Program Transformation: Functional Programming Perspective.

Carlos C. Martínez

Worcester Polytechnic Institute
Department of Computer Science

Wesleyan University
Department of Mathematics and Computer Science

cmartinez@wesleyan.edu

August 2003

Abstract

The purpose of this talk is to give a brief motivation of some program transformations in the context of Functional Programming.

Abstract

The purpose of this talk is to give a brief motivation of some program transformations in the context of Functional Programming.

Program transformation is used in a wide range of applications including compiler constructions, optimization, program synthesis, refactoring, software renovation, among others. Complex program transformation are achieved though a number of consecutive modifications of a program, Transformations rules define basic modifications. A transformation strategy is an algorithm for choosing a path in a rewrite relation induced by a set of rules.

Carlos C. Martínez

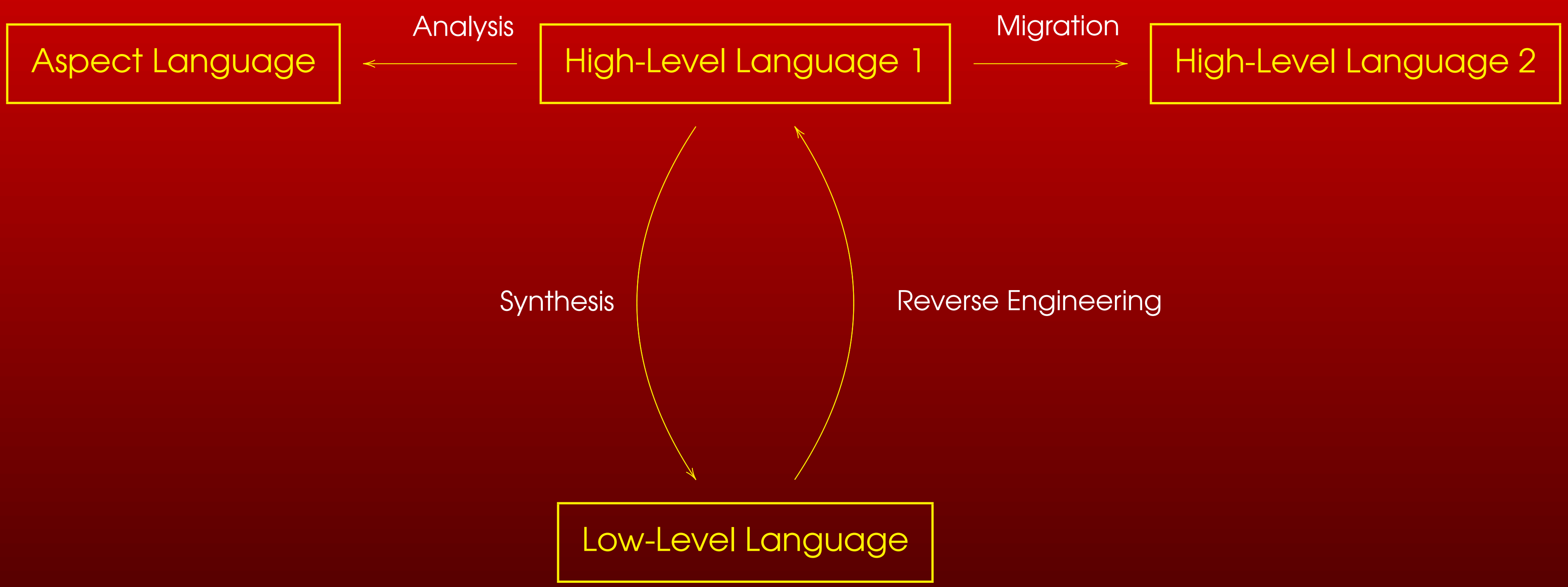
Program Transformation in a more general framework

Table 1: Taxonomy

Translation	Rephrasing
Migration	Normalization
Synthesis	◦ Simplification
◦ Refinement	◦ Desugaring
◦ Compilation	◦ Weaving
Reverse engineering	Optimization
◦ Decompilation	◦ Specialization
◦ Architecture extraction	◦ Inlining
◦ Documentation generation	◦ Fusion
◦ Software visualization	Refactoring
Analysis	◦ Design improvements
◦ Control-flow analysis	◦ Obfuscation
◦ Data-flow analysis	Renovation

Program Transformation in a more general framework

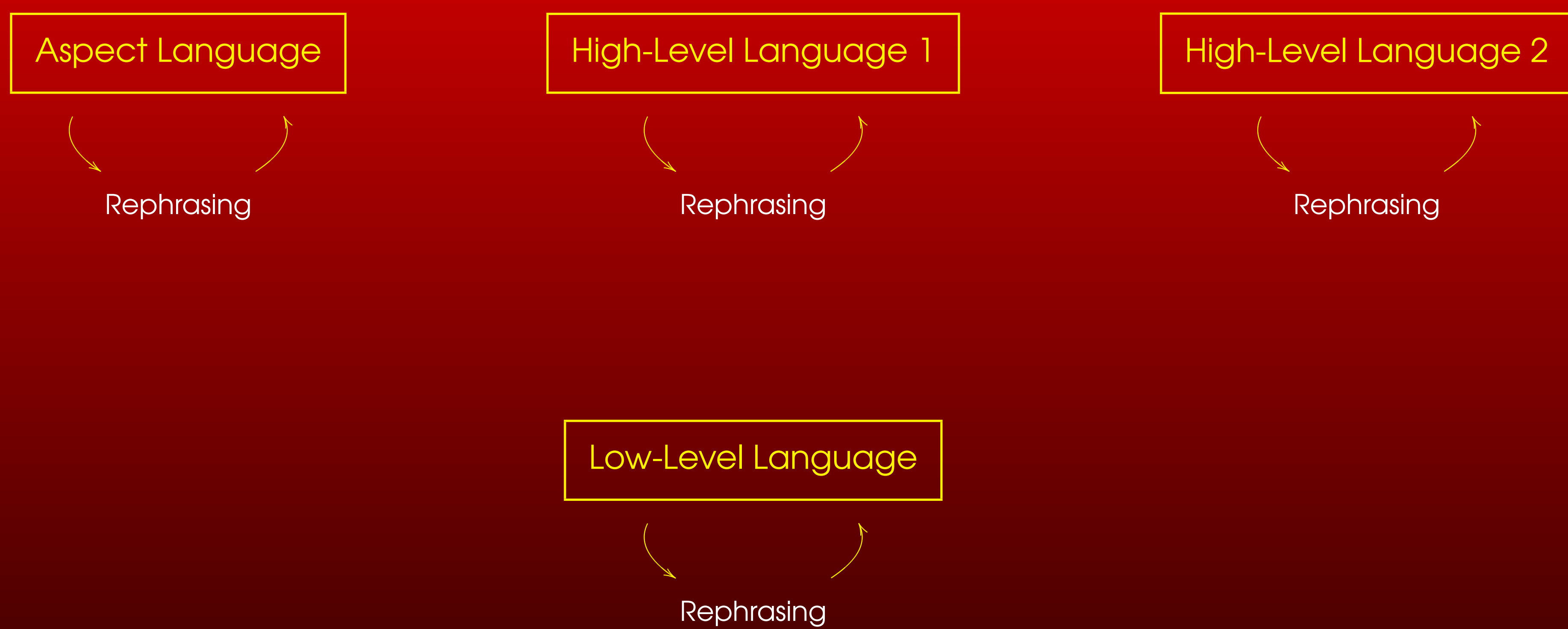
Table 2: Translation



Carlos C. Martínez

Program Transformation in a more general framework

Table 3: Rephrasing



Carlos C. Martínez

Motivation.

Gérard Huet and **Bernand Lang** (1978) *"Proving and Applying Program Transformations Expressed with Second-Order Patterns"*

Motivation.

Gérard Huet and **Bernand Lang** (1978) *"Proving and Applying Program Transformations Expressed with Second-Order Patterns"*

" There is a huge gap between software certification techniques and the theoretical tool defined for formal proofs of programs.."

Example 1.

Structural Recursion:

```
fact(x) = if (x==0) return 1 else return x * fact (x-1)
```

Example 1.

Structural Recursion:

```
fact(x) = if (x==0) return 1 else return x * fact (x-1)
```

Iteration:

```
fact(x) = if (x==0) return 1
      else {
        result = x;
        x = x-1;
        While ( x !== 0) do {
          result = result * x;
          x = x-1};
        return result * 1 }
```

Example 2.

Structural Recursion:

```
(rev x) = if (x== nil) return nil
     else return (append (rev (tail(x)), [head(x)]))
```

Example 2.

Structural Recursion:

```
(rev x) = if (x== nil) return nil
    else return (append (rev (tail(x)), [head(x)]))
```

Iteration:

```
(rev x) = if (x == nil) return nil
     else {
       result = [(head x)];
       x = (tail x);
       While ( x !== nil) do {
         result = (append [(head x)] result);
         x = (tail x)};
       return (append result nil) }
```

Abstracting.

Σ:

```
f(x) = if a(x) return b(x)
     else return h(d(x), f(c(x)))
```

Abstracting.

Σ:

```
f(x) = if a(x) return b(x)
     else return h(d(x), f(c(x)))
```

Σ':

```
f(x) = if a(x) return b(x)
      else {
        result = d(x);
        x = c(x);
        While ( not a(x)) do {
          result = h(result, d(x));
          x = c(x)};
        return h(result, b(x)) }
```

Instantiating patterns.

Σ:

```
f(x) = if a(x) return b(x)
     else return h(d(x), f(c(x)))
```

$$\sigma_1 \begin{cases} \mathtt{f}(\mathtt{x}) \leftarrow \mathtt{fact}(\mathtt{x}); \\ \mathtt{a}(\mathtt{x}) \leftarrow (\mathtt{x} == 0); \\ \mathtt{b}(\mathtt{x}) \leftarrow 1; \\ \mathtt{c}(\mathtt{x}) \leftarrow \mathtt{x} - 1; \\ \mathtt{d}(\mathtt{x}) \leftarrow \mathtt{x}; \\ \mathtt{h}(\mathtt{u}, \mathtt{v}) \leftarrow \mathtt{u} * \mathtt{v}; \end{cases}$$

Instantiating patterns.

Σ:

```
f(x) = if a(x) return b(x)
     else return h(d(x), f(c(x)))
```

$$\sigma_1 \left\{ \begin{array}{l} \mathtt{f}(\mathtt{x}) \leftarrow \mathtt{fact}(\mathtt{x}); \\ \mathtt{a}(\mathtt{x}) \leftarrow (\mathtt{x} == 0); \\ \mathtt{b}(\mathtt{x}) \leftarrow 1; \\ \mathtt{c}(\mathtt{x}) \leftarrow \mathtt{x} - 1; \\ \mathtt{d}(\mathtt{x}) \leftarrow \mathtt{x}; \\ \mathtt{h}(\mathtt{u}, \mathtt{v}) \leftarrow \mathtt{u} * \mathtt{v}; \end{array} \right.$$

```
fact(x) = if (x==0) return 1 else return x * fact (x-1)
```

Instantiating patterns.

Σ:

```
f(x) = if a(x) return b(x)
     else return h(d(x), f(c(x)))
```

$$\sigma_2 \begin{cases} \mathtt{f}(\mathtt{x}) \leftarrow (\mathtt{rev}\,\mathtt{x}); \\ \mathtt{a}(\mathtt{x}) \leftarrow (\mathtt{x} == \mathtt{nil}); \\ \mathtt{b}(\mathtt{x}) \leftarrow \mathtt{nil}; \\ \mathtt{c}(\mathtt{x}) \leftarrow (\mathtt{tail}\,\mathtt{x}); \\ \mathtt{d}(\mathtt{x}) \leftarrow [(\mathtt{head}\,\mathtt{x})]; \\ \mathtt{h}(\mathtt{u}, \mathtt{v}) \leftarrow (\mathtt{append}\,\mathtt{v}\,\mathtt{u}); \end{cases}$$

Instantiating patterns.

Σ:

```
f(x) = if a(x) return b(x)
     else return h(d(x), f(c(x)))
```

$$\sigma_2 \begin{cases} \mathtt{f}(\mathtt{x}) \leftarrow (\mathtt{rev\ x}); \\ \mathtt{a}(\mathtt{x}) \leftarrow (\mathtt{x} == \mathtt{nil}); \\ \mathtt{b}(\mathtt{x}) \leftarrow \mathtt{nil}; \\ \mathtt{c}(\mathtt{x}) \leftarrow (\mathtt{tail\ x}); \\ \mathtt{d}(\mathtt{x}) \leftarrow [(\mathtt{head\ x})]; \\ \mathtt{h}(\mathtt{u}, \mathtt{v}) \leftarrow (\mathtt{append\ v\ u}); \end{cases}$$

```
(rev x) = if (x == nil) return nil
     else return (append (rev (tail(x)), [(head x)]))
```

Transformation.

Example 2.

$$\sigma_2 \begin{cases} \mathtt{f}(\mathtt{x}) \leftarrow (\mathtt{rev}\ \mathtt{x}); \\ \mathtt{a}(\mathtt{x}) \leftarrow (\mathtt{x} == \mathtt{nil}); \\ \mathtt{b}(\mathtt{x}) \leftarrow \mathtt{nil}; \\ \mathtt{c}(\mathtt{x}) \leftarrow (\mathtt{tail}\ \mathtt{x}); \\ \mathtt{d}(\mathtt{x}) \leftarrow [(\mathtt{head}\ \mathtt{x})]; \\ \mathtt{h}(\mathtt{u}, \mathtt{v}) \leftarrow (\mathtt{append}\ \mathtt{v}\ \mathtt{u}); \end{cases}$$

Transformation.

$\sigma_2(\Sigma)$:

```
(rev x) = if a(x) return b(x)
    else return h(d(x), (rev c(x)))
```

$\sigma_2(\Sigma')$:

```
(rev x) = if a(x) return b(x)
     else {
       result = d(x);
       x = c(x);
       While ( not a(x)) do {
         result = h(result, d(x));
         x = c(x)};
       return h(result, b(x)) }
```

Transformation.

Example 2.

$$\sigma_2 \begin{cases} \mathtt{f}(\mathtt{x}) \leftarrow (\mathtt{rev}\;\mathtt{x}); \\ \mathtt{a}(\mathtt{x}) \leftarrow (\mathtt{x} == \mathtt{nil}); \\ \mathtt{b}(\mathtt{x}) \leftarrow \mathtt{nil}; \\ \mathtt{c}(\mathtt{x}) \leftarrow (\mathtt{tail}\;\mathtt{x}); \\ \mathtt{d}(\mathtt{x}) \leftarrow [(\mathtt{head}\;\mathtt{x})]; \\ \mathtt{h}(\mathtt{u},\mathtt{v}) \leftarrow (\mathtt{append}\;\mathtt{v}\;\mathtt{u}); \end{cases}$$

Transformation.

$\sigma_2(\Sigma)$:

```
(rev x) = if (x == nil) return nil
     else return h(d(x),(rev c(x)))
```

$\sigma_2(\Sigma')$:

```
(rev x) = if (x == nil) return nil
      else {
        result = d(x);
        x = c(x);
        While ( x !== nil) do {
          result = h(result, d(x));
          x = c(x)};
        return h(result, nil) }
```

Transformation.

Example 2

$$\sigma_2 \begin{cases} \mathtt{f}(\mathtt{x}) \leftarrow (\mathtt{rev\ x}); \\ \mathtt{a}(\mathtt{x}) \leftarrow (\mathtt{x} == \mathtt{nil}); \\ \mathtt{b}(\mathtt{x}) \leftarrow \mathtt{nil}; \\ \mathtt{c}(\mathtt{x}) \leftarrow (\mathtt{tail\ x}); \\ \mathtt{d}(\mathtt{x}) \leftarrow [(\mathtt{head\ x})]; \\ \mathtt{h}(\mathtt{u}, \mathtt{v}) \leftarrow (\mathtt{append\ v\ u}); \end{cases}$$

Transformation.

$\sigma_2(\Sigma)$:

```
(rev x) = if (x == nil) return nil
     else return (append (rev(tail x)) [(head x)])
```

$\sigma_2(\Sigma')$:

```
(rev x) = if (x == nil) return nil
      else {
        result = [(head x)];
        x =(tail x);
        While ( x !== nil) do {
          result = (append [(head x)] result );
          x =(tail x) };
        return (append nil result) }
```

Comments.

- Factorial uses the same pattern $(\Sigma, \Sigma^{'})$ and σ_1 as its matching.

Comments.

- Factorial uses the same pattern $(\Sigma, \Sigma^{'})$ and σ_1 as its matching.
- $(\Sigma, \Sigma^{'})$ transformation in order to be valid require the following constraints $\mathcal{X}$ on `h`.

1. associative
2. has unity, i.e. `h(x ,*) = h(* ,x) = x`

Comments.

- Factorial uses the same pattern $(\Sigma, \Sigma^{'})$ and σ_1 as its matching.
- $(\Sigma, \Sigma^{'})$ transformation in order to be valid require the following constraints $\mathcal{X}$ on `h`.
  1. associative
  2. has unity, i.e. `h(x ,*) = h(* ,x) = x`
- The Paradigm:
  1. recognize a pattern $(\Sigma, \Sigma^{'}, \mathcal{X})$
  2. validate such pattern.
  3. recognize whether or not a pattern is applicable to a given program.
  4. organize a system applying automatically those patterns.

Huet's Proposal.

Let $(\Sigma, \Sigma^{'}, \mathcal{X})$ be a transformation, and let $\mathcal{P}[t]$ be a program which contains t as subterm. We say the transformation is *applicable* in $\mathcal{P}[t]$ at t if and only if there exists a substitution σ for the free variable of Σ and $\Sigma^{'}$ such that:

Huet's Proposal.

Let $(\Sigma, \Sigma^{'}, \mathcal{X})$ be a transformation, and let $\mathcal{P}[t]$ be a program which contains t as subterm. We say the transformation is *applicable* in $\mathcal{P}[t]$ at t if and only if there exists a substitution σ for the free variable of Σ and $\Sigma^{'}$ such that:

- $t = \sigma(\Sigma)$ (matching)

Huet's Proposal.

Let $(\Sigma, \Sigma^{'}, \mathcal{X})$ be a transformation, and let $\mathcal{P}[t]$ be a program which contains t as subterm. We say the transformation is *applicable* in $\mathcal{P}[t]$ at t if and only if there exists a substitution σ for the free variable of Σ and $\Sigma^{'}$ such that:

- $t = \sigma(\Sigma)$ (matching)
- $\mathcal{M} \models \sigma(\mathcal{X})$, i.e. axioms $\mathcal{X}$ instantiated by σ are valid in the programming semantics $\mathcal{M}$

Huet's Proposal.

Let $(\Sigma, \Sigma', \mathcal{X})$ be a transformation, and let $\mathcal{P}[t]$ be a program which contains t as subterm. We say the transformation is *applicable* in $\mathcal{P}[t]$ at t if and only if there exists a substitution σ for the free variable of Σ and Σ' such that:

- $t = \sigma(\Sigma)$ (matching)
- $\mathcal{M} \models \sigma(\mathcal{X})$, i.e. axioms $\mathcal{X}$ instantiated by σ are valid in the programming semantics $\mathcal{M}$
- Replace in the program $\mathcal{P}$, t by $\sigma(\Sigma')$

Huet's Proposal.

Let $(\Sigma, \Sigma^{'}, \mathcal{X})$ be a transformation, and let $\mathcal{P}[t]$ be a program which contains t as subterm. We say the transformation is *applicable* in $\mathcal{P}[t]$ at t if and only if there exists a substitution σ for the free variable of Σ and $\Sigma^{'}$ such that:

- $t = \sigma(\Sigma)$ (matching)
- $\mathcal{M} \models \sigma(\mathcal{X})$, i.e. axioms $\mathcal{X}$ instantiated by σ are valid in the programming semantics $\mathcal{M}$
- Replace in the program $\mathcal{P}$, t by $\sigma(\Sigma^{'})$

A problem occurs when for a certain context two different transformations are available, leading to two different programs, this is a well known problem for *Rewriting Systems*. In some cases it will possible to prove that our set of transformations is *confluent*.

Glasgow Haskell [1] Compiler: Compilation by transformation.

Compiler' Structure:

- The front end parses the source, does scope analysis and type inference, and translates the program into a small intermediate language called the *Core Language*.

Glasgow Haskell [1] Compiler: Compilation by transformation.

Compiler' Structure:

- The front end parses the source, does scope analysis and type inference, and translates the program into a small intermediate language called the *Core Language*.
- The middle consists of sequences of Core to Core [2] transformations

Glasgow Haskell [1] Compiler: Compilation by transformation.

Compiler' Structure:

- The front end parses the source, does scope analysis and type inference, and translates the program into a small intermediate language called the *Core Language*.
- The middle consists of sequences of Core to Core [2] transformations
- The back end translates the resulting Core program into C, whence it is compiled to the machine code.

---

[1] Haskell is a non-strict, typed, pure functional language
[2] Most of the "optimizations" appears at this level

Glasgow Haskell Compiler: Compilation by transformation.

The Core Language:

Consists essentially of a Polymorphic typed lambda calculus augmented with `let`, `case` and algebraic type declarations, for example

Glasgow Haskell Compiler: Compilation by transformation.

The Core Language:

Consists essentially of a Polymorphic typed lambda calculus augmented with `let`, `case` and algebraic type declarations, for example

```
data Boolean = True | False
```

Glasgow Haskell Compiler: Compilation by transformation.

The Core Language:

Consists essentially of a Polymorphic typed lambda calculus augmented with `let`, `case` and algebraic type declarations, for example

```
data Boolean = True | False

data List a = Nil | Cons a (List a)
```

Glasgow Haskell Compiler: Compilation by transformation.

The Core Language:

Consists essentially of a Polymorphic typed lambda calculus augmented with `let`, `case` and algebraic type declarations, for example

```
data Boolean = True | False

data List a = Nil | Cons a (List a)

data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Glasgow Haskell Compiler: Compilation by transformation.

"Desugarer":

Glasgow Haskell Compiler: Compilation by transformation.

"Desugarer":

Haskell Code	$\Longrightarrow$	**Core Code**
`f (sin x ) y`		`let v = sin x in f v y`
`if C E1 E2`		`case C of {True -> E1; False -> E2}`

Glasgow Haskell Compiler: Compilation by transformation.

A Core to Core Optimization:
Deforestation/Fusion laws are program transformation by means of which intermediate data structures can be eliminated.

Glasgow Haskell Compiler: Compilation by transformation.

A Core to Core Optimization:
Deforestation/Fusion laws are program transformation by means of which intermediate data structures can be eliminated.

The interest in this particular technique is due to the fact that programs are often implemented in a compositional fashion, using intermediate data structures to connect such a components. This compositional style is suitable for modular programming, however may be inefficient both time and space.

Deforestation/fusion

From an example: "Sum of the square of n integers"

```
Version(1) Modular
```

```
SquareList ::  [Int] -> [Int]
SquareList [] = 0
SquareList (x :xs ) = [x^2] ++ SquareList(xs)
```

Deforestation/fusion

From an example: “Sum of the square of n integers”

```
Version(1) Modular

 SquareList ::  [Int] -> [Int]
 SquareList [] = 0
 SquareList (x :xs ) = [x^2] ++ SquareList(xs)

 Sum ::  [Int] -> Int
 Sum [] = 0
 Sum (x :xs) = x + Sum(xs)
```

Deforestation/fusion

From an example: "Sum of the square of n integers"

Version(1) Modular

```
SquareList ::  [Int] -> [Int]
SquareList [] = 0
SquareList (x :xs ) = [x^2] ++ SquareList(xs)

Sum ::  [Int] -> Int
Sum [] = 0
Sum (x :xs) = x + Sum(xs)

SumSqr ::  [Int] -> Int
SumSqr = Sum ∘ SquareList
```

Deforestation/fusion

Version(2) UnModular

```
SumSquare ::  [Int] -> Int
SumSquare [] = 0
SumSquare (x :xs) = x^2 + SumSquare(xs)
```

Deforestation/fusion

Version(2) UnModular

```
SumSquare ::  [Int] -> Int
SumSquare [] = 0
SumSquare (x :xs) = x^2 + SumSquare(xs)
```

A generalization?

Deforestation/fusion

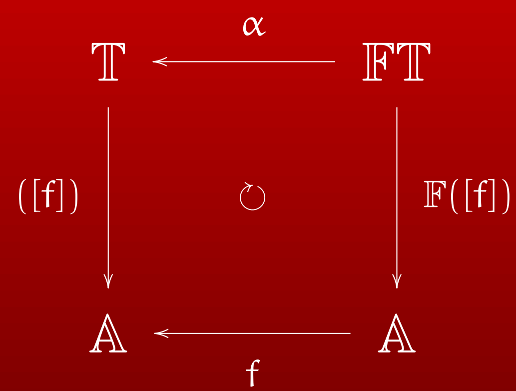
Version(2) UnModular

```
SumSquare ::  [Int] -> Int
SumSquare [] = 0
SumSquare (x :xs) = x^2 + SumSquare(xs)
```

A generalization? Where could we get it?

Deforestation/fusion

Answer: Category theory!!, datatypes [3] as initial algebra for a given functor.



[3] $\alpha : \mathbb{F} \leftarrow \mathbb{F}\mathbb{T}$, is called the *initial algebra* for $\mathbb{F}$. Exists a unique $(\![f]\!)$ called *catamorphism*

Deforestation/fusion

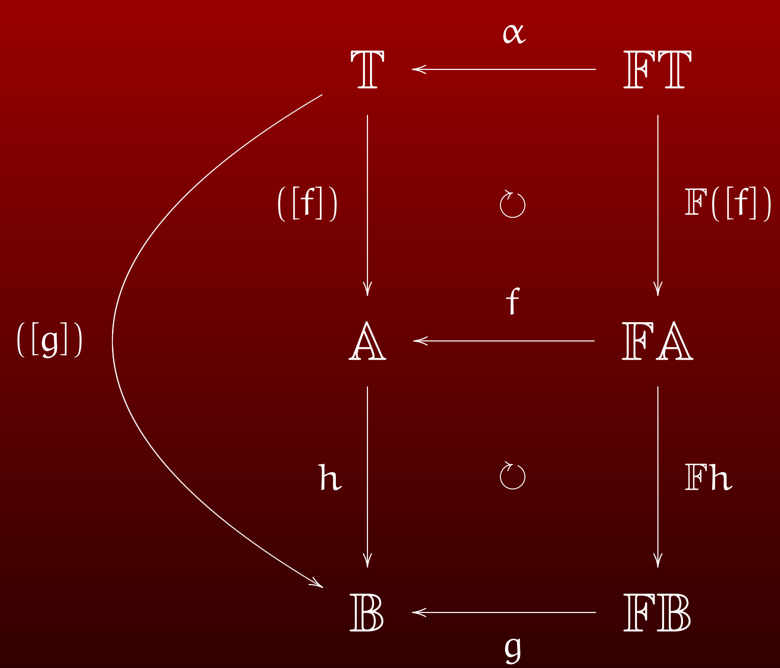
Fusion Law

- $(\![\alpha]\!) = \mathrm{id}$ and $h \circ (\![f]\!) = (\![g]\!) \Leftarrow h \circ f = g \circ \mathbb{F}h.$

Deforestation/fusion

Fusion Law

- $([\alpha]) = id$ and $h \circ ([f]) = ([g]) \Leftarrow h \circ f = g \circ \mathbb{F}h$.

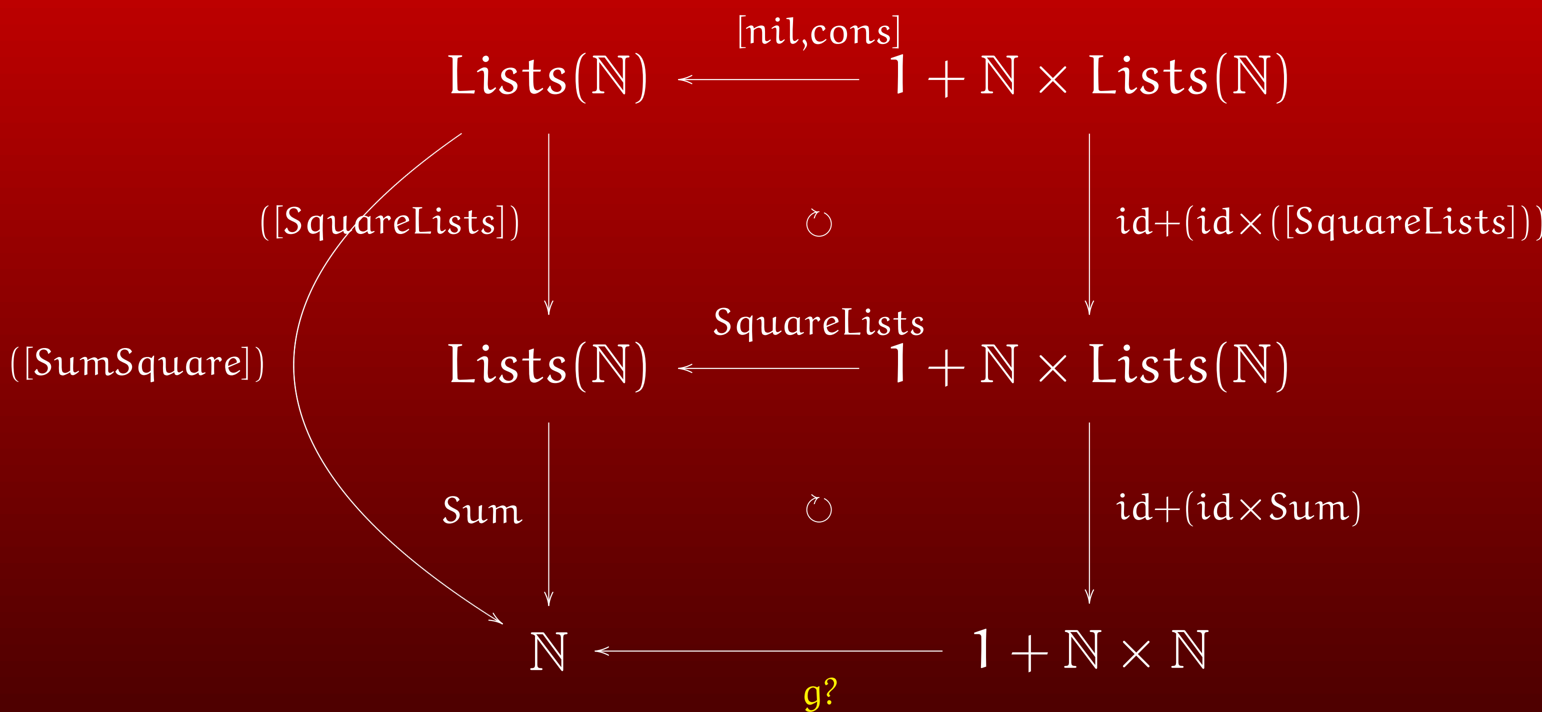


Deforestation/fusion

Back on our example: looking for g

Deforestation/fusion

Back on our example: looking for g

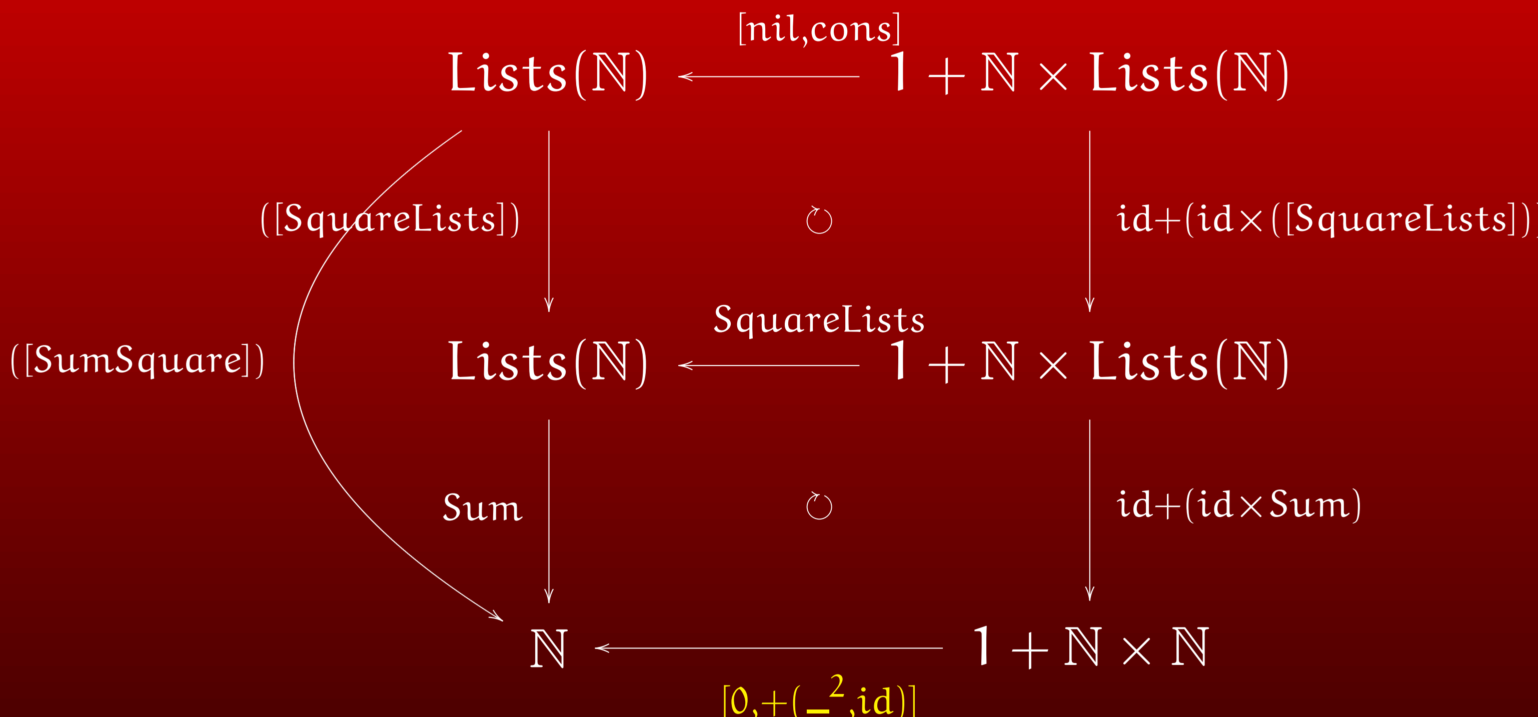


Where $\mathsf{SquareLists} = [\mathsf{nil}, \mathsf{cons}(\_^2, \mathsf{id})]$

Carlos C. Martínez

Deforestation/fusion

Back on our example



Where $\mathsf{SquareLists} = [\mathsf{nil}, \mathsf{cons}(\_^2, \mathsf{id})]$

Deforestation/fusion

The key: observe that we found an appropriate $g = [0, +(\_^2, id)]$ such that the diagram commute i.e. we answer the question.

Is there some g such that $h \circ f = g \circ \mathbb{F}h$.?

Deforestation/fusion

The key: observe that we found an appropriate $g = [0, +(\_^2, id)]$ such that the diagram commute i.e. we answer the question.

Is there some g such that $h \circ f = g \circ \mathbb{F}h$.?

Remark: this can be seen as a Matching problem!!

Deforestation/fusion

The key: observe that we found an appropriate $g = [0, +(\_^2, id)]$ such that the diagram commute i.e. we answer the question.

Is there some g such that $h \circ f = g \circ \mathbb{F}h$.?

Remark: this can be seen as a Matching problem!!

Ref: **O. de Moor** and **G. Sittampalam**(1999) *Generic Program Transformation* (1999)

Deforestation/fusion

From Example:

$$
\begin{array}{ccc}
\mathsf{Lists}(\mathbb{N}) & \xleftarrow{\mathsf{SquareLists}} & 1 + \mathbb{N} \times \mathsf{Lists}(\mathbb{N}) \\
\Big\downarrow \mathsf{Sum} & \circlearrowright & \Big\downarrow \mathsf{id}+(\mathsf{id}\times\mathsf{Sum}) \\
\mathbb{N} & \xleftarrow[g=[g_0,g_1]]{} & 1 + \mathbb{N} \times \mathbb{N}
\end{array}
$$

Deforestation/fusion

From Example:

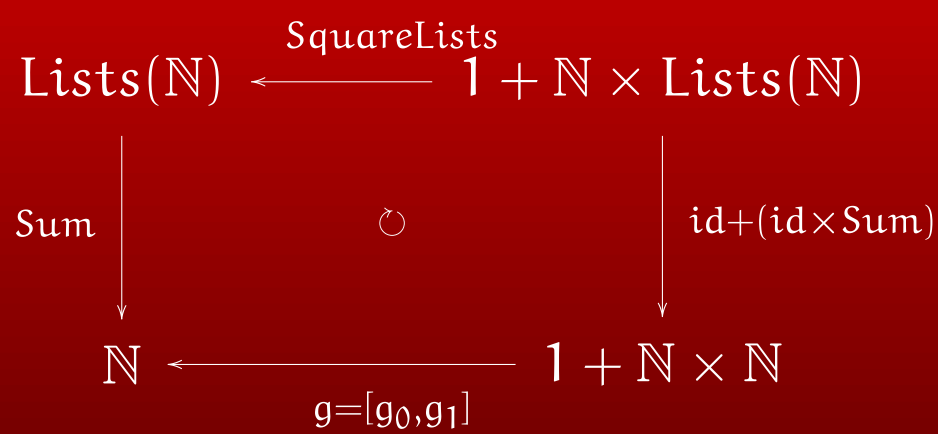
$$\begin{array}{ccc}
\mathsf{Lists}(\mathbb{N}) & \xleftarrow{\mathsf{SquareLists}} & 1 + \mathbb{N} \times \mathsf{Lists}(\mathbb{N}) \\
\Big\downarrow{\scriptstyle \mathsf{Sum}} & \circlearrowright & \Big\downarrow{\scriptstyle \mathsf{id}+(\mathsf{id}\times\mathsf{Sum})} \\
\mathbb{N} & \xleftarrow[g=[g_0,g_1]]{} & 1 + \mathbb{N} \times \mathbb{N}
\end{array}$$

Translated to:

$\mathtt{SquareLists} \Longrightarrow \mathtt{if\ (y\ =\ 1)\ nil\ cons(\ \_^{2},\ id)y}$

Deforestation/fusion

From Example:



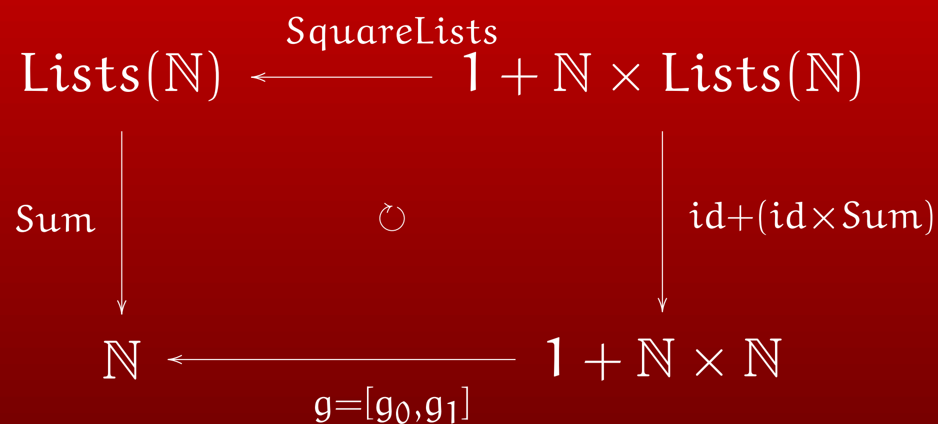
Translated to:

$\texttt{SquareLists} \Longrightarrow \texttt{if (y = 1) nil cons(}\_^{2}\texttt{, id)y}$

$\texttt{Sum} \Longrightarrow \texttt{if (y = nil) 0 +(id , Sum)y}$

Deforestation/fusion

From Example:



Translated to:

$\texttt{SquareLists} \Longrightarrow \texttt{if (y = 1) nil cons(} \_^2\texttt{, id)y}$

$\texttt{Sum} \Longrightarrow \texttt{if (y = nil) 0 +(id , Sum)y}$

$\texttt{g} \Longrightarrow \texttt{if (y = 1)}\ g_0\ g_1$

Carlos C. Martínez

Higher Order Matching and HOAS

From this it's not hard to see that those programs can be encoded as a lambda terms (HOAS) i.e. the fusion law can be expressed as:

$$(t_{Sum}\ \ t_{SquareLists}) =^{?} (t_g\quad [id, \langle id, t_{Sum}\rangle])$$

Higher Order Matching and HOAS

Results on Matching

Higher Order Matching and HOAS

Results on Matching

- **Gérard Huet** (1976) Higher Order Matching conjecture. "they are decidable"

Higher Order Matching and HOAS

Results on Matching

- **Gérard Huet** (1976) Higher Order Matching conjecture. "they are decidable"
- **Gérard Huet**, **B. Lang**(1978) found an algorithm that terminates in the case of 2^{nd}.O M.

Higher Order Matching and HOAS

Results on Matching

- **Gérard Huet** (1976) Higher Order Matching conjecture. "they are decidable"
- **Gérard Huet**, **B. Lang**(1978) found an algorithm that terminates in the case of 2^{nd}.O M.
- **Gilles Dowek**(1994): decidability of 3^{rd}.O.M.

Higher Order Matching and HOAS

Results on Matching

- **Gérard Huet** (1976) Higher Order Matching conjecture. "they are decidable"
- **Gérard Huet**, **B. Lang**(1978) found an algorithm that terminates in the case of 2^{nd}.O M.
- **Gilles Dowek**(1994): decidability of 3^{rd}.O.M.
- **V. Padovani**(1995): decidability of 4^{th}.O.M.

Higher Order Matching and HOAS

Results on Matching

- **Gérard Huet** (1976) Higher Order Matching conjecture. "they are decidable"
- **Gérard Huet**, **B. Lang**(1978) found an algorithm that terminates in the case of 2^{nd}.O M.
- **Gilles Dowek**(1994): decidability of 3^{rd}.O.M.
- **V. Padovani**(1995): decidability of 4^{th}.O.M.
- More coming...

References

(1) **Richard Bird** and **Oege de Moor**(1997);
Algebra of Programming, Prentice Hall International Series in Computer Science.

References

(1) **Richard Bird** and **Oege de Moor**(1997);
Algebra of Programming, Prentice Hall International Series in Computer Science.

(2) **Simon Marlow**;
http://www.haskell.org/ghc/ Glasgow Haskell Compiler.

References

(1) **Richard Bird** and **Oege de Moor**(1997);
Algebra of Programming, Prentice Hall International Series in Computer Science.

(2) **Simon Marlow**;
http://www.haskell.org/ghc/ Glasgow Haskell Compiler.

(3) **Gérard Huet** and **Bernard Lang**(1978);
Proving and Applying Program Transformations Expressed with Second -Order Patterns, Informatica by Springer-Verlag.

References

(1) **Richard Bird** and **Oege de Moor**(1997);
Algebra of Programming, Prentice Hall International Series in Computer Science.

(2) **Simon Marlow**;
http://www.haskell.org/ghc/ Glasgow Haskell Compiler.

(3) **Gérard Huet** and **Bernard Lang**(1978);
Proving and Applying Program Transformations Expressed with Second -Order Patterns, Informatica by Springer-Verlag.

(4) **Simon P. Jones**, **André M. Santos**(1998);
A transformation-based optimiser for Haskell, Science of Computer Programming.

(5) **O. de Moor** and **G. Sittampalam**(1999);
Generic Program Transformation. Proceedings of the 3rd International Summer School on Advanced Functional Programming. Springer Lecture Notes in Computer Science, Vol 1608, pages 116–149.

(5) **O. de Moor** and **G. Sittampalam**(1999); *Generic Program Transformation.* Proceedings of the 3rd International Summer School on Advanced Functional Programming. Springer Lecture Notes in Computer Science, Vol 1608, pages 116–149.

(6) **Simon Thompson** and **Claus Rienke**; http://www.cs.kent.ac.uk/projects/refactor-fp/ Refactoring Functional Programming: *"Improving the design of existing code"*.

(5) **O. de Moor** and **G. Sittampalam**(1999);
Generic Program Transformation. Proceedings of the 3rd International Summer School on Advanced Functional Programming. Springer Lecture Notes in Computer Science, Vol 1608, pages 116–149.

(6) **Simon Thompson** and **Claus Rienke**;
http://www.cs.kent.ac.uk/projects/refactor-fp/ Refactoring Functional Programming: *"Improving the design of existing code"*.

(7) **Eelco Visser**;
http://www.stratego-language.org/ Stratego: *"Strategies for Program Transformation"*.

(5) **O. de Moor** and **G. Sittampalam**(1999);
Generic Program Transformation. Proceedings of the 3rd International Summer School on Advanced Functional Programming. Springer Lecture Notes in Computer Science, Vol 1608, pages 116–149.

(6) **Simon Thompson** and **Claus Rienke**;
http://www.cs.kent.ac.uk/projects/refactor-fp/ Refactoring Functional Programming: *"Improving the design of existing code"*.

(7) **Eelco Visser**;
http://www.stratego-language.org/ Stratego: *"Strategies for Program Transformation"*.

(8) **Eelco Visser**(2001);
A Survey of Strategies in Program Transformation Systems, Proceddings of WRS'01.