Type-Based Reasoning for Real Languages

Janis Voigtländer

University of Bonn

PPL'10

Parametric Polymorphism in Haskell

A standard function:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism in Haskell

A standard function:

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

Some invocations:

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

Parametric Polymorphism in Haskell

A standard function:

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

Some invocations:

$$\begin{array}{llll}
\texttt{map succ} & [1,2,3] & = [2,3,4] & \text{— } \alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int} \\
\texttt{map not} & [\mathsf{True}, \mathsf{False}] & = [\mathsf{False}, \mathsf{True}] & \text{— } \alpha, \beta \mapsto \mathsf{Bool}, \mathsf{Bool} \\
\texttt{map even} & [1,2,3] & = [\mathsf{False}, \mathsf{True}, \mathsf{False}] & \text{— } \alpha, \beta \mapsto \mathsf{Int}, \mathsf{Bool} \\
\texttt{map not} & [1,2,3] & &
\end{array}$$

Parametric Polymorphism in Haskell

A standard function:

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

Some invocations:

$$\begin{array}{llll}
\texttt{map succ} & [1,2,3] & = [2,3,4] & \text{— } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Int} \\
\texttt{map not} & [\mathsf{True},\mathsf{False}] & = [\mathsf{False},\mathsf{True}] & \text{— } \alpha,\beta \mapsto \mathsf{Bool},\mathsf{Bool} \\
\texttt{map even} & [1,2,3] & = [\mathsf{False},\mathsf{True},\mathsf{False}] & \text{— } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Bool} \\
\texttt{map not} & [1,2,3] & \lightning \text{ rejected at compile-time} &
\end{array}$$

Parametric Polymorphism in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

Some invocations:

`map succ`	$[1, 2, 3]$	$= [2, 3, 4]$	— $\alpha, \beta \mapsto$ Int, Int
`map not`	$[\mathsf{True}, \mathsf{False}]$	$= [\mathsf{False}, \mathsf{True}]$	— $\alpha, \beta \mapsto$ Bool, Bool
`map even`	$[1, 2, 3]$	$= [\mathsf{False}, \mathsf{True}, \mathsf{False}]$	— $\alpha, \beta \mapsto$ Int, Bool
`map not`	$[1, 2, 3]$	$\lightning$ rejected at compile-time	

Another Example

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

Another Example

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

For every choice of p, f, and l:

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

Provable by induction.

Another Example

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

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

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

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

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

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

For every choice of p, f, and l:

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

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

Another Example

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

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

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

For every choice of p, f, and l:

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

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

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

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

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

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

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

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

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

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ 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 $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ 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 $(\texttt{map}\ f\ l)$ and l always have equal length.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ 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 $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ 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 $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l,

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ 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 $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ 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 $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.
- $\texttt{g}\ p\ (\texttt{map}\ f\ l)$ is equivalent to $\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$.

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ 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 $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\mathtt{g}$ with p always chooses "the same" elements from $(\mathtt{map}\ f\ l)$ for output as does $\mathtt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.
- $\mathtt{g}\ p\ (\mathtt{map}\ f\ l)$ is equivalent to $\mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$.
- That is what was claimed!

Automatic Generation of Free Theorems

At `http://www-ps.iai.uni-bonn.de/ft`:

This tool allows to generate free theorems for sublanguages of Haskell as described here.

The source code of the underlying library and a shell-based application using it is available here and here.

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]" or simply "filter":

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

Please choose a sublanguage of Haskell:

- (•) no bottoms (hence no general recursion and no selective strictness)
- ( ) general recursion but no selective strictness
- ( ) general recursion and selective strictness

Please choose a theorem style (without effect in the sublanguage with no bottoms):

- (•) equational
- ( ) inequational

[Generate]

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

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]
- Bidirectionalisation [V., POPL'09]

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]
- Bidirectionalisation [V., POPL'09]
- Reasoning about invariants for monadic programs [V., ICFP'09]

Automatic Generation of Free Theorems

At `http://www-ps.iai.uni-bonn.de/ft`:

This tool allows to generate free theorems for sublanguages of Haskell as described here.

The source code of the underlying library and a shell-based application using it is available here and here.

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]" or simply "filter":

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

Please choose a sublanguage of Haskell:

- (•) no bottoms (hence no general recursion and no selective strictness)
- ( ) general recursion but no selective strictness
- ( ) general recursion and selective strictness

Please choose a theorem style (without effect in the sublanguage with no bottoms):

- (•) equational
- ( ) inequational

Generate

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd*

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd* (or semantically equivalent to one of them).

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd* (or semantically equivalent to one of them).

But how to give a formal answer?

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd* (or semantically equivalent to one of them).

But how to give a formal answer?

Clearly, any g of that type must semantically be a collection of functions of types $(\tau, \tau) \to \tau$, indexed over by τ.

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd* (or semantically equivalent to one of them).

But how to give a formal answer?

Clearly, any $\mathtt{g}$ of that type must semantically be a collection of functions of types $(\tau, \tau) \to \tau$, indexed over by τ.

But this also allows a $\mathtt{g}$ with

$$\begin{aligned} \mathtt{g}_{\mathsf{Bool}}\ (x, y) &= \mathtt{not}\ x \quad \text{and} \\ \mathtt{g}_{\mathsf{Int}}\ (x, y) &= y + 1 \end{aligned}$$

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd* (or semantically equivalent to one of them).

But how to give a formal answer?

Clearly, any g of that type must semantically be a collection of functions of types $(\tau, \tau) \to \tau$, indexed over by τ.

But this also allows a g with

$$\begin{aligned} \mathtt{g}_{\mathsf{Bool}}\,(x, y) &= \mathtt{not}\ x \quad \text{and} \\ \mathtt{g}_{\mathsf{Int}}\,(x, y) &= y + 1\,, \end{aligned}$$

which is not possible in Haskell at type $(\alpha, \alpha) \to \alpha$!

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd* (or semantically equivalent to one of them).

But how to give a formal answer?

Clearly, any $\mathtt{g}$ of that type must semantically be a collection of functions of types $(\tau, \tau) \to \tau$, indexed over by τ.

But this also allows a $\mathtt{g}$ with

$$\begin{aligned} \mathtt{g}_{\mathsf{Bool}}\ (x, y) &= \mathtt{not}\ x \quad \text{and} \\ \mathtt{g}_{\mathsf{Int}}\ (x, y) &= y + 1\,, \end{aligned}$$

which is not possible in Haskell at type $(\alpha, \alpha) \to \alpha$!

To prevent this, we have to compare

$$\mathtt{g}_{\mathsf{Bool}} :: (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool} \quad \text{vs.} \quad \mathtt{g}_{\mathsf{Int}} :: (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}$$

and ensure that they "behave identically".

A Simpler Example

Question: What do we know about functions of type $(\alpha, \alpha) \to \alpha$?

Intuitively: Can only be *fst* or *snd* (or semantically equivalent to one of them).

But how to give a formal answer?

Clearly, any `g` of that type must semantically be a collection of functions of types $(\tau, \tau) \to \tau$, indexed over by τ.

But this also allows a `g` with

$$\begin{aligned} \mathtt{g}_{\mathsf{Bool}}\ (x, y) &= \mathtt{not}\ x \quad \text{and} \\ \mathtt{g}_{\mathsf{Int}}\ \ (x, y) &= y + 1\,, \end{aligned}$$

which is not possible in Haskell at type $(\alpha, \alpha) \to \alpha$!

To prevent this, we have to compare

$$\mathtt{g}_{\mathsf{Bool}} :: (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool} \quad \text{vs.} \quad \mathtt{g}_{\mathsf{Int}} :: (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}$$

and ensure that they "behave identically". But how?

Key Idea [Reynolds 1983]

Use arbitrary relations to tie instances together!

Key Idea [Reynolds 1983]

Use arbitrary relations to tie instances together!

In the example ($\mathrm{g} :: (\alpha, \alpha) \to \alpha$):

- Choose a relation $\mathcal{R} \subseteq \mathsf{Bool} \times \mathsf{Int}$.

Key Idea [Reynolds 1983]

Use arbitrary relations to tie instances together!

In the example ($\mathrm{g} :: (\alpha, \alpha) \to \alpha$):

- Choose a relation $\mathcal{R} \subseteq \mathsf{Bool} \times \mathsf{Int}$.
- Call $(x_1, x_2) :: (\mathsf{Bool}, \mathsf{Bool})$ and $(y_1, y_2) :: (\mathsf{Int}, \mathsf{Int})$ related if $(x_1, y_1) \in \mathcal{R}$ and $(x_2, y_2) \in \mathcal{R}$.

Key Idea [Reynolds 1983]

Use arbitrary relations to tie instances together!

In the example ($g :: (\alpha, \alpha) \to \alpha$):

- Choose a relation $\mathcal{R} \subseteq \mathsf{Bool} \times \mathsf{Int}$.
- Call $(x_1, x_2) :: (\mathsf{Bool}, \mathsf{Bool})$ and $(y_1, y_2) :: (\mathsf{Int}, \mathsf{Int})$ related if $(x_1, y_1) \in \mathcal{R}$ and $(x_2, y_2) \in \mathcal{R}$.
- Call $f_1 :: (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}$, $f_2 :: (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}$ related if related inputs lead to related outputs.

Key Idea [Reynolds 1983]

Use arbitrary relations to tie instances together!

In the example ($g :: (\alpha, \alpha) \to \alpha$):

- Choose a relation $\mathcal{R} \subseteq \mathsf{Bool} \times \mathsf{Int}$.
- Call $(x_1, x_2) :: (\mathsf{Bool}, \mathsf{Bool})$ and $(y_1, y_2) :: (\mathsf{Int}, \mathsf{Int})$ related if $(x_1, y_1) \in \mathcal{R}$ and $(x_2, y_2) \in \mathcal{R}$.
- Call $f_1 :: (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}$, $f_2 :: (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}$ related if related inputs lead to related outputs.
- Then g_{Bool} and g_{Int} with

$$\begin{aligned} g_{\mathsf{Bool}}\ (x, y) &= \texttt{not}\ x \\ g_{\mathsf{Int}}\ (x, y) &= y + 1 \end{aligned}$$

  are not related for choice of, e.g., $\mathcal{R} = \{(\mathsf{True}, 1)\}$.

Key Idea [Reynolds 1983]

Use arbitrary relations to tie instances together!

In the example ($g :: (\alpha, \alpha) \to \alpha$):

- Choose a relation $\mathcal{R} \subseteq \text{Bool} \times \text{Int}$.
- Call $(x_1, x_2) :: (\text{Bool}, \text{Bool})$ and $(y_1, y_2) :: (\text{Int}, \text{Int})$ related if $(x_1, y_1) \in \mathcal{R}$ and $(x_2, y_2) \in \mathcal{R}$.
- Call $f_1 :: (\text{Bool}, \text{Bool}) \to \text{Bool}$, $f_2 :: (\text{Int}, \text{Int}) \to \text{Int}$ related if related inputs lead to related outputs.
- Then g_{Bool} and g_{Int} with

$$\begin{aligned} g_{\text{Bool}}\ (x, y) &= \mathtt{not}\ x \\ g_{\text{Int}}\ (x, y) &= y + 1 \end{aligned}$$

  are not related for choice of, e.g., $\mathcal{R} = \{(\text{True}, 1)\}$.

Reynolds: $g :: \tau$, with α free in τ, iff for every $\tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2$, g_{τ_1} is related to g_{τ_2} by the "propagation" of $\mathcal{R}$ (replaced for α) along τ.

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$(\mathcal{R}, \mathcal{S}) \quad = \quad \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\}$$

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\} \\
[\mathcal{R}] & = & \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\}
\end{array}
$$

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R},\mathcal{S}) & = & \{((x_1,x_2),(y_1,y_2)) \mid (x_1,y_1)\in\mathcal{R},(x_2,y_2)\in\mathcal{S}\} \\
[\mathcal{R}] & = & \{([x_1,\dots,x_n],[y_1,\dots,y_n]) \mid n\geq 0,(x_i,y_i)\in\mathcal{R}\} \\
\mathcal{R}\to\mathcal{S} & = & \{(f_1,f_2) \mid \forall (a_1,a_2)\in\mathcal{R}.\ (f_1\ a_1,f_2\ a_2)\in\mathcal{S}\}
\end{array}
$$

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\} \\
[\mathcal{R}] & = & \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\} \\
\mathcal{R} \to \mathcal{S} & = & \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}.\ (f_1\ a_1, f_2\ a_2) \in \mathcal{S}\} \\
\forall \mathcal{R}.\mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}(\mathcal{R})\}
\end{array}
$$

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\} \\
[\mathcal{R}] & = & \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\} \\
\mathcal{R} \rightarrow \mathcal{S} & = & \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}.\ (f_1\ a_1, f_2\ a_2) \in \mathcal{S}\} \\
\forall \mathcal{R}.\mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}(\mathcal{R})\}
\end{array}
$$

Then for every $\mathrm{g} :: \tau$, the pair (g, g) is contained in the relational interpretation of τ.

Now Formal Counterpart to Intuitive Reasoning

Let $\mathrm{g} :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.

Now Formal Counterpart to Intuitive Reasoning

Let $\mathrm{g} :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$(\mathrm{g}, \mathrm{g}) \in \forall \mathcal{R}.(\mathcal{R} \to \mathit{id}_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}])$$

Now Formal Counterpart to Intuitive Reasoning

Let $g :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$(g, g) \in \forall \mathcal{R}. (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}])$$
$$\Leftrightarrow \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}])$$

by definition of $\forall \mathcal{R}. \mathcal{F}(\mathcal{R})$

Now Formal Counterpart to Intuitive Reasoning

Let $\mathrm{g} :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$(\mathrm{g}, \mathrm{g}) \in \forall \mathcal{R}.(\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}])$$
$$\Leftrightarrow \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (\mathrm{g}_{\tau_1}, \mathrm{g}_{\tau_2}) \in (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}])$$
$$\Leftrightarrow \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ (\mathrm{g}_{\tau_1}\ a_1, \mathrm{g}_{\tau_2}\ a_2) \in ([\mathcal{R}] \to [\mathcal{R}])$$

by definition of $\mathcal{R} \to \mathcal{S}$

Now Formal Counterpart to Intuitive Reasoning

Let $\mathrm{g} :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$
\begin{aligned}
& (\mathrm{g}, \mathrm{g}) \in \forall \mathcal{R}. (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (\mathrm{g}_{\tau_1}, \mathrm{g}_{\tau_2}) \in (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ (\mathrm{g}_{\tau_1}\ a_1, \mathrm{g}_{\tau_2}\ a_2) \in ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ \forall (l_1, l_2) \in [\mathcal{R}]. \\
& \qquad\qquad\qquad\qquad (\mathrm{g}_{\tau_1}\ a_1\ l_1, \mathrm{g}_{\tau_2}\ a_2\ l_2) \in [\mathcal{R}]
\end{aligned}
$$

by definition of $\mathcal{R} \to \mathcal{S}$

Now Formal Counterpart to Intuitive Reasoning

Let $\mathrm{g} :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$
\begin{aligned}
& (\mathrm{g}, \mathrm{g}) \in \forall \mathcal{R}.(\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (\mathrm{g}_{\tau_1}, \mathrm{g}_{\tau_2}) \in (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ (\mathrm{g}_{\tau_1}\ a_1, \mathrm{g}_{\tau_2}\ a_2) \in ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ \forall (l_1, l_2) \in [\mathcal{R}]. \\
& \qquad\qquad\qquad\qquad (\mathrm{g}_{\tau_1}\ a_1\ l_1, \mathrm{g}_{\tau_2}\ a_2\ l_2) \in [\mathcal{R}] \\
\Rightarrow\ & \forall (a_1, a_2) \in (f \to id_{\mathsf{Bool}}).\ \forall (l_1, l_2) \in (\mathtt{map}\ f). \\
& \qquad\qquad\qquad\qquad (\mathrm{g}_{\tau_1}\ a_1\ l_1, \mathrm{g}_{\tau_2}\ a_2\ l_2) \in (\mathtt{map}\ f)
\end{aligned}
$$

by instantiating $\mathcal{R} = f$ and realising that then $[\mathcal{R}] = (\mathtt{map}\ f)$

for every function $f :: \tau_1 \to \tau_2$

Now Formal Counterpart to Intuitive Reasoning

Let $\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$
\begin{aligned}
& (\mathtt{g}, \mathtt{g}) \in \forall \mathcal{R}. (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (\mathtt{g}_{\tau_1}, \mathtt{g}_{\tau_2}) \in (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ (\mathtt{g}_{\tau_1}\ a_1, \mathtt{g}_{\tau_2}\ a_2) \in ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ \forall (l_1, l_2) \in [\mathcal{R}]. \\
& \qquad\qquad\qquad\qquad (\mathtt{g}_{\tau_1}\ a_1\ l_1, \mathtt{g}_{\tau_2}\ a_2\ l_2) \in [\mathcal{R}] \\
\Rightarrow\ & \forall (a_1, a_2) \in (f \to id_{\mathsf{Bool}}).\ \forall (l_1, l_2) \in (\mathtt{map}\ f). \\
& \qquad\qquad\qquad (\mathtt{g}_{\tau_1}\ a_1\ l_1, \mathtt{g}_{\tau_2}\ a_2\ l_2) \in (\mathtt{map}\ f) \\
\Rightarrow\ & \forall (l_1, l_2) \in (\mathtt{map}\ f).\ (\mathtt{g}_{\tau_1}\ (p \circ f)\ l_1, \mathtt{g}_{\tau_2}\ p\ l_2) \in (\mathtt{map}\ f) \\
& \text{by instantiating } (a_1, a_2) = (p \circ f, p) \in (f \to id_{\mathsf{Bool}})
\end{aligned}
$$

for every function $f :: \tau_1 \to \tau_2$ and predicate $p :: \tau_2 \to \mathsf{Bool}$.

Now Formal Counterpart to Intuitive Reasoning

Let $\texttt{g} :: (\alpha \to \text{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$
\begin{aligned}
& (\texttt{g}, \texttt{g}) \in \forall \mathcal{R}.(\mathcal{R} \to id_{\text{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (\texttt{g}_{\tau_1}, \texttt{g}_{\tau_2}) \in (\mathcal{R} \to id_{\text{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\text{Bool}}).\ (\texttt{g}_{\tau_1}\ a_1, \texttt{g}_{\tau_2}\ a_2) \in ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\text{Bool}}).\ \forall (l_1, l_2) \in [\mathcal{R}]. \\
& \qquad (\texttt{g}_{\tau_1}\ a_1\ l_1, \texttt{g}_{\tau_2}\ a_2\ l_2) \in [\mathcal{R}] \\
\Rightarrow\ & \forall (a_1, a_2) \in (f \to id_{\text{Bool}}).\ \forall (l_1, l_2) \in (\texttt{map}\ f). \\
& \qquad (\texttt{g}_{\tau_1}\ a_1\ l_1, \texttt{g}_{\tau_2}\ a_2\ l_2) \in (\texttt{map}\ f) \\
\Rightarrow\ & \forall (l_1, l_2) \in (\texttt{map}\ f).\ (\texttt{g}_{\tau_1}\ (p \circ f)\ l_1, \texttt{g}_{\tau_2}\ p\ l_2) \in (\texttt{map}\ f) \\
\Leftrightarrow\ & \forall l_1 :: [\tau_1].\ \texttt{map}\ f\ (\texttt{g}_{\tau_1}\ (p \circ f)\ l_1) = \texttt{g}_{\tau_2}\ p\ (\texttt{map}\ f\ l_1)
\end{aligned}
$$

by inlining

for every function $f :: \tau_1 \to \tau_2$ and predicate $p :: \tau_2 \to \text{Bool}$.

Now Formal Counterpart to Intuitive Reasoning

Let $\mathrm{g} :: (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.
Then:

$$
\begin{aligned}
& (\mathrm{g}, \mathrm{g}) \in \forall \mathcal{R}. (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (\mathrm{g}_{\tau_1}, \mathrm{g}_{\tau_2}) \in (\mathcal{R} \to id_{\mathsf{Bool}}) \to ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ (\mathrm{g}_{\tau_1}\ a_1, \mathrm{g}_{\tau_2}\ a_2) \in ([\mathcal{R}] \to [\mathcal{R}]) \\
\Leftrightarrow\ & \forall \mathcal{R}.\ \forall (a_1, a_2) \in (\mathcal{R} \to id_{\mathsf{Bool}}).\ \forall (l_1, l_2) \in [\mathcal{R}]. \\
& \qquad\qquad\qquad\qquad (\mathrm{g}_{\tau_1}\ a_1\ l_1, \mathrm{g}_{\tau_2}\ a_2\ l_2) \in [\mathcal{R}] \\
\Rightarrow\ & \forall (a_1, a_2) \in (f \to id_{\mathsf{Bool}}).\ \forall (l_1, l_2) \in (\mathtt{map}\ f). \\
& \qquad\qquad\qquad\qquad (\mathrm{g}_{\tau_1}\ a_1\ l_1, \mathrm{g}_{\tau_2}\ a_2\ l_2) \in (\mathtt{map}\ f) \\
\Rightarrow\ & \forall (l_1, l_2) \in (\mathtt{map}\ f).\ (\mathrm{g}_{\tau_1}\ (p \circ f)\ l_1, \mathrm{g}_{\tau_2}\ p\ l_2) \in (\mathtt{map}\ f) \\
\Leftrightarrow\ & \forall l_1 :: [\tau_1].\ \mathtt{map}\ f\ (\mathrm{g}_{\tau_1}\ (p \circ f)\ l_1) = \mathrm{g}_{\tau_2}\ p\ (\mathtt{map}\ f\ l_1)
\end{aligned}
$$

for every function $f :: \tau_1 \to \tau_2$ and predicate $p :: \tau_2 \to \mathsf{Bool}$.

That is what was claimed!

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\} \\
[\mathcal{R}] & = & \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\} \\
\mathcal{R} \rightarrow \mathcal{S} & = & \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}.\ (f_1\ a_1, f_2\ a_2) \in \mathcal{S}\} \\
\forall \mathcal{R}.\mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}(\mathcal{R})\}
\end{array}
$$

Then for every $\mathrm{g} :: \tau$, the pair (g, g) is contained in the relational interpretation of τ.

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall\alpha.\tau$

Terms: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall \alpha . \tau$
Terms: $t := x \mid \lambda x : \tau . t \mid t\ t \mid \Lambda \alpha . t \mid t\ \tau$

$$\Gamma, x : \tau \vdash x : \tau$$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall \alpha.\tau$

Terms: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda \alpha.t \mid t\ \tau$

$$\Gamma, x : \tau \vdash x : \tau \qquad\qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2}$$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall \alpha . \tau$

Terms: $t := x \mid \lambda x : \tau . t \mid t \; t \mid \Lambda \alpha . t \mid t \; \tau$

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . t) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t \; u) : \tau_2}$$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall\alpha.\tau$
Terms: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau$

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2} \qquad \frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau}$$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall \alpha . \tau$

Terms: $t := x \mid \lambda x : \tau . t \mid t \; t \mid \Lambda \alpha . t \mid t \; \tau$

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . t) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t \; u) : \tau_2}$$

$$\frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda \alpha . t) : \forall \alpha . \tau}$$

$$\frac{\Gamma \vdash t : \forall \alpha . \tau}{\Gamma \vdash (t \; \tau') : \tau[\tau' / \alpha]}$$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid \mathsf{Bool} \mid [\tau]$

Terms: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau$

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2}$$

$$\frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau}$$

$$\frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash (t\ \tau') : \tau[\tau'/\alpha]}$$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \rightarrow \tau \mid \forall\alpha.\tau \mid \mathsf{Bool} \mid [\tau]$

Terms: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau \mid$
$\quad \mathsf{True} \mid \mathsf{False} \mid [\,]_\tau \mid t : t \mid \textbf{case}\ t\ \textbf{of}\ \{\cdots\}$

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2} \qquad \frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau}$$

$$\frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash (t\ \tau') : \tau[\tau'/\alpha]}$$

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid \mathsf{Bool} \mid [\tau]$

Terms: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau \mid$
$\mathsf{True} \mid \mathsf{False} \mid [\,]_\tau \mid t : t \mid \textbf{case}\ t\ \textbf{of}\ \{\cdots\}$

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2} \qquad \frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau}$$

$$\frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash (t\ \tau') : \tau[\tau'/\alpha]} \qquad \frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash u : [\tau]}{\Gamma \vdash (t : u) : [\tau]}$$

$$\Gamma \vdash \mathsf{True} : \mathsf{Bool} \quad , \quad \Gamma \vdash \mathsf{False} : \mathsf{Bool} \quad , \quad \Gamma \vdash [\,]_\tau : [\tau]$$

$$\frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma \vdash u : \tau \qquad \Gamma \vdash v : \tau}{\Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{\mathsf{True} \to u\,; \mathsf{False} \to v\}) : \tau}$$

$$\frac{\Gamma \vdash t : [\tau'] \qquad \Gamma \vdash u : \tau \qquad \Gamma, x_1 : \tau', x_2 : [\tau'] \vdash v : \tau}{\Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{[] \to u\,; (x_1 : x_2) \to v\}) : \tau}$$

Deriving Free Theorems, in General

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R},\mathcal{S}) & = & \{((x_1,x_2),(y_1,y_2)) \mid (x_1,y_1)\in\mathcal{R},(x_2,y_2)\in\mathcal{S}\} \\
[\mathcal{R}] & = & \{([x_1,\ldots,x_n],[y_1,\ldots,y_n]) \mid n\geq 0,(x_i,y_i)\in\mathcal{R}\} \\
\mathcal{R}\rightarrow\mathcal{S} & = & \{(f_1,f_2) \mid \forall(a_1,a_2)\in\mathcal{R}.\ (f_1\ a_1,f_2\ a_2)\in\mathcal{S}\} \\
\forall\mathcal{R}.\mathcal{F}(\mathcal{R}) & = & \{(u,v) \mid \forall\tau_1,\tau_2,\mathcal{R}\subseteq\tau_1\times\tau_2.\ (u_{\tau_1},v_{\tau_2})\in\mathcal{F}(\mathcal{R})\}
\end{array}
$$

Then for every $\mathrm{g} :: \tau$, the pair (g,g) is contained in the relational interpretation of τ.

General Recursion

We had that for every

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

it holds

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

for every choice of p, f, and l.

General Recursion

We had that for every

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

it holds

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

for every choice of p, f, and l.

What about

$$\begin{aligned} &\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\ &\texttt{g}\ p\ l = [\texttt{head}\ (\texttt{g}\ p\ l)] \end{aligned} \qquad ?$$

General Recursion

We had that for every

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

it holds

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

for every choice of p, f, and l.

What about

$$\begin{array}{l} \texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\ \texttt{g}\ p\ l = [\texttt{head}\ (\texttt{g}\ p\ l)] \end{array} \qquad ?$$

The above free theorem fails!

Consider, e.g., $p = \texttt{id}$, $f = \texttt{const}$ True, and $l = [\,]$.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ 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 $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.
- $\texttt{g}\ p\ (\texttt{map}\ f\ l)$ is equivalent to $\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$.
- That is what was claimed!

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

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

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

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

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

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

↯ Not true! Also possible: $\perp$.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ 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.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

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

- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\perp$.

- 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.

Why g p (map f l) = map f (g ($p \circ f$) l), 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.

$\lightning$ 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.

$\lightning$ Not true! Also possible: checking outcome of p on $\perp$.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ 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 outcome of p on $\bot$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ 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 outcome of p on $\bot$.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

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

$\lightning$ 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.

$\lightning$ Not true! Also possible: checking outcome of p on $\bot$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\bot$ has the same outcome as applying $(p \circ f)$ to $\bot$, provided f is strict ($f\ \bot = \bot$).

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ 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 outcome of p on $\bot$.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\bot$ has the same outcome as applying $(p \circ f)$ to $\bot$, provided f is strict $(f\ \bot = \bot)$.

- $\mathtt{g}$ with p always chooses "the same" elements from $(\mathtt{map}\ f\ l)$ for output as does $\mathtt{g}$ with $(p \circ f)$ from l,

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- 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 outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- 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 outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\perp$.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

↯ 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 outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\perp$.

- $\texttt{g}\ p\ (\texttt{map}\ f\ l)$ is equivalent to $\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$,

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- 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 outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\perp$.

- $\texttt{g}\ p\ (\texttt{map}\ f\ l)$ is equivalent to $\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, if f is strict.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- 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.

$\lightning$ Not true! Also possible: checking outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\perp$.

- $\texttt{g}\ p\ (\texttt{map}\ f\ l)$ is equivalent to $\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, if f is strict.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

$\lightning$ Not true! Also possible: checking outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\perp$.

- $\texttt{g}\ p\ (\texttt{map}\ f\ l)$ is equivalent to $\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, if f is strict.
- This gives a revised free theorem.

The Polymorphic λ-Calculus [Girard 1972, Reynolds 1974]

Types: $\tau := \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid \mathsf{Bool} \mid [\tau]$

Terms: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau \mid$
$\mathsf{True} \mid \mathsf{False} \mid [\,]_\tau \mid t : t \mid \textbf{case}\ t\ \textbf{of}\ \{\cdots\}$

$$\Gamma, x : \tau \vdash x : \tau \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2} \qquad \frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau}$$

$$\frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash (t\ \tau') : \tau[\tau'/\alpha]} \qquad \frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash u : [\tau]}{\Gamma \vdash (t : u) : [\tau]}$$

$$\Gamma \vdash \mathsf{True} : \mathsf{Bool} \quad , \quad \Gamma \vdash \mathsf{False} : \mathsf{Bool} \quad , \quad \Gamma \vdash [\,]_\tau : [\tau]$$

$$\frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma \vdash u : \tau \qquad \Gamma \vdash v : \tau}{\Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{\mathsf{True} \to u\,;\mathsf{False} \to v\}) : \tau}$$

$$\frac{\Gamma \vdash t : [\tau'] \qquad \Gamma \vdash u : \tau \qquad \Gamma, x_1 : \tau', x_2 : [\tau'] \vdash v : \tau}{\Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{[] \to u\,;(x_1 : x_2) \to v\}) : \tau}$$

Adding General Recursion

Terms: $t := \;\; \cdots \mid \mathbf{fix}\ t$

Adding General Recursion

Terms: $t := \quad \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

Adding General Recursion

Terms: $t := \; \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders, and:

$$\textbf{fix}\ t \quad = \quad \bigsqcup_{i \geq 0} (t^i\ \bot)$$

Use in an Example

The function

$$\begin{array}{l}
\texttt{filter} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
\texttt{filter}\ p\ [\,] \qquad\quad = [\,] \\
\texttt{filter}\ p\ (a : as) = \textbf{if}\ p\ a\ \textbf{then}\ a : (\texttt{filter}\ p\ as) \\
\qquad\qquad\qquad\qquad\quad\ \ \textbf{else}\ \ \texttt{filter}\ p\ as
\end{array}$$

has a “desugaring” in the (extended) calculus as follows:

$$\begin{array}{l}
\textbf{fix}\ (\lambda f : (\forall\alpha.(\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]). \\
\quad \Lambda\alpha.\lambda p : (\alpha \to \mathsf{Bool}).\lambda l : [\alpha]. \\
\quad \textbf{case}\ l\ \textbf{of}\ \{[\,] \qquad \to [\,]_\alpha\,; \\
\qquad\qquad\quad (a : as) \to \textbf{case}\ p\ a\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \{\mathsf{True} \to a : (f\ \alpha\ p\ as)\,; \\
\qquad\qquad\qquad\qquad\quad\ \ \mathsf{False} \to f\ \alpha\ p\ as\}\})
\end{array}$$

Adding General Recursion

Terms: $t := \;\; \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders, and:

$$\textbf{fix}\ t \;\; = \;\; \bigsqcup_{i \geq 0} (t^i\ \bot)$$

And what about free theorems?

Adding General Recursion

Terms: $t := \quad \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders, and:

$$\textbf{fix}\ t \quad = \quad \bigsqcup_{i \geq 0} (t^i\ \bot)$$

And what about free theorems?

Let us check the one for, essentially, **fix** $:: (\alpha \to \alpha) \to \alpha$, namely:

$$\begin{array}{l} \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. \forall t_1 :: \tau_1 \to \tau_1, t_2 :: \tau_2 \to \tau_2. \\ \qquad (\forall (a_1, a_2) \in \mathcal{R}.\ (t_1\ a_1, t_2\ a_2) \in \mathcal{R}) \\ \quad \Rightarrow (\textbf{fix}\ t_1, \textbf{fix}\ t_2) \in \mathcal{R} \end{array}$$

Adding General Recursion

Terms: $t := \quad \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders, and:

$$\textbf{fix}\ t \quad = \quad \bigsqcup_{i \geq 0} (t^i\ \bot)$$

And what about free theorems?

Let us check the one for, essentially, **fix** $:: (\alpha \to \alpha) \to \alpha$, namely:

$$\begin{array}{c} \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. \forall t_1 :: \tau_1 \to \tau_1, t_2 :: \tau_2 \to \tau_2. \\ (\forall (a_1, a_2) \in \mathcal{R}.\ (t_1\ a_1, t_2\ a_2) \in \mathcal{R}) \\ \Rightarrow (\bigsqcup_{i \geq 0} (t_1^i\ \bot), \bigsqcup_{i \geq 0} (t_2^i\ \bot)) \in \mathcal{R} \end{array}$$

Adding General Recursion

To provide semantics, types are interpreted as pointed complete partial orders, and:

$$\textbf{fix}\ t \quad = \quad \bigsqcup_{i \geq 0} (t^i\ \bot)$$

And what about free theorems?

Let us check the one for, essentially, **fix** $:: (\alpha \to \alpha) \to \alpha$, namely:

$$\begin{array}{l}\forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. \forall t_1 :: \tau_1 \to \tau_1, t_2 :: \tau_2 \to \tau_2. \\ \qquad (\forall (a_1, a_2) \in \mathcal{R}.\ (t_1\ a_1, t_2\ a_2) \in \mathcal{R}) \\ \quad \Rightarrow (\bigsqcup_{i \geq 0} (t_1^i\ \bot), \bigsqcup_{i \geq 0} (t_2^i\ \bot)) \in \mathcal{R}\end{array}$$

Adding General Recursion

To provide semantics, types are interpreted as pointed complete partial orders, and:

$$\textbf{fix}\ t \quad = \quad \bigsqcup_{i \geq 0} (t^i\ \bot)$$

And what about free theorems?

Let us check the one for, essentially, **fix** $:: (\alpha \to \alpha) \to \alpha$, namely:

$$\begin{array}{l} \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. \forall t_1 :: \tau_1 \to \tau_1, t_2 :: \tau_2 \to \tau_2. \\ \qquad (\forall (a_1, a_2) \in \mathcal{R}.\ (t_1\ a_1, t_2\ a_2) \in \mathcal{R}) \\ \quad \Rightarrow (\bigsqcup_{i \geq 0} (t_1^i\ \bot), \bigsqcup_{i \geq 0} (t_2^i\ \bot)) \in \mathcal{R} \end{array}$$

We can guarantee the above, provided all relations are restricted to be strict and continuous.

Deriving Free Theorems in Presence of General Recursion

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R},\mathcal{S}) & = & \{(\bot,\bot)\} \cup \{((x_1,x_2),(y_1,y_2)) \mid \cdots\} \\
[\mathcal{R}] & = & \{(\bot,\bot)\} \cup \{([x_1,\ldots,x_n],[y_1,\ldots,y_n]) \mid \cdots\} \\
\mathcal{R} \to \mathcal{S} & = & \{(f_1,f_2) \mid \forall (a_1,a_2) \in \mathcal{R}.\ (f_1\ a_1, f_2\ a_2) \in \mathcal{S}\} \\
\forall \mathcal{R}.\mathcal{F}(\mathcal{R}) & = & \{(u,v) \mid \forall \tau_1,\tau_2,\mathcal{R}\ \text{strict and continuous}.\ \cdots\}
\end{array}
$$

Deriving Free Theorems in Presence of General Recursion

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{(\bot, \bot)\} \cup \{((x_1, x_2), (y_1, y_2)) \mid \cdots\} \\
[\mathcal{R}] & = & \{(\bot, \bot)\} \cup \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid \cdots\} \\
\mathcal{R} \to \mathcal{S} & = & \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}.\ (f_1\ a_1, f_2\ a_2) \in \mathcal{S}\} \\
\forall \mathcal{R}.\mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \text{ strict and continuous}.\ \cdots\}
\end{array}
$$

Then for every $\mathrm{g} :: \tau$, the pair (g, g) is contained in the (adapted) relational interpretation of τ.

Automatic Generation of Free Theorems

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

This tool allows to generate free theorems for sublanguages of Haskell as described here.

The source code of the underlying library and a shell-based application using it is available here and here.

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]" or simply "filter":

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

Please choose a sublanguage of Haskell:

- no bottoms (hence no general recursion and no selective strictness)
- general recursion but no selective strictness
- general recursion and selective strictness

Please choose a theorem style (without effect in the sublanguage with no bottoms):

- equational
- inequational

Generate

Adding Selective Strictness

Terms: $t := \ \cdots \ \mid \ \mathbf{seq}\ t\ t$

Adding Selective Strictness

Terms: $t := \quad \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Adding Selective Strictness

Terms: $t := \;\; \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Semantics: **seq** t_1 t_2 tries to evaluate t_1; if/after that succeeds, projects to t_2.

Adding Selective Strictness

Terms: $t := \quad \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Semantics: $\textbf{seq}\ t_1\ t_2$ tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Adding Selective Strictness

Terms: $t := \quad \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Semantics: $\textbf{seq}\ t_1\ t_2$ tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

There are counterexamples, again.

Without **seq**, $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$

- $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\bot$.
- 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 and on $\bot$.
- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- Applying p to $\bot$ has the same outcome as applying $(p \circ f)$, provided f is strict.
- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f, and that they may also choose, at the same positions, to output $\bot$.
- $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, if f is strict.

With **seq**, $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$?

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.

With **seq**, $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$?

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\bot$.

With **seq**, $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$?

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

With **seq**, $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$?

- $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.
- 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 and on $\perp$.

With **seq**, $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$?

- $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\bot$.
- 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 and on $\bot$.

$\lightning$ Not true! Also possible:

  - "checking" elements from l for being $\bot$
  - "checking" p for being $\bot$

With **seq**, $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$?

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.
- 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 and on $\perp$.

$\lightning$ Not true! Also possible:

- “checking” elements from l for being $\perp$
- “checking” p for being $\perp$

. . . ???

Adding Selective Strictness

Terms: $t := \;\; \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Semantics: **seq** t_1 t_2 tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Adding Selective Strictness

Terms: $t := \; \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Semantics: $\textbf{seq}\ t_1\ t_2$ tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Let us try the same strategy as before, looking at the free theorem for, essentially, $\textbf{seq} :: \alpha \to \beta \to \beta$, namely:

$$\forall \mathcal{R}, \mathcal{S} \text{ strict and continuous.}$$
$$\forall (t_1, t_1') \in \mathcal{R}, (t_2, t_2') \in \mathcal{S}.\ (\textbf{seq}\ t_1\ t_2, \textbf{seq}\ t_1'\ t_2') \in \mathcal{S}$$

Adding Selective Strictness

Semantics: $\mathbf{seq}\ t_1\ t_2$ tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Let us try the same strategy as before, looking at the free theorem for, essentially, $\mathbf{seq} :: \alpha \to \beta \to \beta$, namely:

$$\forall \mathcal{R}, \mathcal{S} \text{ strict and continuous.}$$
$$\forall (t_1, t_1') \in \mathcal{R}, (t_2, t_2') \in \mathcal{S}.\ (\mathbf{seq}\ t_1\ t_2, \mathbf{seq}\ t_1'\ t_2') \in \mathcal{S}$$

Adding Selective Strictness

Semantics: **seq** t_1 t_2 tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Let us try the same strategy as before, looking at the free theorem for, essentially, **seq** :: $\alpha \to \beta \to \beta$, namely:

$$\forall \mathcal{R}, \mathcal{S} \text{ strict and continuous.}$$
$$\forall (t_1, t_1') \in \mathcal{R}, (t_2, t_2') \in \mathcal{S}.\ (\mathbf{seq}\ t_1\ t_2, \mathbf{seq}\ t_1'\ t_2') \in \mathcal{S}$$

Case distinction:

t_1	t_1'	**seq** t_1 t_2	**seq** t_1' t_2'	$\in \mathcal{S}$
$\perp$	$\perp$	$\perp$	$\perp$	
$\perp$	$\not\perp$	$\perp$	t_2'	
$\not\perp$	$\perp$	t_2	$\perp$	
$\not\perp$	$\not\perp$	t_2	t_2'	

Adding Selective Strictness

Semantics: **seq** t_1 t_2 tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Let us try the same strategy as before, looking at the free theorem for, essentially, **seq** :: $\alpha \to \beta \to \beta$, namely:

$$\forall \mathcal{R}, \mathcal{S} \text{ strict and continuous.}$$
$$\forall (t_1, t_1') \in \mathcal{R}, (t_2, t_2') \in \mathcal{S}.\ (\mathbf{seq}\ t_1\ t_2, \mathbf{seq}\ t_1'\ t_2') \in \mathcal{S}$$

Case distinction:

t_1	t_1'	**seq** t_1 t_2	**seq** t_1' t_2'	$\in \mathcal{S}$
$\perp$	$\perp$	$\perp$	$\perp$	✓
$\perp$	$\not\perp$	$\perp$	t_2'	
$\not\perp$	$\perp$	t_2	$\perp$	
$\not\perp$	$\not\perp$	t_2	t_2'	

Adding Selective Strictness

Semantics: **seq** t_1 t_2 tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Let us try the same strategy as before, looking at the free theorem for, essentially, **seq** $:: \alpha \to \beta \to \beta$, namely:

$$\forall \mathcal{R}, \mathcal{S} \text{ strict and continuous.}$$
$$\forall (t_1, t_1') \in \mathcal{R}, (t_2, t_2') \in \mathcal{S}.\ (\mathbf{seq}\ t_1\ t_2, \mathbf{seq}\ t_1'\ t_2') \in \mathcal{S}$$

Case distinction:

t_1	t_1'	**seq** t_1 t_2	**seq** t_1' t_2'	$\in \mathcal{S}$
$\bot$	$\bot$	$\bot$	$\bot$	✓
$\bot$	$\not\bot$	$\bot$	t_2'	
$\not\bot$	$\bot$	t_2	$\bot$	
$\not\bot$	$\not\bot$	t_2	t_2'	✓

Adding Selective Strictness

Semantics: **seq** $t_1\ t_2$ tries to evaluate t_1; if/after that succeeds, projects to t_2.

What about free theorems?

Let us try the same strategy as before, looking at the free theorem for, essentially, **seq** :: $\alpha \to \beta \to \beta$, namely:

$$\forall \mathcal{R}, \mathcal{S} \text{ strict and continuous.}$$
$$\forall (t_1, t_1') \in \mathcal{R}, (t_2, t_2') \in \mathcal{S}.\ (\mathbf{seq}\ t_1\ t_2, \mathbf{seq}\ t_1'\ t_2') \in \mathcal{S}$$

Case distinction:

t_1	t_1'	$\mathbf{seq}\ t_1\ t_2$	$\mathbf{seq}\ t_1'\ t_2'$	$\in \mathcal{S}$
$\perp$	$\perp$	$\perp$	$\perp$	✓
$\perp$	$\not\perp$	$\perp$	t_2'	?
$\not\perp$	$\perp$	t_2	$\perp$	?
$\not\perp$	$\not\perp$	t_2	t_2'	✓

(Equational) Free Theorems in the Presence of **seq** [Johann & V., POPL'04]

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{(\bot, \bot)\} \cup \{((x_1, x_2), (y_1, y_2)) \mid \cdots\} \\
[\mathcal{R}] & = & \{(\bot, \bot)\} \cup \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid \cdots\} \\
\mathcal{R} \to \mathcal{S} & = & \{(f_1, f_2) \mid (f_1 = \bot \Leftrightarrow f_2 = \bot) \wedge \cdots\} \\
\forall \mathcal{R}. \mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \text{ strict, continuous,} \\
 & & \qquad \text{and bottom-reflecting}. \ \cdots\}
\end{array}
$$

(Equational) Free Theorems in the Presence of **seq** [Johann & V., POPL'04]

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{(\bot, \bot)\} \cup \{((x_1, x_2), (y_1, y_2)) \mid \cdots\} \\
[\mathcal{R}] & = & \{(\bot, \bot)\} \cup \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid \cdots\} \\
\mathcal{R} \rightarrow \mathcal{S} & = & \{(f_1, f_2) \mid (f_1 = \bot \Leftrightarrow f_2 = \bot) \wedge \cdots\} \\
\forall \mathcal{R}.\mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \text{ strict, continuous,} \\
 & & \qquad \text{and bottom-reflecting}.\ \cdots\}
\end{array}
$$

(Equational) Free Theorems in the Presence of **seq** [Johann & V., POPL'04]

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$
\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{(\bot, \bot)\} \cup \{((x_1, x_2), (y_1, y_2)) \mid \cdots\} \\
[\mathcal{R}] & = & \{(\bot, \bot)\} \cup \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid \cdots\} \\
\mathcal{R} \to \mathcal{S} & = & \{(f_1, f_2) \mid (f_1 = \bot \Leftrightarrow f_2 = \bot) \wedge \cdots\} \\
\forall \mathcal{R}. \mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \text{ strict, continuous,} \\
 & & \qquad \text{and bottom-reflecting}. \ \cdots\}
\end{array}
$$

Then for every $g :: \tau$, the pair (g, g) is contained in the (adapted) relational interpretation of τ.

Revising Free Theorems

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

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

- if f strict.

Revising Free Theorems

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

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

- if f strict.

[Johann & V., POPL'04] : in presence of **seq**, if additionally:

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

Revising Free Theorems

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

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

- if f strict.

[Johann & V., POPL'04] : in presence of **seq**, if additionally:

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

⋮

[Stenger & V., TLCA'09] : take finite failures with imprecise error semantics into account

Revising Free Theorems

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

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

- if f strict.

[Johann & V., POPL'04] : in presence of **seq**, if additionally:

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

$\vdots$

[Stenger & V., TLCA'09] : take finite failures with imprecise error semantics into account

[Christiansen et al., PLPV'10] : functional logic programs in Curry

Necessity of Certain Restrictions?

We have, with **fix**:

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

for every $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$, if

- f strict.

Necessity of Certain Restrictions?

We have, with **fix**:

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

for every $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$, if

- f strict.

We have, with **fix** and **seq**: ..., if

- $p \neq \bot$,
- f strict, and
- f total.

Necessity of Certain Restrictions?

We have, with **fix**:

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

for every $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$, if

- f strict.

We have, with **fix** and **seq**: . . . , if

- $p \neq \perp$,
- f strict, and
- f total.

We have, with . . . , if . . .

Necessity of Certain Restrictions?

We have, with **fix**:

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

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

- f strict.

We have, with **fix** and **seq**: . . . , if

- $p \neq \bot$,
- f strict, and
- f total.

We have, with . . . , if . . .

Natural questions in each case:

1. Are the conditions necessary for every $\texttt{g}$?

Necessity of Certain Restrictions?

We have, with **fix**:

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

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

- f strict.

We have, with **fix** and **seq**: . . . , if

- $p \neq \perp$,
- f strict, and
- f total.

We have, with . . . , if . . .

Natural questions in each case:

1. Are the conditions necessary for every $\texttt{g}$?
2. Are they for any $\texttt{g}$?

Question 1, for (only) **fix**

Are all strictness conditions necessary for every g?

Question 1, for (only) **fix**

Are all strictness conditions necessary for every g? No!

Question 1, for (only) **fix**

Are all strictness conditions necessary for every g? No!

Systematic approach: replace

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

by

$$\frac{\Gamma \vdash \tau \in \text{Pointed} \qquad \Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

Question 1, for (only) **fix**

Are all strictness conditions necessary for every g ? No!

Systematic approach: replace

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau}$$

by

$$\frac{\Gamma \vdash \tau \in \text{Pointed} \qquad \Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau},$$

where

$$\text{Pointed}\ \alpha, \Gamma \vdash \alpha \in \text{Pointed} \qquad \frac{\Gamma \vdash \tau_2 \in \text{Pointed}}{\Gamma \vdash \tau_1 \to \tau_2 \in \text{Pointed}}$$

$$\Gamma \vdash \text{Bool} \in \text{Pointed} \qquad \Gamma \vdash [\tau] \in \text{Pointed}$$

Question 1, for (only) **fix**

Are all strictness conditions necessary for every g ? No!

Systematic approach: replace

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau}$$

by

$$\frac{\Gamma \vdash \tau \in \mathrm{Pointed} \qquad \Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau},$$

where

$$\mathrm{Pointed}\ \alpha, \Gamma \vdash \alpha \in \mathrm{Pointed} \qquad \frac{\Gamma \vdash \tau_2 \in \mathrm{Pointed}}{\Gamma \vdash \tau_1 \to \tau_2 \in \mathrm{Pointed}}$$

$$\Gamma \vdash \mathrm{Bool} \in \mathrm{Pointed} \qquad \Gamma \vdash [\tau] \in \mathrm{Pointed}$$

Gain: Even if relations for un-Pointed types not strict anymore, free theorems continue to hold! [Launchbury & Paterson, ESOP'96]

Question 1, for (only) **fix**

For example, we get:

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

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

  if f strict.

Question 1, for (only) **fix**

For example, we get:

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

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

  if f strict.

- For every $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ (in the new system),

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

  without conditions on f.

Question 2, for (only) **fix**

For a given type, is there a g such that the strictness conditions are really necessary?

Question 2, for (only) **fix**

For a given type, is there a g such that the strictness conditions are really necessary? Not always!

Question 2, for (only) **fix**

For a given type, is there a g such that the strictness conditions are really necessary? Not always!

The ideal scenario, automatic generation of counterexamples:

- I give the system a type, say $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$.

Question 2, for (only) **fix**

For a given type, is there a $\mathtt{g}$ such that the strictness conditions are really necessary? Not always!

The ideal scenario, automatic generation of counterexamples:

- I give the system a type, say $\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$.
- The system gives me the free theorem. Here:

  for strict f, $\quad \mathtt{g}\ p\ (\mathtt{map}\ f\ l) \;=\; \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$

Question 2, for (only) **fix**

For a given type, is there a `g` such that the strictness conditions are really necessary? Not always!

The ideal scenario, automatic generation of counterexamples:

- I give the system a type, say $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$.
- The system gives me the free theorem. Here:

  for strict f, $\quad \texttt{g}\ p\ (\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$

- I ask: why must f be strict? What if it were not?

Question 2, for (only) **fix**

For a given type, is there a $\texttt{g}$ such that the strictness conditions are really necessary? Not always!

The ideal scenario, automatic generation of counterexamples:

- I give the system a type, say $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$.
- The system gives me the free theorem. Here:

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

- I ask: why must f be strict? What if it were not?
- The system gives me concrete $\texttt{g}$, as well as p, l, and (non-strict) f that refute the thus naivified free theorem.

Idea 1: Use the Pointed-Approach

For example, search for a g such that

$$\text{Pointed } \alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

Idea 1: Use the Pointed-Approach

For example, search for a g such that

$$\text{Pointed } \alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\textbf{fix}\ (\lambda x : \tau.x)) : \tau}$$

Idea 1: Use the Pointed-Approach

For example, search for a g such that

$$\text{Pointed } \alpha \vdash g : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash g : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\textbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Otherwise, search further depending on type.

Idea 1: Use the Pointed-Approach

For example, search for a g such that

$$\text{Pointed } \alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\textbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Otherwise, search further depending on type.

Problem: For term search, rules are not “syntax-directed” enough.

Idea 1: Use the Pointed-Approach

For example, search for a g such that

$$\text{Pointed } \alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\textbf{fix } (\lambda x : \tau . x)) : \tau}$$

Otherwise, search further depending on type.

Problem: For term search, rules are not “syntax-directed” enough.

Particularly:

$$\frac{\Gamma \Vdash t : \tau_1 \to \tau_2 \qquad \Gamma \Vdash u : \tau_1}{\Gamma \Vdash (t \ u) : \tau_2}$$

Idea 1: Use the Pointed-Approach

For example, search for a g such that

$$\text{Pointed } \alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\textbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Otherwise, search further depending on type.

Problem: For term search, rules are not "syntax-directed" enough.

Particularly:

$$\frac{\Gamma \Vdash t : \tau_1 \to \tau_2 \qquad \Gamma \Vdash u : \tau_1}{\Gamma \Vdash (t\ u) : \tau_2}$$

Idea 1: Use the Pointed-Approach

For example, search for a g such that

$$\text{Pointed } \alpha \vdash g : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash g : (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\mathbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Otherwise, search further depending on type.

Problem: For term search, rules are not "syntax-directed" enough.

Particularly:

$$\frac{\Gamma \Vdash \qquad \tau_1 \to \tau_2 \qquad \Gamma \Vdash \qquad \tau_1}{\Gamma \Vdash \qquad \tau_2}$$

Idea 2: Use the Curry/Howard-Isomorphism

- [Dyckhoff 1992] gives a proof search procedure for intuitionistic propositional logic.

Idea 2: Use the Curry/Howard-Isomorphism

- [Dyckhoff 1992] gives a proof search procedure for intuitionistic propositional logic.

- It has been turned into a **fix**-free term generator for given polymorphic types [Augustsson, AAIP'09].

Idea 2: Use the Curry/Howard-Isomorphism

- [Dyckhoff 1992] gives a proof search procedure for intuitionistic propositional logic.
- It has been turned into a **fix**-free term generator for given polymorphic types [Augustsson, AAIP'09].
- We mix it with our rule

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\textbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

and perform further adaptations ...

Idea 2: Use the Curry/Howard-Isomorphism

- [Dyckhoff 1992] gives a proof search procedure for intuitionistic propositional logic.
- It has been turned into a **fix**-free term generator for given polymorphic types [Augustsson, AAIP'09].
- We mix it with our rule

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash (\mathbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

  and perform further adaptations ...
  [Seidel & V., FLOPS'10]

The Tool on an Example

The Free Theorem

The theorem generated for functions of the type

```
f :: (a -> Int) -> Int
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
 forall p :: t1 -> Int.
  forall q :: t2 -> Int.
   (forall x :: t1. p x = q (g x)) ==> (f p = f q)
```

The Counterexample

By disregarding the strictness condition on `g` the theorem becomes wrong. The term

```
f = (\x1 -> (x1 _|_))
```

is a counterexample.

By setting `t1 = t2 = ... = ()` and

```
g = const ()
```

the following would be a consequence of the thus "naivified" free theorem:

```
(f p) = (f q)

where

p        = (\x1 -> 0)
q        = (\x1 -> (case x1 of {() -> 0}))
```

But this is wrong since with the above `f` it reduces to:

```
0 = _|_
```

Another Example

The Free Theorem

The theorem generated for functions of the type

```
f :: [a] -> Int
```

is:

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

The Counterexample

Disregarding the strictness condition on `g` the algorithm found no counterexample.

Question 1, for (**fix** and) **seq**

Are all totality and "$\neq \bot$"- conditions necessary for every g ?

Question 1, for (**fix** and) **seq**

Are all totality and "$\neq \perp$"- conditions necessary for every g ? No!

Question 1, for (**fix** and) **seq**

Are all totality and "$\neq \bot$"- conditions necessary for every g? No!

Natural approach: replace

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{seq}\ t_1\ t_2) : \tau_2}$$

by

$$\frac{\Gamma \vdash \tau_1 \in \text{Seqable} \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{seq}\ t_1\ t_2) : \tau_2}$$

Question 1, for (**fix** and) **seq**

Are all totality and "$\neq \bot$"- conditions necessary for every g? No!

Natural approach: replace

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

by

$$\frac{\Gamma \vdash \tau_1 \in \text{Seqable} \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2},$$

where

$$\text{Seqable}\ \alpha, \Gamma \vdash \alpha \in \text{Seqable} \qquad \frac{???}{\Gamma \vdash (\tau_1 \to \tau_2) \in \text{Seqable}}$$

$$\Gamma \vdash \text{Bool} \in \text{Seqable} \qquad \Gamma \vdash [\tau] \in \text{Seqable}$$

Question 1, for (**fix** and) **seq**

Are all totality and "$\neq \bot$"- conditions necessary for every g? No!

Natural approach: replace

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{seq}\ t_1\ t_2) : \tau_2}$$

by

$$\frac{\Gamma \vdash \tau_1 \in \text{Seqable} \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{seq}\ t_1\ t_2) : \tau_2},$$

where

$$\text{Seqable}\ \alpha, \Gamma \vdash \alpha \in \text{Seqable} \qquad \frac{???}{\Gamma \vdash (\tau_1 \to \tau_2) \in \text{Seqable}}$$

$$\Gamma \vdash \text{Bool} \in \text{Seqable} \qquad \Gamma \vdash [\tau] \in \text{Seqable}$$

Problem: Completely new approach needed due to complications with function types.

(Equational) Free Theorems in the Presence of **seq** [Johann & V., POPL'04]

For interpreting types as relations:

1. Replace (implicit quantification over) type variables by (explicit) quantification over relation variables.
2. Replace types without any polymorphism by identity relations.
3. Use the following rules:

$$\begin{array}{lcl}
(\mathcal{R}, \mathcal{S}) & = & \{(\bot, \bot)\} \cup \{((x_1, x_2), (y_1, y_2)) \mid \cdots\} \\
[\mathcal{R}] & = & \{(\bot, \bot)\} \cup \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid \cdots\} \\
\mathcal{R} \to \mathcal{S} & = & \{(f_1, f_2) \mid (f_1 = \bot \Leftrightarrow f_2 = \bot) \wedge \cdots\} \\
\forall \mathcal{R}. \mathcal{F}(\mathcal{R}) & = & \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \text{ strict, continuous,} \\
& & \qquad \text{and bottom-reflecting. } \cdots\}
\end{array}$$

Then for every $g :: \tau$, the pair (g, g) is contained in the (adapted) relational interpretation of τ.

Question 1, for (**fix** and) **seq**

Are all totality and "$\neq \bot$"- conditions necessary for every g? No!

Natural approach: replace

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{seq}\ t_1\ t_2) : \tau_2}$$

by

$$\frac{\Gamma \vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{seq}\ t_1\ t_2) : \tau_2},$$

where

$$\mathsf{Seqable}\ \alpha, \Gamma \vdash \alpha \in \mathsf{Seqable} \qquad \frac{???}{\Gamma \vdash (\tau_1 \to \tau_2) \in \mathsf{Seqable}}$$

$$\Gamma \vdash \mathsf{Bool} \in \mathsf{Seqable} \qquad \Gamma \vdash [\tau] \in \mathsf{Seqable}$$

Problem: Completely new approach needed due to complications with function types.

. . . But it Can be Done [Seidel & V., ATPS'09]

At `http://www-ps.iai.uni-bonn.de/cgi-bin/polyseq.cgi`:

The term

```
t = (/\a.
     (/\b.
      (\c::(a -> (b -> a)).
       (fix (\h::(a -> ([b] -> a)).
             (\n::a.
              (\ys::[b].
               (seq (c n) (case ys of {[] -> n; x:xs ->
                           (seq xs (seq x (let n' = ((c n) x) in
                                          ((h n') xs))))}))))))))))
```

can be typed to the optimal type

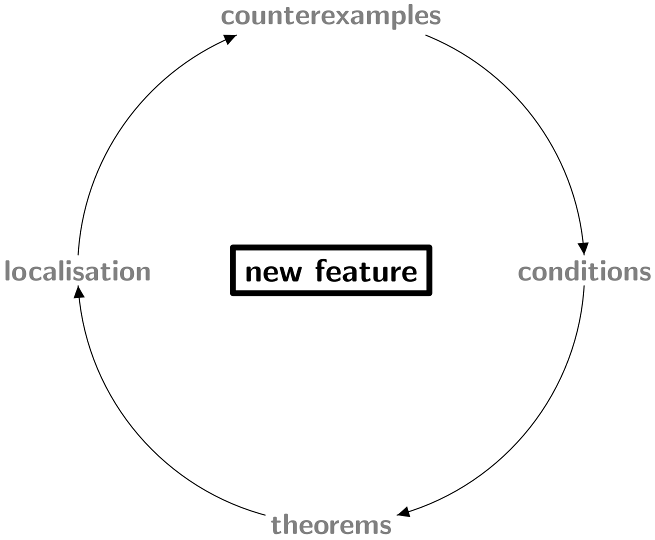
```
(forall^n a. (forall^e b. ((a ->^n (b ->^e a)) ->^e (a ->^e ([b] ->^e a)))))
```

with the free theorem

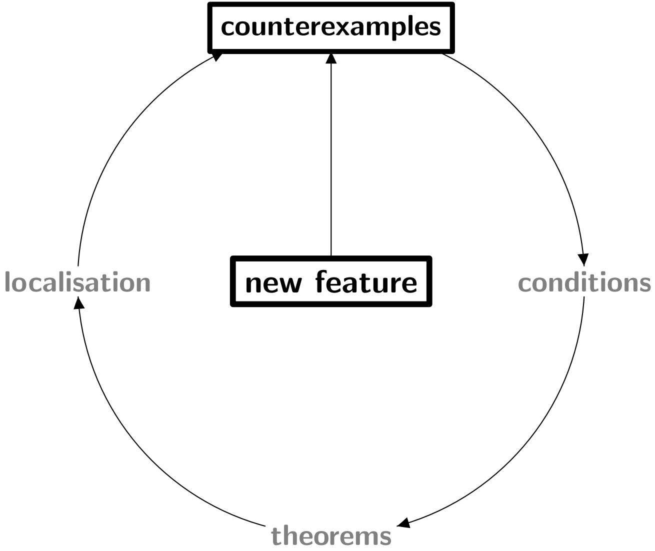
```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  ((t_{t1}_{t3} /= _|_) <=> (t_{t2}_{t4} /= _|_))
  && (forall p :: t1 -> (t3 -> t1).
       forall q :: t2 -> (t4 -> t2).
        (forall x :: t1.
          ((p x /= _|_) <=> (q (f x) /= _|_))
          && (forall y :: t3. f (p x y) = q (f x) (g y)))
        ==> (((t_{t1}_{t3} p /= _|_) <=> (t_{t2}_{t4} q /= _|_))
             && (forall z :: t1.
                  ((t_{t1}_{t3} p z /= _|_) <=> (t_{t2}_{t4} q (f z) /= _|_))
                  && (forall v :: [t3].
                       f (t_{t1}_{t3} p z v) = t_{t2}_{t4} q (f z) (map_{t3}_{t4} g v)))))
```

The normal free theorem for the type without marks would be:

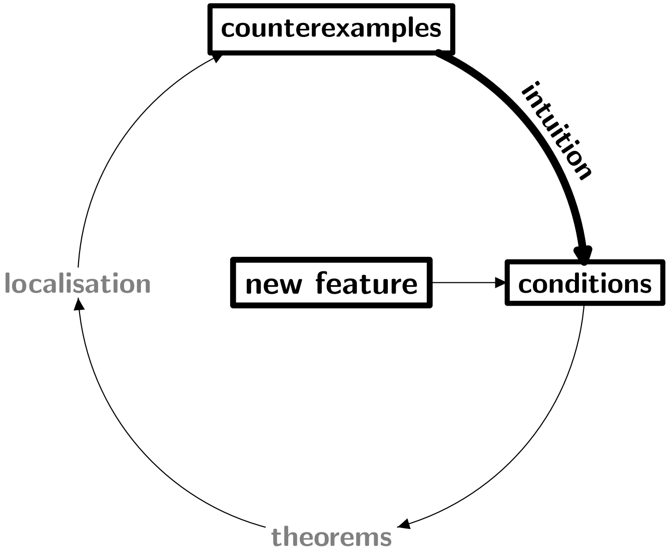
Investigating the Impact of a New Feature



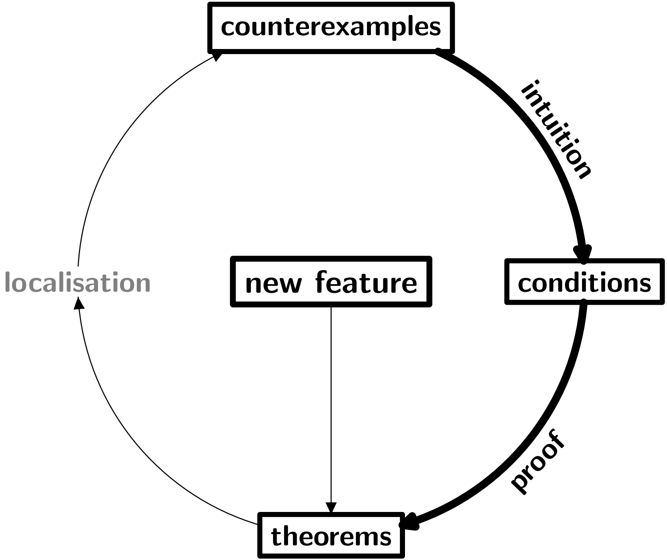
Investigating the Impact of a New Feature



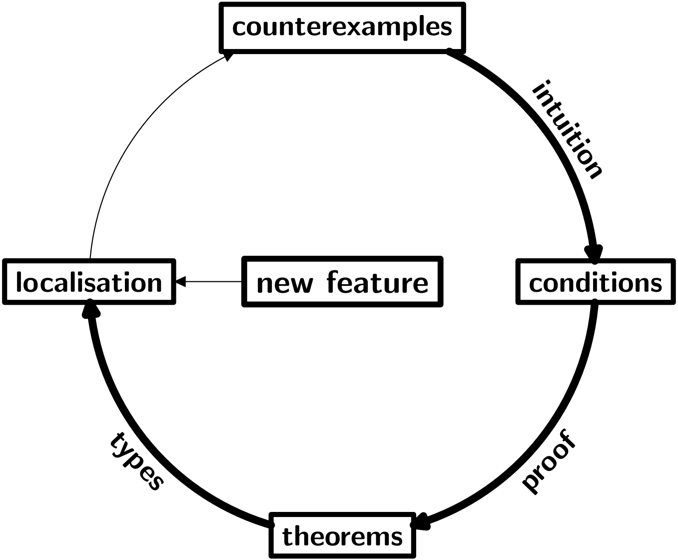
Investigating the Impact of a New Feature



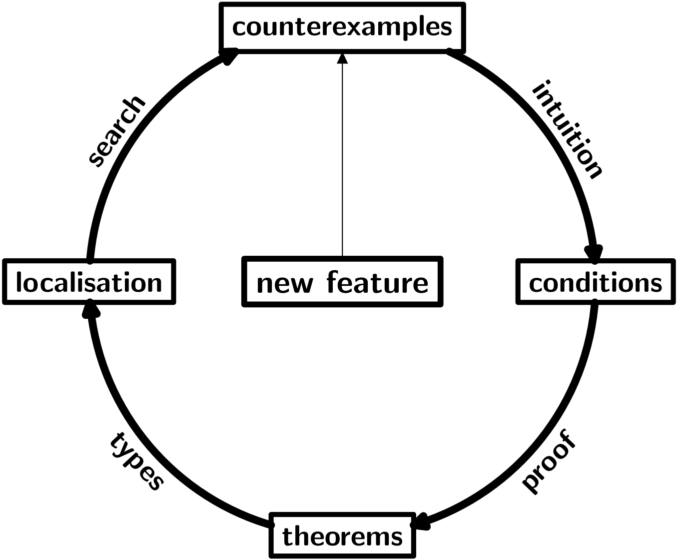
Investigating the Impact of a New Feature



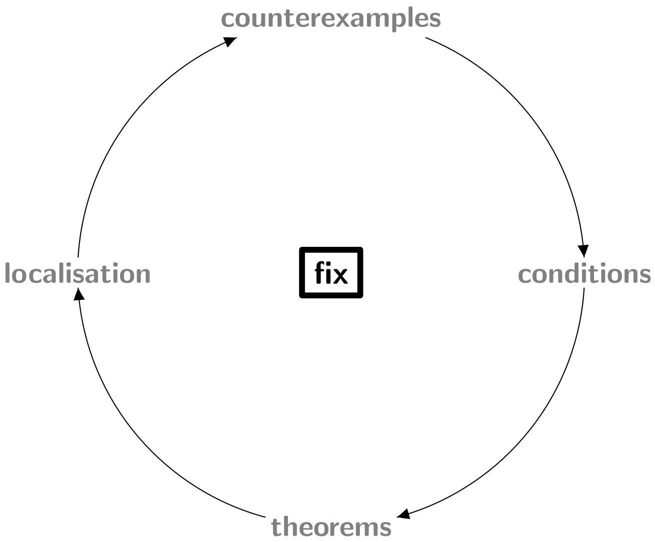
Investigating the Impact of a New Feature



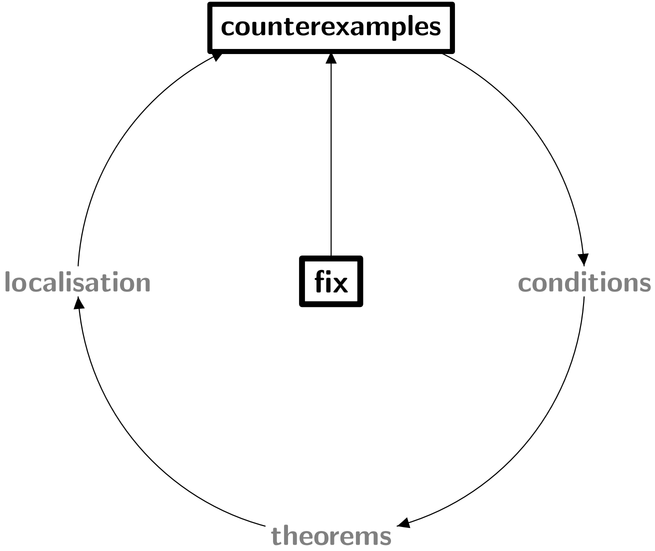
Investigating the Impact of a New Feature



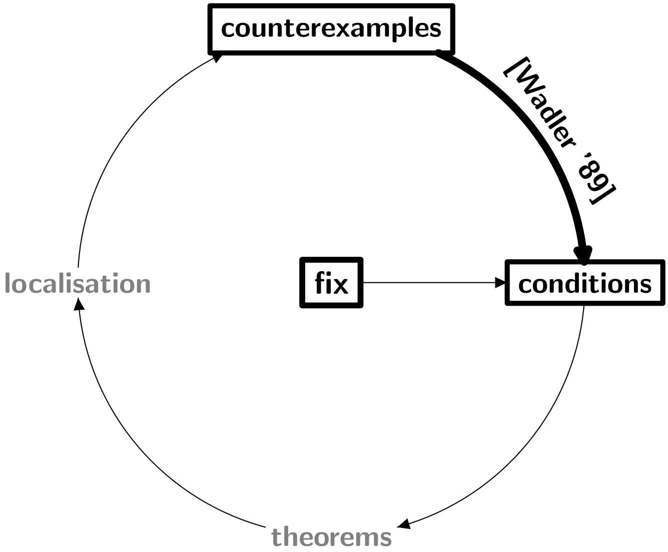
Progress for General Recursion



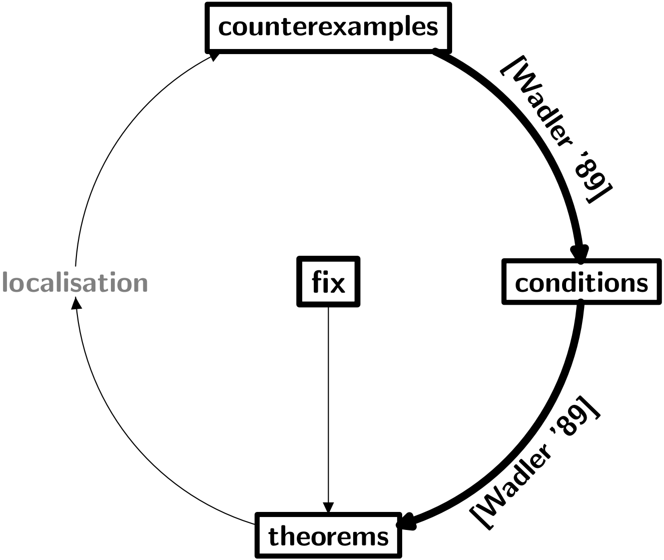
Progress for General Recursion



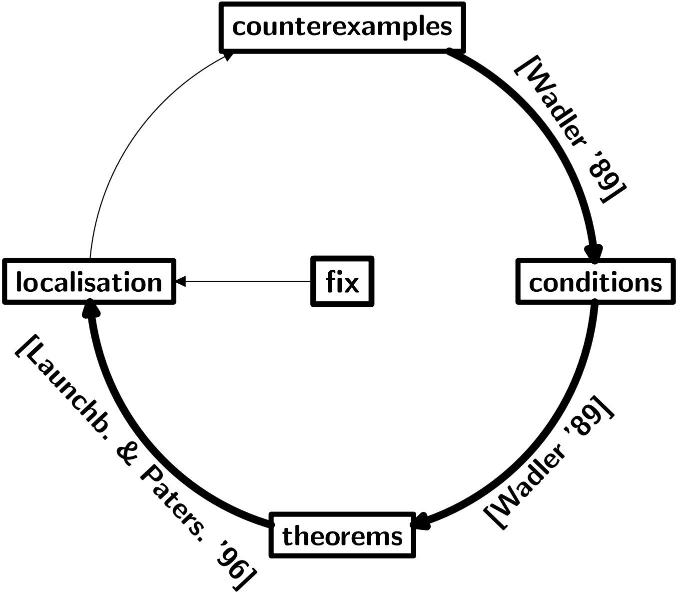
Progress for General Recursion



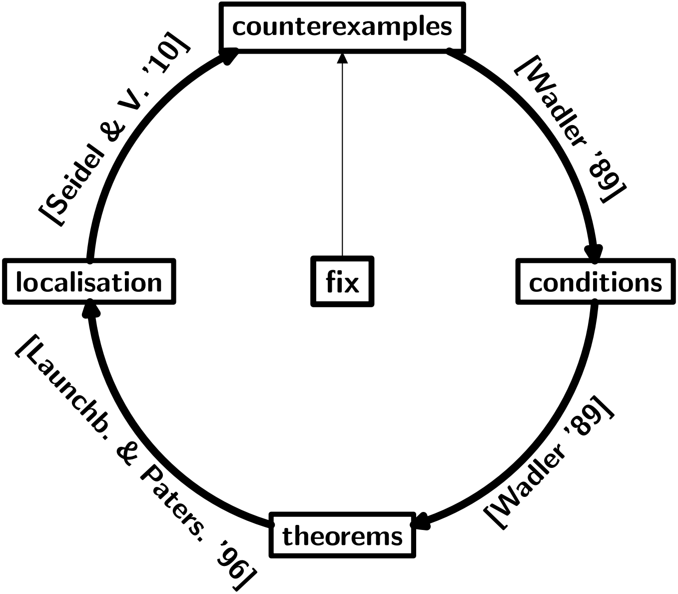
Progress for General Recursion



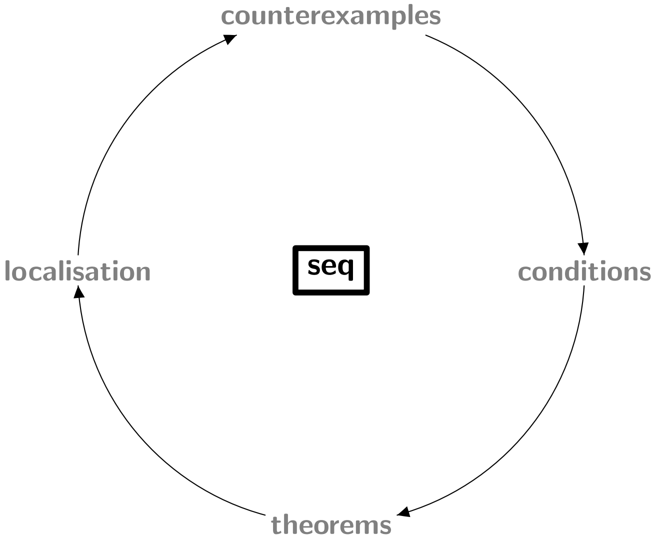
Progress for General Recursion



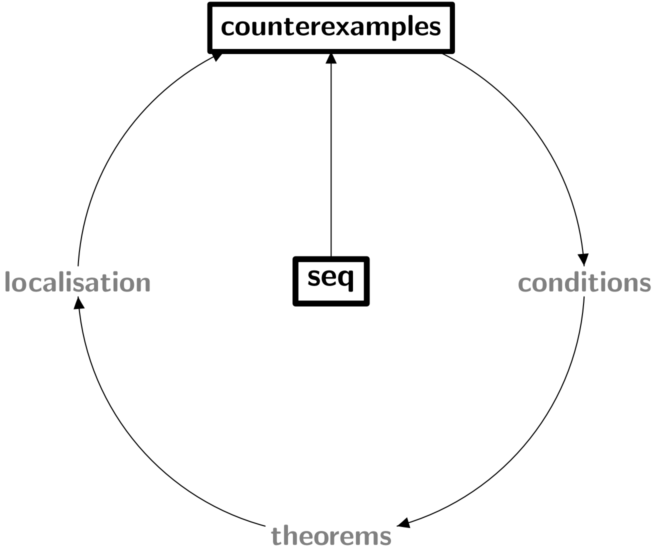
Progress for General Recursion



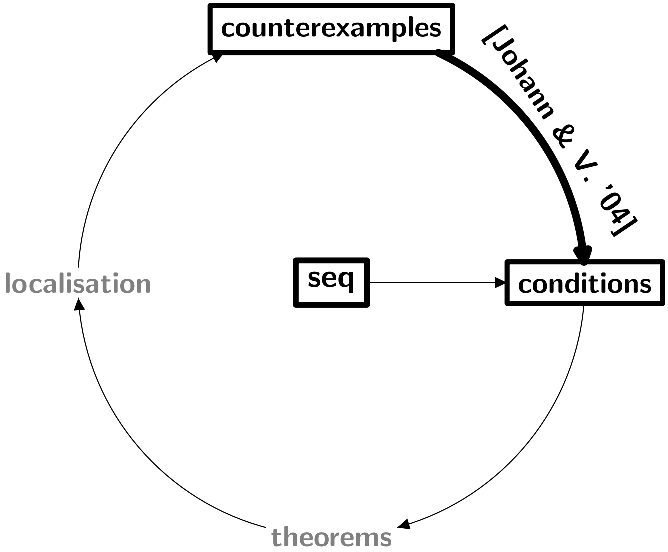
Progress for Selective Strictness



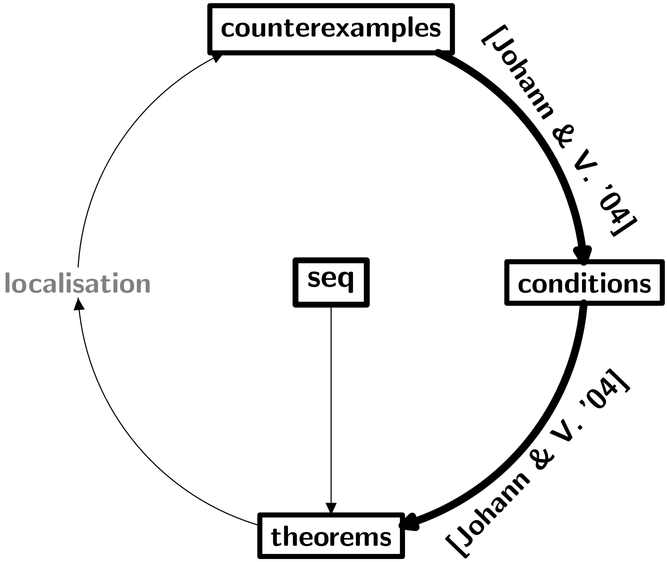
Progress for Selective Strictness



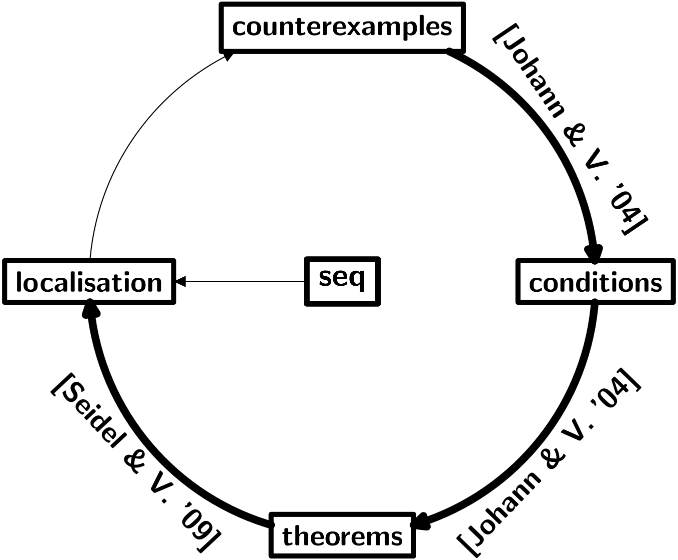
Progress for Selective Strictness



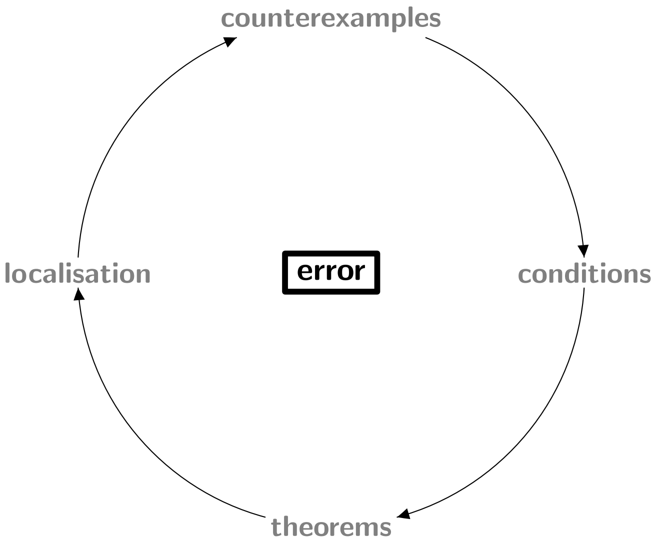
Progress for Selective Strictness



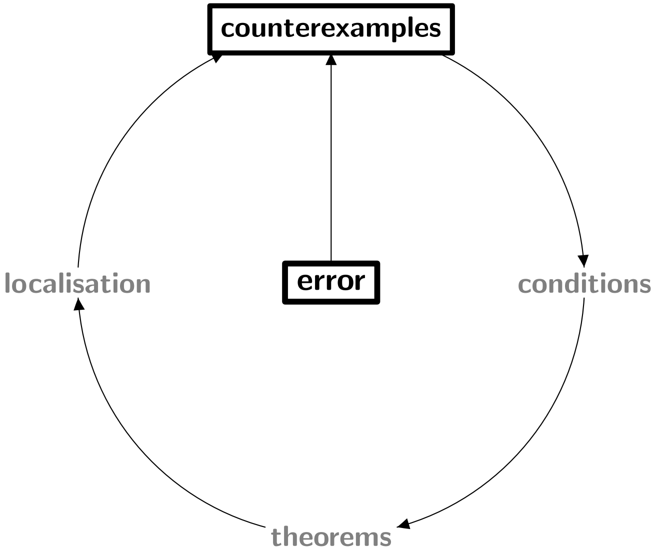
Progress for Selective Strictness



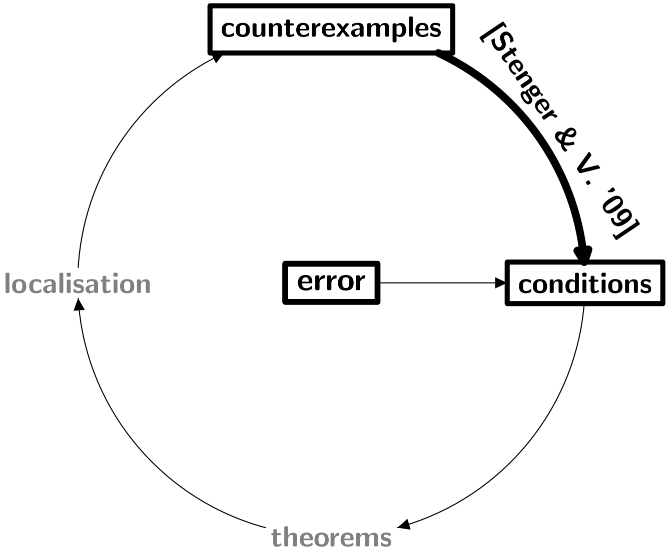
Progress for Imprecise Errors



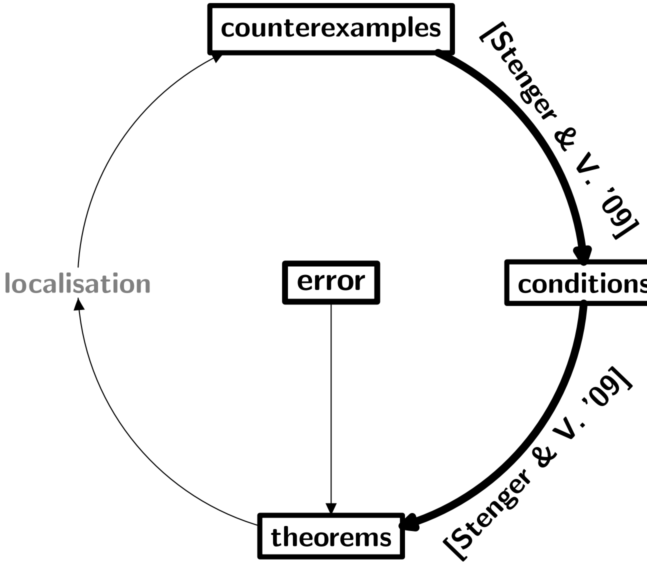
Progress for Imprecise Errors



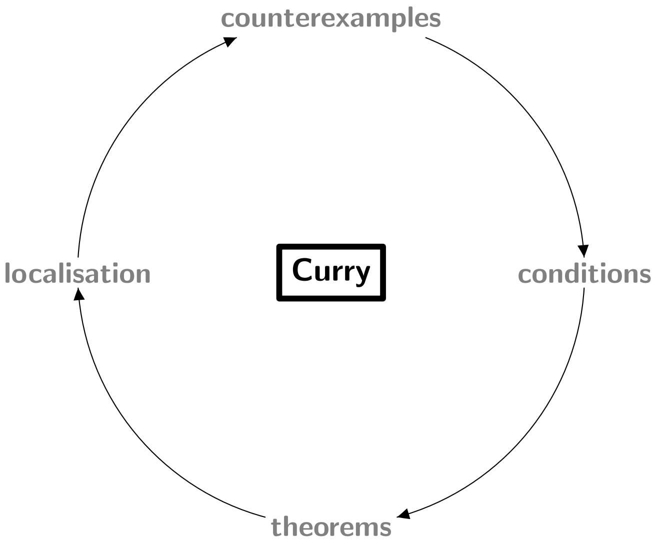
Progress for Imprecise Errors



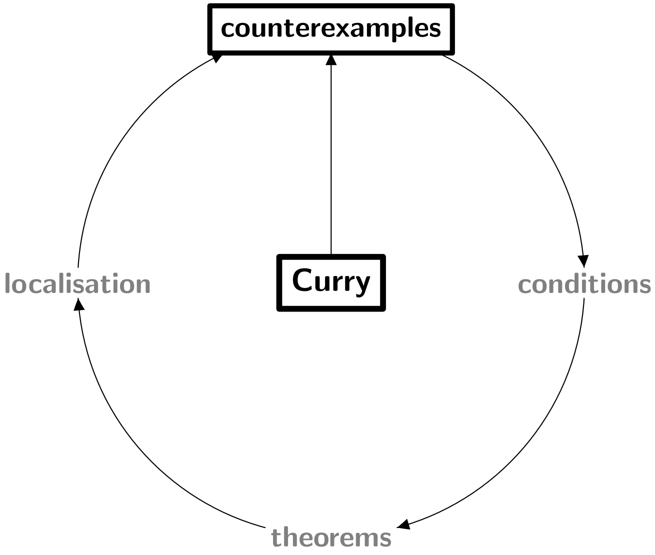
Progress for Imprecise Errors



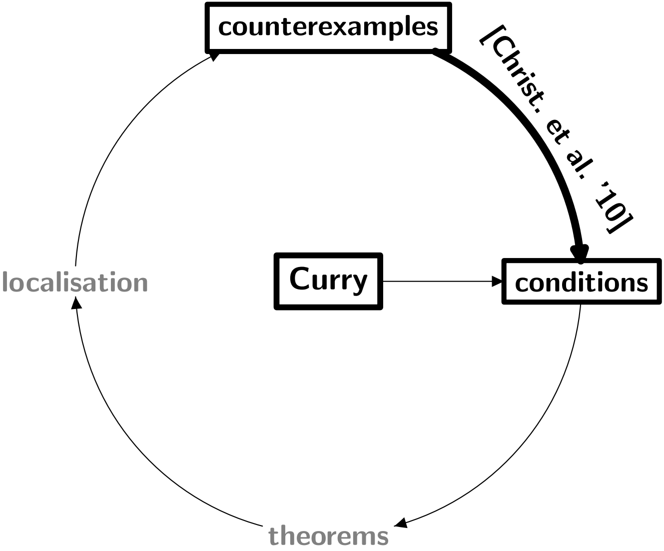
Progress for Functional Logic Programs



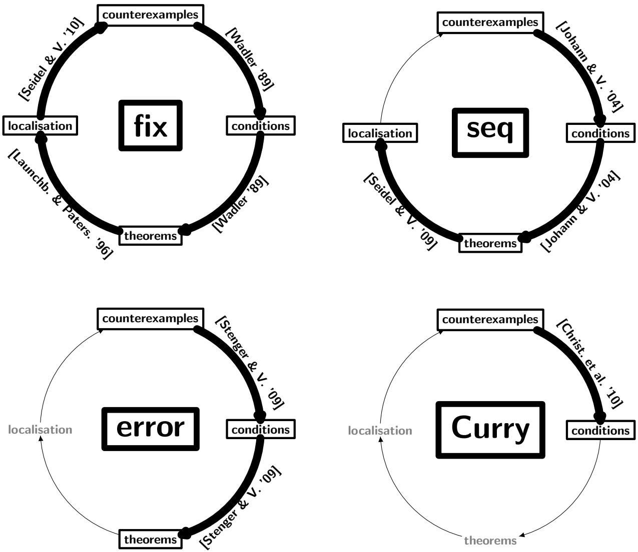
Progress for Functional Logic Programs



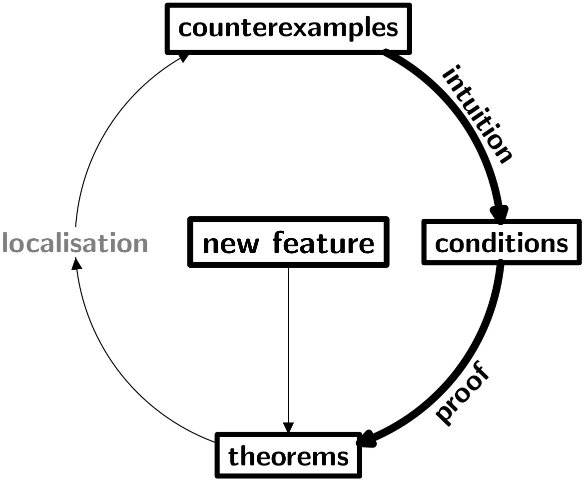
Progress for Functional Logic Programs



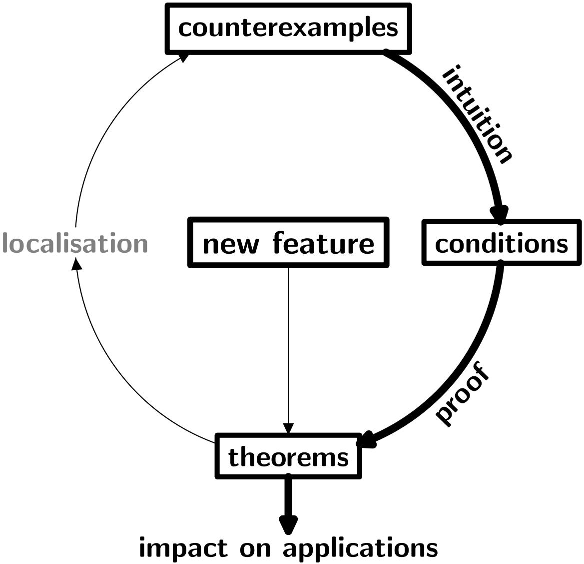
An Overview (and Challenges)



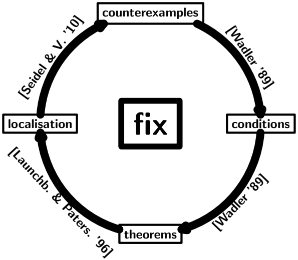
Impact on Applications

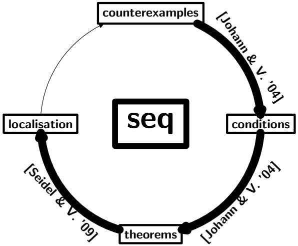


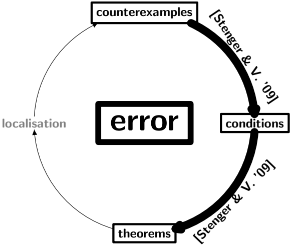
Impact on Applications



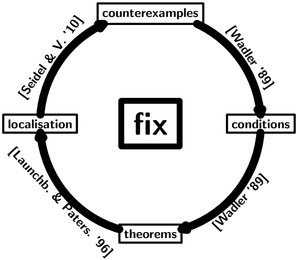
Specific Extensions and Specific Applications

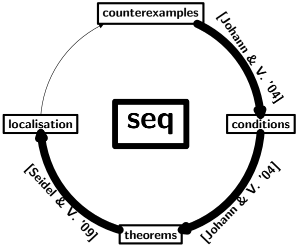




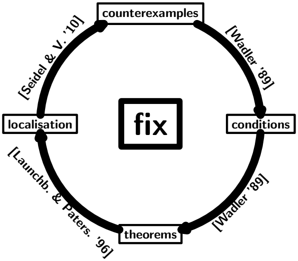


Specific Extensions and Specific Applications

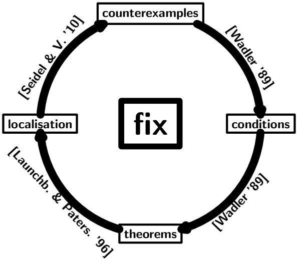




Specific Extensions and Specific Applications



Specific Extensions and Specific Applications



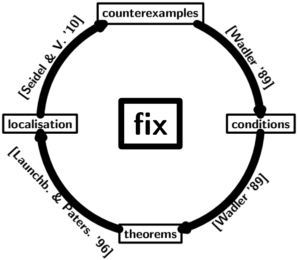
✓ Short Cut Fusion [Gill et al., FPCA'93]

✓ The Dual of Short Cut Fusion [Svenningsson, ICFP'02]

✓ Circular Short Cut Fusion [Fernandes et al., Haskell'07]

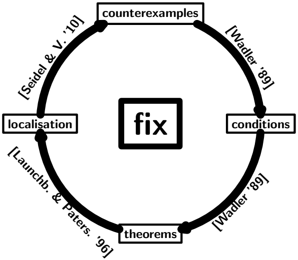
...

Specific Extensions and Specific Applications



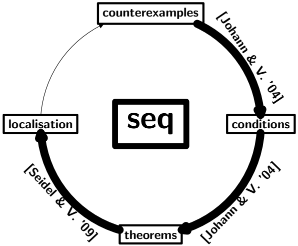
- ✓ Short Cut Fusion [Gill et al., FPCA'93]
- ✓ The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- ✓ Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- ? Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]
- ? Bidirectionalisation [V., POPL'09]

Specific Extensions and Specific Applications

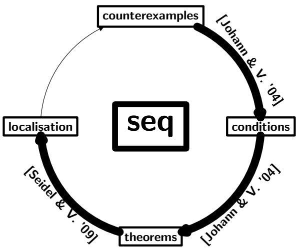


- ✓ Short Cut Fusion [Gill et al., FPCA'93]
- ✓ The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- ✓ Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- ? Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]
- ? Bidirectionalisation [V., POPL'09]
- ?/✓ Reasoning about invariants for monadic programs [V., ICFP'09]

Specific Extensions and Specific Applications



Specific Extensions and Specific Applications



- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]

Conclusion

Types:

- constrain the behaviour of programs

Conclusion

Types:

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

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- (enable lightweight, semantic analysis methods)
- (combine well with algebraic techniques, equational reasoning)

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- (enable lightweight, semantic analysis methods)
- (combine well with algebraic techniques, equational reasoning)

On the programming language side:

- push towards full programming languages

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- (enable lightweight, semantic analysis methods)
- (combine well with algebraic techniques, equational reasoning)

On the programming language side:

- push towards full programming languages
- aim for exploiting more expressive type systems

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- (enable lightweight, semantic analysis methods)
- (combine well with algebraic techniques, equational reasoning)

On the programming language side:

- push towards full programming languages
- aim for exploiting more expressive type systems

On the practical side:

- efficiency-improving program transformations

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- (enable lightweight, semantic analysis methods)
- (combine well with algebraic techniques, equational reasoning)

On the programming language side:

- push towards full programming languages
- aim for exploiting more expressive type systems

On the practical side:

- efficiency-improving program transformations
- applications in specific domains (more out there?)

References I

L. Augustsson.
Putting Curry-Howard to work (Invited talk).
At *Approaches and Applications of Inductive Programming*, 2009.

J. Christiansen, D. Seidel, and J. Voigtländer.
Free theorems for functional logic programs.
In *Programming Languages meets Program Verification, Proceedings*, pages 39–48. ACM Press, 2010.

N.A. Day, J. Launchbury, and J. Lewis.
Logical abstractions in Haskell.
In *Haskell Workshop, Proceedings*. Technical Report UU-CS-1999-28, Utrecht University, 1999.

References II

R. Dyckhoff.
Contraction-free sequent calculi for intuitionistic logic.
Journal of Symbolic Logic, 57(3):795–807, 1992.

J.P. Fernandes, A. Pardo, and J. Saraiva.
A shortcut fusion rule for circular program calculation.
In *Haskell Workshop, Proceedings*, pages 95–106. ACM Press, 2007.

A. Gill, J. Launchbury, and S.L. Peyton Jones.
A short cut to deforestation.
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.

References III

J.-Y. Girard.
Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.
PhD thesis, Université Paris VII, 1972.

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

J. Launchbury and R. Paterson.
Parametricity and unboxing with unpointed types.
In *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996.

References IV

J.C. Reynolds.
Towards a theory of type structure.
In *Colloque sur la Programmation, Proceedings*, volume 19 of *LNCS*, pages 408–423. Springer-Verlag, 1974.

J.C. Reynolds.
Types, abstraction and parametric polymorphism.
In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

D. Seidel and J. Voigtländer.
Taming selective strictness.
In *Arbeitstagung Programmiersprachen, Proceedings*, volume 154 of *Lecture Notes in Informatics*, pages 2916–2930. GI, 2009.

References V

D. Seidel and J. Voigtländer.
Automatically generating counterexamples to naive free theorems.
In *Functional and Logic Programming, Proceedings*, volume 6009 of *LNCS*, pages 175–190. Springer-Verlag, 2010.

F. Stenger and J. Voigtländer.
Parametricity for Haskell with imprecise error semantics.
In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009.

J. Svenningsson.
Shortcut fusion for accumulating parameters & zip-like functions.
In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM Press, 2002.

References VI

J. Voigtländer.
Much ado about two: A pearl on parallel prefix computation.
In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.

J. Voigtländer.
Bidirectionalization for free!
In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

J. Voigtländer.
Free theorems involving type constructor classes.
In *International Conference on Functional Programming, Proceedings*, pages 173–184. ACM Press, 2009.

References VII

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