Transformaciones de Programas: Una Taxonomía, un punto de vista clásico, un punto de vista nuevo.

Carlos Martínez Méndez

Wesleyan University
Department of Mathematics and Computer Science

cmartinez@wesleyan.edu

August 2006

Informatics Civil Engineering, Universidad de Valparaiso, Chile

Abstract.

El propósito de esta conferencia es presentar desde una perspectiva lo más general posible una tema que ha tenido un ferviente desarrollo en estas últimas decadas en la área de diseño de languages de programación.

Abstract.

El propósito de esta conferencia es presentar desde una perspectiva lo más general posible una tema que ha tenido un ferviente desarrollo en estas últimas decadas en la área de diseño de languages de programación. Me centrare en un punto de vista originario de G. Huet y F. Lang que en 1978 presentaron esquemas de transformaciones que motivaron futuras implementaciones y sistemas que automatización de dichas tranformaciones.

Abstract.

El propósito de esta conferencia es presentar desde una perspectiva lo más general posible una tema que ha tenido un ferviente desarrollo en estas últimas decadas en la área de diseño de languages de programación. Me centrare en un punto de vista originario de G. Huet y F. Lang que en 1978 presentaron esquemas de transformaciones que motivaron futuras implementaciones y sistemas que automatización de dichas tranformaciones. Por último exploremos futuras extensiones que motivan mi disertación doctoral.

En esta presentación.

- Transformaciones de Programas?

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.
- Huet-Lang punto de vista: Transformaciones de Programas y *Higher Order Matching*.

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.
- Huet-Lang punto de vista: Transformaciones de Programas y *Higher Order Matching*.
  - ⋆ Recursión V/S Iteratión.

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.
- Huet-Lang punto de vista: Transformaciones de Programas y *Higher Order Matching*.
  - ⋆ Recursión V/S Iteratión.
  - ⋆ Compilación via Transformaciones. *(opcional)*

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.
- Huet-Lang punto de vista: Transformaciones de Programas y *Higher Order Matching*.
  - ⋆ Recursión V/S Iteratión.
  - ⋆ Compilación via Transformaciones. *(opcional)*
  - ⋆ Desforestación/Fusion.*(opcional)*

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.
- Huet-Lang punto de vista: Transformaciones de Programas y *Higher Order Matching*.
  - ⋆ Recursión V/S Iteratión.
  - ⋆ Compilación via Transformaciones. *(opcional)*
  - ⋆ Desforestación/Fusion.*(opcional)*
- Type Isomorphisms y Program Isomorphisms.

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.
- Huet-Lang punto de vista: Transformaciones de Programas y *Higher Order Matching*.
  - ⋆ Recursión V/S Iteratión.
  - ⋆ Compilación via Transformaciones. *(opcional)*
  - ⋆ Desforestación/Fusion.*(opcional)*
- Type Isomorphisms y Program Isomorphisms.
- Refinamiento de Transformaciones de Programas.

En esta presentación.

- Transformaciones de Programas?
- Taxonomía de Transformaciones de Programas.
- Huet-Lang punto de vista: Transformaciones de Programas y *Higher Order Matching*.
  - ⋆ Recursión V/S Iteratión.
  - ⋆ Compilación via Transformaciones. *(opcional)*
  - ⋆ Desforestación/Fusion.*(opcional)*
- Type Isomorphisms y Program Isomorphisms.
- Refinamiento de Transformaciones de Programas.
- Referencias Principales.

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software,

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores,

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software,

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software, Generación de Documentación,

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software, Generación de Documentación, Renovación automatizada de Software.

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software, Generación de Documentación, Renovación automatizada de Software.

Escenarios posibles: Transformaciones desde

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software, Generación de Documentación, Renovación automatizada de Software.

Escenarios posibles: Transformaciones desde

- Un lenguaje a otro lenguaje diferente,

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software, Generación de Documentación, Renovación automatizada de Software.

Escenarios posibles: Transformaciones desde

- Un lenguaje a otro lenguaje diferente,Traduciones*(Translations)*.

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software, Generación de Documentación, Renovación automatizada de Software.

Escenarios posibles: Transformaciones desde

- Un lenguaje a otro lenguaje diferente,Traduciones*(Translations)*.
- Un lenguaje en si mismo,

Transformaciones de Programas?

Definición: Una *transformación de programas* es el acto de cambiar un programa en otro.

Motivación: Aumentar la *productividad* de programadores al automatizar ciertas tareas de la programación, esto permite una programación de una mayor nivel de abstracción, y alcanzar mayor *reusabilidad* y *mantenibilidad* de estos.

Aplicación: Ingenieria en Software, Construction y Diseño de Compiladores, Visualización de Software, Generación de Documentación, Renovación automatizada de Software.

Escenarios posibles: Transformaciones desde

- Un lenguaje a otro lenguaje diferente,Traduciones*(Translations)*.
- Un lenguaje en si mismo, Reescritura*(Rephrasing)*.

Transformación de Programa:

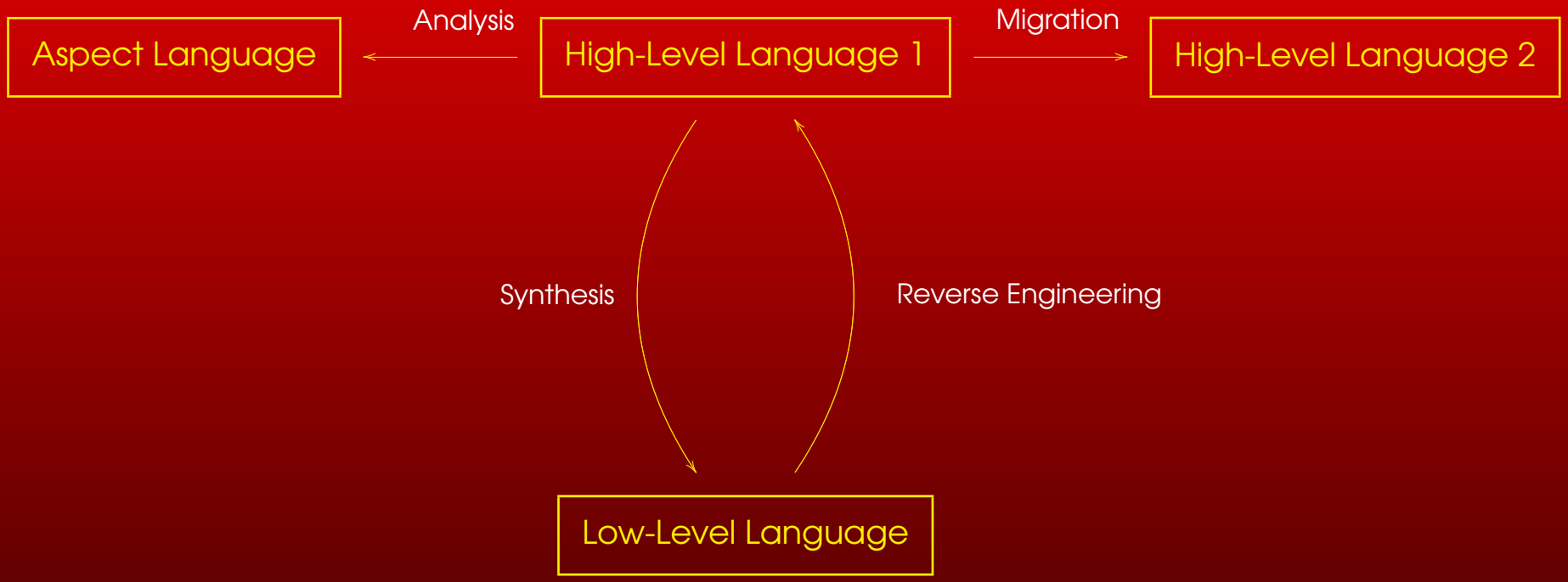
Aspect Language

High-Level Language 1

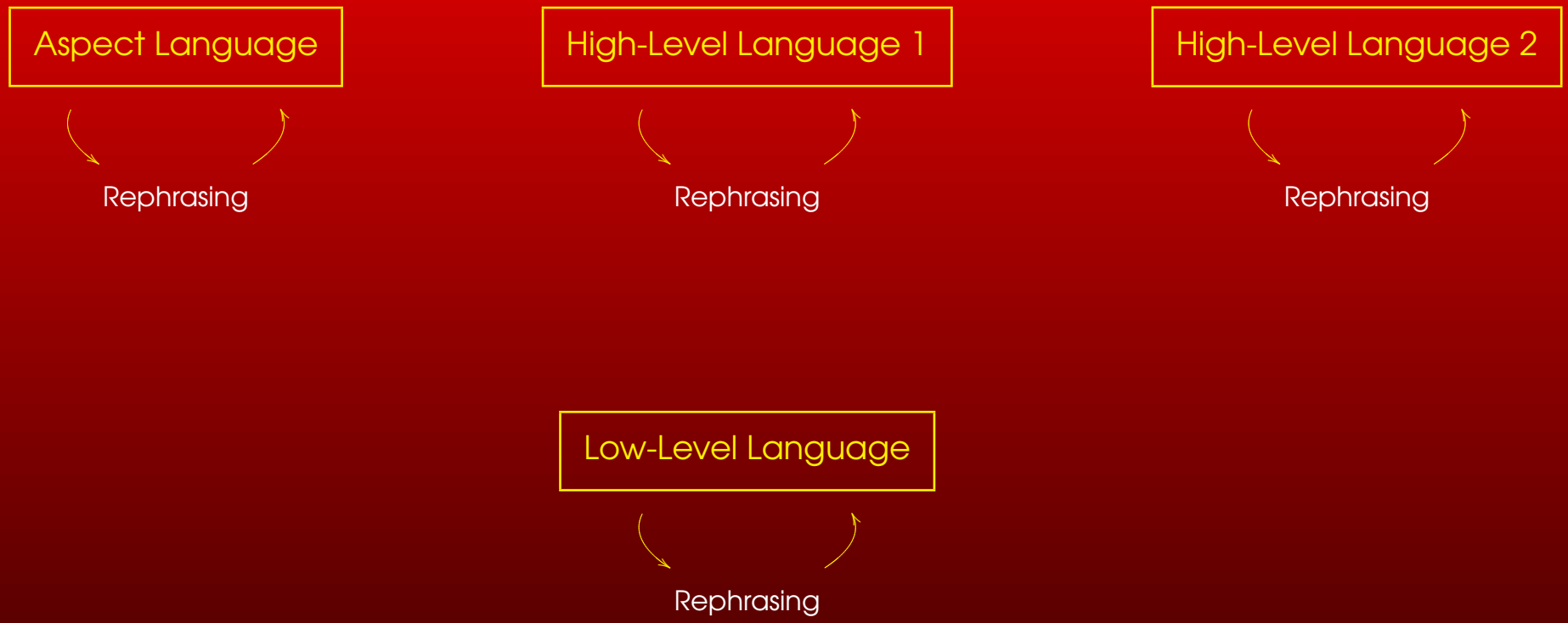
High-Level Language 2

Low-Level Language

Transformación de Programa: Translation.



Transformación de Programa: Rephrasing.



Taxonomía de Transformaciones de Programas

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

Taxonomía de Transformaciones de Programas

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

Huet-Lang punto de vista.

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

Huet-Lang punto de vista.

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.."

Huet-Lang punto de vista.

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.."

"Hay una brecha entre técnicas de certificación de software y herramientas teóricas definiendo pruebas formales de programas.."

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Caso 1:

Structural Recursion:

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Caso 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 }
```

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Caso 2:

Structural Recursion:

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Caso 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) }
```

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Abstración:

Σ:

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Abstración:

Σ:

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

Definición: Decimos que `a, b, c, d, f` y `h` son *variable libres* del patrón Σ.

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Abstración:

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)) }
```

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 1

Σ:

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

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 1

Σ:

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

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

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

Σ:

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

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

Σ:

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

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

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

$\sigma_2(\Sigma)$:

```
(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)) }
```

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

$\sigma_2(\Sigma)$:

```
(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) }
```

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

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

Huet-Lang punto de vista.

Ejemplo: Recursión V/S Iteración.

Instanciación: Caso 2

$\sigma_2(\Sigma)$:

```
(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) }
```

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación $\mathtt{h}$ para ser aplicada *validamente*.

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación $\mathtt{h}$ para ser aplicada *validamente*.

$\mathcal{X}.1$ $\mathtt{h}$ es asociativa.

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación `h` para ser aplicada *validamente*.

$\mathcal{X}$.1 `h` es asociativa.
$\mathcal{X}$.2 `h` tiene unidad ($\exists$ `*: h(x,*) = h(*,x) = x`).

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación `h` para ser aplicada *validamente*.

$\mathcal{X}$.1 `h` es asociativa.
$\mathcal{X}$.2 `h` tiene unidad ($\exists$ `*: h(x,*) = h(*,x) = x`).

- El Paradigma:

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación `h` para ser aplicada *validamente*.

$\mathcal{X}$.1 `h` es asociativa.
$\mathcal{X}$.2 `h` tiene unidad ($\exists$ `*: h(x,*) = h(*,x) = x`).

- El Paradigma:

1. Reconocer un patrón: $(\Sigma, \Sigma', \mathcal{X})$

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación `h` para ser aplicada *validamente*.

$\mathcal{X}$.1 `h` es asociativa.
$\mathcal{X}$.2 `h` tiene unidad ($\exists$ `*: h(x,*) = h(*,x) = x`).

- El Paradigma:

1. Reconocer un patrón: $(\Sigma, \Sigma', \mathcal{X})$
2. Validar tal patrón.

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación `h` para ser aplicada *validamente*.

$\mathcal{X}$.1 `h` es asociativa.
$\mathcal{X}$.2 `h` tiene unidad ($\exists$ `*: h(x,*) = h(*,x) = x`).

- El Paradigma:
  1. Reconocer un patrón: $(\Sigma, \Sigma', \mathcal{X})$
  2. Validar tal patrón.
  3. Reconocer si un patrón es aplicable a un programa dado.

Huet-Lang punto de vista.

Observaciones desde el ejemplo.

- *Factorial* y *Reverse* usan el mismo patrón (Σ, Σ') y las substuciones σ_1 y σ_2 son las *correspondencia* respectivamente.
- La transformación (Σ, Σ') requiere *satisfacer* las siguientes condiciones $\mathcal{X}$ sobre la operación `h` para ser aplicada *validamente*.

$\mathcal{X}$.1 `h` es asociativa.
$\mathcal{X}$.2 `h` tiene unidad ($\exists$ `*: h(x,*) = h(*,x) = x`).

- El Paradigma:

1. Reconocer un patrón: $(\Sigma, \Sigma', \mathcal{X})$
2. Validar tal patrón.
3. Reconocer si un patrón es aplicable a un programa dado.
4. Organizar la aplicación automática de los patrones.

Huet-Lang punto de vista.

Sea $(\Sigma, \Sigma', \mathcal{X})$ una transformación, y sea $\mathcal{P}[t]$ una programa tal que contiene t como "subrutina" *(fragmento)*.

Huet-Lang punto de vista.

Sea $(\Sigma, \Sigma', \mathcal{X})$ una transformación, y sea $\mathcal{P}[t]$ una programa tal que contiene t como "subrutina" *(fragmento)*.

Definición: Decimos que la transformación es *aplicable* en $\mathcal{P}[t]$ *en* t si y solo si existe una substitución σ para las *variable libres* de Σ y Σ' tal que:

Huet-Lang punto de vista.

Sea $(\Sigma, \Sigma', \mathcal{X})$ una transformación, y sea $\mathcal{P}[t]$ una programa tal que contiene t como "subrutina" *(fragmento)*.

Definición: Decimos que la transformación es *aplicable* en $\mathcal{P}[t]$ *en* t si y solo si existe una substitución σ para las *variable libres* de Σ y Σ' tal que:

1. $t = \sigma(\Sigma)$ *(matching)*

Huet-Lang punto de vista.

Sea $(\Sigma, \Sigma', \mathcal{X})$ una transformación, y sea $\mathcal{P}[\mathrm{t}]$ una programa tal que contiene t como "subrutina" *(fragmento)*.

Definición: Decimos que la transformación es *aplicable* en $\mathcal{P}[\mathrm{t}]$ *en* t si y solo si existe una substitución σ para las *variable libres* de Σ y Σ' tal que:

1. $\mathrm{t} = \sigma(\Sigma)$ *(matching)*

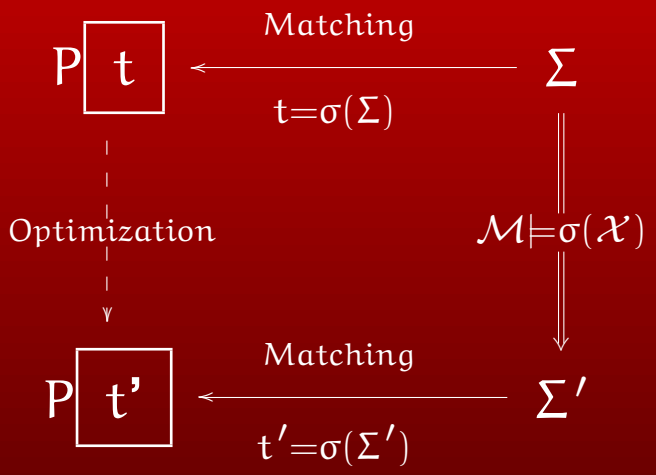
2. $\mathcal{M} \models \sigma(\mathcal{X})$, es decir, las restriciones $\mathcal{X}$ bajo σ son validas en la semántica de programación $\mathcal{M}$.

Huet-Lang punto de vista.

Aplicación de T := $(\Sigma, \Sigma', \mathcal{X})$ a $\mathcal{P}[t]$:

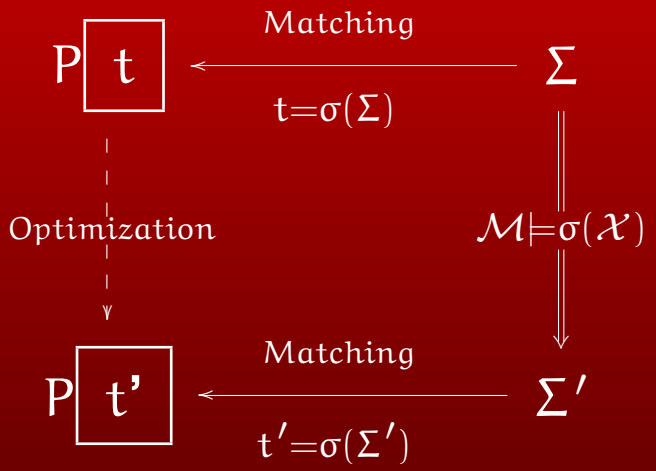
Huet-Lang punto de vista.

Aplicación de $T := (\Sigma, \Sigma', \mathcal{X})$ a $\mathcal{P}[t]$:



Huet-Lang punto de vista.

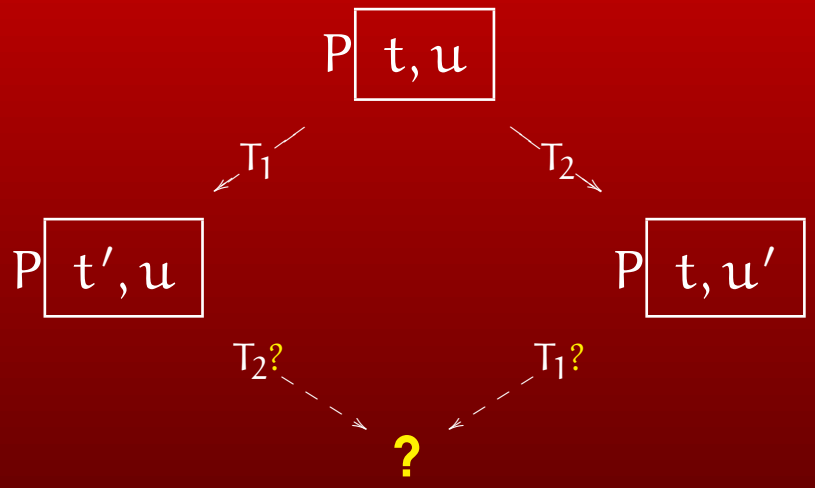
Aplicación de $T := (\Sigma, \Sigma', \mathcal{X})$ a $\mathcal{P}[t]$:



Desafío: Un primer problema de esta perspectiva occurre cuando queremos aplicar en algún contexto dos transformaciones diferentes obteniando dos refinamientos de este programa en principio diferentes.

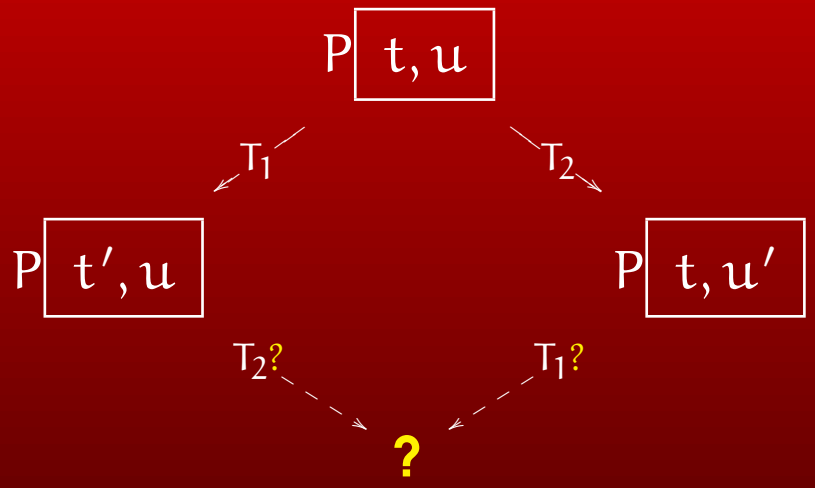
Huet-Lang punto de vista.

Aplicaciones simultáneas?:



Huet-Lang punto de vista.

Aplicaciones simultáneas?:



Donde las transformaciones estan definidas como:

- $T_1 := (\Sigma, \Sigma', \mathcal{X})$ y existe una substitución σ tal que $\sigma(\Sigma) = t$ y $\sigma(\Sigma') = t'$.
- $T_2 := (\Theta, \Theta', \mathcal{Y})$ y existe una substitución θ tal que $\theta(\Theta) = u$ y $\theta(\Theta') = u'$.

Huet-Lang punto de vista.

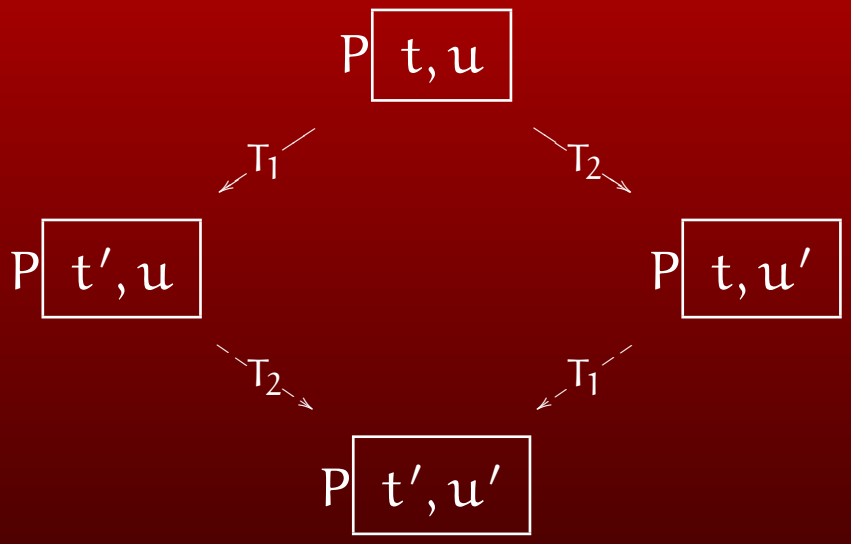
Aplicaciones simultáneas:

Este problema es conocido en *Sistemas de Reescritura (Rewriting Systems)*. En algunos casos podremos concluir lo deseado:

Huet-Lang punto de vista.

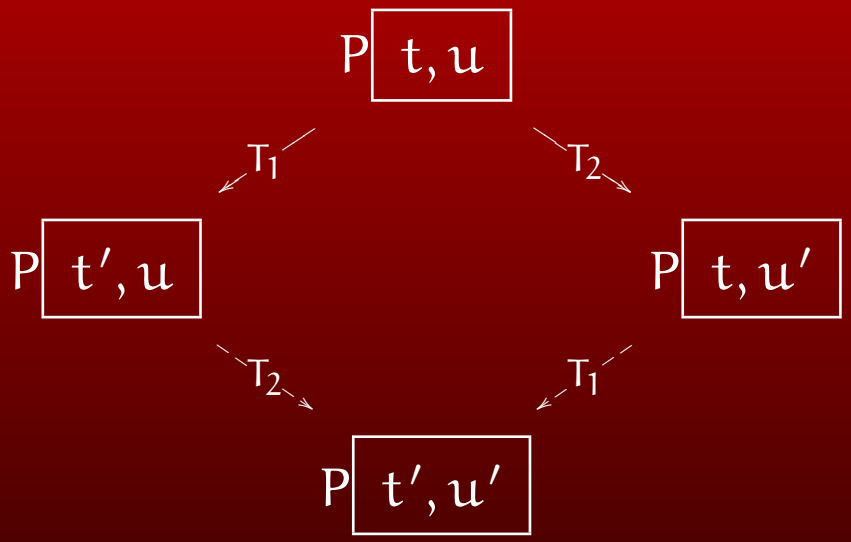
Aplicaciones simultáneas:

Este problema es conocido en *Sistemas de Reescritura (Rewriting Systems)*. En algunos casos podremos concluir lo deseado:



Huet-Lang punto de vista.

Aplicaciones simultáneas:
Este problema es conocido en *Sistemas de Reescritura (Rewriting Systems)*. En algunos casos podremos concluir lo deseado:



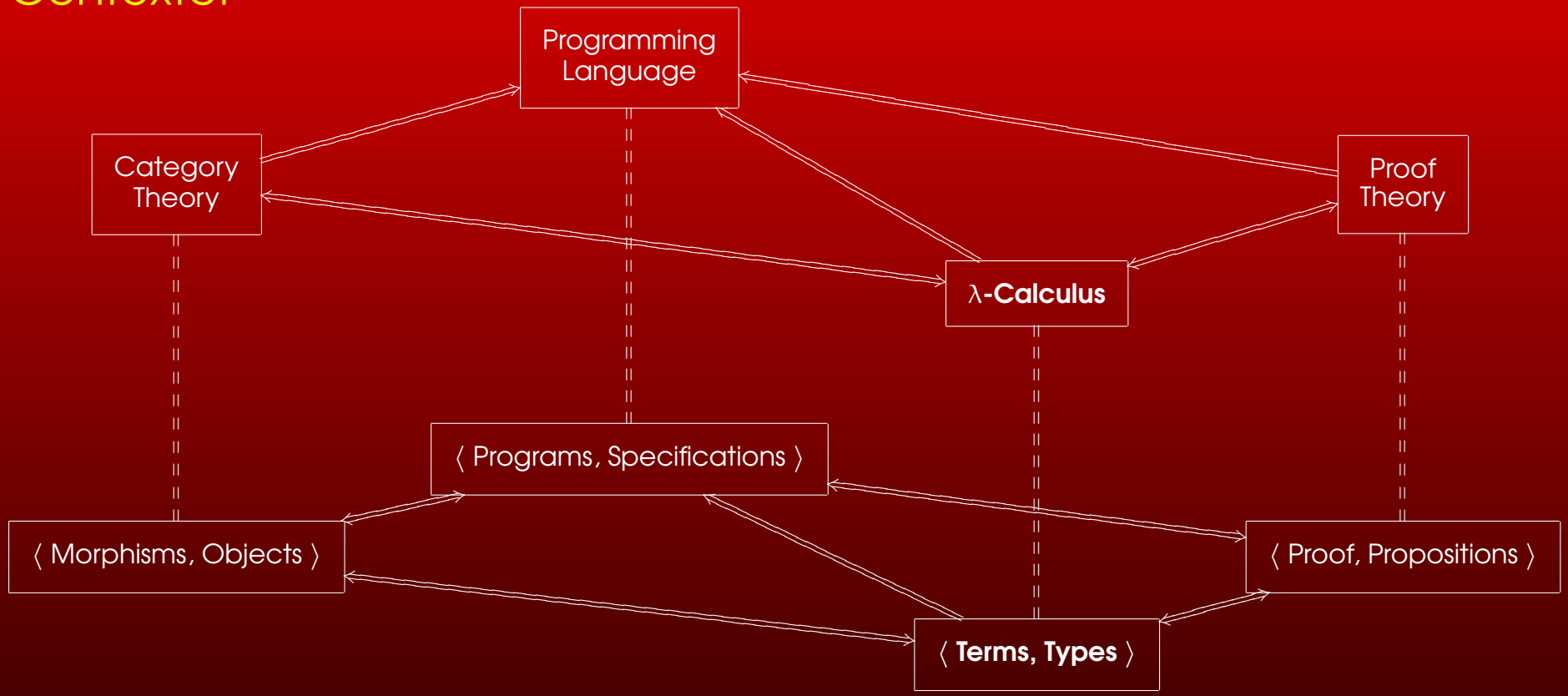
En este caso, sera necesario demostrar que nuestro sistema de transformaciones tiene la propiedad de *confluencia*.

Type Isomorphisms and Program Isomorphism.

Contexto:

Type Isomorphisms and Program Isomorphism.

Contexto:



Type Isomorphisms and Program Isomorphism.

λ-Cálculo:

Type Isomorphisms and Program Isomorphism.

λ-Cálculo:

Tipos: τ

- Sea ι un *tipo básico*

$$\tau ::= \iota \mid (\tau \to \tau) \mid \tau \times \tau$$

Type Isomorphisms and Program Isomorphism.

λ-Cálculo:

Tipos: τ

- Sea ι un *tipo básico*

$$\tau ::= \iota \mid (\tau \to \tau) \mid \tau \times \tau$$

Términos: t

- Sea x una *variable*

$$t ::= x \mid (\lambda x.t) \mid (tt) \mid \langle t, t \rangle$$

Type Isomorphisms and Program Isomorphism.

λ-Cálculo:

Tipos: τ

- Sea ι un *tipo básico*

$$\tau ::= \iota \mid (\tau \to \tau) \mid \tau \times \tau$$

Términos: t

- Sea x una *variable*

$$t ::= x \mid (\lambda x.t) \mid (tt) \mid \langle t, t \rangle$$

Computaciones: $\triangleright$

- $(\lambda x.t)u \triangleright_\beta t[x := u]$ *(beta-reducción)*.
- $(\lambda x.(tx)) \triangleright_\eta t$, if x is not free in t *(eta-reducción)*.

Type Isomorphisms and Program Isomorphism.

Motivación:

Type Isomorphisms and Program Isomorphism.

Motivación:

Considere los siguientes programas:

$$f_1 = \lambda\langle x, y\rangle.2 * x + 3 * y;; \quad f_1 : int \times int \to int$$

$$f_2 = \lambda\langle y, x\rangle.2 * x + 3 * y;; \quad f_2 : int \times int \to int$$

$$f_3 = \lambda x \lambda y.2 * x + 3 * y;; \quad f_3 : int \to (int \to int)$$

$$f_4 = \lambda y.\lambda x.2 * x + 3 * y;; \quad f_4 : int \to (int \to int)$$

Type Isomorphisms and Program Isomorphism.

Motivación:

Considere los siguientes programas:

$$f_1 = \lambda\langle x, y\rangle.2 * x + 3 * y;; \quad f_1 : int \times int \to int$$

$$f_2 = \lambda\langle y, x\rangle.2 * x + 3 * y;; \quad f_2 : int \times int \to int$$

$$f_3 = \lambda x\lambda y.2 * x + 3 * y;; \quad f_3 : int \to (int \to int)$$

$$f_4 = \lambda y.\lambda x.2 * x + 3 * y;; \quad f_4 : int \to (int \to int)$$

Observe que:

$$f_1(4,5) = (\lambda\langle x, y\rangle.2 * x + 3 * y)(4,5) \rhd_\beta 2 * 4 + 3 * 5 = \mathbf{23}$$

Type Isomorphisms and Program Isomorphism.

Motivación:

Considere los siguientes programas:

$$f_1 = \lambda\langle x, y\rangle.2 * x + 3 * y;; \quad f_1 : int \times int \to int$$

$$f_2 = \lambda\langle y, x\rangle.2 * x + 3 * y;; \quad f_2 : int \times int \to int$$

$$f_3 = \lambda x\lambda y.2 * x + 3 * y;; \quad f_3 : int \to (int \to int)$$

$$f_4 = \lambda y.\lambda x.2 * x + 3 * y;; \quad f_4 : int \to (int \to int)$$

Observe que:

$$f_1(4,5) = (\lambda\langle x, y\rangle.2 * x + 3 * y)(4,5) \rhd_\beta 2 * 4 + 3 * 5 = \mathbf{23}$$
$$f_2(5,4) = (\lambda\langle y, x\rangle.2 * x + 3 * y)(5,4) \rhd_\beta 2 * 4 + 3 * 5 = \mathbf{23}$$

Type Isomorphisms and Program Isomorphism.

Motivación:

Considere los siguientes programas:

$$f_1 = \lambda\langle x, y\rangle.2 * x + 3 * y;; \quad f_1 : int \times int \to int$$

$$f_2 = \lambda\langle y, x\rangle.2 * x + 3 * y;; \quad f_2 : int \times int \to int$$

$$f_3 = \lambda x\lambda y.2 * x + 3 * y;; \quad f_3 : int \to (int \to int)$$

$$f_4 = \lambda y.\lambda x.2 * x + 3 * y;; \quad f_4 : int \to (int \to int)$$

Observe que:

$$f_1(4,5) = (\lambda\langle x, y\rangle.2 * x + 3 * y)(4,5) \rhd_\beta 2 * 4 + 3 * 5 = \mathbf{23}$$
$$f_2(5,4) = (\lambda\langle y, x\rangle.2 * x + 3 * y)(5,4) \rhd_\beta 2 * 4 + 3 * 5 = \mathbf{23}$$
$$(f_3 4)5 = ((\lambda x\lambda y.2 * x + 3 * y)4)5 \rhd_\beta (\lambda y.8 + 3 * y)5 \rhd_\beta 8 + 3 * 5 = \mathbf{23}$$

Type Isomorphisms and Program Isomorphism.

Motivación:

Considere los siguientes programas:

$$f_1 = \lambda\langle x, y\rangle.2 * x + 3 * y;; \quad f_1 : int \times int \to int$$

$$f_2 = \lambda\langle y, x\rangle.2 * x + 3 * y;; \quad f_2 : int \times int \to int$$

$$f_3 = \lambda x \lambda y.2 * x + 3 * y;; \quad f_3 : int \to (int \to int)$$

$$f_4 = \lambda y.\lambda x.2 * x + 3 * y;; \quad f_4 : int \to (int \to int)$$

Observe que:

$$f_1(4,5) = (\lambda\langle x, y\rangle.2 * x + 3 * y)(4,5) \rhd_\beta 2 * 4 + 3 * 5 = \mathbf{23}$$
$$f_2(5,4) = (\lambda\langle y, x\rangle.2 * x + 3 * y)(5,4) \rhd_\beta 2 * 4 + 3 * 5 = \mathbf{23}$$
$$(f_3 4)5 = ((\lambda x \lambda y.2 * x + 3 * y)4)5 \rhd_\beta (\lambda y.8 + 3 * y)5 \rhd_\beta 8 + 3 * 5 = \mathbf{23}$$
$$(f_4 5)4 = ((\lambda y \lambda x.2 * x + 3 * y)5)4 \rhd_\beta (\lambda x.2 * x + 15)4 \rhd_\beta 2 * 4 + 15 = \mathbf{23}$$

Type Isomorphisms and Program Isomorphism.

Motivación:

Type Isomorphisms and Program Isomorphism.

Motivación:

Aún más, se tienen los siguientes programas:

$$perm = \lambda z \lambda \langle x, y \rangle . z \langle y, x \rangle \;;; \quad perm : ((A \times B) \to C)) \to ((B \times A) \to C))$$

$$curry = \lambda x \lambda x \lambda y . z \langle x, y \rangle ;; \quad curry : ((A \times B) \to C)) \to (A \to (B \to C))$$

Type Isomorphisms and Program Isomorphism.

Motivación:

Aún más, se tienen los siguientes programas:

$$\begin{array}{l|l} perm = \lambda z \lambda \langle x, y \rangle . z \langle y, x \rangle \;;; & curry = \lambda x \lambda x \lambda y . z \langle x, y \rangle ;; \\ perm : ((A \times B) \to C)) \to ((B \times A) \to C)) & curry : ((A \times B) \to C)) \to (A \to (B \to C)) \end{array}$$

Calculemos $permf_1 = ?$ y $curryf_1 = ?$ *(exercisio)*.

$$\begin{array}{rcl} permf_1 & = & (\lambda z \lambda \langle x, y \rangle . z \langle y, x \rangle) \lambda \langle x, y \rangle . 2 * x + 3 * y \\ & \triangleright_\beta & \lambda \langle x, y \rangle . (\lambda \langle x, y \rangle . 2 * x + 3 * y) \langle y, x \rangle \\ & \triangleright_\beta & \lambda \langle x, y \rangle . 2 * y + 3 * x = f_2 \end{array}$$

Type Isomorphisms and Program Isomorphism.

Definición: ***(Programa Invertible)*** *Un programa* $\mathrm{M} : \mathrm{A} \to \mathrm{B}$ *es invertible, si existe un programa* $\mathrm{N} : \mathrm{B} \to \mathrm{A}$ *tal que* $\mathrm{N} \circ \mathrm{M} = \mathrm{Id}_{\mathrm{A}}$ *and* $\mathrm{M} \circ \mathrm{N} = \mathrm{Id}_{\mathrm{B}}$.

Type Isomorphisms and Program Isomorphism.

Definición: ***(Programa Invertible)*** *Un programa* $M : A \to B$ *es invertible, si existe un programa* $N : B \to A$ *tal que* $N \circ M = Id_A$ *and* $M \circ N = Id_B$.

Ejemplos:

#Let curry f $xy = f(x, y)$;;
curry: $((A \times B) \to C) \to (A \to (B \to C))$ = ⟨**fun**⟩
#Let uncurry f $(x, y) = fxy$;;
uncurry: $(A \to (B \to C)) \to ((A \times B) \to C)$ = ⟨**fun**⟩

Type Isomorphisms and Program Isomorphism.

Definición: ***(Programa Invertible)*** *Un programa* $M : A \to B$ *es invertible, si existe un programa* $N : B \to A$ *tal que* $N \circ M = Id_A$ *and* $M \circ N = Id_B$.

Ejemplos:

#Let curry f $xy = f(x, y)$;;
curry: $((A \times B) \to C) \to (A \to (B \to C))$ = $\langle$**fun**$\rangle$
#Let uncurry f $(x, y) = fxy$;;
uncurry: $(A \to (B \to C)) \to ((A \times B) \to C)$ = $\langle$**fun**$\rangle$

#Let perm f $xy = fyx$;;
perm : $(A \to (B \to C)) \to (B \to (A \to C))$ = $\langle$**fun**$\rangle$

Type Isomorphisms and Program Isomorphism.

Definición: ***(Programa Invertible)*** *Un programa* $M : A \to B$ *es invertible, si existe un programa* $N : B \to A$ *tal que* $N \circ M = Id_A$ *and* $M \circ N = Id_B$.

Ejemplos:

#Let curry f $xy = f(x, y)$;;
curry: $((A \times B) \to C) \to (A \to (B \to C))$ = $\langle$**fun**$\rangle$
#Let uncurry f $(x, y) = fxy$;;
uncurry: $(A \to (B \to C)) \to ((A \times B) \to C)$ = $\langle$**fun**$\rangle$

#Let perm f $xy = fyx$;;
perm : $(A \to (B \to C)) \to (B \to (A \to C))$ = $\langle$**fun**$\rangle$

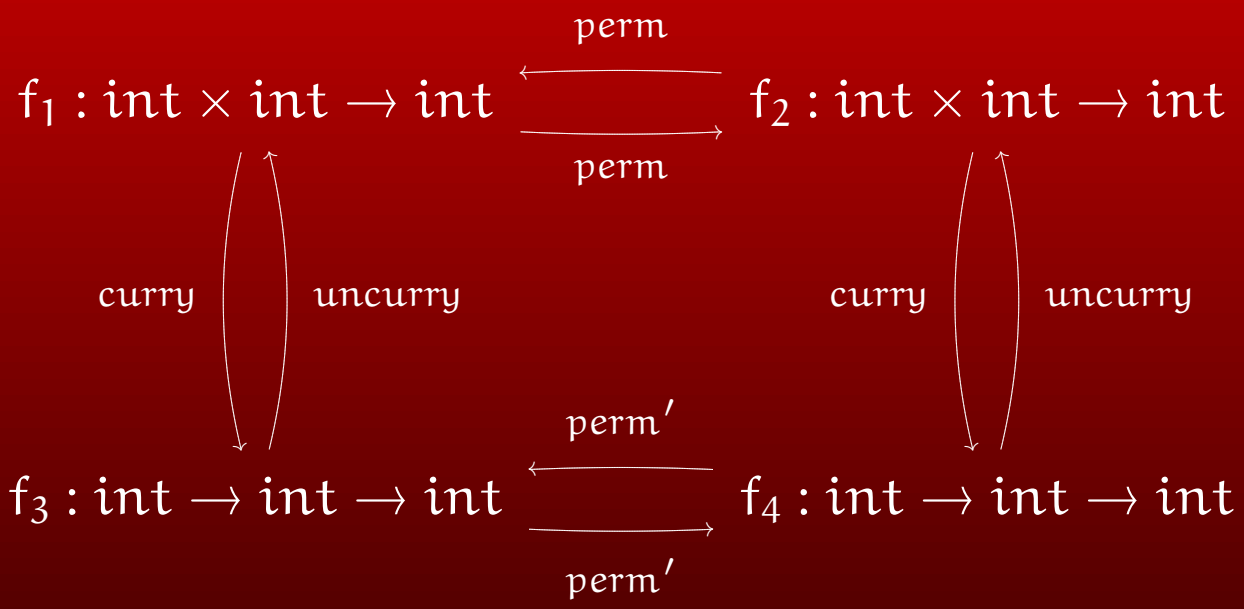
#Let perm′ f $(x, y) = f(y, x)$;;
perm′ : $((A \times B) \to C)) \to ((B \times A) \to C))$= $\langle$**fun**$\rangle$

Type Isomorphisms and Program Isomorphism.

En resumen:

Type Isomorphisms and Program Isomorphism.

En resumen:



Type Isomorphisms and Program Isomorphism.

Definición ***(Programas Invertibles)*** *Un programa* $\mathrm{M} : \mathrm{A} \to \mathrm{B}$ *es invertible, Si existe un programa* $\mathrm{N} : \mathrm{B} \to \mathrm{A}$ *tal que* $\mathrm{N} \circ \mathrm{M} = \mathrm{Id}_{\mathrm{A}}$ *and* $\mathrm{M} \circ \mathrm{N} = \mathrm{Id}_{\mathrm{B}}$.

Type Isomorphisms and Program Isomorphism.

Definición ***(Programas Invertibles)*** *Un programa* $\mathrm{M} : \mathrm{A} \to \mathrm{B}$ *es invertible, Si existe un programa* $\mathrm{N} : \mathrm{B} \to \mathrm{A}$ *tal que* $\mathrm{N} \circ \mathrm{M} = \mathrm{Id}_{\mathrm{A}}$ *and* $\mathrm{M} \circ \mathrm{N} = \mathrm{Id}_{\mathrm{B}}$.

Definición ***(Tipos Isomórficos)*** *Dos tipos* A *y* B *son tipos isomórficos, denotado* $\mathrm{A} \simeq \mathrm{B}$*, si y solo si existen programas* $\mathrm{M} : \mathrm{A} \to \mathrm{B}$ *y* $\mathrm{N} : \mathrm{B} \to \mathrm{A}$ *tal que* $\mathrm{N} \circ \mathrm{M} = \mathrm{Id}_{\mathrm{A}}$ *and* $\mathrm{M} \circ \mathrm{N} = \mathrm{Id}_{\mathrm{B}}$.

Type Isomorphisms and Program Isomorphism.

Definición ***(Programas Invertibles)*** *Un programa* $\mathrm{M} : \mathrm{A} \to \mathrm{B}$ *es invertible, Si existe un programa* $\mathrm{N} : \mathrm{B} \to \mathrm{A}$ *tal que* $\mathrm{N} \circ \mathrm{M} = \mathrm{Id}_{\mathrm{A}}$ *and* $\mathrm{M} \circ \mathrm{N} = \mathrm{Id}_{\mathrm{B}}$.

Definición ***(Tipos Isomórficos)*** *Dos tipos* A *y* B *son tipos isomórficos, denotado* $\mathrm{A} \simeq \mathrm{B}$*, si y solo si existen programas* $\mathrm{M} : \mathrm{A} \to \mathrm{B}$ *y* $\mathrm{N} : \mathrm{B} \to \mathrm{A}$ *tal que* $\mathrm{N} \circ \mathrm{M} = \mathrm{Id}_{\mathrm{A}}$ *and* $\mathrm{M} \circ \mathrm{N} = \mathrm{Id}_{\mathrm{B}}$.

Definición ***(Programas Isomórfiscos)*** *Dos programas* $\mathrm{P} : \mathrm{A}$ *y* $\mathrm{Q} : \mathrm{B}$ *son programas isomórficos, denotado* $\mathrm{P} \simeq \mathrm{Q}$*, si y solo si* $\mathrm{A} \simeq \mathrm{B}$ *y existe un programa invertible* $\mathrm{F} : \mathrm{A} \to \mathrm{B}$ *tal que* $\mathrm{FP} = \mathrm{Q}$.

Transformación de Programas Revisada.

Definición: Decimos que la transformación es *aplicable* en $\mathcal{P}[t]$ *en* t si y solo si existe una substitución σ para las *variable libres* de Σ y Σ' tal que:

Transformación de Programas Revisada.

Definición: Decimos que la transformación es *aplicable* en $\mathcal{P}[t]$ *en* t si y solo si existe una substitución σ para las *variable libres* de Σ y Σ' tal que:

1. $t \simeq \sigma(\Sigma)$ *(matching)*

Transformación de Programas Revisada.

Definición: Decimos que la transformación es *aplicable* en $\mathcal{P}[t]$ *en* t si y solo si existe una substitución σ para las *variable libres* de Σ y Σ' tal que:

1. $t \simeq \sigma(\Sigma)$ *(matching)*

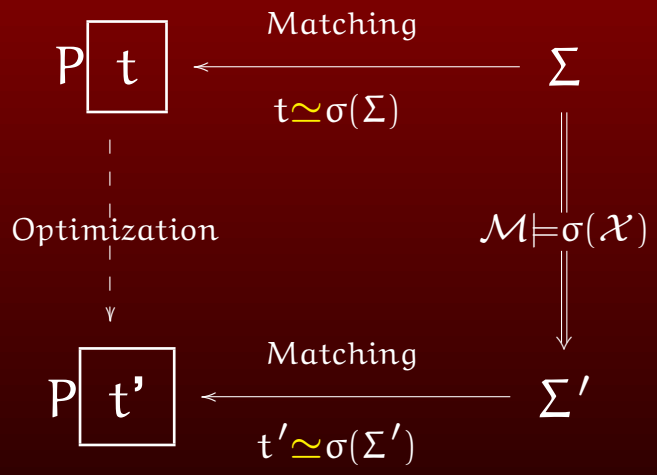
2. $\mathcal{M} \models \sigma(\mathcal{X})$, es decir, las restriciones $\mathcal{X}$ bajo σ son validas en la semántica de programación $\mathcal{M}$.

Transformación de Programas Revisada.

Definición: Decimos que la transformación es *aplicable* en $\mathcal{P}[t]$ *en* t si y solo si existe una substitución σ para las *variable libres* de Σ y Σ' tal que:

1. $t \simeq \sigma(\Sigma)$ *(matching)*

2. $\mathcal{M} \models \sigma(\mathcal{X})$, es decir, las restriciones $\mathcal{X}$ bajo σ son validas en la semántica de programación $\mathcal{M}$.



Higher Order Matching.

Results on Matching

Higher Order Matching.

Results on Matching

- **Gérard Huet** (1976): Higher Order Matching conjectura positivamente, es decir el problema es decidible.

Higher Order Matching.

Results on Matching

- **Gérard Huet** (1976): Higher Order Matching conjectura positivamente, es decir el problema es decidible.
- **Gérard Huet**, **B. Lang**(1978) encontraron un algoritmo que termina (Converge) en el caso of 2^{nd}.O M.

Higher Order Matching.

Results on Matching

- **Gérard Huet** (1976): Higher Order Matching conjectura positivamente, es decir el problema es decidible.
- **Gérard Huet**, **B. Lang**(1978) encontraron un algoritmo que termina (Converge) en el caso of 2^{nd}.O M.
- **Gilles Dowek**(1994): decidible 3^{rd}.O.M.

Higher Order Matching.

Results on Matching

- **Gérard Huet** (1976): Higher Order Matching conjectura positivamente, es decir el problema es decidible.
- **Gérard Huet**, **B. Lang**(1978) encontraron un algoritmo que termina (Converge) en el caso of 2^{nd}.O M.
- **Gilles Dowek**(1994): decidible 3^{rd}.O.M.
- **V. Padovani**(1995): decidible 4^{th}.O.M.

Higher Order Matching.

Results on Matching

- **Gérard Huet** (1976): Higher Order Matching conjectura positivamente, es decir el problema es decidible.
- **Gérard Huet**, **B. Lang**(1978) encontraron un algoritmo que termina (Converge) en el caso of 2^{nd}.O M.
- **Gilles Dowek**(1994): decidible 3^{rd}.O.M.
- **V. Padovani**(1995): decidible 4^{th}.O.M.
- More coming...

References

(1) **Roberto Di Cosmo**(1995);
Isomorphism of Types: from λ-calculus to information retrieval and language design, *Birkhäuser*.

References

(1) **Roberto Di Cosmo**(1995);
Isomorphism of Types: from λ-calculus to information retrieval and language design, *Birkhäuser*.

(2) **M. Dezani-Ciancaglini**(1976);
Characterization of Normal Forms Possessing Inverse in the $\lambda_{\beta\eta}$-Calculus, *Theoretical Computer Science*, **2**, 323 - 337.

References

(1) **Roberto Di Cosmo**(1995);
Isomorphism of Types: from λ-calculus to information retrieval and language design, *Birkhäuser*.

(2) **M. Dezani-Ciancaglini**(1976);
Characterization of Normal Forms Possessing Inverse in the $\lambda_{\beta\eta}$-Calculus, *Theoretical Computer Science*, **2**, 323 - 337.

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

References

(1) **Roberto Di Cosmo**(1995);
Isomorphism of Types: from λ-calculus to information retrieval and language design, *Birkhäuser*.

(2) **M. Dezani-Ciancaglini**(1976);
Characterization of Normal Forms Possessing Inverse in the $\lambda_{\beta\eta}$-Calculus, *Theoretical Computer Science*, **2**, 323 - 337.

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

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

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

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

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

(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) **M. Rittri**(1993);
Retrieving Library Functions by Unifying Types Modulo Linear Type Isomorphism, *Theoretical Informatics and Applications*, **27**, 71 -89.

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

(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) **M. Rittri**(1993);
Retrieving Library Functions by Unifying Types Modulo Linear Type Isomorphism, *Theoretical Informatics and Applications*, **27**, 71 -89.

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

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

(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) **M. Rittri**(1993);
Retrieving Library Functions by Unifying Types Modulo Linear Type Isomorphism, *Theoretical Informatics and Applications*, **27**, 71 -89.

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

(9) **Y. Zibin, J. Gil, J. Considine**(2003);
Efficient Algorithm for Isomorphism of Simple Types, *Proceedings POPL'03 in ACM SIGPLAN,* **38**, 160 - 171, 2003.

Apéndice.

- Compilacion via Transformaciones: Glasgow Haskell Compiler.

Apéndice.

- Compilacion via Transformaciones: Glasgow Haskell Compiler.
- Deforestation/Fusion.

Apéndice.

- Compilacion via Transformaciones: Glasgow Haskell Compiler.
- Deforestation/Fusion.
- Higher Order Matching and HOAS.

GHC [1]: 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*.

GHC [1]: 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

GHC [1]: 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

GHC: Compilation by transformation.

The Core Language:

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

GHC: 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
```

GHC: 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)
```

GHC: 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)
```

GHC: Compilation by transformation.

"Desugarer":

GHC: 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}`

GHC: Compilation by transformation.

A Core to Core Optimization:

Deforestation/Fusion laws are program transformation by means of which intermediate data structures can be eliminated.

GHC: 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

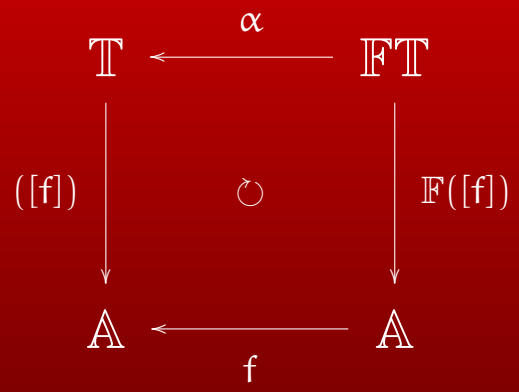
Version(2) UnModular

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

Deforestation/Fusion

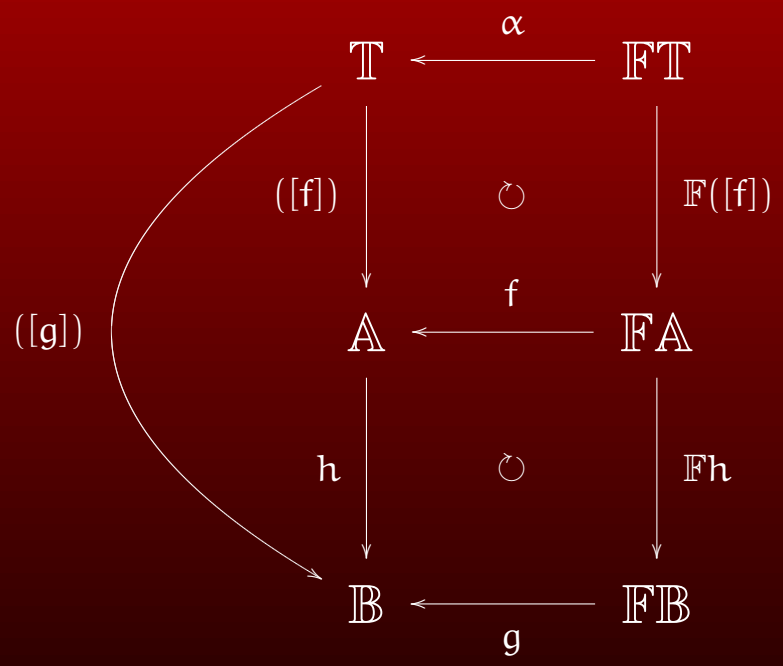
Fusion Law

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

Deforestation/Fusion

Fusion Law

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

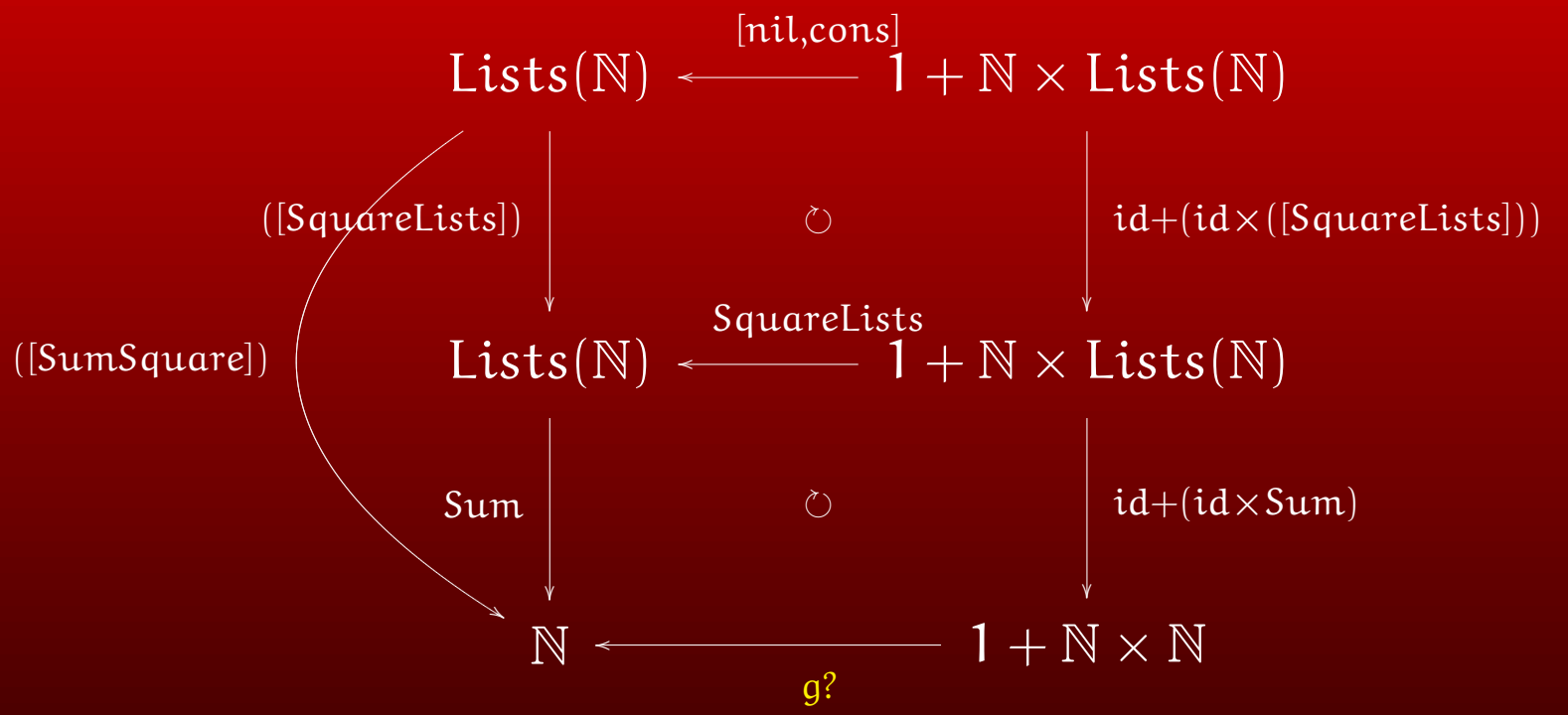


Deforestation/fusion

Back on our example: looking for g

Deforestation/fusion

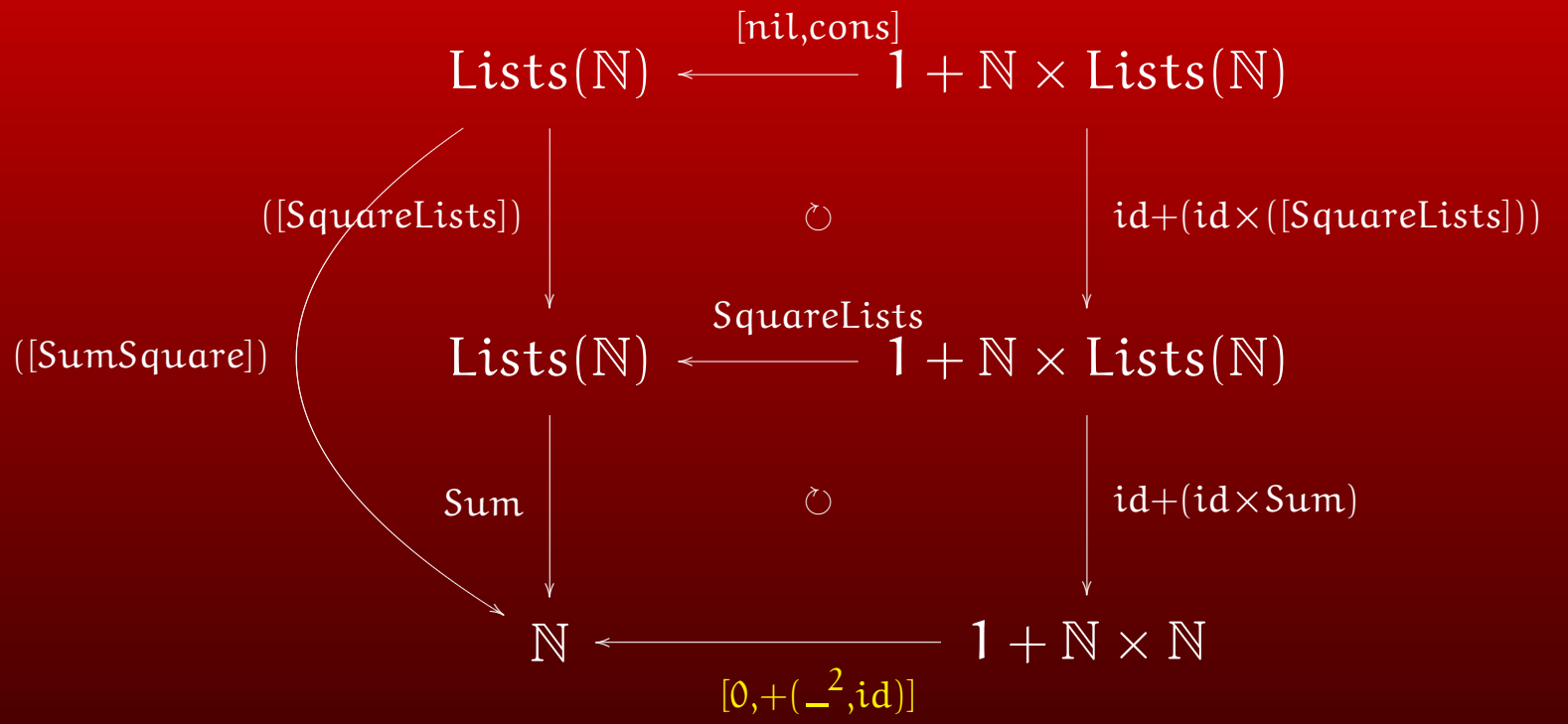
Back on our example: looking for g



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

Deforestation/Fusion

Back on our example



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

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.

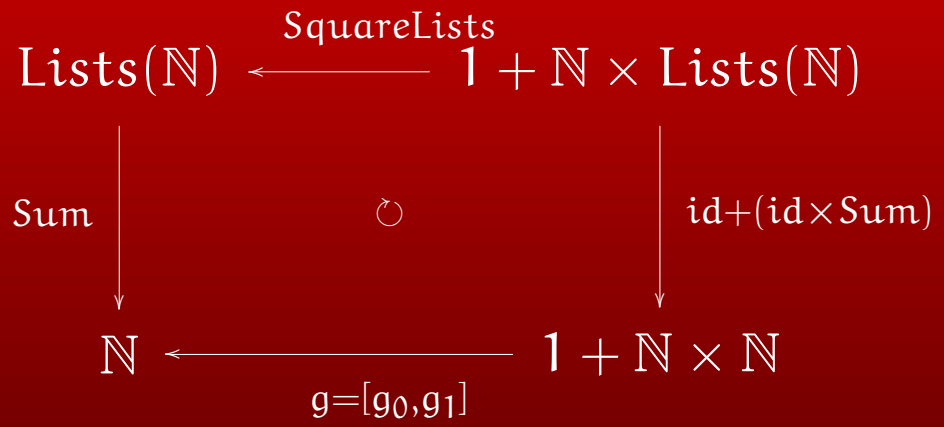
Is there some g such that $h \circ f = g \circ \mathbb{F}h$.?

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

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

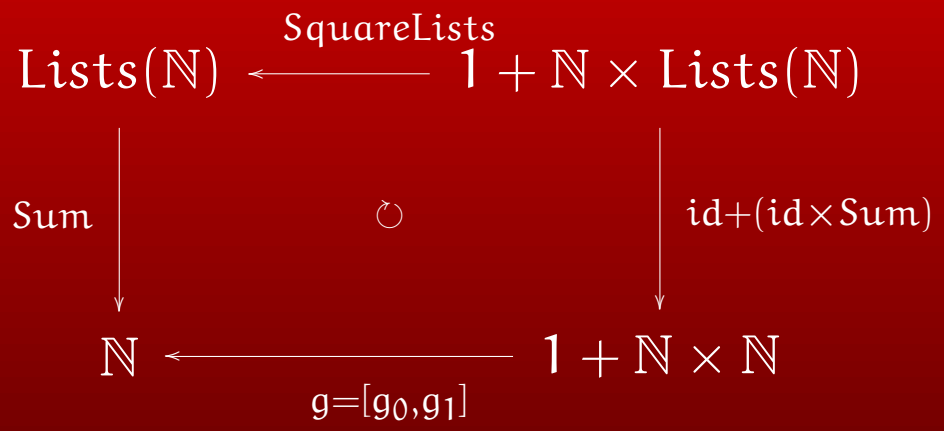
Deforestation/Fusion

From Example:



Deforestation/Fusion

From Example:

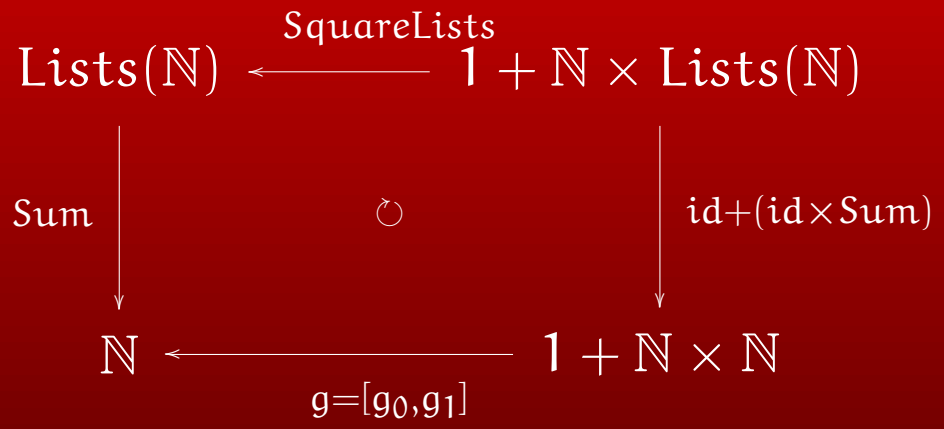


Translated to:

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

Deforestation/Fusion

From Example:



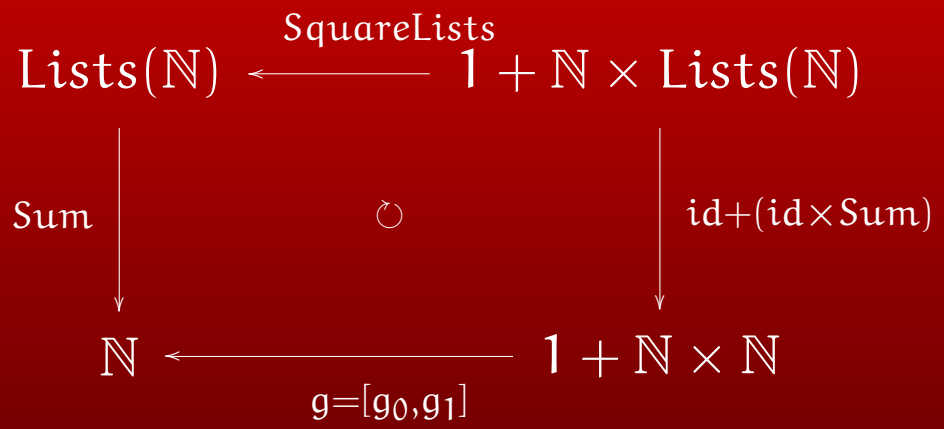
Translated to:

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

`Sum` $\Longrightarrow$ `if (y = nil) 0 +(id , Sum)y`

Deforestation/Fusion

From Example:



Translated to:

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

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

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

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...