Type-Based Reasoning and Imprecise Errors

Janis Voigtländer

Technische Universität Dresden

March 6th, 2009

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned}\texttt{map}\ f\ [\,] &= [\,]\\ \texttt{map}\ f\ (a : as) &= (f\ \ a) : (\texttt{map}\ f\ \ as)\end{aligned}$$

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned} \texttt{map}\ f\ [\,] &= [\,] \\ \texttt{map}\ f\ (a : as) &= (f\ \ a) : (\texttt{map}\ f\ \ as) \end{aligned}$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] \qquad = [2, 3, 4]$

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned} \texttt{map}\ f\ [\,] &= [\,] \\ \texttt{map}\ f\ (a : as) &= (f\ a) : (\texttt{map}\ f\ as) \end{aligned}$$

Some invocations:

$$\begin{aligned} \texttt{map succ}\ [1, 2, 3] &= [2, 3, 4] \\ \texttt{map not}\ \ [\mathsf{True}, \mathsf{False}] &= [\mathsf{False}, \mathsf{True}] \end{aligned}$$

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned} \texttt{map}\ f\ [\,] &= [\,] \\ \texttt{map}\ f\ (a : as) &= (f\ a) : (\texttt{map}\ f\ as) \end{aligned}$$

Some invocations:

$$\begin{aligned} \texttt{map succ}\ [1, 2, 3] &= [2, 3, 4] \\ \texttt{map not}\ \ [\text{True}, \text{False}] &= [\text{False}, \text{True}] \\ \texttt{map even}\ [1, 2, 3] &= [\text{False}, \text{True}, \text{False}] \end{aligned}$$

Polymorphic Types: An Example in Haskell

A standard function:

$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$$
$$\texttt{map not}\ [\text{True}, \text{False}] = [\text{False}, \text{True}]$$
$$\texttt{map even}\ [1, 2, 3] = [\text{False}, \text{True}, \text{False}]$$
$$\texttt{map not}\ [1, 2, 3]$$

Polymorphic Types: An Example in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$$
$$\texttt{map not}\ [\mathsf{True}, \mathsf{False}] = [\mathsf{False}, \mathsf{True}]$$
$$\texttt{map even}\ [1, 2, 3] = [\mathsf{False}, \mathsf{True}, \mathsf{False}]$$
$$\texttt{map not}\ [1, 2, 3]$$

Polymorphic Types: An Example in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int}$

$\texttt{map not}\ [\mathsf{True}, \mathsf{False}] = [\mathsf{False}, \mathsf{True}]$

$\texttt{map even}\ [1, 2, 3] = [\mathsf{False}, \mathsf{True}, \mathsf{False}]$

$\texttt{map not}\ [1, 2, 3]$

Polymorphic Types: An Example in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$\texttt{map succ}\ [1,2,3] = [2,3,4]$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int}$

$\texttt{map not}\ [\mathsf{True}, \mathsf{False}] = [\mathsf{False}, \mathsf{True}]$ — $\alpha, \beta \mapsto \mathsf{Bool}, \mathsf{Bool}$

$\texttt{map even}\ [1,2,3] = [\mathsf{False}, \mathsf{True}, \mathsf{False}]$

$\texttt{map not}\ [1,2,3]$

Polymorphic Types: An Example in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$ — $\alpha, \beta \mapsto$ Int, Int

$\texttt{map not}\ [\text{True}, \text{False}] = [\text{False}, \text{True}]$ — $\alpha, \beta \mapsto$ Bool, Bool

$\texttt{map even}\ [1, 2, 3] = [\text{False}, \text{True}, \text{False}]$ — $\alpha, \beta \mapsto$ Int, Bool

$\texttt{map not}\ [1, 2, 3]$

Polymorphic Types: An Example in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int}$

$\texttt{map not}\ [\mathsf{True}, \mathsf{False}] = [\mathsf{False}, \mathsf{True}]$ — $\alpha, \beta \mapsto \mathsf{Bool}, \mathsf{Bool}$

$\texttt{map even}\ [1, 2, 3] = [\mathsf{False}, \mathsf{True}, \mathsf{False}]$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Bool}$

$\texttt{map not}\ [1, 2, 3]$ $\lightning$ rejected at compile-time

Another Example

$$
\begin{array}{l}
\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
\texttt{takeWhile}\ p\ [\,] = [\,] \\
\texttt{takeWhile}\ p\ (a : as) \mid p\ a = a : (\texttt{takeWhile}\ p\ as) \\
\qquad\qquad\qquad\qquad\quad \mid \texttt{otherwise} = [\,]
\end{array}
$$

Another Example

$$
\begin{array}{l}
\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
\texttt{takeWhile}\ p\ [\,] \qquad\quad = [\,] \\
\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a \qquad\quad\; = a : (\texttt{takeWhile}\ p\ as) \\
\qquad\qquad\qquad\qquad\quad\ \mid\ \texttt{otherwise} = [\,]
\end{array}
$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \ = \ \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ l)$$

Provable by induction.

Another Example

$$
\begin{aligned}
&\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
&\texttt{takeWhile}\ p\ [\,] = [\,] \\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a = a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\ \ \mid\ \texttt{otherwise} = [\,]
\end{aligned}
$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \ = \ \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ \ l)$$

Provable by induction.

Or as a "free theorem" [Wadler, FPCA'89].

Another Example

$$\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ l)$$

Provable by induction.

Or as a "free theorem" [Wadler, FPCA'89].

Another Example

$$\texttt{takeWhile} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

$$\texttt{filter} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ \ l)$$

$$\texttt{filter}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{filter}\ (p \circ f)\ \ l)$$

Another Example

$$\texttt{takeWhile} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$
$$\texttt{filter} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$
$$\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \ = \ \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ \ l)$$
$$\texttt{filter}\ p\ (\texttt{map}\ f\ \ l) \ = \ \texttt{map}\ f\ (\texttt{filter}\ (p \circ f)\ \ l)$$
$$\texttt{g}\ p\ (\texttt{map}\ f\ \ l) \ = \ \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ \ l)$$

Why, Intuitively

- $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

Why, Intuitively

- $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

Why, Intuitively

- $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list *l*.
- Which, and in which order/multiplicity, can only be decided based on *l* and the input predicate *p*.

Why, Intuitively

- g :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list *l*.
- Which, and in which order/multiplicity, can only be decided based on *l* and the input predicate *p*.
- The only means for this decision are to inspect the length of *l* and to check the outcome of *p* on its elements.

Why, Intuitively

- `g` :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.

Why, Intuitively

- `g` :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.
- Applying p to an element of (`map` f l) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Why, Intuitively

- `g` $:: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.
- Applying p to an element of (`map` f l) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- `g` with p always chooses "the same" elements from (`map` f l) for output as does `g` with $(p \circ f)$ from l,

Why, Intuitively

- `g` :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.
- Applying p to an element of (`map` f l) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- `g` with p always chooses "the same" elements from (`map` f l) for output as does `g` with $(p \circ f)$ from l, except that it outputs their images under f.

Why, Intuitively

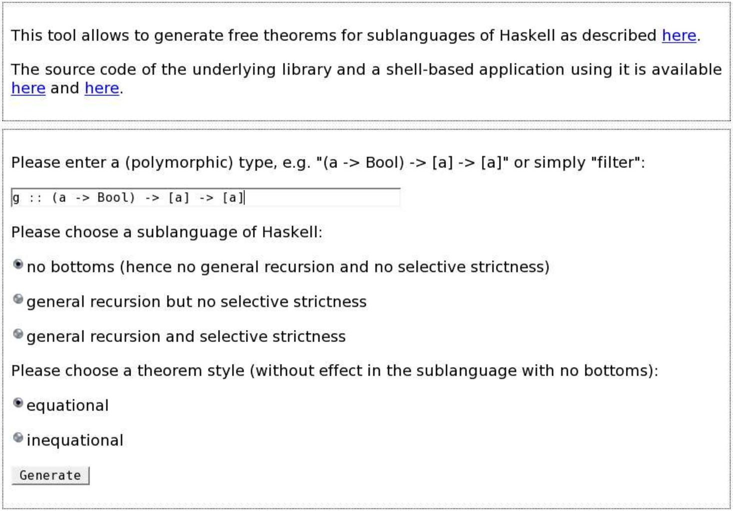
- `g` :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.
- Applying p to an element of (`map` f l) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- `g` with p always chooses "the same" elements from (`map` f l) for output as does `g` with $(p \circ f)$ from l, except that it outputs their images under f.
- (`g` p (`map` f l)) is equivalent to (`map` f (`g` $(p \circ f)$ l)).

Why, Intuitively

- `g` :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.
- Applying p to an element of (`map` f l) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- `g` with p always chooses "the same" elements from (`map` f l) for output as does `g` with $(p \circ f)$ from l, except that it outputs their images under f.
- (`g` p (`map` f l)) is equivalent to (`map` f (`g` $(p \circ f)$ l)).
- That is what was claimed!

Automatic Generation of Free Theorems

At `http://linux.tcs.inf.tu-dresden.de/~voigt/ft`:



Automatic Generation of Free Theorems

The theorem generated for functions of the type

```
g :: forall a . (a -> Bool) -> [a] -> [a]
```

in the sublanguage of Haskell with no bottoms is:

```
forall t1,t2 in TYPES, R in REL(t1,t2).
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall (x, y) in R. p x = q y)
   ==> (forall (z, v) in lift{[]}(R).
         (g p z, g q v) in lift{[]}(R))
```

The structural lifting occurring therein is defined as follows:

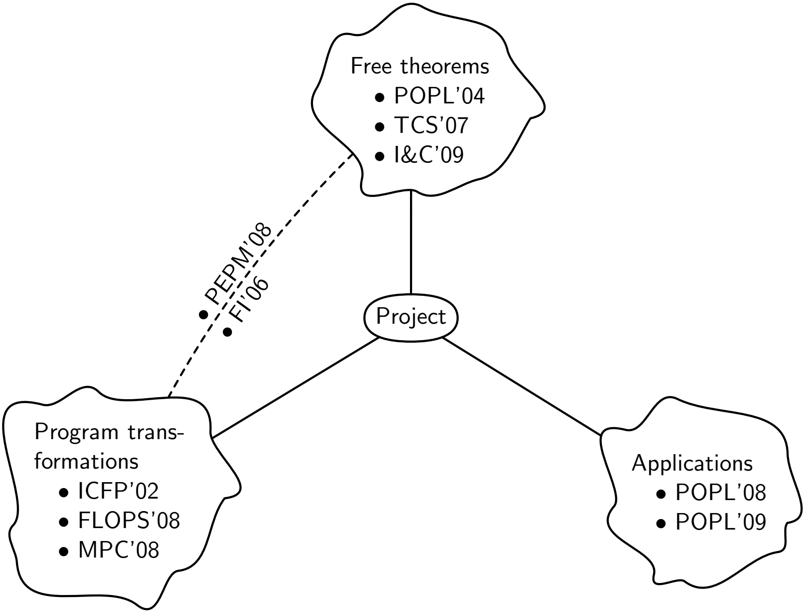
```
lift{[]}(R)
  = {([], [])}
  u {(x : xs, y : ys) |
       ((x, y) in R) && ((xs, ys) in lift{[]}(R))}
```

Reducing all permissible relation variables to functions yields:

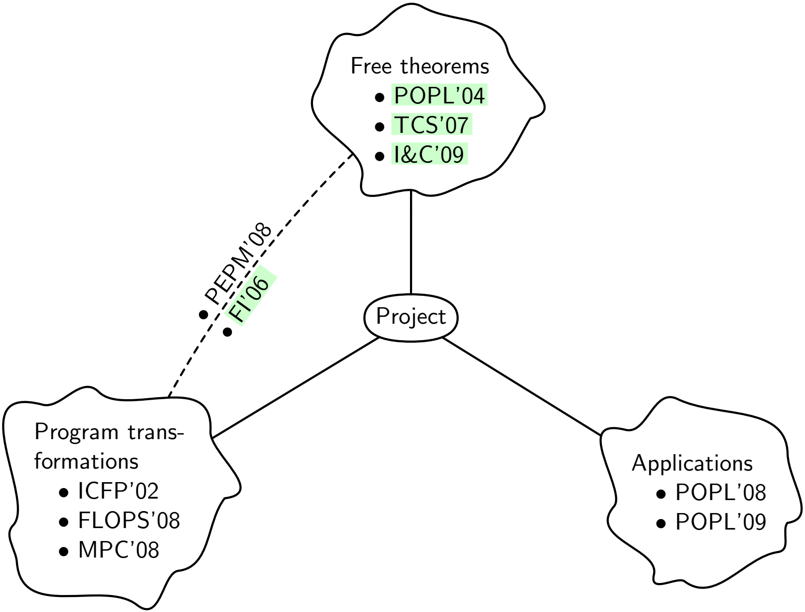
```
forall t1,t2 in TYPES, f :: t1 -> t2.
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall x :: t1. p x = q (f x))
   ==> (forall y :: [t1]. map f (g p y) = g q (map f y))
```

Export as PDF | Show type instantiations | Enter a new type | Help page

DFG-Project VO 1512/1-1



DFG-Project VO 1512/1-1



Where is the Problem?

- g :: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.

Where is the Problem?

- $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

Where is the Problem?

- g :: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list *l*.

$\lightning$ Not true! Also possible: $\bot$

Where is the Problem?

- $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\bot$

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.

Where is the Problem?

- g :: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\perp$

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

Where is the Problem?

- g :: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list *l*.

↯ Not true! Also possible: $\perp$

- Which, and in which order/multiplicity, can only be decided based on *l* and the input predicate *p*.
- The only means for this decision are to inspect the length of *l* and to check the outcome of *p* on its elements.

↯ Not true! Also possible:

- checking elements from *l* for being $\perp$
- checking *p* for being $\perp$
- checking outcome of *p* on $\perp$

Where is the Problem?

- $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\bot$

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

↯ Not true! Also possible:

- checking elements from l for being $\bot$
- checking p for being $\bot$
- checking outcome of p on $\bot$

... ???

Revising Free Theorems

[Wadler, FPCA'89] : for every $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

Revising Free Theorems

[Wadler, FPCA'89] : for every $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

[Johann & V., POPL'04] : in Haskell only provable if:

- $p \neq \bot$,
- f strict ($f\ \bot = \bot$), and
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

Revising Free Theorems

[Wadler, FPCA'89] : for every $\mathtt{g} :: (\alpha \rightarrow \mathrm{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$,

$$\mathtt{g}\ p\ (\mathtt{map}\ f\ l) \quad = \quad \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$$

[Johann & V., POPL'04] : in Haskell only provable if:

- $p \neq \bot$,
- f strict ($f\ \bot = \bot$), and
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

[Johann & V., I&C'09] : taking finite failures into account

Revising Free Theorems

[Wadler, FPCA'89] : for every $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

[Johann & V., POPL'04] : in Haskell only provable if:

- $p \neq \bot$,
- f strict ($f\ \bot = \bot$), and
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

[Johann & V., I&C'09] : taking finite failures into account

[Stenger & V., TR] : taking imprecise error semantics into account

Errors in Haskell

- **let** `average` $l =$ `div` $($`sum` $l)$ $($`length` $l)$
  in `average` $[\,]$

Errors in Haskell

- **let** $\texttt{average}\ l = \texttt{div}\ (\texttt{sum}\ l)\ (\texttt{length}\ l)$
  in $\texttt{average}\ [\,]$

- **let** $\texttt{tail}\ (a : as) = as$
  in $\texttt{tail}\ [\,]$

Errors in Haskell

- **let** `average` $l = $ `div` (`sum` l) (`length` l)
  in `average` [ ]

- **let** `tail` $(a : as) = as$
  in `tail` [ ]

- **if** $\cdots$ **then** `error` "some string" **else** $\cdots$

Errors in Haskell

- **let** `average` l = `div` (`sum` l) (`length` l)
  in `average` []

- **let** `tail` $(a : as) = as$
  in `tail` []

- **if** $\cdots$ **then** `error` "some string" **else** $\cdots$

- **let** `loop` = `loop`
  in `loop`

Errors in Haskell

- **let** `average` l = `div` (`sum` l) (`length` l)
  in `average` []
- **let** `tail` $(a : as) = as$
  in `tail` []
- **if** $\cdots$ **then** `error` "some string" **else** $\cdots$
- **let** `loop` = `loop`
  in `loop`

Traditionally, all error causes subsumed under $\bot$.

Errors in Haskell

- **let** `average` l = `div` (`sum` l) (`length` l)
  in `average` []

- **let** `tail` $(a : as) = as$
  in `tail` []

- **if** $\cdots$ **then** `error` "some string" **else** $\cdots$

- **let** `loop` = `loop`
  in `loop`

Traditionally, all error causes subsumed under $\bot$.

Better, explicit distinction. Like:

Ok v	:	nonerroneous
Bad "$\cdots$"	:	finitely failing
$\bot$	:	nonterminating

Propagation of Errors

- `tail` $[1/0, 2.5] \rightsquigarrow \mathit{Ok}\ [\mathit{Ok}\ 2.5]$

Propagation of Errors

- `tail` $[1/0, 2.5] \ \leadsto \ \mathit{Ok}\ [\mathit{Ok}\ 2.5]$

- $(\lambda x \rightarrow 3)$ (`error` "$\cdots$") $\ \leadsto \ \mathit{Ok}\ 3$

Propagation of Errors

- $\texttt{tail}\ [1/0, 2.5] \ \leadsto\ \mathit{Ok}\ [\mathit{Ok}\ 2.5]$
- $(\lambda x \rightarrow 3)\ (\texttt{error}\ \text{``}\cdots\text{''}) \ \leadsto\ \mathit{Ok}\ 3$
- $(\texttt{error}\ s)\ (\cdots) \ \leadsto\ \mathit{Bad}\ s$

Propagation of Errors

- $\texttt{tail}\ [1/0, 2.5] \ \rightsquigarrow\ \mathit{Ok}\ [\mathit{Ok}\ 2.5]$
- $(\lambda x \rightarrow 3)\ (\texttt{error}\ \text{"}\cdots\text{"}) \ \rightsquigarrow\ \mathit{Ok}\ 3$
- $(\texttt{error}\ s)\ (\cdots) \ \rightsquigarrow\ \mathit{Bad}\ s$
- $\textbf{case}\ (\texttt{error}\ s)\ \textbf{of}\ \{\cdots\} \ \rightsquigarrow\ \mathit{Bad}\ s$

Propagation of Errors

- `tail` $[1/0, 2.5] \rightsquigarrow \mathit{Ok}\ [\mathit{Ok}\ 2.5]$

- $(\lambda x \to 3)$ (`error` "$\cdots$") $\rightsquigarrow \mathit{Ok}\ 3$

- (`error` s) $(\cdots) \rightsquigarrow \mathit{Bad}\ s$

- **case** (`error` s) **of** $\{\cdots\} \rightsquigarrow \mathit{Bad}\ s$

- (`error` s_1) $+$ (`error` s_2) $\rightsquigarrow$???

Propagation of Errors

- $\texttt{tail}\ [1/0, 2.5] \ \rightsquigarrow\ \mathit{Ok}\ [\mathit{Ok}\ 2.5]$
- $(\lambda x \rightarrow 3)\ (\texttt{error}\ \text{“}\cdots\text{”}) \ \rightsquigarrow\ \mathit{Ok}\ 3$
- $(\texttt{error}\ s)\ (\cdots) \ \rightsquigarrow\ \mathit{Bad}\ s$
- $\textbf{case}\ (\texttt{error}\ s)\ \textbf{of}\ \{\cdots\} \ \rightsquigarrow\ \mathit{Bad}\ s$
- $(\texttt{error}\ s_1) + (\texttt{error}\ s_2) \ \rightsquigarrow\ ???$

Dependence on evaluation order leads to considerably less freedom for implementors to rearrange computations, to optimise!

Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Basic idea:

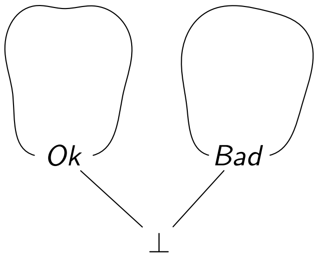
$$
\begin{aligned}
\mathit{Ok}\ v &: \text{nonerroneous} \\
\mathit{Bad}\ \{\cdots\} &: \text{finitely failing, nondeterministic} \\
\bot &: \text{nonterminating}
\end{aligned}
$$

Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Basic idea:

$$
\begin{aligned}
\mathit{Ok}\ v &: \text{nonerroneous} \\
\mathit{Bad}\ \{\cdots\} &: \text{finitely failing, nondeterministic} \\
\bot &: \text{nonterminating}
\end{aligned}
$$

Definedness order:

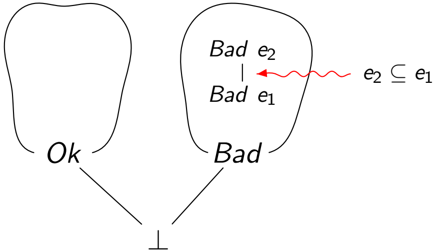


Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Basic idea:

$$\begin{aligned} \mathit{Ok}\ v &: \text{nonerroneous} \\ \mathit{Bad}\ \{\cdots\} &: \text{finitely failing, nondeterministic} \\ \bot &: \text{nonterminating} \end{aligned}$$

Definedness order:



Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Propagation of Errors:

- $(\texttt{error}\ s_1) + (\texttt{error}\ s_2) \rightsquigarrow \mathit{Bad}\ \{s_1, s_2\}$

Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Propagation of Errors:

- $(\texttt{error}\ s_1) + (\texttt{error}\ s_2) \ \rightsquigarrow\ \mathit{Bad}\ \{s_1, s_2\}$
- $3 + (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$

Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Propagation of Errors:

- $(\texttt{error}\ s_1) + (\texttt{error}\ s_2) \ \rightsquigarrow\ \mathit{Bad}\ \{s_1, s_2\}$

- $3 + (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$

- $\texttt{loop} + (\texttt{error}\ s) \ \rightsquigarrow\ \bot$

Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Propagation of Errors:

- $(\texttt{error}\ s_1) + (\texttt{error}\ s_2) \ \rightsquigarrow\ \mathit{Bad}\ \{s_1, s_2\}$
- $3 + (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$
- $\texttt{loop} + (\texttt{error}\ s) \ \rightsquigarrow\ \bot$
- $(\texttt{error}\ s_1)\ (\texttt{error}\ s_2) \ \rightsquigarrow\ \mathit{Bad}\ \{s_1, s_2\}$

Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Propagation of Errors:

- $(\texttt{error}\ s_1) + (\texttt{error}\ s_2) \ \rightsquigarrow\ \mathit{Bad}\ \{s_1, s_2\}$
- $3 + (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$
- $\texttt{loop} + (\texttt{error}\ s) \ \rightsquigarrow\ \bot$
- $(\texttt{error}\ s_1)\ (\texttt{error}\ s_2) \ \rightsquigarrow\ \mathit{Bad}\ \{s_1, s_2\}$
- $(\lambda x \rightarrow 3)\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Ok}\ 3$

Imprecise Error Semantics [Peyton Jones et al., PLDI'99]

Propagation of Errors:

- $(\texttt{error}\ s_1) + (\texttt{error}\ s_2) \ \leadsto\ \mathit{Bad}\ \{s_1, s_2\}$
- $3 + (\texttt{error}\ s) \ \leadsto\ \mathit{Bad}\ \{s\}$
- $\texttt{loop} + (\texttt{error}\ s) \ \leadsto\ \bot$
- $(\texttt{error}\ s_1)\ (\texttt{error}\ s_2) \ \leadsto\ \mathit{Bad}\ \{s_1, s_2\}$
- $(\lambda x \to 3)\ (\texttt{error}\ s) \ \leadsto\ \mathit{Ok}\ 3$
- $\textbf{case}\ (\texttt{error}\ s_1)\ \textbf{of}\ \{(x, y) \to \texttt{error}\ s_2\} \ \leadsto\ \mathit{Bad}\ \{s_1, s_2\}$

Impact on Program Equivalence

"Normally":

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ l)$$

where:

$$\begin{array}{l}
\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
\texttt{takeWhile}\ p\ [\,] \qquad = [\,] \\
\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a \qquad\quad = a : (\texttt{takeWhile}\ p\ as) \\
\qquad\qquad\qquad\qquad\quad \mid\ \texttt{otherwise} = [\,]
\end{array}$$

$$\begin{array}{l}
\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\texttt{map}\ f\ [\,] \qquad = [\,] \\
\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{array}$$

Impact on Program Equivalence

"Normally":

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ l)$$

where:

$$\begin{array}{l}
\texttt{takeWhile} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha] \\
\texttt{takeWhile}\ p\ [\,] \qquad\quad = [\,] \\
\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a \qquad\qquad = a : (\texttt{takeWhile}\ p\ as) \\
\qquad\qquad\qquad\qquad\quad \mid\ \texttt{otherwise} = [\,]
\end{array}$$

$$\begin{array}{l}
\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{array}$$

But now:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s))$$

$$\neq$$

$$\texttt{map}\ \texttt{tail}\ (\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s))$$

Impact on Program Equivalence

"Normally":

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ l)$$

where:

$$\begin{array}{l}
\texttt{takeWhile} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha] \\
\texttt{takeWhile}\ p\ [\,] \qquad = [\,] \\
\texttt{takeWhile}\ p\ (a : as) \ \mid\ p\ a \qquad\quad = a : (\texttt{takeWhile}\ p\ as) \\
\qquad\qquad\qquad\qquad\quad \mid\ \texttt{otherwise} = [\,]
\end{array}$$

$$\begin{array}{l}
\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\texttt{map}\ f\ [\,] \qquad = [\,] \\
\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{array}$$

But now:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \qquad \lightning\, s$$

$$\neq$$

$$\texttt{map}\ \texttt{tail}\ (\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s)) \qquad \lightning\, s \text{ or } \lightning\, \text{“empty list”}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \leadsto \mathit{Bad}\ \{s, \text{“empty list”}\}$$

where:

$$\begin{array}{lll} \texttt{takeWhile}\ p\ [\,] & & = [\,] \\ \texttt{takeWhile}\ p\ (a : as) & \mid\ p\ a & = a : (\texttt{takeWhile}\ p\ as) \\ & \mid\ \texttt{otherwise} & = [\,] \end{array}$$

$$\begin{array}{ll} \texttt{tail}\ [\,] & = \texttt{error}\ \text{“empty list”} \\ \texttt{tail}\ (a : as) & = as \end{array}$$

$$\begin{array}{ll} \texttt{null}\ [\,] & = \text{True} \\ \texttt{null}\ (a : as) & = \text{False} \end{array}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ [\,] &&= [\,] \\
&\texttt{takeWhile}\ p\ (a : as) \mid p\ a &&= a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\ \mid \texttt{otherwise} &&= [\,]
\end{aligned}$$

$$\begin{aligned}
&\texttt{tail}\ [\,] &&= \texttt{error}\ \text{“empty list”} \\
&\texttt{tail}\ (a : as) &&= as
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [\,] &&= \text{True} \\
&\texttt{null}\ (a : as) &&= \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ [\,] &&= [\,] \\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a &&= a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\ \mid\ \texttt{otherwise} &&= [\,]
\end{aligned}$$

$$\begin{aligned}
&\texttt{tail}\ [\,] &&= \texttt{error}\ \text{“empty list”} \\
&\texttt{tail}\ (a : as) &&= as
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [\,] &&= \text{True} \\
&\texttt{null}\ (a : as) &&= \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{``empty list''}\}$$

where:

$$\begin{array}{lll} \texttt{takeWhile}\ p\ [\,] & = [\,] & \\ \texttt{takeWhile}\ p\ (a : as) & \mid\ p\ a & = a : (\texttt{takeWhile}\ p\ as) \\ & \mid\ \texttt{otherwise} & = [\,] \end{array}$$

$$\begin{array}{ll} \texttt{tail}\ [\,] & = \texttt{error}\ \text{``empty list''} \\ \texttt{tail}\ (a : as) & = as \end{array}$$

$$\begin{array}{ll} \texttt{null}\ [\,] & = \text{True} \\ \texttt{null}\ (a : as) & = \text{False} \end{array}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{``empty list''}\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ []\ &&= []\\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a &&= a : (\texttt{takeWhile}\ p\ as)\\
&\qquad\qquad\qquad\quad\ \ \mid\ \texttt{otherwise} &&= []
\end{aligned}$$

$$\begin{aligned}
&\texttt{tail}\ [] &&= \texttt{error}\ \text{``empty list''}\\
&\texttt{tail}\ (a : as) &&= as
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [] &&= \text{True}\\
&\texttt{null}\ (a : as) &&= \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ [\,] && = [\,] \\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a && = a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\ \mid\ \texttt{otherwise} && = [\,]
\end{aligned}$$

$$\begin{aligned}
&\texttt{tail}\ [\,] && = \texttt{error}\ \text{“empty list”} \\
&\texttt{tail}\ (a : as) && = as
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [\,] && = \text{True} \\
&\texttt{null}\ (a : as) && = \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ [\,] && = [\,] \\
&\texttt{takeWhile}\ p\ (a : as) \mid p\ a && = a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\ \mid \texttt{otherwise} && = [\,]
\end{aligned}$$

$$\begin{aligned}
&\texttt{tail}\ [\,] && = \texttt{error}\ \text{“empty list”} \\
&\texttt{tail}\ (a : as) && = as
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [\,] && = \text{True} \\
&\texttt{null}\ (a : as) && = \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$$

where:

$$\begin{array}{lll}
\texttt{takeWhile}\ p\ [\,] & & = [\,] \\
\texttt{takeWhile}\ p\ (a : as) & \mid\ p\ a & = a : (\texttt{takeWhile}\ p\ as) \\
 & \mid\ \texttt{otherwise} & = [\,]
\end{array}$$

$$\begin{array}{ll}
\texttt{map}\ f\ [\,] & = [\,] \\
\texttt{map}\ f\ (a : as) & = (f\ a) : (\texttt{map}\ f\ as)
\end{array}$$

$$\begin{array}{ll}
\texttt{null}\ [\,] & = \text{True} \\
\texttt{null}\ (a : as) & = \text{False}
\end{array}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{"empty list"}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ [\,] && = [\,] \\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a && = a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\quad \mid\ \texttt{otherwise} && = [\,]
\end{aligned}$$

$$\begin{aligned}
&\texttt{map}\ f\ [\,] && = [\,] \\
&\texttt{map}\ f\ (a : as) && = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [\,] && = \text{True} \\
&\texttt{null}\ (a : as) && = \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s)\ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s))\ \rightsquigarrow\ \mathit{Bad}\ \{s\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ []\ &&= []\\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a &&= a : (\texttt{takeWhile}\ p\ as)\\
&\qquad\qquad\qquad\quad \mid\ \texttt{otherwise} &&= []
\end{aligned}$$

$$\begin{aligned}
&\texttt{map}\ f\ [] &&= []\\
&\texttt{map}\ f\ (a : as) &&= (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [] &&= \text{True}\\
&\texttt{null}\ (a : as) &&= \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s)\ \leadsto\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s))\ \leadsto\ \mathit{Bad}\ \{s\}$$

where:

$$\begin{array}{lll} \texttt{takeWhile}\ p\ [\,] & & = [\,] \\ \texttt{takeWhile}\ p\ (a : as) & \mid\ p\ a & = a : (\texttt{takeWhile}\ p\ as) \\ & \mid\ \texttt{otherwise} & = [\,] \end{array}$$

$$\begin{array}{ll} \texttt{map}\ f\ [\,] & = [\,] \\ \texttt{map}\ f\ (a : as) & = (f\ a) : (\texttt{map}\ f\ as) \end{array}$$

$$\begin{array}{ll} \texttt{null}\ [\,] & = \text{True} \\ \texttt{null}\ (a : as) & = \text{False} \end{array}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$$

where:

$$\begin{array}{lll}
\texttt{takeWhile}\ p\ [\,] & & = [\,] \\
\texttt{takeWhile}\ p\ (a : as) & \mid\ p\ a & = a : (\texttt{takeWhile}\ p\ as) \\
 & \mid\ \texttt{otherwise} & = [\,]
\end{array}$$

$$\begin{array}{ll}
\texttt{map}\ f\ [\,] & = [\,] \\
\texttt{map}\ f\ (a : as) & = (f\ a) : (\texttt{map}\ f\ as)
\end{array}$$

$$\begin{array}{ll}
\texttt{null}\ [\,] & = \text{True} \\
\texttt{null}\ (a : as) & = \text{False}
\end{array}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$$

where:

$$\begin{aligned}
&\texttt{takeWhile}\ p\ [\,] && = [\,] \\
&\texttt{takeWhile}\ p\ (a : as) \mid p\ a && = a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\quad\ \mid \texttt{otherwise} && = [\,]
\end{aligned}$$

$$\begin{aligned}
&\texttt{map}\ f\ [\,] && = [\,] \\
&\texttt{map}\ f\ (a : as) && = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

$$\begin{aligned}
&\texttt{null}\ [\,] && = \text{True} \\
&\texttt{null}\ (a : as) && = \text{False}
\end{aligned}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s) \ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \ \rightsquigarrow\ \mathit{Bad}\ \{s\}$$

Thus:

$$\begin{array}{c} \texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \\ \neq \\ \texttt{map}\ \texttt{tail}\ (\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s)) \end{array}$$

Impact on Program Equivalence

Because:

$$\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s)\ \rightsquigarrow\ \mathit{Bad}\ \{s, \text{“empty list”}\}$$

while:

$$\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s))\ \rightsquigarrow\ \mathit{Bad}\ \{s\}$$

Thus:

$$\begin{gathered}\texttt{takeWhile}\ \texttt{null}\ (\texttt{map}\ \texttt{tail}\ (\texttt{error}\ s)) \\ \neq \\ \texttt{map}\ \texttt{tail}\ (\texttt{takeWhile}\ (\texttt{null} \circ \texttt{tail})\ (\texttt{error}\ s))\end{gathered}$$

Now, imagine this in the following program context:

$$\begin{aligned}&\texttt{catchJust}\ \texttt{errorCalls}\ (\texttt{evaluate}\ \cdots) \\ &\qquad (\lambda s \rightarrow \textbf{if}\ s == \text{“empty list”} \\ &\qquad\qquad \textbf{then}\ \texttt{return}\ [[42]] \\ &\qquad\qquad \textbf{else}\ \ \texttt{return}\ [\,])\end{aligned}$$

How to Revise Free Theorems?

[Wadler, FPCA'89] : for every $\texttt{g} :: (\alpha \rightarrow \mathsf{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

How to Revise Free Theorems?

[Wadler, FPCA'89] : for every $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

[Johann & V., POPL'04] : in Haskell only provable if:

- $p \neq \bot$,
- f strict ($f\ \bot = \bot$), and
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

How to Revise Free Theorems?

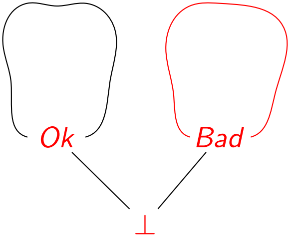
[Wadler, FPCA'89] : for every g :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

[Johann & V., POPL'04] : in Haskell only provable if:

- $p \neq \bot$,
- f strict ($f\ \bot = \bot$), and
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

What are corresponding conditions "in real"?



Sweat and Tears ...

...provide full formalisation

Sweat and Tears . . .

. . . provide full formalisation

. . . enter general proof of parametricity theorem

Sweat and Tears . . .

. . . provide full formalisation

. . . enter general proof of parametricity theorem

. . . identify appropriate restrictions on the level of relations

Sweat and Tears . . .

. . . provide full formalisation

. . . enter general proof of parametricity theorem

. . . identify appropriate restrictions on the level of relations

. . . adapt relational actions

Sweat and Tears . . .

. . . provide full formalisation

. . . enter general proof of parametricity theorem

. . . identify appropriate restrictions on the level of relations

. . . adapt relational actions

. . . complete general proof

Sweat and Tears . . .

. . . provide full formalisation

. . . enter general proof of parametricity theorem

. . . identify appropriate restrictions on the level of relations

. . . adapt relational actions

. . . complete general proof

. . . transfer restrictions to the level of functions

Sweat and Tears . . .

. . . provide full formalisation

. . . enter general proof of parametricity theorem

. . . identify appropriate restrictions on the level of relations

. . . adapt relational actions

. . . complete general proof

. . . transfer restrictions to the level of functions

. . . apply to concrete functions

Sweat and Tears . . .

. . . provide full formalisation

. . . enter general proof of parametricity theorem

. . . identify appropriate restrictions on the level of relations

. . . adapt relational actions

. . . complete general proof

. . . transfer restrictions to the level of functions

. . . apply to concrete functions

(. . . similarly for "asymmetric" scenarios as well)

... Application to `takeWhile`

For every $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

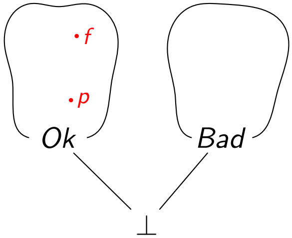
... Application to takeWhile

For every $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

provided

- p and f are nonerroneous,



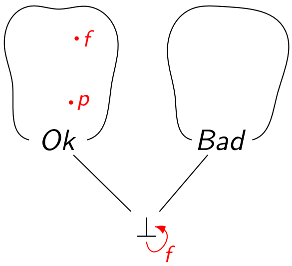
. . . Application to takeWhile

For every $\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$,

$$\mathtt{g}\ p\ (\mathtt{map}\ f\ l) \quad = \quad \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$$

provided

- p and f are nonerroneous,
- $f\ \bot = \bot$,



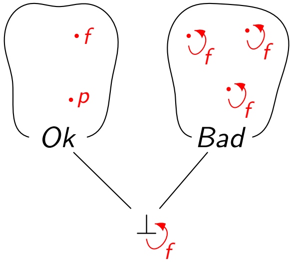
... Application to `takeWhile`

For every $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

provided

- p and f are nonerroneous,
- $f\ \bot = \bot$,
- f acts as identity on erroneous values, and



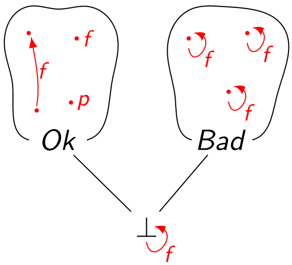
...Application to takeWhile

For every $\mathtt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\mathtt{g}\ p\ (\mathtt{map}\ f\ l) \quad = \quad \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$$

provided

- p and f are nonerroneous,
- $f\ \bot = \bot$,
- f acts as identity on erroneous values, and
- f maps nonerroneous values to nonerroneous values.



Summary and Outlook

Types:

- constrain the behaviour of programs

Summary and Outlook

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs

Summary and Outlook

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

Summary and Outlook

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages

Summary and Outlook

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages
- strive for more expressive type systems

Summary and Outlook

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages
- strive for more expressive type systems

On the practical side:

- efficiency-improving program transformations

Summary and Outlook

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages
- strive for more expressive type systems

On the practical side:

- efficiency-improving program transformations
- applications in specific domains

References I

P. Hudak, R.J.M. Hughes, S.L. Peyton Jones, and P. Wadler.
A history of Haskell: Being lazy with class.
In *History of Programming Languages, Proceedings*, pages 12-1–12-55. ACM Press, 2007.

P. Johann and J. Voigtländer.
Free theorems in the presence of seq.
In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

P. Johann and J. Voigtländer.
A family of syntactic logical relations for the semantics of Haskell-like languages.
Information and Computation, 2009.

References II

S.L. Peyton Jones, A. Reid, C.A.R. Hoare, S. Marlow, and F. Henderson.
A semantics for imprecise exceptions.
In *Programming Language Design and Implementation, Proceedings*, pages 25–36. ACM Press, 1999.

F. Stenger and J. Voigtländer.
Parametricity for Haskell with imprecise error semantics.
Technical Report TUD-FI08-08, Technische Universität Dresden, 2008.

P. Wadler.
Theorems for free!
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.