Taming Selective Strictness

Daniel Seidel[1] and Janis Voigtländer

Institute for Computer Science
Department III
University of Bonn, Germany

{ds,jv}@informatik.uni-bonn.de

April 7, 2010

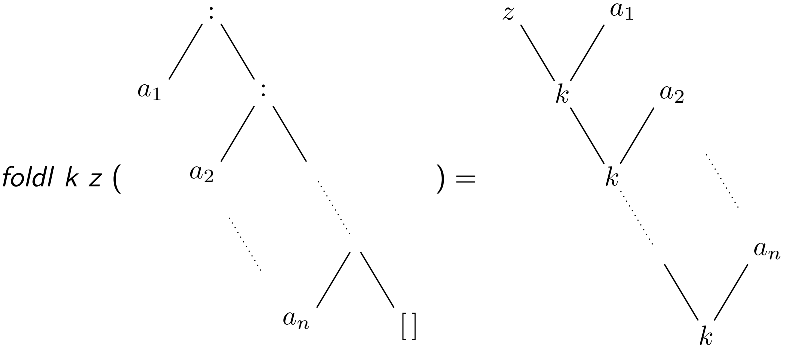
[1]This author was supported by the DFG under grant VO 1512/1-1.

The Polymorphic Function *foldl*

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$
$$\mathit{foldl}\ k\ z\ [\,] = z$$
$$\mathit{foldl}\ k\ z\ (x : \mathit{xs}) = \mathit{foldl}\ k\ (k\ z\ x)\ \mathit{xs}$$

The Polymorphic Function *foldl*

$$\textit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$
$$\textit{foldl}\ k\ z\ [\,] = z$$
$$\textit{foldl}\ k\ z\ (x : xs) = \textit{foldl}\ k\ (k\ z\ x)\ xs$$



The Polymorphic Function *foldl*

$$\begin{aligned}
&\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}\ k\ z\ [\,] = z \\
&\mathit{foldl}\ k\ z\ (x : xs) = \mathit{foldl}\ k\ (k\ z\ x)\ xs
\end{aligned}$$

Example

$$\mathit{sum} = \mathit{foldl}\ (+)\ 0$$

The Polymorphic Function *foldl*

$$
\begin{aligned}
&\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}\ k\ z\ [\,] = z \\
&\mathit{foldl}\ k\ z\ (x : xs) = \mathit{foldl}\ k\ (k\ z\ x)\ xs
\end{aligned}
$$

Example

$$\mathit{sum} = \mathit{foldl}\ (+)\ 0$$

$$\mathit{sum}\ [1, 2, 3] = \mathit{foldl}\ (+)\ 0\ [1, 2, 3]$$

The Polymorphic Function *foldl*

$$
\begin{aligned}
&\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}\ k\ z\ [\,] = z \\
&\mathit{foldl}\ k\ z\ (x : xs) = \mathit{foldl}\ k\ (k\ z\ x)\ xs
\end{aligned}
$$

Example

$$\mathit{sum} = \mathit{foldl}\ (+)\ 0$$

$$
\begin{aligned}
\mathit{sum}\ [1,2,3] &= \mathit{foldl}\ (+)\ 0\ [1,2,3] \\
&= \mathit{foldl}\ (+)\ (0+1)\ [2,3]
\end{aligned}
$$

The Polymorphic Function *foldl*

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$
$$\mathit{foldl}\ k\ z\ [\,] = z$$
$$\mathit{foldl}\ k\ z\ (x : xs) = \mathit{foldl}\ k\ (k\ z\ x)\ xs$$

Example

$$\mathit{sum} = \mathit{foldl}\ (+)\ 0$$

$$\begin{aligned} \mathit{sum}\ [1,2,3] &= \mathit{foldl}\ (+)\ 0\ [1,2,3] \\ &= \mathit{foldl}\ (+)\ (0+1)\ [2,3] \\ &= \mathit{foldl}\ (+)\ ((0+1)+2)\ [3] \end{aligned}$$

The Polymorphic Function *foldl*

$$
\begin{aligned}
&\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}\ k\ z\ [\,] = z \\
&\mathit{foldl}\ k\ z\ (x : xs) = \mathit{foldl}\ k\ (k\ z\ x)\ xs
\end{aligned}
$$

Example

$$
\mathit{sum} = \mathit{foldl}\ (+)\ 0
$$

$$
\begin{aligned}
\mathit{sum}\ [1,2,3] &= \mathit{foldl}\ (+)\ 0\ [1,2,3] \\
&= \mathit{foldl}\ (+)\ (0+1)\ [2,3] \\
&= \mathit{foldl}\ (+)\ ((0+1)+2)\ [3] \\
&= \mathit{foldl}\ (+)\ (((0+1)+2)+3)\ [\,]
\end{aligned}
$$

The Polymorphic Function *foldl*

$$
\begin{aligned}
&\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}\ k\ z\ [\,] = z \\
&\mathit{foldl}\ k\ z\ (x : xs) = \mathit{foldl}\ k\ (k\ z\ x)\ xs
\end{aligned}
$$

Example

$$\mathit{sum} = \mathit{foldl}\ (+)\ 0$$

$$
\begin{aligned}
\mathit{sum}\ [1,2,3] &= \mathit{foldl}\ (+)\ 0\ [1,2,3] \\
&= \mathit{foldl}\ (+)\ (0+1)\ [2,3] \\
&= \mathit{foldl}\ (+)\ ((0+1)+2)\ [3] \\
&= \mathit{foldl}\ (+)\ (((0+1)+2)+3)\ [\,] \\
&= (((0+1)+2)+3)
\end{aligned}
$$

The Polymorphic Function *foldl*

$$
\begin{aligned}
&\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}\ k\ z\ [\,] = z \\
&\mathit{foldl}\ k\ z\ (x : xs) = \mathit{foldl}\ k\ (k\ z\ x)\ xs
\end{aligned}
$$

Example

$$\mathit{sum} = \mathit{foldl}\ (+)\ 0$$

$$
\begin{aligned}
\mathit{sum}\ [1,2,3] &= \mathit{foldl}\ (+)\ 0\ [1,2,3] \\
&= \mathit{foldl}\ (+)\ (0+1)\ [2,3] \\
&= \mathit{foldl}\ (+)\ ((0+1)+2)\ [3] \\
&= \mathit{foldl}\ (+)\ (((0+1)+2)+3)\ [\,] \\
&= (((0+1)+2)+3) \\
&= 6
\end{aligned}
$$

The Fusion Property

Consider a simple program transformation:

$$(+1) \circ \mathit{sum}$$

The Fusion Property

Consider a simple program transformation:

$$(+1) \circ \mathit{sum} = (+1) \circ (\mathit{foldl}\ (+)\ 0)$$

The Fusion Property

Consider a simple program transformation:

$$(+1) \circ \mathit{sum} = (+1) \circ (\mathit{foldl}\ (+)\ 0) = \mathit{foldl}\ (+)\ 1$$

The Fusion Property

Consider a simple program transformation:

$$(+1) \circ \mathit{sum} = (+1) \circ (\mathit{foldl}\ (+)\ 0) = \mathit{foldl}\ (+)\ 1$$

More generally:

$$f \circ (\mathit{foldl}\ k\ z) = \mathit{foldl}\ k'\ z'$$

The Fusion Property

Consider a simple program transformation:

$$(+1) \circ \mathit{sum} = (+1) \circ (\mathit{foldl}\ (+)\ 0) = \mathit{foldl}\ (+)\ 1$$

More generally:

$$f \circ (\mathit{foldl}\ k\ z) = \mathit{foldl}\ k'\ z'$$

For an inductive proof the conditions

$$f\ z = z'$$

$$\forall x, y.\ f\ (k\ x\ y) = k'\ (f\ x)\ y$$

are sufficient.

Free Theorems [Wadler, 1989]

With free theorems we can prove the fusion property automatically only using *foldl*'s type

[2] `http://www-ps.iai.uni-bonn.de/ft/`

Free Theorems [Wadler, 1989]

With free theorems we can prove the fusion property automatically only using *foldl*'s type

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha\,.$$

[2] `http://www-ps.iai.uni-bonn.de/ft/`

Free Theorems [Wadler, 1989]

With free theorems we can prove the fusion property automatically only using *foldl*'s type

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha\,.$$

Output of a generator[2] for *foldl*'s type as input:

```
forall t1,t2 in TYPES,  f :: t1 -> t2 .
 forall t3,t4 in TYPES, g :: t3 -> t4.
  forall  k :: t1 -> t3 -> t1 .
   forall  k' :: t2 -> t4 -> t2 .
    (forall x :: t1. forall y :: t3.
        f (k x y) = k' (f x) (g y))
    ==> (forall z :: t1.
          forall xs :: [t3].
           f (foldl k  z  xs)
             = foldl k'  (f z)  (map g xs))
```

[2] http://www-ps.iai.uni-bonn.de/ft/

Free Theorems [Wadler, 1989]

With free theorems we can prove the fusion property automatically only using *foldl*'s type

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha\,.$$

Output of a generator[2] for *foldl*'s type as input:

```
forall t1,t2 in TYPES, f :: t1 -> t2.
 forall t3,t4 in TYPES, g :: t3 -> t4.
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (forall x :: t1. forall y :: t3.
       f (k x y) = k' (f x) (g y))
    ==> (forall z :: t1.
          forall xs :: [t3].
           f (foldl k z xs)
             = foldl k' (f z) (map g xs))
```

[2] http://www-ps.iai.uni-bonn.de/ft/

Free Theorems [Wadler, 1989]

With free theorems we can prove the fusion property automatically only using *foldl*'s type

$$foldl :: (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha\,.$$

Output of a generator[2] for *foldl*'s type as input:

```
forall t1,t2 in TYPES,  f :: t1 -> t2 .

  forall  k :: t1 -> t3 -> t1 .
   forall  k' :: t2 -> t3 -> t2 .
    (forall x :: t1. forall y :: t3.
        f (k x y) = k' (f x) (  y))
    ==> (forall z :: t1.
          forall xs :: [t3].
           f (foldl k  z  xs)
             = foldl k'  (f z)  (       xs))
```

[2] http://www-ps.iai.uni-bonn.de/ft/

Speed Up with Selective Strictness

Example (*sum* reconsidered)

$$\begin{aligned} \textit{sum}\ [1,2,3] &= \textit{foldl}\ (+)\ 0\ [1,2,3] \\ &= \textit{foldl}\ (+)\ (0+1)\ [2,3] \\ &= \textit{foldl}\ (+)\ ((0+1)+2)\ [3] \\ &\dots \end{aligned}$$

Speed Up with Selective Strictness

Example (*sum* reconsidered)

$$\begin{aligned} \textit{sum}\ [1,2,3] &= \textit{foldl}\ (+)\ 0\ [1,2,3] \\ &= \textit{foldl}\ (+)\ (0+1)\ [2,3] \\ &= \textit{foldl}\ (+)\ ((0+1)+2)\ [3] \\ &\ldots \end{aligned}$$

Lazy evaluation results in a huge overhead.

Speed Up with Selective Strictness

Example (*sum* reconsidered)

$$\begin{aligned} \mathit{sum}\ [1,2,3] &= \mathit{foldl}\ (+)\ 0\ [1,2,3] \\ &= \mathit{foldl}\ (+)\ (0+1)\ [2,3] \\ &= \mathit{foldl}\ (+)\ ((0+1)+2)\ [3] \\ &\dots \end{aligned}$$

Lazy evaluation results in a huge overhead.

$\Rightarrow$ Strict evaluation is desirable.

Speed Up with Selective Strictness

Example (*sum* reconsidered)

$$\begin{aligned} \mathit{sum}\ [1,2,3] &= \mathit{foldl}\ (+)\ 0\ [1,2,3] \\ &= \mathit{foldl}\ (+)\ (0+1)\ [2,3] \\ &= \mathit{foldl}\ (+)\ ((0+1)+2)\ [3] \\ &\dots \end{aligned}$$

Lazy evaluation results in a huge overhead.

$\Rightarrow$ Strict evaluation is desirable.

Haskell provides strict evaluation by the function *seq* $:: \alpha \to \beta \to \beta$:

$$\mathit{seq}\ a\ b = \begin{cases} b & \text{if } a \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

foldl' — A Strict Version of foldl

$$
\begin{aligned}
&\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}'\ k\ z\ [\,] = z \\
&\mathit{foldl}'\ k\ z\ (x : xs) = \textbf{let}\ z' = k\ z\ x\ \textbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ xs)
\end{aligned}
$$

foldl' — A Strict Version of *foldl*

$$\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$
$$\mathit{foldl}'\ k\ z\ [\,] = z$$
$$\mathit{foldl}'\ k\ z\ (x : xs) = \mathbf{let}\ z' = k\ z\ x\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ xs)$$

Example (strict sum')

$$\mathit{sum}'\ [1,2,3] = \mathit{foldl}'\ (+)\ 0\ [1,2,3]$$

foldl' — A Strict Version of *foldl*

$$
\begin{aligned}
&\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}'\ k\ z\ [\,] = z \\
&\mathit{foldl}'\ k\ z\ (x : xs) = \mathbf{let}\ z' = k\ z\ x\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ xs)
\end{aligned}
$$

Example (strict *sum'*)

$$
\begin{aligned}
\mathit{sum}'\ [1, 2, 3] &= \mathit{foldl}'\ (+)\ 0\ [1, 2, 3] \\
&= \mathbf{let}\ z' = 0 + 1\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [2, 3])
\end{aligned}
$$

foldl′ — A Strict Version of *foldl*

$$
\begin{aligned}
&\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}'\ k\ z\ [\,] = z \\
&\mathit{foldl}'\ k\ z\ (x : \mathit{xs}) = \mathbf{let}\ z' = k\ z\ x\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ \mathit{xs})
\end{aligned}
$$

Example (strict *sum′*)

$$
\begin{aligned}
\mathit{sum}'\ [1,2,3] &= \mathit{foldl}'\ (+)\ 0\ [1,2,3] \\
&= \mathbf{let}\ z' = 0 + 1\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [2,3]) \\
&= \mathit{foldl}'\ (+)\ {\color{red}1}\ [2,3]
\end{aligned}
$$

foldl′ — A Strict Version of *foldl*

$$
\begin{aligned}
&\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}'\ k\ z\ [\,] = z \\
&\mathit{foldl}'\ k\ z\ (x : xs) = \textbf{let}\ z' = k\ z\ x\ \textbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ xs)
\end{aligned}
$$

Example (strict *sum′*)

$$
\begin{aligned}
\mathit{sum}'\ [1,2,3] &= \mathit{foldl}'\ (+)\ 0\ [1,2,3] \\
&= \textbf{let}\ z' = 0 + 1\ \textbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [2,3]) \\
&= \mathit{foldl}'\ (+)\ 1\ [2,3] \\
&= \textbf{let}\ z' = 1 + 2\ \textbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [3])
\end{aligned}
$$

foldl′ — A Strict Version of *foldl*

$$
\begin{aligned}
&\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}'\ k\ z\ [\,] = z \\
&\mathit{foldl}'\ k\ z\ (x : xs) = \textbf{let}\ z' = k\ z\ x\ \textbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ xs)
\end{aligned}
$$

Example (strict *sum′*)

$$
\begin{aligned}
\mathit{sum}'\ [1,2,3] &= \mathit{foldl}'\ (+)\ 0\ [1,2,3] \\
&= \textbf{let}\ z' = 0 + 1\ \textbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [2,3]) \\
&= \mathit{foldl}'\ (+)\ 1\ [2,3] \\
&= \textbf{let}\ z' = 1 + 2\ \textbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [3]) \\
&= \mathit{foldl}'\ (+)\ \color{red}{3}\ [3]
\end{aligned}
$$

$foldl'$ — A Strict Version of $foldl$

$$foldl' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$
$$foldl'\ k\ z\ [\,] = z$$
$$foldl'\ k\ z\ (x : xs) = \textbf{let}\ z' = k\ z\ x\ \textbf{in}\ seq\ z'\ (foldl'\ k\ z'\ xs)$$

Example (strict sum')

$$\begin{aligned}
sum'\ [1,2,3] &= foldl'\ (+)\ 0\ [1,2,3] \\
&= \textbf{let}\ z' = 0+1\ \textbf{in}\ seq\ z'\ (foldl'\ (+)\ z'\ [2,3]) \\
&= foldl'\ (+)\ 1\ [2,3] \\
&= \textbf{let}\ z' = 1+2\ \textbf{in}\ seq\ z'\ (foldl'\ (+)\ z'\ [3]) \\
&= foldl'\ (+)\ 3\ [3] \\
&\ldots
\end{aligned}$$

foldl' — A Strict Version of *foldl*

$$\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$
$$\mathit{foldl}'\ k\ z\ [\,] = z$$
$$\mathit{foldl}'\ k\ z\ (x : xs) = \mathbf{let}\ z' = k\ z\ x\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ xs)$$

Example (strict sum')

$$\begin{aligned}
\mathit{sum}'\ [1,2,3] &= \mathit{foldl}'\ (+)\ 0\ [1,2,3] \\
&= \mathbf{let}\ z' = 0 + 1\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [2,3]) \\
&= \mathit{foldl}'\ (+)\ 1\ [2,3] \\
&= \mathbf{let}\ z' = 1 + 2\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [3]) \\
&= \mathit{foldl}'\ (+)\ 3\ [3] \\
&\ldots
\end{aligned}$$

sum' evaluates the addition whenever possible.

$foldl'$ — A Strict Version of $foldl$

$$
\begin{aligned}
&foldl' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&foldl'\ k\ z\ [\,] = z \\
&foldl'\ k\ z\ (x : xs) = \mathbf{let}\ z' = k\ z\ x\ \mathbf{in}\ seq\ z'\ (foldl'\ k\ z'\ xs)
\end{aligned}
$$

Example (strict sum')

$$
\begin{aligned}
sum'\ [1,2,3] &= foldl'\ (+)\ 0\ [1,2,3] \\
&= \mathbf{let}\ z' = 0 + 1\ \mathbf{in}\ seq\ z'\ (foldl'\ (+)\ z'\ [2,3]) \\
&= foldl'\ (+)\ 1\ [2,3] \\
&= \mathbf{let}\ z' = 1 + 2\ \mathbf{in}\ seq\ z'\ (foldl'\ (+)\ z'\ [3]) \\
&= foldl'\ (+)\ 3\ [3] \\
&\ldots
\end{aligned}
$$

sum' evaluates the addition whenever possible.

$\Rightarrow$ Saving space (and time)

foldl' — A Strict Version of *foldl*

$$\begin{aligned}
&\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}'\ k\ z\ [\,] = z \\
&\mathit{foldl}'\ k\ z\ (x : xs) = \mathbf{let}\ z' = k\ z\ x\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ k\ z'\ xs)
\end{aligned}$$

Example (strict *sum'*)

$$\begin{aligned}
\mathit{sum}'\ [1,2,3] &= \mathit{foldl}'\ (+)\ 0\ [1,2,3] \\
&= \mathbf{let}\ z' = 0 + 1\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [2,3]) \\
&= \mathit{foldl}'\ (+)\ 1\ [2,3] \\
&= \mathbf{let}\ z' = 1 + 2\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ (+)\ z'\ [3]) \\
&= \mathit{foldl}'\ (+)\ 3\ [3] \\
&\ldots
\end{aligned}$$

sum' evaluates the addition whenever possible.

⇒ Saving space (and time)

⇒ Strict evaluation pays off here.

Drawbacks of Selective Strictness

Question:

$$f\ (foldl'\ k\ z\ xs) \stackrel{?}{=} foldl'\ k'\ (f\ z)\ xs$$

$$\text{if } f\ (k\ x\ y) = k'\ (f\ x)\ y$$

Drawbacks of Selective Strictness

Question:

$$f\ (foldl'\ k\ z\ xs) \overset{?}{=} foldl'\ k'\ (f\ z)\ xs$$

$$\text{if } f\ (k\ x\ y) = k'\ (f\ x)\ y$$

Consider an example instantiation:

$$\begin{aligned} f &= \lambda x \rightarrow x \vee \bot \\ k = k' &= \lambda x\ y \rightarrow y \vee x \\ xs &= [False, True] \\ z &= False \end{aligned}$$

Drawbacks of Selective Strictness

Question:

$$f\ (foldl'\ k\ z\ xs) \stackrel{?}{=} foldl'\ k'\ (f\ z)\ xs$$
$$\text{if } f\ (k\ x\ y) = k'\ (f\ x)\ y$$

Consider an example instantiation:

$$\begin{aligned} f &= \lambda x \rightarrow x \vee \bot \\ k = k' &= \lambda x\ y \rightarrow y \vee x \\ xs &= [False, True] \\ z &= False \end{aligned}$$

$$f\ (foldl'\ k\ False\ [False, True]) = True$$

Drawbacks of Selective Strictness

Question:

$$f\ (foldl'\ k\ z\ xs) \stackrel{?}{=} foldl'\ k'\ (f\ z)\ xs$$
$$\text{if } f\ (k\ x\ y) = k'\ (f\ x)\ y$$

Consider an example instantiation:

$$\begin{aligned} f &= \lambda x \rightarrow x \vee \bot \\ k = k' &= \lambda x\ y \rightarrow y \vee x \\ xs &= [False, True] \\ z &= False \end{aligned}$$

$$f\ (foldl'\ k\ False\ [False, True]) = True$$

$$foldl'\ k'\ (f\ False)\ [False, True] = \bot$$

Drawbacks of Selective Strictness

Question:

$$f\ (foldl'\ k\ z\ xs) \stackrel{?}{=} foldl'\ k'\ (f\ z)\ xs$$

$$\text{if } f\ (k\ x\ y) = k'\ (f\ x)\ y$$

Consider an example instantiation:

$$\begin{aligned} f &= \lambda x \to x \vee \bot \\ k = k' &= \lambda x\ y \to y \vee x \\ xs &= [False, True] \\ z &= False \end{aligned}$$

$$f\ (foldl'\ k\ False\ [False, True]) = True$$

$$\neq$$

$$foldl'\ k'\ (f\ False)\ [False, True] = \bot$$

Drawbacks of Selective Strictness

Question:

$$f\ (foldl'\ k\ z\ xs)\ \boxed{\neq}\ foldl'\ k'\ (f\ z)\ xs$$
$$\text{if } f\ (k\ x\ y) = k'\ (f\ x)\ y$$

Consider an example instantiation:

$$\begin{aligned} f &= \lambda x \rightarrow x \vee \bot \\ k = k' &= \lambda x\ y \rightarrow y \vee x \\ xs &= [False, True] \\ z &= False \end{aligned}$$

Answer: No!

$$\begin{gathered} f\ (foldl'\ k\ False\ [False, True]) = True \\ \neq \\ foldl'\ k'\ (f\ False)\ [False, True] = \bot \end{gathered}$$

Analyzing the Problem

The strictness-aware free theorem:

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict and total.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (((k /= _|_) <=> (k' /= _|_))
     && (forall x :: t1.
          ((k x /= _|_) <=> (k' (f x) /= _|_))
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

Analyzing the Problem

The strictness-aware free theorem:

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict and total.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

Question: Are all these restrictions necessary?

Analyzing the Problem

The strictness-aware free theorem:

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict and total.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

Question: Are all these restrictions necessary?

An inductive proof for

$$f\ (foldl'\ k\ z\ xs) = foldl'\ k'\ (f\ z)\ xs$$

shows that $f\ x = \perp \Leftrightarrow x = \perp$ suffices (i.e. f is strict and total).

Why so Many Restrictions?

Free theorems depend only on the type.

Why so Many Restrictions?

Free theorems depend only on the type.

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

Why so Many Restrictions?

Free theorems depend only on the type.

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

Why so Many Restrictions?

Free theorems depend only on the type.

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$\begin{aligned}
&\mathit{foldl}'' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&\mathit{foldl}''\ k\ z\ [\,] = \mathit{seq}\ k\ z \\
&\mathit{foldl}''\ k\ z\ (x : \mathit{xs}) = \mathit{foldl}''\ (k\ z\ x)\ \mathit{xs}
\end{aligned}$$

Why so Many Restrictions?

Free theorems depend only on the type.

$$\mathit{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$\mathit{foldl}' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$\begin{aligned}
&\mathit{foldl}'' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha\\
&\mathit{foldl}''\ k\ z\ [\,] = \mathit{seq}\ k\ z\\
&\mathit{foldl}''\ k\ z\ (x : \mathit{xs}) = \mathit{foldl}''\ (k\ z\ x)\ \mathit{xs}
\end{aligned}$$

Problem: The free theorem is only aware of the potential risks of *seq*, but not of its concrete use.

Why so Many Restrictions?

Free theorems depend only on the type.

$$foldl :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$foldl' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$\begin{aligned}
&foldl'' :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha \\
&foldl''\ k\ z\ [\,] = seq\ k\ z \\
&foldl''\ k\ z\ (x : xs) = foldl''\ (k\ z\ x)\ xs
\end{aligned}$$

Problem: The free theorem is only aware of the potential risks of *seq*, but not of its concrete use.

Solution: Make the use of *seq* visible from the type. In particular where it is used.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

Distinguish types whos terms are / are not allowed
to be strictly evaluated.

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

Distinguish types whos terms are / are not allowed
to be strictly evaluated.

A former approach (Haskell 1.3)

$$\textit{foldl}' :: \textit{Eval}\ \alpha \Rightarrow (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$$

was not sufficient [Johann and Voigtländer, 2004].

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

Distinguish types whos terms are / are not allowed
to be strictly evaluated.

A former approach (Haskell 1.3)

$$\mathit{foldl}' :: \mathit{Eval}\ \alpha \Rightarrow (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$$

was not sufficient [Johann and Voigtländer, 2004].

The difference: two function types.

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict and total.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (((k /= _|_) <=> (k' /= _|_))
     && (forall x :: t1.
          ((k x /= _|_) <=> (k' (f x) /= _|_))
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System ...

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    ((  (k /= _|_) <=> (k' /= _|_)  )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \rightarrow^{\varepsilon} \tau \mid \tau \rightarrow^{\circ} \tau \mid \ldots$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
 forall t3,t4 in TYPES, g :: t3 -> t4, g  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t4 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) (g y))))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f (foldl' k z xs) = foldl' k' (f z) (map g xs))
```

The difference: two function types.

A Refined Type System . . .

Add new type constructors.

$$\tau ::= \alpha \mid \forall^{\varepsilon}\alpha.\tau \mid \forall^{\circ}\alpha.\tau \mid \tau \to^{\varepsilon} \tau \mid \tau \to^{\circ} \tau \mid \ldots$$

The difference: two function types.

. . . and its Effects on the Typing Rules (1)

A rule system for $\Gamma \vdash \tau \in \mathsf{Seqable}$:

$$\Gamma \vdash [\tau] \in \mathsf{Seqable} \qquad \Gamma \vdash (\tau_1 \rightarrow^{\varepsilon} \tau_2) \in \mathsf{Seqable}$$

$$\frac{\alpha^{\varepsilon} \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Seqable}} \qquad \frac{\alpha^{\varepsilon}, \Gamma \vdash \tau \in \mathsf{Seqable}}{\Gamma \vdash (\forall \alpha^{\nu}.\tau) \in \mathsf{Seqable}}$$

... and its Effects on the Typing Rules (1)

A rule system for $\Gamma \vdash \tau \in \mathsf{Seqable}$:

$$\Gamma \vdash [\tau] \in \mathsf{Seqable} \qquad \Gamma \vdash (\tau_1 \rightarrow^{\varepsilon} \tau_2) \in \mathsf{Seqable}$$

$$\frac{\alpha^{\varepsilon} \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Seqable}} \qquad \frac{\alpha^{\varepsilon}, \Gamma \vdash \tau \in \mathsf{Seqable}}{\Gamma \vdash (\forall \alpha^{\nu}.\tau) \in \mathsf{Seqable}}$$

Restricting $(\textsc{SLet})$

$$\frac{\color{red}{\Gamma \vdash \tau_1 \in \mathsf{Seqable}} \qquad \Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let}!\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}\ (\textsc{SLet}')$$

...and its Effects on the Typing Rules (1)

A rule system for $\Gamma \vdash \tau \in \mathsf{Seqable}$:

$$\Gamma \vdash [\tau] \in \mathsf{Seqable} \qquad \Gamma \vdash (\tau_1 \to^{\varepsilon} \tau_2) \in \mathsf{Seqable}$$

$$\frac{\alpha^{\varepsilon} \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Seqable}} \qquad \frac{\alpha^{\varepsilon}, \Gamma \vdash \tau \in \mathsf{Seqable}}{\Gamma \vdash (\forall \alpha^{\nu}.\tau) \in \mathsf{Seqable}}$$

Restricting $(\textsc{SLet})$

$$\frac{\Gamma \vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let}!\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}\ (\textsc{SLet}')$$

with $\Gamma = \alpha_1^{\nu_1}, \ldots \alpha_n^{\nu_n}, x_1 :: \tau_1, \ldots x_n :: \tau_n$ and $\nu_i \in \{\circ, \varepsilon\}$.

. . . and its Effects on the Typing Rules (2)

More typing rules because of new constructors:

$$\frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow^{\varepsilon} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow^{\circ} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

. . . and its Effects on the Typing Rules (2)

More typing rules because of new constructors:

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^{\varepsilon} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^{\circ} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

A term can have more than one type.

$$(\lambda x :: \mathit{Int}.\, x) :: \mathit{Int} \to^{\varepsilon} \mathit{Int}$$
$$(\lambda x :: \mathit{Int}.\, x) :: \mathit{Int} \to^{\circ} \mathit{Int}$$

...and its Effects on the Typing Rules (2)

More typing rules because of new constructors:

$$\frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow^{\varepsilon} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow^{\circ} \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

A term can have more than one type.

$$(\lambda x :: Int.\, x) :: Int \rightarrow^{\varepsilon} Int$$
$$(\lambda x :: Int.\, x) :: Int \rightarrow^{\circ} Int$$

We introduce subtyping.

$$\frac{\Gamma \vdash t :: \tau_1 \qquad \tau_1 \preceq \tau_2}{\Gamma \vdash t :: \tau_2}\ (\textsc{Sub})$$

Refinement Pays Off

The use of selective strictness becomes visible from the type ($\circ$- and ε-marks):

$$\begin{array}{r}
\mathit{foldl} :: \forall^{\circ}\alpha.\forall^{\circ}\beta.(\alpha \to^{\circ} \beta \to^{\circ} \alpha) \to^{\varepsilon} \alpha \to^{\varepsilon} [\beta] \to^{\varepsilon} \alpha \\
\mathit{foldl}' :: \forall^{\varepsilon}\alpha.\forall^{\circ}\beta.(\alpha \to^{\circ} \beta \to^{\circ} \alpha) \to^{\varepsilon} \alpha \to^{\varepsilon} [\beta] \to^{\varepsilon} \alpha
\end{array}$$

Refinement Pays Off

The use of selective strictness becomes visible from the type ($\circ$- and ε-marks):

$$\mathit{foldl} :: \forall^{\circ}\alpha.\,\forall^{\circ}\beta.\,(\alpha \rightarrow^{\circ} \beta \rightarrow^{\circ} \alpha) \rightarrow^{\varepsilon} \alpha \rightarrow^{\varepsilon} [\beta] \rightarrow^{\varepsilon} \alpha$$
$$\mathit{foldl}' :: \forall^{\varepsilon}\alpha.\,\forall^{\circ}\beta.\,(\alpha \rightarrow^{\circ} \beta \rightarrow^{\circ} \alpha) \rightarrow^{\varepsilon} \alpha \rightarrow^{\varepsilon} [\beta] \rightarrow^{\varepsilon} \alpha$$

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t3 -> t2.
    (( (k /= _|_) <=> (k' /= _|_) )
     && (forall x :: t1.
          ( (k x /= _|_) <=> (k' (f x) /= _|_) )
          && (forall y :: t3. f (k x y) = k' (f x) y)))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f ( foldl  k z xs) = foldl  k' (f z) xs)
```

Refinement Pays Off

The use of selective strictness becomes visible from the type (∘- and ε-marks):

$$\mathit{foldl} :: \forall^{\circ}\alpha.\,\forall^{\circ}\beta.\,(\alpha \to^{\circ} \beta \to^{\circ} \alpha) \to^{\varepsilon} \alpha \to^{\varepsilon} [\beta] \to^{\varepsilon} \alpha$$
$$\mathit{foldl}' :: \forall^{\varepsilon}\alpha.\,\forall^{\circ}\beta.\,(\alpha \to^{\circ} \beta \to^{\circ} \alpha) \to^{\varepsilon} \alpha \to^{\varepsilon} [\beta] \to^{\varepsilon} \alpha$$

Restrictions on free theorems can be dropped if the type guarantees selective strictness is not used.

```
forall t1,t2 in TYPES, f :: t1 -> t2                          .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t3 -> t2.
    (
        (forall x :: t1.

             (forall y :: t3. f (k x y) = k' (f x) y)))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f ( foldl   k z xs) = foldl   k' (f z) xs)
```

Refinement Pays Off

The use of selective strictness becomes visible from the type ($\circ$- and ε-marks):

$$\begin{aligned} \mathit{foldl} &:: \forall^{\circ}\alpha.\,\forall^{\circ}\beta.\,(\alpha \to^{\circ} \beta \to^{\circ} \alpha) \to^{\varepsilon} \alpha \to^{\varepsilon} [\beta] \to^{\varepsilon} \alpha \\ \mathit{foldl}' &:: \forall^{\varepsilon}\alpha.\,\forall^{\circ}\beta.\,(\alpha \to^{\circ} \beta \to^{\circ} \alpha) \to^{\varepsilon} \alpha \to^{\varepsilon} [\beta] \to^{\varepsilon} \alpha \end{aligned}$$

Restrictions on free theorems can be dropped if the type guarantees selective strictness is not used.

```
forall t1,t2 in TYPES, f :: t1 -> t2, f  strict and total .
  forall k :: t1 -> t3 -> t1.
   forall k' :: t2 -> t3 -> t2.
    (
        (forall x :: t1.

             (forall y :: t3. f (k x y) = k' (f x) y)))
    ==> (forall z :: t1.
          forall xs :: [t3].
            f ( foldl'  k z xs) =  foldl'  k' (f z) xs)
```

Going Algorithmic

Goal: An algorithm retyping from standard types to (minimal) refined types.

Going Algorithmic

Goal: An algorithm retyping from standard types
to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Going Algorithmic

Goal: An algorithm retyping from standard types to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Problems:

- Type annotations are standard in the input term, but we need refined ones.

Going Algorithmic

Goal: An algorithm retyping from standard types to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Problems:

- Type annotations are standard in the input term, but we need refined ones.
- Typing rules are in competition, especially the $(\textsc{Sub})$-rule is always applicable.

Going Algorithmic

Goal: An algorithm retyping from standard types
to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Problems:

- Type annotations are standard in the input term, but we need refined ones.
- Typing rules are in competition, especially the $(\textsc{Sub})$-rule is always applicable.

Solutions:

- Remove the $(\textsc{Sub})$-rule by integrating subtyping into the other rules.

Going Algorithmic

Goal: An algorithm retyping from standard types
to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Problems:

- Type annotations are standard in the input term, but we need refined ones.
- Typing rules are in competition, especially the $(\textsc{Sub})$-rule is always applicable.

Solutions:

- Remove the $(\textsc{Sub})$-rule by integrating subtyping into the other rules.
- Start the algorithm with all possible refined type annotations.

Going Algorithmic

Goal: An algorithm retyping from standard types to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Problems:

- Type annotations are standard in the input term, but we need refined ones.
- Typing rules are in competition, especially the $(\textsc{Sub})$-rule is always applicable.

Solutions:

- Remove the $(\textsc{Sub})$-rule by integrating subtyping into the other rules.
- Start the algorithm with all possible refined type annotations.

Are these good ideas?

Going Algorithmic

Goal: An algorithm retyping from standard types
to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Problems:

- Type annotations are standard in the input term, but we need refined ones.
- Typing rules are in competition, especially the $(\textsc{Sub})$-rule is always applicable.

Solutions:

- Remove the $(\textsc{Sub})$-rule by integrating subtyping into the other rules.
- Start the algorithm with all possible refined type annotations.

Are these good ideas?

Going Algorithmic

Goal: An algorithm retyping from standard types
to (minimal) refined types.

Idea: Use typing rules backwards to (re)type a term.

Problems:

- Type annotations are standard in the input term, but we need refined ones.
- Typing rules are in competition, especially the $(\textsc{Sub})$-rule is always applicable.

Solutions:

- Remove the $(\textsc{Sub})$-rule by integrating subtyping into the other rules.
- Start the algorithm with all possible refined type annotations.

Are these good ideas?

How to Deal with Refined Type Annotations?

Switch to variable marks at the type annotations:

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^{\varepsilon} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to^{\circ} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

How to Deal with Refined Type Annotations?

Switch to variable marks at the type annotations:

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^{\varepsilon} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to^{\circ} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

$\Downarrow$ combine two rules into one

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^{\nu} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

How to Deal with Refined Type Annotations?

Switch to variable marks at the type annotations:

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^{\varepsilon} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to^{\circ} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

$\Downarrow$ combine two rules into one

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^{\nu} \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

$\Downarrow$ add constraints for the mark variables

$$\frac{\langle \dot{\Gamma} \vdash \dot{t}_1 \rangle \Rrightarrow (C_1, \dot{\tau}_1 \to^{\nu} \dot{\tau}_2) \quad \langle \dot{\Gamma} \vdash \dot{t}_2 \rangle \Rrightarrow (C_2, \dot{\tau}_1') \quad \langle \dot{\tau}_1 = \dot{\tau}_1' \rangle \Rrightarrow C_3}{\langle \dot{\Gamma} \vdash \dot{t}_1\ \dot{t}_2 \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot{\tau}_2)}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow (?, ?)$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\frac{\langle \alpha \rightarrow^{\nu_2} \alpha \preceq \cdot \rangle \Rrightarrow (?, ?)}{\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow (?, ?)}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\dfrac{\dfrac{\langle \cdot \preceq \alpha \rangle \Rrightarrow (?, ?) \qquad \langle \alpha \preceq \cdot \rangle \Rrightarrow (?, ?)}{\langle \alpha \rightarrow^{\nu_2} \alpha \preceq \cdot \rangle \Rrightarrow (?, ?)}}{\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow (?, ?)}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\frac{\dfrac{\langle \cdot \preceq \alpha \rangle \Rrightarrow (\mathsf{True}, \alpha) \qquad \langle \alpha \preceq \cdot \rangle \Rrightarrow (?, ?)}{\langle \alpha \rightarrow^{\nu_2} \alpha \preceq \cdot \rangle \Rrightarrow (?, ?)}}{\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow (?, ?)}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\cfrac{\cfrac{\langle \cdot \preceq \alpha \rangle \Rrightarrow (\mathsf{True}, \alpha) \qquad \langle \alpha \preceq \cdot \rangle \Rrightarrow {\color{red}(\mathsf{True}, \alpha)}}{\langle \alpha \rightarrow^{\nu_2} \alpha \preceq \cdot \rangle \Rrightarrow (?, ?)}}{\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow (?, ?)}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\dfrac{\dfrac{\langle \cdot \preceq \alpha \rangle \Rrightarrow (\mathsf{True}, \alpha) \qquad \langle \alpha \preceq \cdot \rangle \Rrightarrow (\mathsf{True}, \alpha)}{\langle \alpha \rightarrow^{\nu_2} \alpha \preceq \cdot \rangle \Rrightarrow (\mathsf{True} \wedge \mathsf{True} \wedge (\nu_3 \leqslant \nu_2), \alpha \rightarrow^{\nu_3} \alpha)}}{\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow (?, ?)}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\dfrac{\dfrac{\langle \cdot \preceq \alpha \rangle \Rrightarrow (\mathsf{True}, \alpha) \qquad \langle \alpha \preceq \cdot \rangle \Rrightarrow (\mathsf{True}, \alpha)}{\langle \alpha \rightarrow^{\nu_2} \alpha \preceq \cdot \rangle \Rrightarrow (\mathsf{True} \wedge \mathsf{True} \wedge (\nu_3 \leqslant \nu_2), \alpha \rightarrow^{\nu_3} \alpha)}}{\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow ((\nu_3 \leqslant \nu_2), \alpha \rightarrow^{\nu_3} \alpha)}$$

The Resulting (Re)Typing Algorithm

A deterministic typing algorithm.

How does it work?

$$\text{input} \Rrightarrow \text{output}$$

$$\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$$

$$\dfrac{\dfrac{\langle \cdot \preceq \alpha \rangle \Rrightarrow (\mathsf{True}, \alpha) \qquad \langle \alpha \preceq \cdot \rangle \Rrightarrow (\mathsf{True}, \alpha)}{\langle \alpha \rightarrow^{\nu_2} \alpha \preceq \cdot \rangle \Rrightarrow (\mathsf{True} \wedge \mathsf{True} \wedge (\nu_3 \leqslant \nu_2), \alpha \rightarrow^{\nu_3} \alpha)}}{\langle \alpha^{\nu_1}, x :: \alpha \rightarrow^{\nu_2} \alpha \vdash x \rangle \Rrightarrow ((\nu_3 \leqslant \nu_2), \alpha \rightarrow^{\nu_3} \alpha)}$$

How to get back to concrete types, without mark variables?

Back to Concrete Typability (Example)

We have:

$$\begin{array}{l}\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \rightarrow^{\nu_1} \beta.\, \lambda x :: \alpha.\, \textbf{let}!\ x' = x \textbf{ in } f\ x' \rangle \Rrightarrow \\ ((\nu_2 = \varepsilon) \wedge (\nu_4 \leqslant \nu_1) \wedge (\nu_1 \leqslant \nu_6), \\ \qquad\qquad\qquad \forall^{\nu_2}\alpha.\, \forall^{\nu_3}\beta.\, (\alpha \rightarrow^{\nu_6} \beta) \rightarrow^{\nu_7} \alpha \rightarrow^{\nu_5} \beta)\end{array}$$

with $\circ < \varepsilon$.

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f :: \alpha \rightarrow^{\nu_1} \beta.\,\lambda x :: \alpha.\,\textbf{let}!\ x' = x\ \textbf{in}\ f\ x' \rangle \Rrightarrow$$
$$((\nu_2 = \varepsilon) \wedge (\nu_4 \leqslant \nu_1) \wedge (\nu_1 \leqslant \nu_6),$$
$$\forall^{\nu_2}\alpha.\,\forall^{\nu_3}\beta.\,(\alpha \rightarrow^{\nu_6} \beta) \rightarrow^{\nu_7} \alpha \rightarrow^{\nu_5} \beta)$$

with $\circ < \varepsilon$.

Back to Concrete Typability (Example)

We have:

$$\begin{aligned}&\langle \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f :: \alpha \rightarrow^{\nu_1} \beta.\,\lambda x :: \alpha.\,\mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x' \rangle \Rrightarrow \\ &((\nu_2 = \varepsilon) \wedge (\nu_4 \leqslant \nu_1) \wedge (\nu_1 \leqslant \nu_6), \\ &\qquad\qquad \forall^{\nu_2}\alpha.\,\forall^{\nu_3}\beta.\,(\alpha \rightarrow^{\nu_6} \beta) \rightarrow^{\nu_7} \alpha \rightarrow^{\nu_5} \beta)\end{aligned}$$

with $\circ < \varepsilon$.

Back to Concrete Typability (Example)

We have:

$$\begin{aligned}
&\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \rightarrow^{\nu_1} \beta.\, \lambda x :: \alpha.\, \textbf{let}!\ x' = x\ \textbf{in}\ f\ x' \rangle \Rrightarrow \\
&((\nu_2 = \varepsilon) \wedge (\nu_4 \leqslant \nu_1) \wedge (\nu_1 \leqslant \nu_6), \\
&\qquad\qquad\qquad \forall^{\nu_2}\alpha.\, \forall^{\nu_3}\beta.\, (\alpha \rightarrow^{\nu_6} \beta) \rightarrow^{\nu_7} \alpha \rightarrow^{\nu_5} \beta)
\end{aligned}$$

with $\circ < \varepsilon$.

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f :: \alpha \rightarrow^{\nu_1} \beta.\,\lambda x :: \alpha.\,\textbf{let}!\ x' = x\ \textbf{in}\ f\ x'\rangle \Rrightarrow$$
$$((\nu_2 = \varepsilon) \wedge (\nu_4 \leqslant \nu_1) \wedge (\nu_1 \leqslant \nu_6),$$
$$\forall^{\nu_2}\alpha.\,\forall^{\nu_3}\beta.\,(\alpha \rightarrow^{\nu_6} \beta) \rightarrow^{\nu_7} \alpha \rightarrow^{\nu_5} \beta)$$

with $\circ < \varepsilon$.

Back to Concrete Typability (Example)

We have:

$$\begin{aligned}&\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \rightarrow^{\nu_1} \beta.\, \lambda x :: \alpha.\, \textbf{let}!\ x' = x \textbf{ in } f\ x' \rangle \Rrightarrow \\ &((\nu_2 = \varepsilon) \wedge (\nu_4 \leqslant \nu_1) \wedge (\nu_1 \leqslant \nu_6), \\ &\qquad\qquad\qquad \forall^{\nu_2}\alpha.\, \forall^{\nu_3}\beta.\, (\alpha \rightarrow^{\nu_6} \beta) \rightarrow^{\nu_7} \alpha \rightarrow^{\nu_5} \beta)\end{aligned}$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f :: \alpha \to^{\nu_1} \beta.\,\lambda x :: \alpha.\,\mathbf{let!}\ x' = x \ \mathbf{in}\ f\ x' \rangle \Rrightarrow$$
$$((\nu_2 = \varepsilon) \land (\nu_4 \leqslant \nu_1) \land (\nu_1 \leqslant \nu_6),$$
$$\forall^{\nu_2}\alpha.\,\forall^{\nu_3}\beta.\,(\alpha \to^{\nu_6} \beta) \to^{\nu_7} \alpha \to^{\nu_5} \beta)$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{aligned}
\nu_1 &= \circ \\
\nu_2 &= \circ \\
\nu_3 &= \circ \\
\nu_4 &= \circ \\
\nu_5 &= \circ \\
\nu_6 &= \circ \\
\nu_7 &= \circ
\end{aligned}$$

Back to Concrete Typability (Example)

We have:

$$\begin{array}{l}\langle \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f :: \alpha \rightarrow^{\circ} \beta.\,\lambda x :: \alpha.\,\textbf{let}!\ x' = x\ \textbf{in}\ f\ x'\rangle \Rrightarrow \\ ((\circ = \varepsilon) \wedge (\circ \leqslant \circ) \wedge (\circ \leqslant \circ), \\ \qquad\qquad\qquad\qquad \forall^{\circ}\alpha.\,\forall^{\circ}\beta.\,(\alpha \rightarrow^{\circ} \beta) \rightarrow^{\circ} \alpha \rightarrow^{\circ} \beta)\end{array}$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{array}{rcl}\nu_1 & = & \circ \\ \nu_2 & = & \circ \\ \nu_3 & = & \circ \\ \nu_4 & = & \circ \\ \nu_5 & = & \circ \\ \nu_6 & = & \circ \\ \nu_7 & = & \circ\end{array}$$

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \rightarrow^{\circ} \beta.\, \lambda x :: \alpha.\, \mathbf{let}!\ x' = x\ \mathbf{in}\ f\ x' \rangle \Rrightarrow$$
$$(\mathit{False},$$
$$\forall^{\circ}\alpha.\, \forall^{\circ}\beta.\, (\alpha \rightarrow^{\circ} \beta) \rightarrow^{\circ} \alpha \rightarrow^{\circ} \beta)$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{aligned} \nu_1 &= \circ \\ \nu_2 &= \circ \\ \nu_3 &= \circ \\ \nu_4 &= \circ \\ \nu_5 &= \circ \\ \nu_6 &= \circ \\ \nu_7 &= \circ \end{aligned}$$

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \to^{\varepsilon} \beta.\, \lambda x :: \alpha.\, \textbf{let}!\ x' = x\ \textbf{in}\ f\ x' \rangle \Rrightarrow$$
$$((\varepsilon = \varepsilon) \wedge (\varepsilon \leqslant \varepsilon) \wedge (\varepsilon \leqslant \varepsilon),$$
$$\forall^{\varepsilon}\alpha.\, \forall^{\varepsilon}\beta.\, (\alpha \to^{\varepsilon} \beta) \to^{\varepsilon} \alpha \to^{\varepsilon} \beta)$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{array}{lllcllll} \nu_1 & = & \circ & \quad & \nu_1 & = & \varepsilon \\ \nu_2 & = & \circ & & \nu_2 & = & \varepsilon \\ \nu_3 & = & \circ & & \nu_3 & = & \varepsilon \\ \nu_4 & = & \circ & & \nu_4 & = & \varepsilon \\ \nu_5 & = & \circ & & \nu_5 & = & \varepsilon \\ \nu_6 & = & \circ & & \nu_6 & = & \varepsilon \\ \nu_7 & = & \circ & & \nu_7 & = & \varepsilon \end{array}$$

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \rightarrow^{\varepsilon} \beta.\, \lambda x :: \alpha.\, \textbf{let}!\ x' = x \textbf{ in } f\ x' \rangle \Rrightarrow$$
$$(\mathit{True},$$
$$\forall^{\varepsilon}\alpha.\, \forall^{\varepsilon}\beta.\, (\alpha \rightarrow^{\varepsilon} \beta) \rightarrow^{\varepsilon} \alpha \rightarrow^{\varepsilon} \beta)$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{array}{lll} \nu_1 = \circ & & \nu_1 = \varepsilon \\ \nu_2 = \circ & & \nu_2 = \varepsilon \\ \nu_3 = \circ & & \nu_3 = \varepsilon \\ \nu_4 = \circ & & \nu_4 = \varepsilon \\ \nu_5 = \circ & & \nu_5 = \varepsilon \\ \nu_6 = \circ & & \nu_6 = \varepsilon \\ \nu_7 = \circ & & \nu_7 = \varepsilon \end{array}$$

Back to Concrete Typability (Example)

We have:

$$\begin{aligned}&\langle \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f :: \alpha \to^{\circ} \beta.\,\lambda x :: \alpha.\,\textbf{let}!\ x' = x \textbf{ in } f\ x'\rangle \Rrightarrow \\ &((\varepsilon = \varepsilon) \wedge (\circ \leqslant \circ) \wedge (\circ \leqslant \circ), \\ &\qquad\qquad\qquad \forall^{\varepsilon}\alpha.\,\forall^{\circ}\beta.\,(\alpha \to^{\circ} \beta) \to^{\varepsilon} \alpha \to^{\varepsilon} \beta)\end{aligned}$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{array}{lll} \nu_1 = \circ & \nu_1 = \varepsilon & \nu_1 = \circ \\ \nu_2 = \circ & \nu_2 = \varepsilon & \nu_2 = \varepsilon \\ \nu_3 = \circ & \nu_3 = \varepsilon & \nu_3 = \circ \\ \nu_4 = \circ & \nu_4 = \varepsilon & \nu_4 = \circ \\ \nu_5 = \circ & \nu_5 = \varepsilon & \nu_5 = \varepsilon \\ \nu_6 = \circ & \nu_6 = \varepsilon & \nu_6 = \circ \\ \nu_7 = \circ & \nu_7 = \varepsilon & \nu_7 = \varepsilon \end{array}$$

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \rightarrow^{\circ} \beta.\, \lambda x :: \alpha.\, \textbf{let}!\ x' = x \ \textbf{in}\ f\ x' \rangle \Rrightarrow$$
$$(\mathit{True},$$
$$\forall^{\varepsilon}\alpha.\, \forall^{\circ}\beta.\, (\alpha \rightarrow^{\circ} \beta) \rightarrow^{\varepsilon} \alpha \rightarrow^{\varepsilon} \beta)$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{array}{lll} \nu_1 = \circ & \nu_1 = \varepsilon & \nu_1 = \circ \\ \nu_2 = \circ & \nu_2 = \varepsilon & \nu_2 = \varepsilon \\ \nu_3 = \circ & \nu_3 = \varepsilon & \nu_3 = \circ \\ \nu_4 = \circ & \nu_4 = \varepsilon & \nu_4 = \circ \\ \nu_5 = \circ & \nu_5 = \varepsilon & \nu_5 = \varepsilon \\ \nu_6 = \circ & \nu_6 = \varepsilon & \nu_6 = \circ \\ \nu_7 = \circ & \nu_7 = \varepsilon & \nu_7 = \varepsilon \end{array}$$

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f :: \alpha \rightarrow^{\circ} \beta.\,\lambda x :: \alpha.\,\mathbf{let!}\ x' = x\ \mathbf{in}\ f\ x' \rangle \Rrightarrow$$
$$(\mathit{True},$$
$$\forall^{\varepsilon}\alpha.\,\forall^{\circ}\beta.\,(\alpha \rightarrow^{\circ} \beta) \rightarrow^{\varepsilon} \alpha \rightarrow^{\varepsilon} \beta)$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{array}{lcl@{\qquad}lcl}
\nu_1 & = & \varepsilon & \nu_1 & = & \circ \\
\nu_2 & = & \varepsilon & \nu_2 & = & \varepsilon \\
\nu_3 & = & \varepsilon & \nu_3 & = & \circ \\
\nu_4 & = & \varepsilon & \nu_4 & = & \circ \\
\nu_5 & = & \varepsilon & \nu_5 & = & \varepsilon \\
\nu_6 & = & \varepsilon & \nu_6 & = & \circ \\
\nu_7 & = & \varepsilon & \nu_7 & = & \varepsilon
\end{array}$$

Back to Concrete Typability (Example)

We have:

$$\langle \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda f :: \alpha \rightarrow^{\circ} \beta.\, \lambda x :: \alpha.\, \textbf{let}!\ x' = x\ \textbf{in}\ f\ x' \rangle \Rrightarrow$$
$$(\mathit{True},$$
$$\forall^{\varepsilon}\alpha.\, \forall^{\circ}\beta.\, (\alpha \rightarrow^{\circ} \beta) \rightarrow^{\varepsilon} \alpha \rightarrow^{\varepsilon} \beta)$$

with $\circ < \varepsilon$.

We test all possible instantiations for the variable marks.

$$\begin{aligned} \nu_1 &= \circ \\ \nu_2 &= \varepsilon \\ \nu_3 &= \circ \\ \nu_4 &= \circ \\ \nu_5 &= \varepsilon \\ \nu_6 &= \circ \\ \nu_7 &= \varepsilon \end{aligned}$$

We take only the minimal solution!

Make it a Type Refinement Algorithm

$\boxed{\text{input: closed term with standard type annotations}}$

Make it a Type Refinement Algorithm

$\boxed{\text{input: closed term with standard type annotations}}$

$\Downarrow$ add variable marks

$\boxed{\text{term with parameterized refined type annotations}}$

Make it a Type Refinement Algorithm

$\boxed{\text{input: closed term with standard type annotations}}$

$\Downarrow$ add variable marks

$\boxed{\text{term with parameterized refined type annotations}}$

$\Downarrow$ the main algorithm

$\boxed{\text{constraint and parameterized type}}$

Make it a Type Refinement Algorithm

$\boxed{\text{input: closed term with standard type annotations}}$

$\Downarrow$ add variable marks

$\boxed{\text{term with parameterized refined type annotations}}$

$\Downarrow$ the main algorithm

$\boxed{\text{constraint and parameterized type}}$

$\Downarrow$ solve constraint

$\boxed{\text{all possible refined types}}$

Make it a Type Refinement Algorithm

$$\boxed{\text{input: closed term with standard type annotations}}$$

$\Downarrow$ add variable marks

$$\boxed{\text{term with parameterized refined type annotations}}$$

$\Downarrow$ the main algorithm

$$\boxed{\text{constraint and parameterized type}}$$

$\Downarrow$ solve constraint

$$\boxed{\text{all possible refined types}}$$

$\Downarrow$ type comparison

$$\boxed{\text{output: the refined types leading to the strongest free theorems}}$$

The Webinterface

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:

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

References I

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

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

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

References II

Seidel, D. and Voigtländer, J. (2009).
Taming selective strictness.
Technical Report TUD-FI09-06, Technische Universität Dresden.
http://wwwtcs.inf.tu-dresden.de/~voigt/TUD-FI09-06.pdf.

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