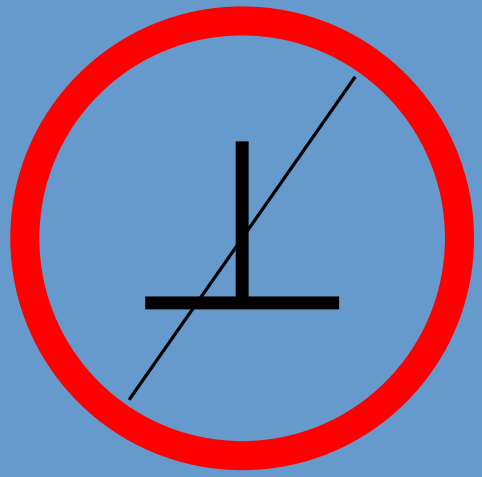
The University of Nottingham

Stop thinking about bottoms when writing programs . . .



Thorsten Altenkirch
University of Nottingham

Trouble with $\perp$

Trouble with $\bot$

$$
\begin{aligned}
&(*) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&0 \qquad\quad * n = 0 \\
&(m+1) * n = m * n + n
\end{aligned}
$$

Trouble with $\perp$

$$(*) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

$$0 \qquad * n = 0$$

$$(m+1) * n = m * n + n$$

$$x * y = y * x \qquad ?$$

Trouble with $\bot$

$$
\begin{aligned}
&(*) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&0 \qquad\quad * n = 0 \\
&(m+1) * n = m * n + n
\end{aligned}
$$

$$x * y = y * x \qquad ?$$

No, because

$$
\begin{aligned}
0 * \bot &= 0 \\
\bot * 0 &= \bot
\end{aligned}
$$

Trouble with $\perp$ . . .

Trouble with $\perp$ . . .

- Many useful algebraic properties do not hold.

Trouble with $\perp$ . . .

- Many useful algebraic properties do not hold.
- Correctness proofs get obliterated with reasoning about $\perp$.

Trouble with $\perp$ . . .

- Many useful algebraic properties do not hold.
- Correctness proofs get obliterated with reasoning about $\perp$.
- Do we actually care about non-terminating programs?

Trouble with $\perp$ . . .

- Many useful algebraic properties do not hold.
- Correctness proofs get obliterated with reasoning about $\perp$.
- Do we actually care about non-terminating programs?
- Programs are **not** natural phenomena. . .

Trouble with $\perp$ . . .

- Many useful algebraic properties do not hold.
- Correctness proofs get obliterated with reasoning about $\perp$.
- Do we actually care about non-terminating programs?
- Programs are **not** natural phenomena. . .
- Programs are **constructed**!

Do we need $\perp$ to be lazy?

Do we need $\perp$ to be lazy?

$$from :: \mathbb{N} \to [\mathbb{N}]$$

$$from\ n = n : (from\ (n + 1))$$

Do we need $\perp$ to be lazy?

$$from :: \mathbb{N} \to [\mathbb{N}]$$
$$from\ n = n : (from\ (n+1))$$

- *from* is total, **if** we interpret lists as a terminal coalgebra.

$$[A] = \nu X.1 + A \times X$$

data vs codata

data vs codata

$$
\begin{aligned}
&evenLength :: [a] \to \mathbf{Bool} \\
&evenLength\ [\,] \qquad\quad = \mathrm{True} \\
&evenLength\ (a : as) = \neg\ (evenLength\ n)
\end{aligned}
$$

data vs codata

$$
\begin{aligned}
&\mathit{evenLength} :: [a] \to \mathbf{Bool} \\
&\mathit{evenLength}\ [\,] \qquad\quad = \mathrm{True} \\
&\mathit{evenLength}\ (a : as) = \neg\ (\mathit{evenLength}\ n)
\end{aligned}
$$

- $\mathit{evenLength}$ is total, ...

data vs codata

$$
\begin{aligned}
&\mathit{evenLength} :: [a] \to \mathbf{Bool} \\
&\mathit{evenLength}\ [\,] \qquad\quad = \mathrm{True} \\
&\mathit{evenLength}\ (a : as) = \neg\ (\mathit{evenLength}\ n)
\end{aligned}
$$

- $\mathit{evenLength}$ is total, ...
- **if** we interpret lists as initial algebra:

$$[A] = \mu X.1 + A \times X$$

data vs codata

$$
\begin{aligned}
&\mathit{evenLength} :: [a] \to \mathbf{Bool} \\
&\mathit{evenLength}\ [\,] \qquad\quad = \mathrm{True} \\
&\mathit{evenLength}\ (a : as) = \neg\ (\mathit{evenLength}\ n)
\end{aligned}
$$

- $\mathit{evenLength}$ is total, ...
- **if** we interpret lists as initial algebra:

$$[A] = \mu X.1 + A \times X$$

- Problem:

$$\mathit{evenLength}\ (\mathit{from}\ 0) = \bot$$

data vs codata

data vs codata

- Finite lists

$$\mathbf{data}\ [a] = [\,] \mid a : [a]$$

data vs codata

- Finite lists

  $$\mathbf{data}\ [a] = [\,] \mid a : [a]$$

- Potentially infinite lists:

  $$\mathbf{codata}\ [a]^{\omega}\ \ a = [\,] \mid a : [a]^{\omega}$$

data vs codata

- Finite lists

  $$\mathbf{data}\ [a] = [\,] \mid a : [a]$$

- Potentially infinite lists:

  $$\mathbf{codata}\ [a]^{\omega}\ \ a = [\,] \mid a : [a]^{\omega}$$

- Better types

  $$\begin{aligned} \mathit{from} &:: \mathbb{N} \to [a]^{\omega} \\ \mathit{evenLength} &:: [a] \to \mathbf{Bool} \end{aligned}$$

data vs codata

- Finite lists
  $$\mathbf{data}\ [a] = [\,] \mid a : [a]$$
- Potentially infinite lists:
  $$\mathbf{codata}\ [a]^{\omega}\ \ a = [\,] \mid a : [a]^{\omega}$$
- Better types
  $$\begin{aligned} \mathit{from} &:: \mathbb{N} \to [a]^{\omega} \\ \mathit{evenLength} &:: [a] \to \mathbf{Bool} \end{aligned}$$
- $\mathit{evenLength}\ (\mathit{from}\ 0)$ doesn't typecheck.

Can we always avoid $\perp$?

Can we always avoid $\bot$?

$$\mathbf{data}\ \mathbf{SK} = \mathrm{S} \mid \mathrm{K} \mid \mathbf{SK} : @\ \mathbf{SK}$$

$$\begin{aligned}
&nf :: \mathbf{SK} \to \mathbf{SK} \\
&nf\ \mathrm{S} &&= \mathrm{S} \\
&nf\ \mathrm{K} &&= \mathrm{K} \\
&nf\ (t : @\ u) &&= (nf\ t)@(nf\ u)
\end{aligned}$$

$$\begin{aligned}
&(@) :: \mathbf{SK} \to \mathbf{SK} \to \mathbf{SK} \\
&\mathrm{K} &@t\ &= \mathrm{K} : @\ t \\
&(\mathrm{K} : @\ t) &@u\ &= t \\
&\mathrm{S} &@t\ &= \mathrm{S} : @\ t \\
&(\mathrm{S} : @\ t) &@u\ &= (\mathrm{S} : @\ t) : @\ u \\
&((\mathrm{S} : @\ t) : @\ u)@v &&= (t@v)@(u@v)
\end{aligned}$$

Computational Reals

Computational Reals

- Define computational reals ($\mathbb{R}$) using Cauchy sequences.

Computational Reals

- Define computational reals ($\mathbb{R}$) using Cauchy sequences.
- We cannot implement

$$pos :: \mathbb{R} \to \mathbf{Bool}$$

Computational Reals

- Define computational reals ($\mathbb{R}$) using Cauchy sequences.
- We cannot implement
$$pos :: \mathbb{R} \to \mathbf{Bool}$$
- Indeed, all total computable functions of type $\mathbb{R} \to \mathbf{Bool}$ are constant (Brouwer).

Computational Reals

- Define computational reals ($\mathbb{R}$) using Cauchy sequences.
- We cannot implement
$$pos :: \mathbb{R} \to \mathbf{Bool}$$
- Indeed, all total computable functions of type $\mathbb{R} \to \mathbf{Bool}$ are constant (Brouwer).
- However, there are perfectly reasonable partial implementations of pos.

We need $\perp$ for:

We need $\perp$ for:

- Interpreters.

We need $\perp$ for:

- Interpreters.
- Functions on $\mathbb{R}$.

We need $\perp$ for:

- Interpreters.
- Functions on $\mathbb{R}$.
- more examples ?

Epigram

Epigram

- Epigram is a dependently typed programming language...

Epigram

- Epigram is a dependently typed programming language...
- All Epigram programs are total (i.e. no $\perp$).

Epigram

- Epigram is a dependently typed programming language...
- All Epigram programs are total (i.e. no $\perp$).
- It is **not** a programming language in Peter Mosses sense.

Epigram

- Epigram is a dependently typed programming language...
- All Epigram programs are total (i.e. no $\perp$).
- It is **not** a programming language in Peter Mosses sense.
- because not all computable functions can be expressed.

Epigram

- Epigram is a dependently typed programming language...
- All Epigram programs are total (i.e. no $\perp$).
- It is **not** a programming language in Peter Mosses sense.
- because not all computable functions can be expressed.
- I am going to show how we can fix this...

Epigram

- Epigram is a dependently typed programming language...
- All Epigram programs are total (i.e. no $\perp$).
- It is **not** a programming language in Peter Mosses sense.
- because not all computable functions can be expressed.
- I am going to show how we can fix this...
- without making Epigram partial.

Monads. . .

Monads...

- A monad $m :: * \to *$ is given by

  $$return :: a \to m\ a$$
  $$(\gg) \quad :: (m\ a) \to (a \to m\ b) \to m\ b$$

  subject to some equations.

Monads...

- A monad $m :: * \to *$ is given by

  $$return :: a \to m\ a$$
  $$(\gg) \quad :: (m\ a) \to (a \to m\ b) \to m\ b$$

  subject to some equations.
- We can use monads to *encapsulate* effects (e.g. state)

  $$newIORef\ :: a \to \mathbf{IO}\ (\mathbf{IORef}\ a)$$
  $$readIORef :: \mathbf{IORef}\ a \to \mathbf{IO}\ a$$
  $$writeIORef :: \mathbf{IORef}\ a \to a \to \mathbf{IO}\ ()$$

Monads...

- A monad $m :: * \to *$ is given by

$$\begin{aligned} &\mathit{return} :: a \to m\ a \\ &(\gg\!=) \quad :: (m\ a) \to (a \to m\ b) \to m\ b \end{aligned}$$

  subject to some equations.

- We can use monads to *encapsulate* effects (e.g. state)

$$\begin{aligned} &\mathit{newIORef} :: a \to \mathbf{IO}\ (\mathbf{IORef}\ a) \\ &\mathit{readIORef} :: \mathbf{IORef}\ a \to \mathbf{IO}\ a \\ &\mathit{writeIORef} :: \mathbf{IORef}\ a \to a \to \mathbf{IO}\ () \end{aligned}$$

- and to *model* effects (e.g. state) :

$$\begin{aligned} &\mathbf{data}\ \mathbf{ST}\ s\ a = \mathrm{M}\ (s \to (a, s)) \\ &\mathbf{instance}\ \mathit{Monad}\ (\mathbf{ST}\ s)\ \mathbf{where} \\ &\quad \mathit{return}\ a \qquad = \mathrm{M}\ (\lambda s \to (a, s)) \\ &\quad (\mathbf{ST}\ f) \gg\!= g = \mathrm{M}\ (\lambda s \to \mathbf{let}\ (a, s') = f\ s \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathrm{M}\ g' = g\ a \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ g'\ s') \end{aligned}$$

The Delay monad

The Delay monad

$$\mathbf{codata}\ \mathbf{D}\ a = \mathrm{Now}\ a \mid \mathrm{Later}\ (\mathbf{D}\ a)$$

The Delay monad

$$\textbf{codata } \mathbf{D}\ a = \mathrm{Now}\ a \mid \mathrm{Later}\ (\mathbf{D}\ a)$$

$$\textbf{instance } \mathit{Monad}\ \mathbf{D}\ \textbf{where}$$

$$\mathit{return} = \mathrm{Now}$$

$$\mathrm{Now}\ a \gg\!= k \ = k\ a$$

$$\mathrm{Later}\ d \gg\!= k = \mathrm{Later}\ (d \gg\!= k)$$

The Delay monad

$$\mathbf{codata}\ \mathbf{D}\ a = \mathrm{Now}\ a \mid \mathrm{Later}\ (\mathbf{D}\ a)$$

$$\mathbf{instance}\ \mathit{Monad}\ \mathbf{D}\ \mathbf{where}$$

$$\mathit{return} = \mathrm{Now}$$

$$\mathrm{Now}\ a \gg\!= k \ = k\ a$$

$$\mathrm{Later}\ d \gg\!= k = \mathrm{Later}\ (d \gg\!= k)$$

$$\bot :: \mathbf{D}\ a$$

$$\bot = \mathrm{Later}\ \bot$$

Iteration with Delay

$$
\begin{aligned}
&rep :: (a \to \mathbf{D}\ (Either\ b\ a)) \to a \to \mathbf{D}\ b \\
&rep\ k\ a = k\ a \gg\!= \lambda ba \to \\
&\quad \mathbf{case}\ ba\ \mathbf{of} \\
&\qquad \mathrm{Left}\ b \to \mathrm{Now}\ b \\
&\qquad \mathrm{Right}\ a \to \mathrm{Later}\ (rep\ k\ a)
\end{aligned}
$$

Fixpoints with Delay

Fixpoints with Delay

$$rec :: ((a \to \mathbf{D}\ b) \to (a \to \mathbf{D}\ b)) \to a \to \mathbf{D}\ b$$

Fixpoints with Delay

$$
\begin{aligned}
&rec :: ((a \to \mathbf{D}\ b) \to (a \to \mathbf{D}\ b)) \to a \to \mathbf{D}\ b \\
&rec\ \phi\ a = aux\ (\lambda\_ \to \bot) \\
&\quad \mathbf{where}\ aux :: (a \to \mathbf{D}\ b) \to \mathbf{D}\ b \\
&\qquad aux\ k = race\ (k\ a)\ (\mathrm{Later}\ (aux\ (\phi\ k)))
\end{aligned}
$$

Fixpoints with Delay

$$
\begin{aligned}
&rec :: ((a \to \mathbf{D}\ b) \to (a \to \mathbf{D}\ b)) \to a \to \mathbf{D}\ b \\
&rec\ \phi\ a = aux\ (\lambda\_ \to \bot) \\
&\quad \mathbf{where}\ aux :: (a \to \mathbf{D}\ b) \to \mathbf{D}\ b \\
&\qquad aux\ k = race\ (k\ a)\ (\mathrm{Later}\ (aux\ (\phi\ k))) \\
&race :: (\mathbf{D}\ a) \to (\mathbf{D}\ a) \to (\mathbf{D}\ a) \\
&race\ (\mathrm{Now}\ a)\ \_ = \mathrm{Now}\ a \\
&race\ (\mathrm{Later}\ \_)\ (\mathrm{Now}\ a) = \mathrm{Now}\ a \\
&race\ (\mathrm{Later}\ d)\ (\mathrm{Later}\ d') = \mathrm{Later}\ (race\ d\ d')
\end{aligned}
$$

From Delay to Partial

From Delay to Partial

- D is too intensional. . .

From Delay to Partial

- $\mathbb{D}$ is too intensional...
- We can observe how fast a function terminates.

From Delay to Partial

- $\mathbf{D}$ is too intensional...
- We can observe how fast a function terminates.
- Hence $rec\ f \neq f\ (rec\ f)$

From Delay to Partial

- $\mathbf{D}$ is too intensional...
- We can observe how fast a function terminates.
- Hence $rec\ f \neq f\ (rec\ f)$
- We define

$$\mathbf{P}\ a = \mathbf{D}\ a/\simeq$$

where $\simeq \subseteq \mathbf{D}\ a \times \mathbf{D}\ a$ identifies values with different finite delay.

From Delay to Partial

- $\mathbf{D}$ is too intensional...
- We can observe how fast a function terminates.
- Hence $rec\ f \neq f\ (rec\ f)$
- We define

$$\mathbf{P}\ a = \mathbf{D}\ a/\simeq$$

  where $\simeq \subseteq \mathbf{D}\ a \times \mathbf{D}\ a$ identifies values with different finite delay.
- We have to show that $\gg\!=$,rep,rec preserve $\simeq$.

From Delay to Partial

- $\mathbf{D}$ is too intensional...
- We can observe how fast a function terminates.
- Hence $rec\ f \neq f\ (rec\ f)$
- We define

$$\mathbf{P}\ a = \mathbf{D}\ a/\simeq$$

  where $\simeq \subseteq \mathbf{D}\ a \times \mathbf{D}\ a$ identifies values with different finite delay.
- We have to show that $\gg\!=$,rep,rec preserve $\simeq$.
- We have $rec\ f \simeq f\ (rec\ f)$

From Delay to Partial

- $\mathbf{D}$ is too intensional...
- We can observe how fast a function terminates.
- Hence $rec\ f \neq f\ (rec\ f)$
- We define

$$\mathbf{P}\ a = \mathbf{D}\ a/\simeq$$

  where $\simeq \subseteq \mathbf{D}\ a \times \mathbf{D}\ a$ identifies values with different finite delay.
- We have to show that $\gg\!=$,rep,rec preserve $\simeq$.
- We have $rec\ f \simeq f\ (rec\ f)$
  if f is ω-continuous,
  however all definable f are.

Defining $\simeq$

Defining $\simeq$

- $(\downarrow) \subseteq \mathbf{D}\ a \times a$ is defined inductively.

$$\frac{}{\text{Now}\ a \downarrow a} \qquad \frac{d \downarrow a}{\text{Later}\ d \downarrow a}$$

Defining $\simeq$

- $(\downarrow) \subseteq \mathbf{D}\ a \times a$ is defined inductively.

$$\frac{}{\text{Now}\ a \downarrow a} \qquad \frac{d \downarrow a}{\text{Later}\ d \downarrow a}$$

-

$$\begin{array}{rcl} \sqsubseteq & \subseteq & \mathbf{D}\ a \times \mathbf{D}\ a \\ d \sqsubseteq d' & = & \forall a.d \downarrow a \implies d' \downarrow a \end{array}$$

Defining $\simeq$

- $(\downarrow) \subseteq \mathbf{D}\ a \times a$ is defined inductively.

$$\frac{}{\text{Now}\ a \downarrow a} \qquad \frac{d \downarrow a}{\text{Later}\ d \downarrow a}$$

-

$$\begin{array}{rcl} \sqsubseteq & \subseteq & \mathbf{D}\ a \times \mathbf{D}\ a \\ d \sqsubseteq d' & = & \forall a.d \downarrow a \implies d' \downarrow a \end{array}$$

-

$$\begin{array}{rcl} \simeq & \subseteq & \mathbf{D}\ a \times \mathbf{D}\ a \\ d \simeq d' & = & d \sqsubseteq d' \wedge d' \sqsubseteq d \end{array}$$

Deja vu ?

Deja vu ?

- Constructive Domain Theory!

Deja vu ?

- Constructive Domain Theory!
- $\mathbf{P}\ a = a_\perp$

Deja vu ?

- Constructive Domain Theory!
- $\mathbf{P}\ a = a_\perp$
- Note that constructively

$$a_\perp \neq a + \{\perp\}$$

because we cannot observe non-termination.

Deja vu ?

- Constructive Domain Theory!
- $\mathbf{P}\ a = a_\perp$
- Note that constructively

$$a_\perp \neq a + \{\perp\}$$

  because we cannot observe non-termination.
- $\mathbf{P}\ a$ and hence $a \to \mathbf{P}\ b$ are ωCPOs.

Deja vu ?

- Constructive Domain Theory!
- $\mathbf{P}\ a = a_\perp$
- Note that constructively

$$a_\perp \neq a + \{\perp\}$$

  because we cannot observe non-termination.
- $\mathbf{P}\ a$ and hence $a \to \mathbf{P}\ b$ are ωCPOs.
- $rec\ f = \sqcup_{i \in \mathbb{N}} f^i \perp$ the code before constructs $\sqcup$ in $a \to \mathbf{P}\ b$.

Conclusions and further work

Conclusions and further work

- Using the partiality monad we can encapsulate partial programs in a total language.

Conclusions and further work

- Using the partiality monad we can encapsulate partial programs in a total language.
- *Partiality is an effect*

Conclusions and further work

- Using the partiality monad we can encapsulate partial programs in a total language.
- *Partiality is an effect*
- We can reason about partial programs at compile time using the definition of $\mathbf{P}\ a$.

Conclusions and further work

- Using the partiality monad we can encapsulate partial programs in a total language.
- *Partiality is an effect*
- We can reason about partial programs at compile time using the definition of $\mathbf{P}\ a$.
- and we can execute non-terminating programs at run-time.

Conclusions and further work

- Using the partiality monad we can encapsulate partial programs in a total language.
- *Partiality is an effect*
- We can reason about partial programs at compile time using the definition of $\mathbf{P}\ a$.
- and we can execute non-terminating programs at run-time.
- In future Epigram could support partiality without giving up the advantages of having a total language for most programs.

Conclusions and further work

- Using the partiality monad we can encapsulate partial programs in a total language.
- *Partiality is an effect*
- We can reason about partial programs at compile time using the definition of $\mathbf{P}\ a$.
- and we can execute non-terminating programs at run-time.
- In future Epigram could support partiality without giving up the advantages of having a total language for most programs.
- Still to do: recursive datatypes by a constructive implementation of the standard domain-theoretic construction.

Thank you

- Thanks to Conor McBride & the Epigram Team
  (James Chapman, Peter Morris, Wouter Swierstra)
  see `www.e-pig.org` for more information on Epigram.
- Acknowledgements to Tarmo Uustalu and Venanzio Capretta
  for joint work on a partial paper...
- Looking for my papers?
  Type "Thorsten" into google...