Interactive Programs

in

Dependent Type Theory

Peter Hancock, Edinburgh
Anton Setzer, Uppsala

1. Motivation.
2. IO-trees.
3. Repeat and redirect.
4. Normalizing version.

1. Motivation

Problem: Ordinary programs in type theory are functions.

- One input
- One output.

Goal: Addition of real interactive programs, eg. an editor.

Concepts for I/O in Functional Programming:

1 Streams.

$\mathrm{IO} = (\mathbb{N} \to \text{Inputtype}) \to (\mathbb{N} \to \text{Outputtype})$.

- Conceptual problems.
- Timing of input/output depends on the evaluation strategy.

2) Constants with side-effects

- Type checking might invoke interaction.

3) IO monad.

- I/O in Haskell.
- Approach taken here.

The IO-monad

IO-monad = triple $(\mathrm{IO}, \eta, *)$, s.t.

- $\mathrm{IO}(A)$: Set (A : Set),
  - Set of interactive program **(-texts)**, which, if they terminate, have as result some $a : A$.
- $\eta_a^A : \mathrm{IO}(A)$ (A : Set, $a : A$)
  - Program with no interaction and result a.
- $p *_{A,B} q : \mathrm{IO}(B)$
  $(A, B : \mathrm{Set}, p : \mathrm{IO}(A), q : A \to \mathrm{IO}(B))$
  - Composition of programs.
  - Starts with p.
  - If termination with result $a : A$, program continues with $q(a)$.
- Additional programs for specific interactions.

2. IO-trees

Higher generality using dependent types:

A **world** w is a pair (C, R) s.t.
- C : Set (Commands).
- $R : C \to$ Set (responses to a command).

Assume $w = (C, R)$ a world.

Goal: Represent IO(A) as a data type existing in dependent type theory.

$\mathbf{IO_w(A)}$ is the set of
(possibly non-well-founded) trees with
- leaves in A.
- nodes marked with elements of C.
- nodes marked with c have branching degree $R(c)$.

$$\frac{A : \mathsf{Set}}{\mathsf{IO}_w(A) : \mathsf{Set}} \qquad \frac{a : A}{\mathsf{leaf}(a) : \mathsf{IO}_w(A)}$$

$$\frac{c : C \qquad p : R(c) \to \mathsf{IO}_w(A)}{\mathsf{do}(c, p) : \mathsf{IO}_w(A)}$$

Definition of η, $*$

$\eta_a = \mathsf{leaf}(a)$.
$\mathsf{leaf}(a) * q = q(a)$.
$\mathsf{do}(c, p) * q = \mathsf{do}(c, \lambda x.(p(x) * q))$.

New operation: execute

Status: External, like "normalize".

Let w_0 be a fixed world (real commands).

execute applied to $p : \mathrm{IO}_{w_0}(A)$ does the following:

- It reduces p to canonical form.
- If $p = \mathsf{leaf}(a)$ it terminates and returns a.
- If $p = \mathsf{do}(c, q)$, then it
  - issues command c;
  - interprets the response as an element $r : R(c)$;
  - then continues with $q(r)$.

3. Repeat, Redirect

3.1. Repeat

Construction for defining infinitely looping programs.
Assume B : Set, $p : B \to \mathrm{IO}(A + B)$, $b : B$.

$\mathbf{repeat(B, p, b)}$ is the following program:

- It computes as $p(b)$.
- If $p(b)$ terminates with result inl(a), it stops with result a.
- If $p(b)$ terminates with result inr(b'), it continues with repeat(B, p, b').

3.2. Redirect

Assume

- $w = (C, R)$, $w' = (C', R')$ are worlds.
- A : Set,
- p : $\mathrm{IO}_w(A)$.
- q : $(c : C) \to \mathrm{IO}_{w'}(R(c))$.

Define $\mathbf{redirect(p, q)}$: $\mathrm{IO}_{w'}A$:

redirect(leaf(a), q) = leaf(a).
redirect(do(c, p), q) = $q(c) * \lambda x$.redirect($p(x), q$).

(More precisely $q(c)$ should be non-leaf tree).

Possibility of writing **libraries.**

Problem: No normalization

Let $A = C = \mathsf{N}$, $\mathsf{R}(c)$ arbitrary.
Assume $f : \mathsf{N} \to \mathsf{N}$.

$$\begin{array}{rcl} p_f & := & \lambda n.\mathsf{do}(f(n), \lambda x.\mathsf{leaf}(\mathsf{inr}(n+1))) \\ & : & \mathsf{N} \to \mathsf{IO}(A + \mathsf{N}). \\ q_f & := & \mathsf{repeat}(B, p_f, 0) \\ & \longrightarrow & \mathsf{do}(f(0), \lambda x.\mathsf{do}(f(1), \lambda y.\mathsf{do}(f(2), \lambda z.\cdots))) \\ & : & \mathsf{IO}(A) \end{array}$$

Observe: Such programs should be definable in the current context.
q_f reduces to head normal form.

Consequence: Type-checking undecidable:

$\lambda B, x.x : (B : \mathsf{IO}(A) \to \mathsf{Set}, B(q_f)) \to B(q_g)$
iff f and g are extensionally equal.

4. Normalizing Version

Add repeat as a constructor
(In the paper while was chosen):

$$\frac{A : \mathsf{Set}}{\mathrm{IO}_w(A) : \mathsf{Set}} \qquad \frac{a : A}{\mathsf{leaf}(a) : \mathrm{IO}(A)}$$

$$\frac{c : C \qquad p : R(c) \to \mathrm{IO}(A)}{\mathsf{do}(c, p) : \mathrm{IO}(A)}$$

$$\frac{B : \mathsf{Set} \qquad p : B \to \mathrm{IO}(A + B) \qquad b : B}{\mathsf{repeat}(B, p, b) : \mathrm{IO}(A)}$$

η, $*$ are now definable.

Split:

split : $\mathrm{IO}(A) \to A + \Sigma c : C.(R(c) \to \mathrm{IO}(A))$
which simulates the decomposition of a non-well-founded tree into the arguments of its constructor.

Execute(p) does now the following:

- If split(p) = inl(a), then terminate
  with result a.
- If split(p) = inr($<c, q>$), then carry out
  command c, get response r and continue
  with $q(r)$.

Example

$$\begin{array}{rcl} \mathsf{quote}(a) & := & \mathsf{leaf}(\mathsf{inr}(a)). \\ \mathsf{terminate}(a) & := & \mathsf{leaf}(\mathsf{inl}(a)). \end{array}$$

```
minieditor=
repeat
(data main(s:string) | onez(s:string))
(λx.case x of
main(s)
  → do(getchar, λa.
        do(writechar(a), λ_.
          if    a='z'
          then  quote(onez(s))
          else  quote(main(append(s, [a])))))
onez(s)
   → do(getchar, λa.
         if    a='z'
         then  do(deletechar, λ_.
                 terminate(s))
         else  do(writechar(a), λ_.
                 quote(main(append(s, ['z', a]))))))
main("")
:IO(string)
```

Normalizing IO Programs

- In the final version all derivable terms are strongly normalizing.
- Therefore after every interaction execute terminates or has a next interaction.
- However, execute might carry out infinitely many IO-commands.
- Notion of “strongly-normalizing IO-programs”.