SIMPLIFREE: Transforming Point-free Programs Using *Strafunski*

José Proença

Department of Informatics
University of Minho

PURe Workshop, 2005

Outline

Outline

The *Point-free* Style

Point-free language

- No variables are used
- Categorically-inspired combinators
- Algebraic data types as fixed points of functors
- Polytypic recursion patterns

Main Goal

Why *Point-free*

Easy to prove

reason equationaly

DrHylo

- Translation from *pointwise* to *point-free*
- Removal of explicit recursion
- Automated process

$\Rightarrow$ Complex *point-free* results

Reason automatically on *point-free* terms

Main Goal

Why *Point-free*

Easy to prove

reason equationaly

DrHylo

- Translation from *pointwise* to *point-free*
- Removal of explicit recursion
- Automated process

$\Rightarrow$ Complex *point-free* results

Reason automatically on *point-free* terms

Main Goal

Why *Point-free*

Easy to prove

reason equationaly

DrHylo

- Translation from *pointwise* to *point-free*
- Removal of explicit recursion
- Automated process

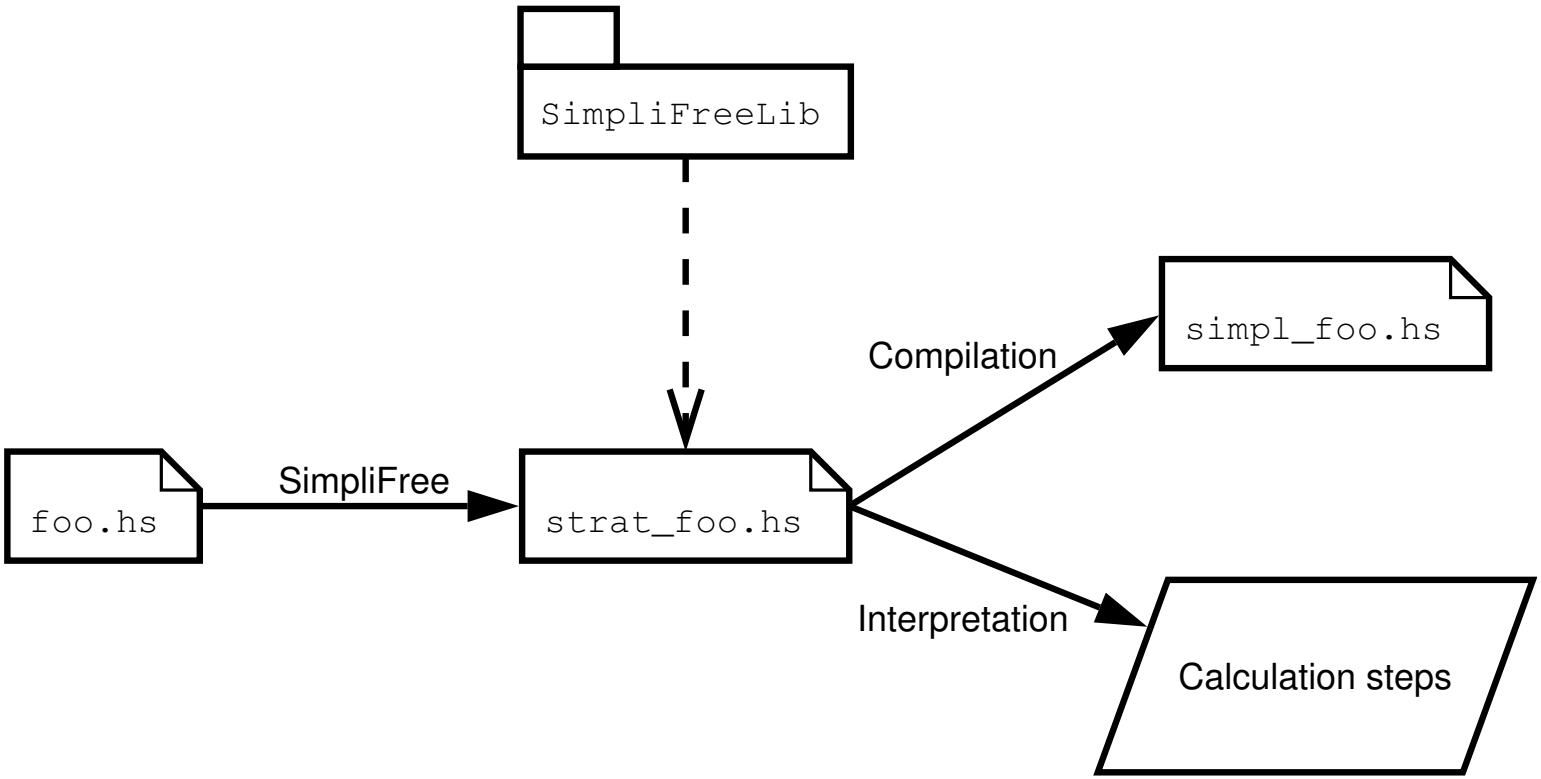
$\Rightarrow$ Complex *point-free* results

Reason automatically on *point-free* terms

General View
SIMPLIFREE

- Simplification and Transformation of *point-free* terms
- *Active source* – Code with special commented blocks
- Rules and strategies to transform terms
- Strategies implemented with generic traversals - *Strafunski*
- Uses *Haskell* patterns matching
- Visualisation of intermediate steps

SimpliFree Architecture



Point-Free Language

```
data Term = ID | Term :.: Term
  | FST | SND | Term :/\: Term
  | INL | INR | Term :\/: Term
  | AP | Curry Term
  | BANG
  | IN | OUT
  | Macro String [Term]
  | Hylo Type Term Term
    deriving Eq

data Type = ...
```

Outline

Basic Concepts

- *Strafunski* generic libraries were used;
- Instances for *Typeable* and *Term* derived with DrIFT;
- Several strategy combinators were defined:
  `rule`, `many`, `or`, `and`, `oneOrMore`, `optional` and `fail`.
- Associativity of composition:
  Terms are reassociated to the right after each transformation

Basic Concepts

- *Strafunski* generic libraries were used;
- Instances for *Typeable* and *Term* derived with DrIFT;
- Several strategy combinators were defined:
  `rule`, `many`, `or`, `and`, `oneOrMore`, `optional` and `fail`.
- Associativity of composition:
  Terms are reassociated to the right after each transformation

Basic Concepts

- *Strafunski* generic libraries were used;
- Instances for *Typeable* and *Term* derived with DrIFT;
- Several strategy combinators were defined:
  `rule`, `many`, `or`, `and`, `oneOrMore`, `optional` and `fail`.
- Associativity of composition:
  Terms are reassociated to the right after each transformation

Computation

Application of a strategy to a *point-free* term:

COMPUTATION

Computation definition

$$Computation = Result\ Steps$$
$$Step = (Term, String)$$

Normalization of the Composition

Reassociation to the right.

```
normalizeStrat :: MonadPlus m => TP m
normalizeStrat = iterateTP strat
  where
    strat = once_tdTP (adhocTP failTP flat)
    flat ((x :.: y) :.: z)
         = return $ x :.: (y :.: z)
    flat _ = fail "no need to flat"
    iterateTP :: MonadPlus m => TP m -> TP m
    iterateTP strat = (strat `seqTP`
                       (iterateTP strat))
                      `choiceTP` idTP
```

Normalization of the Composition

Reassociation to the right.

```
normalizeStrat :: MonadPlus m => TP m
normalizeStrat = iterateTP strat
  where
    strat = once_tdTP (adhocTP failTP flat)
    flat ((x :.: y) :.: z)
         = return $ x :.: (y :.: z)
    flat _ = fail "no need to flat"
    iterateTP :: MonadPlus m => TP m -> TP m
    iterateTP strat = (strat `seqTP`
                       (iterateTP strat))
                      `choiceTP` idTP
```

Normalization of the Composition

Reassociation to the right.

```
normalizeStrat :: MonadPlus m => TP m
normalizeStrat = iterateTP strat
  where
    strat = once_tdTP (adhocTP failTP flat)
    flat ((x :.: y) :.: z)
         = return $ x :.: (y :.: z)
    flat _ = fail "no need to flat"
    iterateTP :: MonadPlus m => TP m -> TP m
    iterateTP strat = (strat `seqTP`
                       (iterateTP strat))
                      `choiceTP` idTP
```


Normalization of the Composition

Reassociation to the right.

```
normalizeStrat :: MonadPlus m => TP m
normalizeStrat = iterateTP strat
  where
    strat = once_tdTP (adhocTP failTP flat)
    flat ((x :.: y) :.: z)
         = return $ x :.: (y :.: z)
    flat _ = fail "no need to flat"
    iterateTP :: MonadPlus m => TP m -> TP m
    iterateTP strat = (strat `seqTP`
                       (iterateTP strat))
                      `choiceTP` idTP
```


Apply a Rule Once

```
rulePF :: (MonadPlus m) =>
  String -> (Term -> m Term)
         -> TU Computation m

rulePF name rule =
    ... collect original term
    (once_tdTP (adhocTP failTP rule)
    `seqTP` normalizestrat
    ... get new term, and create new Computation
```


Apply a Rule Once

```
rulePF :: (MonadPlus m) =>
  String -> (Term -> m Term)
         -> TU Computation m

rulePF name rule =
     ... collect original term
     (once_tdTP (adhocTP failTP rule)
     `seqTP` normalizestrat
     ... get new term, and create new Computation
```

More Complete Justifications

In final version:

- Rules can apply strategies in the right-hand side;
- Justifications are now a string and a list of Computations;
- Redefinition of `Computation`

$$Computation = Result\ Steps$$
$$Step = (Term, Computations, String)$$

- Strategy combinators a bit more complex.

More Complete Justifications

In final version:

- Rules can apply strategies in the right-hand side;
- Justifications are now a string and a list of Computations;
- Redefinition of `Computation`

$$\mathit{Computation} = \mathit{Result}\ \mathit{Steps}$$
$$\mathit{Step} = (\mathit{Term}, \mathit{Computations}, \mathit{String})$$

- Strategy combinators a bit more complex.

More Complete Justifications

In final version:

- Rules can apply strategies in the right-hand side;
- Justifications are now a string and a list of Computations;
- Redefinition of `Computation`

$$\mathit{Computation} = \mathit{Result}\ \mathit{Steps}$$
$$\mathit{Step} = (\mathit{Term}, \mathit{Computations}, \mathit{String})$$

- Strategy combinators a bit more complex.

Examples

```
{- Strategies:
strategy : compute and fold_macros
compute : simplify and (opt
            ((oneOrMore unfold_macros)
             and compute))

simplify : many base_rules
base_rules : rule1 or rule2 or rule3 or ...
fold_macros   : many ( fold_macro1 or
                       fold_macro2 or ... )
unfold_macros : many (unfold_macro1 or
                      unfold_macro2 or ...)
-}
```

Examples

```
{- Strategies:
strategy : compute and fold_macros
compute : simplify and (opt
            ((oneOrMore unfold_macros)
             and compute))

simplify : many base_rules
base_rules : rule1 or rule2 or rule3 or ...
fold_macros   : many ( fold_macro1 or
                       fold_macro2 or ... )
unfold_macros : many (unfold_macro1 or
                      unfold_macro2 or ...)
-}
```

Examples

```
{- Strategies:
strategy : compute and fold_macros
compute : simplify and (opt
            ((oneOrMore unfold_macros)
             and compute))

simplify : many base_rules
base_rules : rule1 or rule2 or rule3 or ...
fold_macros    : many ( fold_macro1 or
                        fold_macro2 or ... )
unfold_macros : many (unfold_macro1 or
                      unfold_macro2 or ...)
-}
```

Examples

```
{- Strategies:
strategy : compute and fold_macros
compute : simplify and (opt
            ((oneOrMore unfold_macros)
             and compute))

simplify : many base_rules
base_rules : rule1 or rule2 or rule3 or ...
fold_macros    : many ( fold_macro1 or
                        fold_macro2 or ... )
unfold_macros : many (unfold_macro1 or
                      unfold_macro2 or ...)
-}
```

Outline

Basic Principles

- A rule is a function with type `Term -> m Term`;
- Tries to apply a transformation to a term or a prefix of a term;
- Assumes composition is *normalized*.

Example: $natId_1 : \mathrm{id} \circ f \longrightarrow f$

```
natId1 (ID :.: f) = return f
natId1 _          = fail "..."
```

Basic Principles

- A rule is a function with type `Term -> m Term`;
- Tries to apply a transformation to a term or a prefix of a term;
- Assumes composition is *normalized*.

Example: $natId_1 : \mathrm{id} \circ f \longrightarrow f$

```
natId1 (ID :.: f) = return f
natId1 _          = fail "..."
```

First Complication

Matching a prefix of a composition.

When the matching term is a composition, not ending on a variable.

Adition of a new match to the function.

Example: $sumCancel_1 : (f \triangledown g) \circ \text{inl} \longrightarrow f$

```
sumCancel1 ((f :: g) :.: INL)
    = return f
sumCancel1 ((f :: g) :.: (INL :.: x))
    = return (f :.: x)
sumCancel1 _
    = fail "..."
```

First Complication

Matching a prefix of a composition.

When the matching term is a composition, not ending on a variable.

Adition of a new match to the function.

Example: $sumCancel_1 : (f \triangledown g) \circ \text{inl} \longrightarrow f$

```
sumCancel1 ((f :: g) :.: INL)
    = return f
sumCancel1 ((f :: g) :.: (INL :.: x))
    = return (f :.: x)
sumCancel1 _
    = fail "..."
```

Left variables
The Problem

Oops: Variables on the left of compositions.

Why?

- Composition is associated to the right;
- Different associations until a pattern matching is found.

Example

$$\overline{f \circ \mathrm{ap}}$$

would not match any subterm of

$$\overline{\mathrm{fst} \circ (\mathrm{snd} \circ \mathrm{ap})}$$

Left variables
The Problem

Oops: Variables on the left of compositions.

Why?

- Composition is associated to the right;
- Different associations until a pattern matching is found.

Example

$$\overline{f \circ \mathsf{ap}}$$

would not match any subterm of

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

Left Variables
Solution

Replace compositions with *left variables* for new variables.

Use auxiliary functions for these substerms.
Still using *Haskell* pattern matching.

The auxiliary functions use an intermediate structure – `Maybe ([Term], Maybe Term)` – to:

- check if the pattern matching succeded;
- return the values of variables inside subterm;
- return the possible *ending*.

Left Variables
Solution

Replace compositions with *left variables* for new variables.

Use auxiliary functions for these substerms.
Still using *Haskell* pattern matching.

The auxiliary functions use an intermediate structure – `Maybe ([Term], Maybe Term)` – to:

- check if the pattern matching succeded;
- return the values of variables inside subterm;
- return the possible *ending*.

Left Variables
Solution

Replace compositions with *left variables* for new variables.

Use auxiliary functions for these substerms.
Still using *Haskell* pattern matching.

The auxiliary functions use an intermediate structure – `Maybe ([Term], Maybe Term)` – to:

- check if the pattern matching succeded;
- return the values of variables inside subterm;
- return the possible *ending*.

Left Variables
Solution

Replace compositions with *left variables* for new variables.

Use auxiliary functions for these substerms.
Still using *Haskell* pattern matching.

The auxiliary functions use an intermediate structure – `Maybe ([Term], Maybe Term)` – to:

- check if the pattern matching succeded;
- return the values of variables inside subterm;
- return the possible *ending*.

Left Variables
Solution

Replace compositions with *left variables* for new variables.

Use auxiliary functions for these substerms.
Still using *Haskell* pattern matching.

The auxiliary functions use an intermediate structure – `Maybe ([Term], Maybe Term)` – to:

- check if the pattern matching succeded;
- return the values of variables inside subterm;
- return the possible *ending*.

Left Variables
Example

$exp\_fold : \overline{f \circ \mathsf{ap}} \longrightarrow f^{\circ}$

```
exp_fold (Curry f') | success (aux f') =
    = return $
        (\f -> getEnd
                left_vars (Macro "exp" [f]))
        (getTerm 0 left_vars)
  where
   left_vars  = aux f'
   aux (f :.: AP) = addTerm f (emptyVar)
   aux (a :.: b) | success (aux b)
     = addComp a (aux b)
   aux _ = noVar

exp_fold _ = fail "rule not applied"
```

Left Variables
Example

$exp\_fold : \overline{f \circ ap} \longrightarrow f^{\circ}$

```
exp_fold (Curry f') | success (aux f') =
    = return $
        (\f -> getEnd
                left_vars (Macro "exp" [f]))
        (getTerm 0 left_vars)
  where
   left_vars  = aux f'
   aux (f :.: AP) = addTerm f (emptyVar)
   aux (a :.: b) | success (aux b)
     = addComp a (aux b)
   aux _ = noVar

exp_fold _ = fail "rule not applied"
```

Left Variables
Example

$exp\_fold : \overline{f \circ \mathsf{ap}} \longrightarrow f^{\circ}$

```
exp_fold (Curry f') | success (aux f') =
    = return $
        (\f -> getEnd
                left_vars (Macro "exp" [f]))
        (getTerm 0 left_vars)
  where
   left_vars  = aux f'
   aux (f :.: AP) = addTerm f (emptyVar)
   aux (a :.: b) | success (aux b)
     = addComp a (aux b)
   aux _ = noVar

exp_fold _ = fail "rule not applied"
```

Left Variables
Example

$exp\_fold : \overline{f \circ ap} \longrightarrow f^{\circ}$

```
exp_fold (Curry f') | success (aux f') =
    = return $
        (\f -> getEnd
                 left_vars (Macro "exp" [f]))
        (getTerm 0 left_vars)
  where
   left_vars  = aux f'
   aux (f :.: AP) = addTerm f (emptyVar)
   aux (a :.: b) | success (aux b)
     = addComp a (aux b)
   aux _ = noVar

exp_fold _ = fail "rule not applied"
```

Left Variables
Example

$exp\_fold : \overline{f \circ \mathsf{ap}} \longrightarrow f^{\circ}$

```
exp_fold (Curry f') | success (aux f') =
    = return $
        (\f -> getEnd
                left_vars (Macro "exp" [f]))
        (getTerm 0 left_vars)
  where
   left_vars  = aux f'
   aux (f :.: AP) = addTerm f (emptyVar)
   aux (a :.: b) | success (aux b)
     = addComp a (aux b)
   aux _ = noVar

exp_fold _ = fail "rule not applied"
```

Conditions
The Problem

Conditions can be introduced by:

- A string starting by “<=”;
- Equal variables;
- Strategies in the right-hand side.

Not enough to add conditions to guards:
Need to backtrack when conditions fail

Conditions
The Problem

Conditions can be introduced by:

- A string starting by “<=”;
- Equal variables;
- Strategies in the right-hand side.

Not enough to add conditions to guards:
Need to backtrack when conditions fail

Conditions
Solution – Not Regarding *Left Variables*

Introduction of a new match when composition ends in a variabe:

```
rule (... ::f ::x) | cond
     = return (exp ::x)
rule (... ::f ::(x1::x2))
     = rule (... ::(f ::x1)::x2)
rule (... ::f ) | cond
     = return exp
```


Conditions

Solution – In the presence of *Left Variables*

Auxiliary funcions collect all possible matches for each variable.

Intermediate structure of auxiliary functions change:
from `Maybe ([Term], Maybe Term)`
to `[ ([Term], Maybe Term) ]`

When a match is found in a composition* ending in a variable:
  Keep searching for more matches.

Use the first match that validates the conditions.

Conditions
Solution – In the presence of *Left Variables*

Auxiliary funcions collect all possible matches for each variable.

Intermediate structure of auxiliary functions change:
from `Maybe ([Term], Maybe Term)`
to `[ ([Term], Maybe Term) ]`

When a match is found in a composition* ending in a variable:
- Keep searching for more matches.

Use the first match that validates the conditions.

Conditions

Solution – In the presence of *Left Variables*

Auxiliary funcions collect all possible matches for each variable.

Intermediate structure of auxiliary functions change:
from `Maybe ([Term], Maybe Term)`
to `[ ([Term], Maybe Term) ]`

When a match is found in a composition* ending in a variable:

Keep searching for more matches.

Use the first match that validates the conditions.

Conditions

Solution – In the presence of *Left Variables*

Auxiliary funcions collect all possible matches for each variable.

Intermediate structure of auxiliary functions change:
from `Maybe ([Term], Maybe Term)`
to `[ ([Term], Maybe Term) ]`

When a match is found in a composition* ending in a variable:
Keep searching for more matches.

Use the first match that validates the conditions.

Syntactic Sugar
Making Life Easier

- Equal variables
- Associativity property:
  Given $\oplus$ produces $\overline{\oplus} \circ \oplus \longrightarrow comp \circ (\overline{\oplus} \times \overline{\oplus})$
- Fold and unfold macros, based on its definition.
- Introduction of lists of terms.

Syntactic Sugar
Making Life Easier

- Equal variables
- Associativity property:
  Given $\oplus$ produces $\overline{\oplus} \circ \oplus \longrightarrow comp \circ (\overline{\oplus} \times \overline{\oplus})$
- Fold and unfold macros, based on its definition.
- Introduction of lists of terms.

Syntactic Sugar
Making Life Easier

- Equal variables
- Associativity property:
  Given $\oplus$ produces $\overline{\oplus} \circ \oplus \longrightarrow comp \circ (\overline{\oplus} \times \overline{\oplus})$
- Fold and unfold macros, based on its definition.
- Introduction of lists of terms.

Syntactic Sugar
Making Life Easier

- Equal variables
- Associativity property:
  Given $\oplus$ produces $\overline{\oplus} \circ \oplus \longrightarrow comp \circ (\overline{\oplus} \times \overline{\oplus})$
- Fold and unfold macros, based on its definition.
- Introduction of lists of terms.

Outline

Simple Strategy
Input

Simple iteration of

$$\text{Prod-Cancel}_1 : \text{fst} \circ (f \vartriangle g) = f$$
$$\text{Prod-Cancel}_2 : \text{snd} \circ (f \vartriangle g) = g$$

Original Code

```
f = curry ((snd.(snd /\ fst)).(fst /\ fst))

{- Rules:
simplify: many (prodCancel1 or prodCancel2)
prodCancel1: fst.(f/\g) -> f
prodCancel2: snd.(f/\g) -> g -}

{- Optimizations: f -> simplify -}
```

Simple Strategy
Input

Simple iteration of

$$\text{Prod-Cancel}_1 : \text{fst} \circ (f \vartriangle g) = f$$
$$\text{Prod-Cancel}_2 : \text{snd} \circ (f \vartriangle g) = g$$

Original Code

```
f = curry ((snd.(snd /\ fst)).(fst /\ fst))

{- Rules:
simplify: many (prodCancel1 or prodCancel2)
prodCancel1: fst.(f/\g) -> f
prodCancel2: snd.(f/\g) -> g -}

{- Optimizations: f -> simplify -}
```

Simple Strategy
Output

```
import SimpliFreeLib
...
f = Curry (SND :.: ((SND :/\: FST) :.: (FST :/\: FST)))
...
prodCancel1  (FST :.: (f :/\: g)) = return (f)
prodCancel1  (FST :.: ((f :/\: g) :.: x))
                        = return (f :.: x)
prodCancel1  _ = fail "rule prodCancel1 not applied"
prodCancel2  (SND :.: (f :/\: g)) = return (g)
prodCancel2  (SND :.: ((f :/\: g) :.: x))
                        = return (g :.: x)
prodCancel2  _ = fail "rule prodCancel2 not applied"

simplify = manyPF ((rulePF "prodCancel1" prodCancel1)
             `orPF` (rulePF "prodCancel2" prodCancel2))

f_simplify = unOk (applyPF simplify f)
...
main = putStrLn ...
```

Simple Strategy
Results

Computation

```
*Main> f_simplify
curry (snd.(snd /\ fst).(fst /\ fst))
  = { prodCancel2 }
curry (fst.(fst /\ fst))
  = { prodCancel1 }
curry fst
```

The `main` function just return the final result.

Simple Strategy
Results

Computation

```
*Main> f_simplify
curry (snd.(snd /\ fst).(fst /\ fst))
  = { prodCancel2 }
curry (fst.(fst /\ fst))
  = { prodCancel1 }
curry fst
```

The `main` function just return the final result.

Cata-FUSION Law

Cata-FUSION

$$f \circ (\!|g|\!)_{\mu F} = (\!|h|\!)_{\mu F} \quad \Leftarrow \quad f \text{ strict} \wedge f \circ g = h \circ F\ f$$

cata-FUSION

For Lists

- $F_{\text{List } A} = \underline{1} \oplus \underline{A} \otimes \text{Id}$
- $g = g_1 \nabla g_2$
- $f \circ (g_1 \nabla g_2) = h \circ (id + id \times f)$

Cata-FUSION Law

Cata-FUSION

$$f \circ (\!|g|\!)_{\mu F} = (\!|h|\!)_{\mu F} \quad \Leftarrow \quad f \text{ strict} \wedge f \circ g = h \circ F\ f$$

cata-FUSION

For Lists

- $\mathsf{F}_{\mathsf{List}\ A} = \underline{1} \oplus \underline{A} \otimes \mathsf{Id}$
- $g = g_1 \triangledown g_2$
- $f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

Some Calculations

$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

$$
\begin{array}{ll}
 & f \circ (g_1 \triangledown g_2) \\
= & \quad \{ \text{ Sum-Fusion } \} \\
 & f \circ g_1 \triangledown f \circ g_2 \\
= & \quad \{ \dagger \} \\
 & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\
= & \quad \{ \text{ Natural Id, Prod-Functor } \} \\
 & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\
= & \quad \{ \text{ Sum-Absortion } \} \\
 & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f) \\
 & (h_1 \triangledown h_2) \circ (id + id \times f)
\end{array}
$$

Strategy for calculating h_2 based on $f \circ g_2$:
Keep trying to perform $\dagger$,
extract i, j and k to h_2 when possible

Some Calculations

$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

$$
\begin{array}{ll}
 & f \circ (g_1 \triangledown g_2) \\
= & \quad \{ \text{Sum-Fusion} \} \\
 & f \circ g_1 \triangledown f \circ g_2 \\
= & \quad \{ \dagger \} \\
 & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\
= & \quad \{ \text{ Natural Id, Prod-Functor} \} \\
 & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\
= & \quad \{ \text{ Sum-Absortion } \} \\
 & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f) \\
 & (h_1 \triangledown h_2) \circ (id + id \times f)
\end{array}
$$

Strategy for calculating h_2 based on $f \circ g_2$:

Keep trying to perform $\dagger$,
extract i, j and k to h_2 when possible

Some Calculations

$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

$$
\begin{array}{ll}
 & f \circ (g_1 \triangledown g_2) \\
= & \quad \{ \text{Sum-Fusion} \} \\
 & f \circ g_1 \triangledown f \circ g_2 \\
= & \quad \{ \dagger \} \\
 & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\
= & \quad \{ \text{ Natural Id, Prod-Functor} \} \\
 & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\
= & \quad \{ \text{ Sum-Absortion } \} \\
 & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f) \\
 & (h_1 \triangledown h_2) \circ (id + id \times f)
\end{array}
$$

Strategy for calculating h_2 based on $f \circ g_2$:
Keep trying to perform $\dagger$,
extract i, j and k to h_2 when possible

Some Calculations

$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

$$
\begin{array}{ll}
 & f \circ (g_1 \triangledown g_2) \\
= & \quad \{ \text{Sum-Fusion} \} \\
 & f \circ g_1 \triangledown f \circ g_2 \\
= & \quad \{ \dagger \} \\
 & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\
= & \quad \{ \text{ Natural Id, Prod-Functor} \} \\
 & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\
= & \quad \{ \text{ Sum-Absortion } \} \\
 & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f)
\end{array}
$$

$$(h_1 \triangledown h_2) \qquad \circ (id + id \times f)$$

Strategy for calculating h_2 based on $f \circ g_2$:
Keep trying to perform $\dagger$,
extract i, j and k to h_2 when possible

Some Calculations

$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

$$
\begin{array}{ll}
 & f \circ (g_1 \triangledown g_2) \\
= & \quad \{ \text{Sum-Fusion} \} \\
 & f \circ g_1 \triangledown f \circ g_2 \\
= & \quad \{ \dagger \} \\
 & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\
= & \quad \{ \text{ Natural Id, Prod-Functor} \} \\
 & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\
= & \quad \{ \text{ Sum-Absortion } \} \\
 & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f) \\
 & (h_1 \triangledown h_2) \qquad \circ (id + id \times f)
\end{array}
$$

Strategy for calculating h_2 based on $f \circ g_2$:
Keep trying to perform $\dagger$,
extract i, j and k to h_2 when possible

Some Calculations

$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

$$
\begin{array}{ll}
 & f \circ (g_1 \triangledown g_2) \\
= & \quad \{ \text{Sum-Fusion} \} \\
 & f \circ g_1 \triangledown f \circ g_2 \\
= & \quad \{ \dagger \} \\
 & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\
= & \quad \{ \text{ Natural Id, Prod-Functor} \} \\
 & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\
= & \quad \{ \text{ Sum-Absortion } \} \\
 & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f) \\
 & (h_1 \triangledown h_2) \qquad\qquad \circ (id + id \times f)
\end{array}
$$

Strategy for calculating h_2 based on $f \circ g_2$:
Keep trying to perform $\dagger$,
extract i, j and k to h_2 when possible

Some Calculations

$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$

$$
\begin{array}{ll}
 & f \circ (g_1 \triangledown g_2) \\
= & \quad \{ \text{Sum-Fusion} \} \\
 & f \circ g_1 \triangledown f \circ g_2 \\
= & \quad \{ \dagger \} \\
 & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\
= & \quad \{ \text{ Natural Id, Prod-Functor} \} \\
 & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\
= & \quad \{ \text{ Sum-Absortion } \} \\
 & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f) \\
 & (h_1 \triangledown h_2) \qquad\qquad \circ (id + id \times f)
\end{array}
$$

Strategy for calculating h_2 based on $f \circ g_2$:

Keep trying to perform $\dagger$,
extract i, j and k to h_2 when possible

SIMPLIFREE Strategy

```
cataList : cataList_rule
cataList_rule : f . ('cataList' [g1\/g2])
 -> 'cataList' [(f.g1) \/ (apply getH2 [f,f.g2])]

getH2 : extractH2 or (cataList_step and getH2)

extractH2 : extractH2A or extractH2B or
            extractH2C or extractH2D
extractH2A : [f,a.(b >< (c.f))] -> a.(b><c)
extractH2B : [f,a.(b >< f)] -> a.(b><id)
extractH2C : [f,b >< (c.f)] -> b><c
extractH2D : [f,b >< f] -> b><id
```

SIMPLIFREE Strategy

```
cataList : cataList_rule
cataList_rule : f . ('cataList' [g1\/g2])
 -> 'cataList' [(f.g1) \/ (apply getH2 [f,f.g2])]

getH2 : extractH2 or (cataList_step and getH2)

extractH2 : extractH2A or extractH2B or
            extractH2C or extractH2D
extractH2A : [f,a.(b >< (c.f))] -> a.(b><c)
extractH2B : [f,a.(b >< f)] -> a.(b><id)
extractH2C : [f,b >< (c.f)] -> b><c
extractH2D : [f,b >< f] -> b><id
```

SIMPLIFREE Strategy
Continuation

```
cataList_step :
    user_cataL_rules or swapLeft or
    base_rule or base_unfMacros

swapLeft :
    (f >< g) . 'swap' -> 'swap' . (g >< f)
```


SIMPLIFREE Strategy
Continuation

```
cataList_step :
    user_cataL_rules or swapLeft or
    base_rule or base_unfMacros

swapLeft :
    (f >< g) . 'swap' -> 'swap' . (g >< f)
```

SIMPLIFREE Strategy
Continuation

```
cataList_step :
    user_cataL_rules or swapLeft or
    base_rule or base_unfMacros

swapLeft :
    (f >< g) . 'swap' -> 'swap' . (g >< f)
```

Example: Reverse in *Point-free*
Cata-FUSION

Pointwise Haskell

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

Point-free

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

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

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

Example: Reverse in *Point-free*
Cata-FUSION

Pointwise Haskell

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

Point-free

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

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

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

Example: Reverse

SIMPLIFREE Input

```
reverse_t =
  = curry cat . (cataList
        ((pnt nil) \/ (cat.swap.(wrap >< id))))

{- Rules:
Assoc catAssoc: 'cat'
user_cataL_rules: catAssoc
-}

{- Optimizations: reverse_t -> cataList -}
```

Example: Reverse

SIMPLIFREE Input

```
reverse_t =
  = curry cat . (cataList
        ((pnt nil) \/ (cat.swap.(wrap >< id))))

{- Rules:
Assoc catAssoc: 'cat'
user_cataL_rules: catAssoc
-}

{- Optimizations: reverse_t -> cataList -}
```

Example: Reverse

SIMPLIFREE Input

```
reverse_t =
  = curry cat . (cataList
        ((pnt nil) \/ (cat.swap.(wrap >< id))))

{- Rules:
Assoc catAssoc: 'cat'
user_cataL_rules: catAssoc
-}

{- Optimizations: reverse_t -> cataList -}
```

Example: Reverse

SimpliFree Results

```
*Main> reverse_t_cataList
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

What was done:

- Automated simplification of *point-free* terms;
- Easy definition of strategies and rules;
- Program transformation – cata-FUSION

Future Work:

- Formal validation;
- Use of type information;
- Generalise cata-FUSION;
- Loop detection;
- Improve rules repository.

Conclusions and Future Work

What was done:

- Automated simplification of *point-free* terms;
- Easy definition of strategies and rules;
- Program transformation – cata-FUSION

Future Work:

- Formal validation;
- Use of type information;
- Generalise cata-FUSION;
- Loop detection;
- Improve rules repository.