A Framework for *Point-free* Program Transformation

Alcino Cunha Jorge Sousa Pinto José Proença

Department of Informatics
University of Minho

IFL, 2005

Outline of Part I
Basic Theory

Outline of Part II
Tools and Libraries

Outline

What Is *Point-free*
In Functional Context

Two different programming styles

- In ***pointwise*** – Using variables
- In ***point-free*** – Without the use of variables

Point-free language

- categorically-inspired combinators
- algebraic data types as fixed points of functors
- polytypic recursion patterns

What Is *Point-free*
In Functional Context

Two different programming styles

- In ***pointwise*** – Using variables
- In ***point-free*** – Without the use of variables

Point-free language

- categorically-inspired combinators
- algebraic data types as fixed points of functors
- polytypic recursion patterns

Advantages

Appropriate for equational reasoning
BUT
difficult for programming in practice.

How to get the best of each style?

Advantages

Appropriate for equational reasoning
BUT
difficult for programming in practice.

How to get the best of each style?

The Framework

Main topics

- Automatic conversion of programs to *point-free*
- *Point-free* manipulation
- *Point-free* rule-driven simplification

Use of the *Haskell* programming language.

The Framework

Main topics

- Automatic conversion of programs to *point-free*
- *Point-free* manipulation
- *Point-free* rule-driven simplification

Use of the *Haskell* programming language.

Tools And Libraries

Developed tools and libraries

Pointless

- *Haskell* library for *point-free* programming
- type-checking and execution of *point-free*
- type-parameterized recursion patterns

- automatic conversion to *point-free*
- recursive functions to *hylomorphisms*
- intermediate data structure inference

- *point-free* code manipulation
- simplification of *point-free* code
- program transformation

Tools And Libraries

Developed tools and libraries

Pointless
- *Haskell* library for *point-free* programming
- type-checking and execution of *point-free*
- type-parameterized recursion patterns

DrHylo
- automatic conversion to *point-free*
- recursive functions to *hylomorphisms*
- intermediate data structure inference

- *point-free* code manipulation
- simplification of *point-free* code
- program transformation

Tools And Libraries

Developed tools and libraries

Pointless	- *Haskell* library for *point-free* programming - type-checking and execution of *point-free* - type-parameterized recursion patterns
DrHylo	- automatic conversion to *point-free* - recursive functions to *hylomorphisms* - intermediate data structure inference
SimpliFree	- *point-free* code manipulation - simplification of *point-free* code - program transformation

Outline

Types
Definition

$$
\begin{array}{rll}
A, B ::= & 1 & - \text{ single element type} \\
| & A \to B & - \text{ functions} \\
| & A \times B & - \text{ cartesian product} \\
| & A + B & - \text{ separated sum} \\
| & \mu F & - \text{ recursive (regular) type,} \\
 & & \quad \text{defined as the fixed point of a functor}
\end{array}
$$

$$
\begin{array}{rll}
F, G ::= & \mathsf{Id} & - \text{ identity functor} \\
| & \underline{A} & - \text{ constant functor} \\
| & F \otimes G & - \text{ lifted product bifunctor} \\
| & F \oplus G & - \text{ lifted sum bifunctor} \\
| & F \circledcirc G & - \text{ composition of functors}
\end{array}
$$

Types
Examples

Booleans, Naturals and Lists

- $\mathsf{Bool} = 1 + 1$
- $\mathsf{Nat} = \mu(\underline{1} \oplus \mathsf{Id})$
- $\mathsf{List}\ A = \mu(\underline{1} \oplus (\underline{A} \otimes \mathsf{Id}))$

Pointwise Language

Syntax

$$
\begin{array}{rcl}
L, M, N & ::= & \star \mid x \mid M\ N \mid \lambda x.\ M \mid \\
& & \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \\
& & \mathsf{case}\ L\ M\ N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \\
& & \mathsf{in}_{\mu F}\ M \mid \mathsf{out}_{\mu F}\ M \mid \mathsf{fix}\ M
\end{array}
$$

Examples

$$\mathsf{in}_{\mathsf{Nat}} : 1 + \mathsf{Nat} \to \mathsf{Nat} \qquad \mathsf{out}_{\mathsf{Nat}} : \mathsf{Nat} \to 1 + \mathsf{Nat}$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{zero} & : & \mathsf{Nat} & \mathsf{succ} & : & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{zero} & = & \mathsf{in}\ (\mathsf{inl}\ \star) & \mathsf{succ} & = & \lambda x.\ \mathsf{in}\ (\mathsf{inr}\ x)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{swap} & : & A \times B \to B \times A \\
\mathsf{swap} & = & \lambda x.\ \langle \mathsf{snd}\ x, \mathsf{fst}\ x \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{length} & : & \mathsf{List}\ A \to \mathsf{Nat} \\
\mathit{length} & = & \mathsf{fix}\ (\lambda f.\ \lambda x.\ \mathsf{case}\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{zero}) \\
& & \qquad (\lambda y.\ \mathsf{succ}\ (f\ y)))
\end{array}
$$

Pointwise Language

Syntax

$$
\begin{array}{rcl}
L, M, N & ::= & \star \mid x \mid M\ N \mid \lambda x.\ M \mid \\
& & \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \\
& & \mathsf{case}\ L\ M\ N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \\
& & \mathsf{in}_{\mu F}\ M \mid \mathsf{out}_{\mu F}\ M \mid \mathsf{fix}\ M
\end{array}
$$

Examples

$$\mathsf{in}_{\mathsf{Nat}} : 1 + \mathsf{Nat} \to \mathsf{Nat} \qquad \mathsf{out}_{\mathsf{Nat}} : \mathsf{Nat} \to 1 + \mathsf{Nat}$$

$$
\begin{array}{lcllcl}
\mathsf{zero} & : & \mathsf{Nat} & \mathsf{succ} & : & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{zero} & = & \mathsf{in}\ (\mathsf{inl}\ \star) & \mathsf{succ} & = & \lambda x.\ \mathsf{in}\ (\mathsf{inr}\ x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{swap} & : & A \times B \to B \times A \\
\mathsf{swap} & = & \lambda x.\ \langle \mathsf{snd}\ x, \mathsf{fst}\ x \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{length} & : & \mathsf{List}\ A \to \mathsf{Nat} \\
\mathit{length} & = & \mathsf{fix}\ (\lambda f.\ \lambda x.\ \mathsf{case}\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{zero}) \\
& & \qquad (\lambda y.\ \mathsf{succ}\ (f\ y)))
\end{array}
$$

Pointwise Language

Syntax

$$
\begin{array}{rcl}
L, M, N & ::= & \star \mid x \mid M\ N \mid \lambda x.\ M \mid \\
& & \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \\
& & \mathsf{case}\ L\ M\ N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \\
& & \mathsf{in}_{\mu F}\ M \mid \mathsf{out}_{\mu F}\ M \mid \mathsf{fix}\ M
\end{array}
$$

Examples

$$\mathsf{in}_{\mathsf{Nat}} : 1 + \mathsf{Nat} \to \mathsf{Nat} \qquad \mathsf{out}_{\mathsf{Nat}} : \mathsf{Nat} \to 1 + \mathsf{Nat}$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{zero} & : & \mathsf{Nat} & \mathsf{succ} & : & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{zero} & = & \mathsf{in}\ (\mathsf{inl}\ \star) & \mathsf{succ} & = & \lambda x.\ \mathsf{in}\ (\mathsf{inr}\ x)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{swap} & : & A \times B \to B \times A \\
\mathsf{swap} & = & \lambda x.\ \langle \mathsf{snd}\ x, \mathsf{fst}\ x \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{length} & : & \mathsf{List}\ A \to \mathsf{Nat} \\
\mathit{length} & = & \mathsf{fix}\ (\lambda f.\ \lambda x.\ \mathsf{case}\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{zero}) \\
& & \qquad\qquad (\lambda y.\ \mathsf{succ}\ (f\ y)))
\end{array}
$$

Point-free Language
Syntax

Constants

fst snd inl inr in out id

$\text{bang} : A \to 1 \qquad \text{ap} : (A \to B) \times A \to B$

Combinators

$(\cdot \circ \cdot) : (B \to C) \to (A \to B) \to A \to C$

$(\cdot \vartriangle \cdot) : (A \to B) \to (A \to C) \to A \to (B \times C)$

$(\cdot \triangledown \cdot) : (A \to C) \to (B \to C) \to (A + B) \to C$

$\overline{\cdot} : (A \times B \to C) \to A \to B \to C$

Combinators for the product, sum and exponention functors on functions were also derived.

Point-free Language
Syntax

Constants

fst snd inl inr in out id

$\text{bang} : A \to 1 \qquad \text{ap} : (A \to B) \times A \to B$

Combinators

$(\cdot \circ \cdot) : (B \to C) \to (A \to B) \to A \to C$

$(\cdot \vartriangle \cdot) : (A \to B) \to (A \to C) \to A \to (B \times C)$

$(\cdot \triangledown \cdot) : (A \to C) \to (B \to C) \to (A + B) \to C$

$\overline{\cdot} : (A \times B \to C) \to A \to B \to C$

Combinators for the product, sum and exponention functors on functions were also derived.

Point-free Language
Implicit Recursion

Recursion through the *hylomorphism* recursion pattern.
Allows the definition of any fixed point.

Defined in *pointwise* as

$$\begin{array}{lcl} \text{hylo}_{\mu F} & : & (F\ B \to B) \to (A \to F\ A) \to A \to B \\ \text{hylo}_{\mu F} & = & \lambda g.\ \lambda h.\ \text{fix}(\lambda f.\ g \circ Ff \circ h) \end{array}$$

Examples

$$\begin{array}{lcl} \text{length} & : & \text{List}\ A \to \text{Nat} \\ \text{length} & = & \text{hylo}_{\text{Nat}}\quad \text{in}_{\text{Nat}}\quad ((\text{id} + \text{snd}) \circ \text{out}_{\text{List}\ A}) \\ (\!|\cdot|\!)_{\mu F} & : & (F\ A \to A) \to \mu F \to A \\ (\!|\cdot|\!)_{\mu F} & = & \lambda g.\ \text{hylo}_{\mu F}\ \ g\ \ \text{out}_{\mu F} \end{array}$$

Point-free Language
Implicit Recursion

Recursion through the *hylomorphism* recursion pattern.
Allows the definition of any fixed point.

Defined in *pointwise* as

$$\begin{array}{lcl} \text{hylo}_{\mu F} & : & (F\ B \to B) \to (A \to F\ A) \to A \to B \\ \text{hylo}_{\mu F} & = & \lambda g.\ \lambda h.\ \text{fix}(\lambda f.\ g \circ Ff \circ h) \end{array}$$

Examples

$$\begin{array}{lcl} \text{length} & : & \text{List}\ A \to \text{Nat} \\ \text{length} & = & \text{hylo}_{\text{Nat}}\ \ \text{in}_{\text{Nat}}\ \ ((\text{id} + \text{snd}) \circ \text{out}_{\text{List}\ A}) \\ (\!|\cdot|\!)_{\mu F} & : & (F\ A \to A) \to \mu F \to A \\ (\!|\cdot|\!)_{\mu F} & = & \lambda g.\ \text{hylo}_{\mu F}\ \ g\ \ \text{out}_{\mu F} \end{array}$$

Point-free Language
Implicit Recursion

Recursion through the *hylomorphism* recursion pattern.
Allows the definition of any fixed point.

Defined in *pointwise* as

$$\begin{array}{lcl} \text{hylo}_{\mu F} & : & (F\ B \to B) \to (A \to F\ A) \to A \to B \\ \text{hylo}_{\mu F} & = & \lambda g.\ \lambda h.\ \text{fix}(\lambda f.\ g \circ Ff \circ h) \end{array}$$

Examples

$$\begin{array}{lcl} \text{length} & : & \text{List}\ A \to \text{Nat} \\ \text{length} & = & \text{hylo}_{\text{Nat}}\quad \text{in}_{\text{Nat}}\quad ((\text{id} + \text{snd}) \circ \text{out}_{\text{List}\ A}) \\ (\!|\cdot|\!)_{\mu F} & : & (F\ A \to A) \to \mu F \to A \\ (\!|\cdot|\!)_{\mu F} & = & \lambda g.\ \text{hylo}_{\mu F}\ g\ \text{out}_{\mu F} \end{array}$$

Point-free Language
Implicit Recursion

Recursion through the *hylomorphism* recursion pattern.
Allows the definition of any fixed point.

Defined in *pointwise* as

$$\begin{array}{lcl} \text{hylo}_{\mu F} & : & (F\ B \to B) \to (A \to F\ A) \to A \to B \\ \text{hylo}_{\mu F} & = & \lambda g.\ \lambda h.\ \text{fix}(\lambda f.\ g \circ Ff \circ h) \end{array}$$

Examples

$$\begin{array}{lcl} \text{length} & : & \text{List}\ A \to \text{Nat} \\ \text{length} & = & \text{hylo}_{\text{Nat}}\quad \text{in}_{\text{Nat}}\quad ((\text{id} + \text{snd}) \circ \text{out}_{\text{List}\ A}) \\ (\!|\cdot|\!)_{\mu F} & : & (F\ A \to A) \to \mu F \to A \\ (\!|\cdot|\!)_{\mu F} & = & \lambda g.\ \text{hylo}_{\mu F}\ \ g\ \ \text{out}_{\mu F} \end{array}$$

Outline

Some *Point-free* Laws

cata-FUSION

$$f \circ (\!|g|\!)_{\mu F} = (\!|h|\!)_{\mu F} \quad \Leftarrow \quad f \circ g = h \circ F\ f \quad \text{cata-FUSION}$$

More laws

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-FUNCTOR}$$

$$(f \triangledown g) \circ (h + i) = f \circ h \triangledown g \circ i \qquad +\text{-ABSOR}$$

$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \qquad +\text{-FUSION}$$

$$(f \times g) \circ swap = swap \circ (g \times f) \qquad \text{swap-NAT}$$

Some *Point-free* Laws

cata-FUSION

$$f \circ (\!|g|\!)_{\mu F} = (\!|h|\!)_{\mu F} \quad \Leftarrow \quad f \circ g = h \circ F\ f \qquad \text{cata-FUSION}$$

More laws

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-FUNCTOR}$$

$$(f \triangledown g) \circ (h + i) = f \circ h \triangledown g \circ i \qquad +\text{-ABSOR}$$

$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \qquad +\text{-FUSION}$$

$$(f \times g) \circ \mathit{swap} = \mathit{swap} \circ (g \times f) \qquad \text{swap-NAT}$$

Example: Reverse

Pointwise Definition

```
reverse [] = []
reverse (x:xs) = cat (reverse xs, wrap x)
```

$$reverse_t\ l\ y = \overline{cat}\ (reverse\ l)\ y$$

$$reverse_t = \overline{cat} \circ reverse$$

Example: Reverse

Pointwise Definition

```
reverse [] = []
reverse (x:xs) = cat (reverse xs, wrap x)
```

$$reverse_t\ l\ y = \overline{cat}\ (reverse\ l)\ y$$

$$reverse_t = \overline{cat} \circ reverse$$

Example: Reverse

Pointwise Definition

```
reverse [] = []
reverse (x:xs) = cat (reverse xs, wrap x)
```

$$reverse_t\ l\ y = \overline{cat}\ (reverse\ l)\ y$$

$$reverse_t = \overline{cat} \circ reverse$$

Example: Reverse

Cata-FUSION

$$reverse = (\!|\, nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id}))\,|\!)_{List\ A}$$

$$reverse_t = \overline{cat} \circ reverse$$

Cata-FUSION for lists

$$f \circ (\!|g|\!)_{\mathrm{List}\ A} = (\!|h|\!)_{\mathrm{List}\ A} \Leftarrow f \circ g = h \circ$$

$$reverse_t = (\!|h|\!)_{\mathrm{List}\ A}$$
$$\overline{cat} \circ nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id})) = h \circ (id + id \times \overline{cat})$$

Cat associativity

$$\overline{cat} \circ cat = comp \circ (\overline{cat} \times \overline{cat}) \qquad \text{cat-ASSOC}$$

Example: Reverse
Cata-FUSION

$$\begin{aligned} reverse &= (\!|\, nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id}))\,|\!)_{List\ A} \\ reverse_t &= \overline{cat} \circ reverse \end{aligned}$$

Cata-FUSION for lists

$$f \circ (\!|g|\!)_{\mathrm{List}\ A} = (\!|h|\!)_{\mathrm{List}\ A} \Leftarrow f \circ g = h \circ$$

$$\begin{aligned} reverse_t &= (\!|h|\!)_{\mathrm{List}\ A} \\ \overline{cat} \circ nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id})) &= h \circ (id + id \times \overline{cat}) \end{aligned}$$

Cat associativity

$$\overline{cat} \circ cat = comp \circ (\overline{cat} \times \overline{cat}) \qquad \text{cat-ASSOC}$$

Example: Reverse
Cata-FUSION

$$\begin{aligned} reverse &= (\!|\, nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id}))\,|\!)_{List\ A} \\ reverse_t &= \overline{cat} \circ reverse \end{aligned}$$

Cata-FUSION for lists

$$\begin{aligned} f \circ (\!|g|\!)_{\mathrm{List}\ A} &= (\!|h|\!)_{\mathrm{List}\ A} \\ &\Leftarrow f \circ g = h \circ F\ f \end{aligned}$$

$$\begin{aligned} reverse_t &= (\!|h|\!)_{\mathrm{List}\ A} \\ \overline{cat} \circ nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id})) &= h \circ (id + id \times \overline{cat}) \end{aligned}$$

Cat associativity

$$\overline{cat} \circ cat = comp \circ (\overline{cat} \times \overline{cat}) \qquad \text{cat-ASSOC}$$

Example: Reverse
Cata-FUSION

$$\begin{aligned} reverse &= (\!|\, nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id}))\,|\!)_{List\ A} \\ reverse_t &= \overline{cat} \circ reverse \end{aligned}$$

Cata-FUSION for lists

$$\begin{aligned} f \circ (\!|g|\!)_{\mathrm{List}\ A} &= (\!|h|\!)_{\mathrm{List}\ A} \\ &\Leftarrow f \circ g = h \circ (\mathrm{id} + \mathrm{id} \times f) \end{aligned}$$

$$\begin{aligned} reverse_t &= (\!|h|\!)_{\mathrm{List}\ A} \\ \overline{cat} \circ nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id})) &= h \circ (id + id \times \overline{cat}) \end{aligned}$$

Cat associativity

$$\overline{cat} \circ cat = comp \circ (\overline{cat} \times \overline{cat}) \qquad \text{cat-ASSOC}$$

Example: Reverse
Cata-FUSION

$$\begin{aligned} reverse &= (\!|\, nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id}))\,|\!)_{List\ A} \\ reverse_t &= \overline{cat} \circ reverse \end{aligned}$$

Cata-FUSION for lists

$$\begin{aligned} f \circ (\!|g|\!)_{\mathrm{List}\ A} &= (\!|h|\!)_{\mathrm{List}\ A} \\ &\Leftarrow f \circ g = h \circ (\mathrm{id} + \mathrm{id} \times f) \end{aligned}$$

$$\begin{aligned} reverse_t &= (\!|h|\!)_{\mathrm{List}\ A} \\ \overline{cat} \circ nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id})) &= h \circ (id + id \times \overline{cat}) \end{aligned}$$

Cat associativity

$$\overline{cat} \circ cat = comp \circ (\overline{cat} \times \overline{cat}) \qquad \text{cat-ASSOC}$$

Example: Reverse
Cata-FUSION

$$\begin{aligned} reverse &= (\!|\, nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id}))\,|\!)_{List\ A} \\ reverse_t &= \overline{cat} \circ reverse \end{aligned}$$

Cata-FUSION for lists

$$\begin{aligned} f \circ (\!|g|\!)_{\mathrm{List}\ A} &= (\!|h|\!)_{\mathrm{List}\ A} \\ &\Leftarrow f \circ g = h \circ (\mathrm{id} + \mathrm{id} \times f) \end{aligned}$$

$$\begin{aligned} reverse_t &= (\!|h|\!)_{\mathrm{List}\ A} \\ \overline{cat} \circ nil \triangledown (cat \circ swap \circ (wrap \times \mathrm{id})) &= h \circ (id + id \times \overline{cat}) \end{aligned}$$

Cat associativity

$$\overline{cat} \circ cat = comp \circ (\overline{cat} \times \overline{cat}) \qquad \text{cat-ASSOC}$$

Example: Reverse

h Calculation

$$h \circ (\mathrm{id} + \mathrm{id} \times \overline{cat}) =$$

$$
\begin{array}{ll}
 & \overline{cat} \circ (nil \triangledown cat \circ swap \circ (wrap \times \mathrm{id})) \\
= & \quad \{ +\text{-Fusion} \} \\
 & \overline{cat} \circ nil \triangledown \qquad\qquad \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{cat-Assoc} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ \qquad\qquad\qquad \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{swap-Nat} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ \\
= & \quad \{ \times\text{-Functor} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ \\
= & \quad \{ \times\text{-Functor} \} \\
 & \overline{cat} \circ nil \quad comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id}) \circ \\
= & \quad \{ +\text{-Absor} \} \\
 & (\overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id})) \circ \\
 & \qquad\qquad\qquad\qquad (id + \mathrm{id} \times \overline{cat})
\end{array}
$$

Example: Reverse

h Calculation

$$h \circ (\mathrm{id} + \mathrm{id} \times \overline{cat}) =$$

$$\begin{bmatrix}
& \overline{cat} \circ (nil \triangledown cat \circ swap \circ (wrap \times \mathrm{id})) \\
= & \quad \{ +\text{-FUSION} \} \\
& \overline{cat} \circ nil \triangledown \overline{cat} \circ cat \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{cat-ASSOC} \} \\
& \overline{cat} \circ nil \triangledown comp \circ \qquad\qquad \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{swap-NAT} \} \\
& \overline{cat} \circ nil \triangledown comp \circ swap \circ \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
& \overline{cat} \circ nil \triangledown comp \circ swap \circ \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
& \overline{cat} \circ nil \quad comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id}) \circ \\
= & \quad \{ +\text{-ABSOR} \} \\
& (\overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id})) \circ \\
& \qquad\qquad\qquad\qquad (id + \mathrm{id} \times \overline{cat})
\end{bmatrix}$$

Example: Reverse

h Calculation

$$h \circ (\mathrm{id} + \mathrm{id} \times \overline{cat}) =$$

$$
\begin{array}{ll}
 & \overline{cat} \circ (nil \triangledown cat \circ swap \circ (wrap \times \mathrm{id})) \\
= & \quad \{ +\text{-FUSION} \} \\
 & \overline{cat} \circ nil \triangledown \overline{cat} \circ cat \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{cat-ASSOC} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ (\overline{cat} \times \overline{cat}) \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{swap-NAT} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \quad comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id}) \circ \\
= & \quad \{ +\text{-ABSOR} \} \\
 & (\overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id})) \circ \\
 & \qquad (id + \mathrm{id} \times \overline{cat})
\end{array}
$$

Example: Reverse

h Calculation

$$h \circ (\mathrm{id} + \mathrm{id} \times \overline{cat}) =$$

$$\begin{bmatrix}
& \overline{cat} \circ (nil \triangledown cat \circ swap \circ (wrap \times \mathrm{id})) \\
= & \quad \{ +\text{-FUSION} \} \\
& \overline{cat} \circ nil \triangledown \overline{cat} \circ cat \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{cat-ASSOC} \} \\
& \overline{cat} \circ nil \triangledown comp \circ (\overline{cat} \times \overline{cat}) \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{swap-NAT} \} \\
& \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \times \overline{cat}) \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
& \overline{cat} \circ nil \triangledown comp \circ swap \circ \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
& \overline{cat} \circ nil \quad comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id}) \circ \\
= & \quad \{ +\text{-ABSOR} \} \\
& (\overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id})) \circ \\
& \qquad\qquad (id + \mathrm{id} \times \overline{cat})
\end{bmatrix}$$

Example: Reverse

h Calculation

$$h \circ (\mathrm{id} + \mathrm{id} \times \overline{cat}) =$$

$$\begin{bmatrix}
 & \overline{cat} \circ (nil \triangledown cat \circ swap \circ (wrap \times \mathrm{id})) \\
= & \quad \{ +\text{-FUSION} \} \\
 & \overline{cat} \circ nil \triangledown \overline{cat} \circ cat \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{cat-ASSOC} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ (\overline{cat} \times \overline{cat}) \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{swap-NAT} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \times \overline{cat}) \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \overline{cat}) \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \quad comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id}) \circ \\
= & \quad \{ +\text{-ABSOR} \} \\
 & (\overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id})) \circ \\
 & \qquad (id + \mathrm{id} \times \overline{cat})
\end{bmatrix}$$

Example: Reverse

h Calculation

$$h \circ (\mathrm{id} + \mathrm{id} \times \overline{cat}) =$$

$$
\begin{array}{ll}
 & \overline{cat} \circ (nil \triangledown cat \circ swap \circ (wrap \times \mathrm{id})) \\
= & \quad \{ +\text{-FUSION} \} \\
 & \overline{cat} \circ nil \triangledown \overline{cat} \circ cat \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{cat-ASSOC} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ (\overline{cat} \times \overline{cat}) \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{swap-NAT} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \times \overline{cat}) \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \overline{cat}) \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id}) \circ (\mathrm{id} \times \overline{cat}) \\
= & \quad \{ +\text{-ABSOR} \} \\
 & (\overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id})) \circ \\
 & \qquad (id + \mathrm{id} \times \overline{cat})
\end{array}
$$

Example: Reverse

h Calculation

$$h \circ (\mathrm{id} + \mathrm{id} \times \overline{cat}) =$$

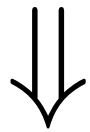
$$
\begin{array}{ll}
 & \overline{cat} \circ (nil \triangledown cat \circ swap \circ (wrap \times \mathrm{id})) \\
= & \quad \{ +\text{-FUSION} \} \\
 & \overline{cat} \circ nil \triangledown \overline{cat} \circ cat \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{cat-ASSOC} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ (\overline{cat} \times \overline{cat}) \circ swap \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \text{swap-NAT} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \times \overline{cat}) \circ (wrap \times \mathrm{id}) \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \overline{cat}) \\
= & \quad \{ \times\text{-FUNCTOR} \} \\
 & \overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id}) \circ (\mathrm{id} \times \overline{cat}) \\
= & \quad \{ +\text{-ABSOR} \} \\
 & (\overline{cat} \circ nil \triangledown comp \circ swap \circ (\overline{cat} \circ wrap \times \mathrm{id})) \circ \\
 & \qquad\qquad\qquad\qquad\qquad\qquad (id + \mathrm{id} \times \overline{cat})
\end{array}
$$

Example: Reverse

Pointwise results

Original function

```
reverse [] = []
reverse (x:xs) = cat (reverse xs, wrap x)
```



Resulting function

```
reverse_t [] y = y
reverse_t (x:xs) y = reverse_t xs (x:y)
```

Outline

Basic Combinators

Allows the definition of *point-free* code in *Haskell*.

Point-free constants and non-recursive combinators directly translated to *Haskell*.

Basic Combinators

Allows the definition of *point-free* code in *Haskell*.

Point-free constants and non-recursive combinators directly translated to *Haskell*.

Recursion
Functors

Based on generic programming library *PolyP*.
Data types viewed as fixed points of functors.

New type class with functional dependency

```
class (Functor f) => FunctorOf f d | d -> f
  where inn' :: f d -> d
        out' :: d -> f d
```

Regular functor combinators

```
newtype Id x          = Id {unId :: x}
newtype Const t x     = Const {unConst :: t}
data (g :+: h) x      = Inl (g x) | Inr (h x)
data (g :*: h) x      = g x :*: h x
newtype (g :@: h) x = Comp {unComp :: g (h x)}
```

Recursion
Functors

Based on generic programming library *PolyP*.
Data types viewed as fixed points of functors.

New type class with functional dependency

```
class (Functor f) => FunctorOf f d | d -> f
  where inn' :: f d -> d
        out' :: d -> f d
```

Regular functor combinators

```
newtype Id x          = Id {unId :: x}
newtype Const t x     = Const {unConst :: t}
data (g :+: h) x      = Inl (g x) | Inr (h x)
data (g :*: h) x      = g x :*: h x
newtype (g :@: h) x = Comp {unComp :: g (h x)}
```

Recursion
Functors

Based on generic programming library *PolyP*.
Data types viewed as fixed points of functors.

New type class with functional dependency

```
class (Functor f) => FunctorOf f d | d -> f
  where inn' :: f d -> d
        out' :: d -> f d
```

Regular functor combinators

```
newtype Id x          = Id {unId :: x}
newtype Const t x     = Const {unConst :: t}
data (g :+: h) x      = Inl (g x) | Inr (h x)
data (g :*: h) x      = g x :*: h x
newtype (g :@: h) x = Comp {unComp :: g (h x)}
```

Implicit Coercion
Functor types representation

To avoid using functor combinators in functions.

Elements defined using functor combinators
$\Rightarrow$ Standard *Haskell* types

Functor evaluation class

```
class Rep fa rep | fa -> rep
  where to :: fa -> rep
        from :: rep -> fa
```

Implicit Coercion
Functor types representation

To avoid using functor combinators in functions.

Elements defined using functor combinators
$\Rightarrow$ Standard *Haskell* types

Functor evaluation class

```
class Rep fa rep | fa -> rep
  where to :: fa -> rep
        from :: rep -> fa
```

Examples

Hylomorphism examples

```
fact :: Int -> Int
fact = hylo (_L :: [Int]) f g
  where g = (id -|- succ /\ id) . out
        f = one \/ mult


fold (_::d) g = hylo (_L::d) g out


length = fold (_L :: [a])
              (inn . (id -|- snd))
```


Outline

What is DrHylo

- Derives *point-free* definitions for a subset of *pointwise Haskell*;
- Resulting code can be executed using the Pointless library;
- 3 important steps:
  - Pattern matching removal;
  - Conversion of non-recursive definitions to *point-free*;
  - Explicit recursion removal.

Pattern Matching Removal

Towards *Haskell* code

- `FunctorOf` instances derived automatically;
- Constructors replaced by fixpoint definitions;
- Enough to pattern match over constructor `in`, sums, pairs and the constant $\star$;
- New constructor in the *pointwise* definition

$$\begin{array}{rcl} P & ::= & \star \mid x \mid \langle P, P \rangle \mid \mathsf{in}\ P \mid \mathsf{inl}\ P \mid \mathsf{inr}\ P \\ M, N & ::= & \ldots \mid \mathsf{match}\ M\ \mathsf{with}\ \{P \to N;\ \ldots; P \to N\} \end{array}$$

- Each *match* is converted to the core λ-calculus.

Pattern Matching Removal

Towards *Haskell* code

- `FunctorOf` instances derived automatically;
- Constructors replaced by fixpoint definitions;
- Enough to pattern match over constructor `in`, sums, pairs and the constant $\star$;
- New constructor in the *pointwise* definition

$$\begin{array}{rcl} P & ::= & \star \mid x \mid \langle P, P \rangle \mid \mathsf{in}\ P \mid \mathsf{inl}\ P \mid \mathsf{inr}\ P \\ M, N & ::= & \ldots \mid \mathsf{match}\ M\ \mathsf{with}\ \{P \to N;\ \ldots; P \to N\} \end{array}$$

- Each *match* is converted to the core λ-calculus.

Pointwise to *Point-free*
Basic Principles

- Based on the equivalence suggested by Lambek in 1980

  simply-typed λ-calculus $\Leftrightarrow$ Cartesian Closed Categories

- Translation inspired the *de Bruijn notation* for λ-calculus
- Typing context as left-nested pairs – $\Gamma ::= \star \mid \langle \Gamma, x : A \rangle$
- Each variable is replaced by the path to its position in the context.

$$\text{path}(\langle c, y \rangle, x) = \begin{cases} \text{snd} & \text{if } x = y \\ \text{path}(c, x) \circ \text{fst} & \text{otherwise} \end{cases}$$

Translation Rules

$$
\begin{array}{lcl}
\Phi(\Gamma \vdash \star) & = & \mathsf{bang} \\
\Phi(\Gamma \vdash x) & = & \mathsf{path}(\Gamma, x) \\
\Phi(\Gamma \vdash M\ N) & = & \mathsf{ap} \circ (\Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N)) \\
\Phi(\Gamma \vdash \lambda x.M) & = & \overline{\Phi(\langle \Gamma, x \rangle \vdash M)} \\
\Phi(\Gamma \vdash \langle M, N \rangle) & = & \Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N) \\
\Phi(\Gamma \vdash \mathsf{fst}\ M) & = & \mathsf{fst} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{snd}\ M) & = & \mathsf{snd} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{inl}\ M) & = & \mathsf{inl} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{inr}\ M) & = & \mathsf{inr} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{case}\ L\ M\ N) & = & \mathsf{ap} \circ (\mathsf{either} \circ (\Phi(\Gamma \vdash M) \\
 & & \qquad \vartriangle \Phi(\Gamma \vdash N)) \vartriangle \Phi(\Gamma \vdash L)) \\
\Phi(\Gamma \vdash \mathsf{in}\ M) & = & \mathsf{in} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{out}\ M) & = & \mathsf{out} \circ \Phi(\Gamma \vdash M)
\end{array}
$$

Final Details

Unpoint

- Translation of $M : A \to B$
  $N : 1 \to (A \to B)$
- Conversion to expected type $A \to B$
  $\mathsf{ap} \circ (\Phi(\star \vdash M) \circ \mathsf{bang} \vartriangle \mathsf{id})$

Translation of swap

$\mathsf{ap} \circ (\overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} \circ \mathsf{bang} \vartriangle \mathsf{id})$

Final Details

Unpoint

- Translation of $M : A \to B$
  $N : 1 \to (A \to B)$
- Conversion to expected type $A \to B$
  $\mathsf{ap} \circ (\Phi(\star \vdash M) \circ \mathsf{bang} \vartriangle \mathsf{id})$

Translation of swap

$\mathsf{ap} \circ (\overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} \circ \mathsf{bang} \vartriangle \mathsf{id})$

Translating Recursive Definitions
Two Methods

1st Method

- fix f viewed as a stream of functions f (f (f (f ...)))
- Stream $A = \mu(\underline{A} \otimes \mathrm{Id})$
- Hylomorphism that builds tree with in $(f, \mathrm{in}\ (f, ...))$ and replaces in by ap.

$$\begin{array}{lcl} \mathrm{fix} & : & (A \to A) \to A \\ \mathrm{fix} & = & \mathrm{hylo}_{\mathrm{Stream}\ (A \to A)}\ \mathrm{ap}\ (\mathrm{id} \vartriangle \mathrm{id}) \end{array}$$

$$\Phi(\Gamma \vdash \mathrm{fix}\ M) = \mathrm{fix} \circ \Phi(\Gamma \vdash M)$$

Translating Recursive Definitions
Two Methods

1st Method

- fix f viewed as a stream of functions f (f (f (f ...)))
- Stream $A = \mu(\underline{A} \otimes \mathrm{Id})$
- Hylomorphism that builds tree with in $(f, \mathrm{in}\ (f, ...))$ and replaces in by ap.

$$\begin{array}{lcl} \mathrm{fix} & : & (A \to A) \to A \\ \mathrm{fix} & = & \mathrm{hylo}_{\mathrm{Stream}\ (A \to A)}\ \mathrm{ap}\ (\mathrm{id} \vartriangle \mathrm{id}) \end{array}$$

$$\Phi(\Gamma \vdash \mathrm{fix}\ M) = \mathrm{fix} \circ \Phi(\Gamma \vdash M)$$

Translating Recursive Definitions
Two Methods

2^{nd} Method

- Algorithm that derives *pointwise hylomorphisms* from explicitly recursive definitions.
- Intermediate data structure models its recursion tree.
- Proposed by Hu, Iwasaki, and Takeichi, in 1996.

Translating Recursive Definitions
Two Methods

2^{nd} Method

- Algorithm that derives *pointwise hylomorphisms* from explicitly recursive definitions.
- Intermediate data structure models its recursion tree.
- Proposed by Hu, Iwasaki, and Takeichi, in 1996.

Example

length

```
len :: [a] -> Nat
len []     = Zero
len (h:t) = Succ (len t)
```

$$
\begin{array}{lcl}
\mathsf{length} & : & \mathsf{List}\ A \rightarrow \mathsf{Nat} \\
\mathsf{length} & = & \mathsf{hylo}_{\mu(\underline{1}\oplus \mathsf{Id})} \\
 & & \quad (\lambda x.\ \mathsf{case}\ x\ (\lambda y.\mathsf{in}\ (\mathsf{inl}\ \star))\ (\lambda y.\ \mathsf{in}\ (\mathsf{inr}\ y))) \\
 & & \quad (\lambda x.\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{inl}\ \star)\ (\lambda y.\ \mathsf{inr}\ (\mathsf{snd}\ y)))
\end{array}
$$

Outline

4. Pointless Haskell
   - Basic Combinators
   - Recursion
5. DrHylo
   - Pattern Matching
   - *Pointwise* to *Point-free*
   - Recursion
6. SimpliFree
   - Basic Principles
   - Rule Construction
   - Strategies
7. Conclusions

Main Topics

- *Active source* – Manipulated code with special commented blocks;
- Rules and strategies can be defined;
- Implemented by generic traversals – with *Strafunski*;
- Uses *Haskell* pattern matching;
- Uses parameterized macros.

Tool Interacion

Input

- *Point-free* functions;
- Special annotated blocks
  - Import strategy pack;
  - Rules;
  - Strategies;
  - Optimisations (functions $\rightharpoonup$ strategies).

Output

- Strategies $\rightarrow$ special combinators;
- Rules $\rightarrow$ functions with type `Term -> Maybe Term`;
- Each optimized function $\rightarrow$ Computation;
- `main` function $\rightarrow$ Transformed file.

Tool Interacion

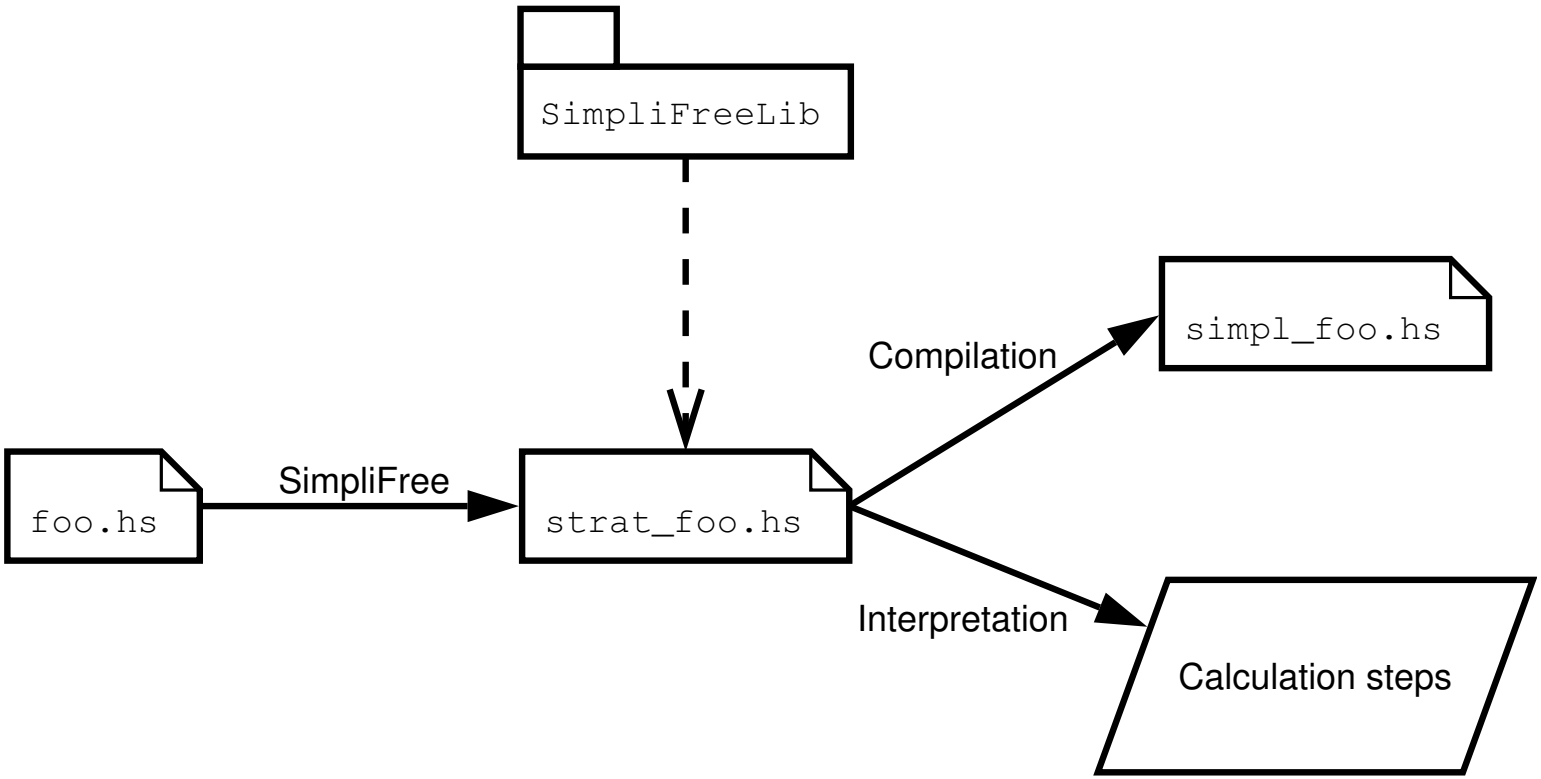
Input

- *Point-free* functions;
- Special annotated blocks
  - Import strategy pack;
  - Rules;
  - Strategies;
  - Optimisations (functions $\rightharpoonup$ strategies).

Output

- Strategies $\longrightarrow$ special combinators;
- Rules $\longrightarrow$ functions with type `Term -> Maybe Term`;
- Each optimized function $\longrightarrow$ Computation;
- `main` function $\longrightarrow$ Transformed file.

SimpliFree Architecture



Rule Construction

```
prodCancel : fst . (f /\ g) -> f
```



```
prodCancel (FST :.: (f :/\: g)) = return f
prodCancel (FST :.: ((f :/\: g) :.: x))
   = return (f :.: x)
prodCancel _
   = fail "rule prodCancel1 not applied"
```

Solved problems

- Variables on the left of a composition and correct association;
- Condition verification may require backtracking;
- Rules produced automatically

Rule Construction

```
prodCancel : fst . (f /\ g) -> f
```

```
prodCancel (FST :.: (f :/\: g)) = return f
prodCancel (FST :.: ((f :/\: g) :.: x))
   = return (f :.: x)
prodCancel _
   = fail "rule prodCancel1 not applied"
```

Solved problems

- Variables on the left of a composition and correct association;
- Condition verification may require backtracking;
- Rules produced automatically

Rule Construction

```
prodCancel : fst . (f /\ g) -> f
```

```
prodCancel (FST :.: (f :/\: g)) = return f
prodCancel (FST :.: ((f :/\: g) :.: x))
   = return (f :.: x)
prodCancel _
   = fail "rule prodCancel1 not applied"
```

Solved problems

- Variables on the left of a composition and correct association;
- Condition verification may require backtracking;
- Rules produced automatically

Defined Strategies

Rules repository with some rules and strategies.
Most depend on a base strategy.

Base Strategy

1. Apply simplification rules while possible;
2. If there are known macros, unfold them and return to (1);
3. Fold known macros.

Addition of new simplification strategies and macros.

Defined Strategies

Rules repository with some rules and strategies.
Most depend on a base strategy.

Base Strategy

1. Apply simplification rules while possible;
2. If there are known macros, unfold them and return to (1);
3. Fold known macros.

Addition of new simplification strategies and macros.

Examples
Swap

```
*Main> swap_adv_strat
app.((curry ((snd.snd) /\ (fst.snd)).bang)
                                        /\ id)
  = { expCancAdv3 }
((snd.snd) /\ (fst.snd)).(bang /\ id)
  = { prodFus }
(snd.snd.(bang /\ id)) /\ (fst.snd.(bang /\ id))
  = { prodCancel2 }
(snd.id) /\ (fst.snd.(bang /\ id))
  = { natId2 }
snd /\ (fst.snd.(bang /\ id))
  = { prodCancel2 }
snd /\ (fst.id)
  = { natId2 }
snd /\ fst
  = { swap_fold }
'swap'
```

Examples
cata-Fusion

```
*Main> reverse_t_cataList_strat
curry 'cat'.('cataList' [('pnt' ['nil']) \/
                ('cat'.'swap'.('wrap' >< id))])
= { cataList
    --- and ---
    [curry 'cat',curry 'cat'.'cat'.'swap'.('wrap' >< id)]
      = { catAssoc }
    [curry 'cat','comp'.(curry 'cat' >< curry 'cat').'swap'.
                                                ('wrap' >< id)]
      = { swapLeft }
    [curry 'cat','comp'.'swap'.(curry 'cat' >< curry 'cat').
                                                ('wrap' >< id)]
      = { prodFun }
    [curry 'cat','comp'.'swap'.((curry 'cat'.'wrap') ><
                                          (curry 'cat'.id))]
      = { natId2 }
    [curry 'cat','comp'.'swap'.((curry 'cat'.'wrap') ><
                                            curry 'cat')]
      = { extractH2B }
    'comp'.'swap'.((curry 'cat'.'wrap') >< id)
  }
'cataList' [(curry 'cat'.('pnt' ['nil'])) \/
       ('comp'.'swap'.((curry 'cat'.'wrap') >< id))]
```

Outline

Conclusions and Future Work

- **Pointless** – reached a stable stage;
- ***DrHylo*** – translate to other recursion patterns
- **SimpliFree**
  – incorporate other laws for recursive patterns
  – make the fusion mechanism more powerful
  – make laws more generic, like the fold fusion

- More components of the framework can be found in the UMinho Haskell Software pages.

  `wiki.di.uminho.pt/wiki/bin/view/PURe`

Conclusions and Future Work

- **Pointless** – reached a stable stage;
- ***DrHylo*** – translate to other recursion patterns
- **SimpliFree**
  – incorporate other laws for recursive patterns
  – make the fusion mechanism more powerful
  – make laws more generic, like the fold fusion

- More components of the framework can be found in the UMinho Haskell Software pages.

  `wiki.di.uminho.pt/wiki/bin/view/PURe`

Conclusions and Future Work

- **Pointless** – reached a stable stage;
- ***DrHylo*** – translate to other recursion patterns
- **SimpliFree**
  – incorporate other laws for recursive patterns
  – make the fusion mechanism more powerful
  – make laws more generic, like the fold fusion

- More components of the framework can be found in the UMinho Haskell Software pages.

  `wiki.di.uminho.pt/wiki/bin/view/PURe`

Conclusions and Future Work

- **Pointless** – reached a stable stage;
- ***DrHylo*** – translate to other recursion patterns
- **SimpliFree**
  – incorporate other laws for recursive patterns
  – make the fusion mechanism more powerful
  – make laws more generic, like the fold fusion

- More components of the framework can be found in the UMinho Haskell Software pages.

  `wiki.di.uminho.pt/wiki/bin/view/PURe`