Stop thinking about bottoms when writing programs . . .

Thorsten Altenkirch
University of Nottingham
Trouble with
Trouble with \( \bot \)

\[
\begin{align*}
(*) & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
0 & \ast n = 0 \\
(m + 1) \ast n & = m \ast n + n
\end{align*}
\]
Trouble with \( \bot \)

\[(*) \colon \mathbb{N} \to \mathbb{N} \to \mathbb{N}\]

\[
0 \ast n = 0 \\
(m + 1) \ast n = m \ast n + n
\]

\[
x \ast y = y \ast x \\
?\]
Trouble with $\bot$

\[(*) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N}\]
\[0 \ast n = 0\]
\[(m + 1) \ast n = m \ast n + n\]

\[x \ast y = y \ast x\quad ?\]

No, because

\[0 \ast \bot = 0\]
\[\bot \ast 0 = \bot\]
Trouble with \( \perp \ldots \)
Trouble with $\bot \ldots$

- Many useful algebraic properties do not hold.
Trouble with $\bot$ . . .

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

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

- Many useful algebraic properties do not hold.
- Correctness proofs get obliterated with reasoning about $\bot$.
- Do we actually care about non-terminating programs?
- Programs are **not** natural phenomena...
Many useful algebraic properties do not hold.
Correctness proofs get obliterated with reasoning about $\bot$.
Do we actually care about non-terminating programs?
Programs are not natural phenomena…
Programs are constructed!
Do we need \( \bot \) to be lazy?
Do we need \( \bot \) to be lazy?

\[
\begin{align*}
\text{from} : \mathbb{N} & \to [\mathbb{N}] \\
\text{from } n & = n : (\text{from } (n + 1))
\end{align*}
\]
Do we need $\bot$ to be lazy?

\[
\begin{align*}
from :: \mathbb{N} & \to [\mathbb{N}] \\
from n & = n : (from (n + 1)) \\
\end{align*}
\]

- \textit{from} is total, \textbf{if} we interpret lists as a terminal coalgebra.

\[
[A] = \nu X.1 + A \times X
\]
data vs codata
data vs codata

\[ \text{evenLength} :: [a] \rightarrow \text{Bool} \]
\[ \text{evenLength} [] = \text{True} \]
\[ \text{evenLength} (a : as) = \neg (\text{evenLength} n) \]
data vs codata

\[\text{evenLength} :: [a] \rightarrow \text{Bool}\]
\[\text{evenLength} [\;] = \text{True}\]
\[\text{evenLength} (a : as) = \neg (\text{evenLength} n)\]

- \text{evenLength} is total, \ldots
data vs codata

\[ evenLength :: [a] \rightarrow \text{Bool} \]
\[ evenLength [] = \text{True} \]
\[ evenLength (a : as) = \neg (evenLength n) \]

- \( evenLength \) is total, ...
- **if** we interpret lists as initial algebra:

\[ [A] = \mu X. 1 + A \times X \]
data vs codata

\[ evenLength :: [a] \rightarrow \text{Bool} \]
\[ evenLength [] = \text{True} \]
\[ evenLength (a : as) = \neg (evenLength \ n) \]

- \( evenLength \) is total, ...
- **if** we interpret lists as initial algebra:

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

- Problem:

\[ evenLength \ (\text{from} \ 0) = \bot \]
data vs codata
data vs codata

- Finite lists
  \[ \text{data} \ [a] = [] \mid a : [a] \]
data vs codata

- Finite lists
  \[ \text{data } [a] = [] \mid a : [a] \]

- Potentially infinite lists:
  \[ \text{codata } [a]^{\omega} a = [] \mid a : [a]^{\omega} \]
data vs codata

- Finite lists
  \[
  \text{data } [a] = [] \mid a : [a]
  \]

- Potentially infinite lists:
  \[
  \text{codata } [a]^{\omega} a = [] \mid a : [a]^{\omega}
  \]

- Better types
  \begin{align*}
  \text{from} & \quad :: \mathbb{N} \to [a]^{\omega} \\
  \text{evenLength} & \quad :: [a] \to \text{Bool}
  \end{align*}
data vs codata

- Finite lists
  \[ \textbf{data} \ [a] = [] \mid a : [a] \]

- Potentially infinite lists:
  \[ \textbf{codata} \ [a]^\omega \ a = [] \mid a : [a]^\omega \]

- Better types
  \[
  \textit{from} \quad :: \mathbb{N} \rightarrow [a]^\omega \\
  \textit{evenLength} :: [a] \rightarrow \textbf{Bool}
  \]

- \textit{evenLength (from 0)} doesn’t typecheck.
Can we always avoid $\bot$?
Can we always avoid $\bot$?

```plaintext
data SK = S | K | SK : @ SK

nf :: SK → SK
nf S = S
nf K = K
nf (t : @ u) = (nf t)@nf u

(@) :: SK → SK → SK
K @t = K : @ t
(K : @ t) @u = t
S @t = S : @ t
(S : @ t) @u = (S : @ t) : @ u
((S : @ t) : @ u)@v = (t@v)@@u@v
```
Computational Reals

Dene computational reals ( ) using Cauchy sequences. We cannot implement . Indeed, all total computable functions of type are constant (Brouwer). However, there are perfectly reasonable partial implementations of .
Computational Reals

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

- Define computational reals ($\mathbb{R}$) using Cauchy sequences.
- We cannot implement
  $$\text{pos} :: \mathbb{R} \to \text{Bool}$$
Computational Reals

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

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

- Interpreters.
We need \( \bot \) for:

- Interpreters.
- Functions on \( \mathbb{R} \).
We need \( \bot \) for:

- Interpreters.
- Functions on \( \mathbb{R} \).
- more examples?
Epigram is a dependently typed programming language...

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 x this. without making Epigram partial.
Epigram

- Epigram is a dependently typed programming language...
Epigram

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

- Epigram is a dependently typed programming language...
- All Epigram programs are total (i.e. no \(\bot\)).
- 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 ⊥).
- 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 \( m :: \ast \rightarrow \ast \) is given by
  
  \[
  \text{return} :: a \rightarrow m a
  \]
  
  \[
  (\geq) :: (m a) \rightarrow (a \rightarrow m b) \rightarrow m b
  \]

  subject to some equations.
Monads...

- A monad $m :: * \to *$ is given by
  
  $\text{return} :: a \to m a$
  
  $(\geq) :: (m a) \to (a \to m b) \to m b$

  subject to some equations.

- We can use monads to *encapsulate* effects (e.g. state)
  
  $\text{newIORef} :: a \to \text{IO} (\text{IORef} a)$
  
  $\text{readIORef} :: \text{IORef} a \to \text{IO} a$
  
  $\text{writeIORef} :: \text{IORef} a \to a \to \text{IO} ()$
Monads...

- A monad $m :: * \rightarrow *$ is given by
  
  $\text{return} :: a \rightarrow m \ a$

  $\left(\left(\left(\left(\left(\left(m \ a\right) \rightarrow \left(\left(a \rightarrow m \ b\right)\right)\right)\rightarrow m \ b\right)\right)\right)\right)$

  subject to some equations.

- We can use monads to *encapsulate* effects (e.g. state)
  
  $\text{newIORef} :: a \rightarrow \text{IO} \ (\text{IORRef} \ a)$

  $\text{readIORef} :: \text{IORRef} \ a \rightarrow \text{IO} \ a$

  $\text{writeIORef} :: \text{IORRef} \ a \rightarrow a \rightarrow \text{IO} ()$

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

  $\text{data ST} \ s \ a = \text{M} \ \left(\left(s \rightarrow (a, s)\right)\right)$

  $\text{instance Monad (ST s) where}$

  $\text{return} \ a = \text{M} \ \left(\lambda s \rightarrow (a, s)\right)$

  $(\text{ST} \ f) \Rightarrow g = \text{M} \ \left(\lambda s \rightarrow \text{let} \ \left(a, s'\right) = f \ s \ 

  \quad \text{M} \ g' = g \ a \ 

  \quad \text{in} \ g' \ s'\right)$
The Delay monad
The Delay monad

codata \( D \ a = \text{Now} \ a \mid \text{Later} \ (D \ a) \)
The Delay monad

codata D a = Now a | Later (D a)
instance Monad D where

  return = Now

  Now a >>= k = k a

  Later d >>= k = Later (d >>= k)
The Delay monad

codata D a = Now a | Later (D a)
instance Monad D where
  return = Now
  Now a ▷◁ k = k a
  Later d ▷◁ k = Later (d ▷◁ k)
⊥ :: D a
⊥ = Later ⊥
Iteration with Delay

\[ rep \, :: \, (a \rightarrow D \, (Either \, b \, a)) \rightarrow a \rightarrow D \, b \]

\[ rep \, k \, a = k \, a \gg \lambda ba \rightarrow \]

\textbf{case} \, ba \, \textbf{of}

\textbf{Left} \, b \rightarrow \textbf{Now} \, b

\textbf{Right} \, a \rightarrow \textbf{Later} \,(rep \, k \, a) \]
Fixpoints with Delay
Fixpoints with Delay

\[ \text{rec} :: ((a \rightarrow \mathbf{D} \, b) \rightarrow (a \rightarrow \mathbf{D} \, b)) \rightarrow a \rightarrow \mathbf{D} \, b \]
Fixpoints with Delay

\[
\text{rec} :: ((a \to \mathbb{D} b) \to (a \to \mathbb{D} b)) \to a \to \mathbb{D} b
\]

\[
\text{rec } \phi \ a = \text{aux } (\lambda \_ \to \bot)
\]

\[\text{where aux} :: (a \to \mathbb{D} b) \to \mathbb{D} b\]

\[\text{aux } k = \text{race } (k \ a) \ (\text{Later } (\text{aux } (\phi \ k)))\]
Fixpoints with Delay

\[
\begin{align*}
\text{rec} &:: ((a \rightarrow \mathbf{D} \ b) \rightarrow (a \rightarrow \mathbf{D} \ b)) \rightarrow a \rightarrow \mathbf{D} \ b \\
\text{rec} \ \phi \ a &= \text{aux} \ (\lambda_\_ \rightarrow \bot) \\
\text{where aux} &:: (a \rightarrow \mathbf{D} \ b) \rightarrow \mathbf{D} \ b \\
\text{aux} \ k &= \text{race} \ (k \ a) \ (\text{Later} \ (\text{aux} \ (\phi \ k))) \\
\text{race} &:: (\mathbf{D} \ a) \rightarrow (\mathbf{D} \ a) \rightarrow (\mathbf{D} \ a) \\
\text{race} \ (\text{Now} \ a) \ _ &= \text{Now} \ a \\
\text{race} \ (\text{Later} \ _) \ (\text{Now} \ a) &= \text{Now} \ a \\
\text{race} \ (\text{Later} \ d) \ (\text{Later} \ d') &= \text{Later} \ (\text{race} \ d \ d')
\end{align*}
\]
From Delay to Partial
From Delay to Partial

- D is too intensional...
From Delay to Partial

- $D$ is too intensional...  
- We can observe how fast a function terminates.
From Delay to Partial

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

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

  where $\sim \subseteq D \ a \times D \ a$ identifies values with different finite delay.
From Delay to Partial

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

\[
\mathbf{P} \ a = D \ a/ \sim
\]

where \(\sim \subseteq D \ a \times D \ a\) identifies values with different finite delay.
- We have to show that \(\gg\), \text{rep}, \text{rec} preserve \(\sim\).
D is too intensional...
We can observe how fast a function terminates.
Hence $\text{rec } f \neq f (\text{rec } f)$
We define

$$P \ a = D \ a/ \sim$$

where $\sim \subseteq D \ a \times D \ a$ identifies values with different finite delay.
We have to show that $\gg$, $\text{rep}$, $\text{rec}$ preserve $\sim$.
We have $\text{rec } f \sim f (\text{rec } f)$
From Delay to Partial

- $\mathcal{D}$ is too intensional...
- We can observe how fast a function terminates.
- Hence $\text{rec } f \not\equiv f \ (\text{rec } f)$
- We define

\[ \mathbf{P} \ a = \mathcal{D} \ a / \sim \]

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

- $(\downarrow) \subseteq D \; a \times a$ is defined inductively.

\[
\begin{align*}
\text{Now } a \downarrow a & \quad \text{Later } d \downarrow a \\
& \quad \text{d} \downarrow a
\end{align*}
\]
Defining $\sim$

- $(\downarrow) \subseteq D \; a \times a$ is defined inductively.

\[
\begin{align*}
\text{Now } a \downarrow a & \quad \text{Later } d \downarrow a \\
& \quad \text{Later } d \downarrow a
\end{align*}
\]

- $d \sqsubseteq d' \quad \Rightarrow \quad \forall a. d \downarrow a \quad \Rightarrow \quad d' \downarrow a$
Defining $\simeq$

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

\[
\begin{align*}
\text{Now } a & \downarrow a \\
\text{Later } d & \downarrow a
\end{align*}
\]

- $\subseteq \subseteq \mathcal{D} a \times \mathcal{D} a$

\[
d \subseteq d' \quad \Rightarrow \quad \forall a.d \downarrow a \quad \Rightarrow \quad d' \downarrow a
\]

- $\simeq \subseteq \mathcal{D} a \times \mathcal{D} a$

\[
d \simeq d' \quad \Rightarrow \quad d \subseteq d' \land d' \subseteq d
\]
Deja vu ?
Deja vu?

- Constructive Domain Theory!
Deja vu?

- Constructive Domain Theory!
- \( P \ a = a_\bot \)
Deja vu?

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

\[ a_\perp \neq a + \{ \perp \} \]

because we cannot observe non-termination.
Deja vu?

- Constructive Domain Theory!
- \( \mathbb{P} \ a = a_\perp \)
- Note that constructively

\[
\begin{align*}
    a_\perp & \neq a + \{\perp\} \\
\end{align*}
\]

because we cannot observe non-termination.
- \( \mathbb{P} \ a \) and hence \( a \rightarrow \mathbb{P} \ b \) are \( \omega \)CPOs.
Deja vu?

- Constructive Domain Theory!
- $P \ a = a_\bot$
- Note that constructively

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

because we cannot observe non-termination.

- $P \ a$ and hence $a \to P \ b$ are $\omega$CPOs.
- $\text{rec } f = \bigsqcup_{i \in \mathbb{N}} f^i \bot$ the code before constructs $\sqcup$ in $a \to P \ b$. 
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 \( \mathbb{P} \ a \).
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 $\mathbb{P}_a$.
- 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 $\mathbb{P}_a$.
- 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 $P_a$.
- 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...