program can execute.
- \( n s c r \) = 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

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

together with operations

$$\eta : A \to \text{IO } A$$

$$\star : \text{IO } A \to (A \to \text{IO } B) \to \text{IO } B$$
We can define a horizontal transformation from a $(S, C, R, n)$-program into a $(S', C', R', n')$-program:

Assume

\[
\text{translate}_c : (s : S) \rightarrow (c : C \ s) \rightarrow \text{IO}_{S', C', R', n'} \ s \ (R \ c)
\]

Then we can define

\[
\text{translate} : \text{IO}_{S, C, R, n} \rightarrow \text{IO}_{S', C', R', n'}
\]

by replacing $c : C$ by an execution of $\text{translate}_c \ c$. 
A Brief Introduction into ML Type Theory

Interactive Programs in Dependent Type Theory

Weakly Final Coalgebras

More on IO

Coalgebras and Bisimulation
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

mutual
record _ ∼ _ : IO → IO → Set where
toproof : (p, p′ : IO)
  → p ∼ p′
  → Bisimaux (c p) (next p) (c p′) (next p′)

data Bisimaux : (c : C)
  → (next : R c → IO)
  → (c′ : C)
  → (next′ : R c′ → IO)
  → Set
  where

eq : (c : C)
  → (next, next′ : R c → IO)
  → (p : (r : R c) → (next r) ∼ (next′ r))
  → Bisimaux c next c next′
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.