The Generalized Intensional Transformation for Implementing Lazy Functional Languages

Georgios Fourtounis\textsuperscript{1}  Nikolaos Papaspyrou\textsuperscript{1}  Panos Rondogiannis\textsuperscript{2}

\textsuperscript{1}National Technical University of Athens  
School of Electrical and Computer Engineering

\textsuperscript{2}University of Athens  
Department of Informatics and Telecommunications

15th International Symposium on  
Practical Aspects of Declarative Languages  
Rome, January 21-22, 2013
Dataflow Programming:

- A program is a directed graph of **data** flowing through a network of **processing units**
- Quite popular in the 1980s due to its implicitly parallel nature

Figure from Joey Paquet’s PhD thesis, “Intensional Scientific Programming” (1999)
Dataflow Programming:

- A program is a directed graph of data flowing through a network of processing units
- Quite popular in the 1980s due to its implicitly parallel nature

Dataflow Languages:

- Mostly functional in nature, encouraging stream processing
- Examples: Val, Id, Lucid, GLU, etc.
Dataflow Programming Languages

**Dataflow Programming:**
- A program is a directed graph of **data** flowing through a network of **processing units**
- Quite popular in the 1980s due to its implicitly parallel nature

**Dataflow Languages:**
- Mostly **functional** in nature, encouraging **stream processing**
- **Examples:** Val, Id, Lucid, GLU, etc.

**Dataflow Machines:**
- **Specialized** parallel architectures for executing dataflow programs, e.g. the MIT Tagged-Token Machine
- Execution is determined by the **availability** of input arguments to operations
The Status of Dataflow

In the 1990s:
- Interest started to decline
- Dataflow architectures could not compete with mainstream
The Status of Dataflow

In the 1990s:
- Interest started to decline
- Dataflow architectures could not compete with mainstream

Today:
- Renewed interest
- Efficient implementation in mainstream multi-core architectures
The Status of Dataflow

In the 1990s:
- Interest started to decline
- Dataflow architectures could not compete with mainstream

Today:
- Renewed interest
- Efficient implementation in mainstream **multi-core** architectures

The Next Day:
- **Map-Reduce**: similarities to Dataflow languages
- A new generation of similar languages/programming models: Dryad, Clustera, Hyrax, etc.
The Intensional Transformation

Alternative technique for implementing \textit{functional languages} by transformation to dataflow programs


Some programming constructs (e.g. full higher-order functions, user-defined data types) are still not satisfactorily handled.
The Original Transformation Algorithm

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

| $\text{result} \quad = \quad f\ 3\ +\ f\ 5$ |
| $f\ x \quad = \quad g\ (x^2)$ |
| $g\ y \quad = \quad y+2$ |
The Original Transformation Algorithm

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>= f 3 + f 5</td>
<td></td>
</tr>
<tr>
<td>f x</td>
<td>= g (x*x)</td>
<td></td>
</tr>
<tr>
<td>g y</td>
<td>= y+2</td>
<td></td>
</tr>
</tbody>
</table>

Step 1: for all functions f

- Replace the $i$th call of $f$ by $\text{call}_i(f)$
- Remove formal parameters from function definitions
The Original Transformation Algorithm

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

result = f 3 + f 5
f x = g (x*x)
g y = y+2

result = call₀(f)+call₁(f)
f = call₀(g)
g = y+2

Step 1: for all functions $f$

- Replace the $i$th call of $f$ by $\text{call}_i(f)$
- Remove formal parameters from function definitions
The Original Transformation Algorithm

The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>result =</td>
<td>result =</td>
</tr>
<tr>
<td>f 3 + f 5</td>
<td>call_0(f)+call_1(f)</td>
</tr>
<tr>
<td>f x = g (x*x)</td>
<td>f = call_0(g)</td>
</tr>
<tr>
<td>g y = y+2</td>
<td>g = y+2</td>
</tr>
</tbody>
</table>

Step 2: for all functions $f$, for all formal parameters $x$

- Find actual parameters corresponding to $x$ in all calls of $f$
- Introduce a new definition for $x$ with an actuals clause, listing the actual parameters in the order of the calls
The input is a first-order functional program. The output is a program with parameterless definitions (intensional program).

### Example

<table>
<thead>
<tr>
<th>result  =  f 3 + f 5</th>
<th>result  =  call₀(f)+call₁(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f x = g (x*x)</td>
<td>f = call₀(g)</td>
</tr>
<tr>
<td>g y = y+2</td>
<td>g = y+2</td>
</tr>
<tr>
<td></td>
<td>x = actuals(3, 5)</td>
</tr>
<tr>
<td></td>
<td>y = actuals(x*x)</td>
</tr>
</tbody>
</table>

### Step 2: for all functions \( f \), for all formal parameters \( x \)

- Find actual parameters corresponding to \( x \) in all calls of \( f \)
- Introduce a new definition for \( x \) with an \texttt{actuals} clause, listing the actual parameters in the order of the calls
**The Semantics of the Target language**

<table>
<thead>
<tr>
<th>Evaluation of expressions: $EVAL(e, w)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intensional</strong>: with respect to a context $w$</td>
</tr>
<tr>
<td>Evaluation contexts are <strong>lists</strong> of natural numbers</td>
</tr>
<tr>
<td>The <strong>initial</strong> context is the empty list</td>
</tr>
</tbody>
</table>
The Semantics of the Target language

Evaluation of expressions: $\text{EVAL}(e, w)$

- **Intensional**: with respect to a context $w$
- Evaluation contexts are lists of natural numbers
- The initial context is the empty list

Context switching: call and actuals

\[
\begin{align*}
\text{EVAL}(\text{call}_i(e), w) & = \text{EVAL}(e, i : w) \\
\text{EVAL}(\text{actuals}(e_0, \ldots, e_{n-1}), i : w) & = \text{EVAL}(e_i, w)
\end{align*}
\]
Example

Evaluation of the target program:

\[
EVAL(result, [ ]) = EVAL(call_0(f) + call_1(f), [ ]) = EVAL(call_0(g), [ ]) + EVAL(call_1(g), [ ]) = EVAL(f, [0]) + EVAL(f, [1]) = EVAL(call_0(g), [0]) + EVAL(call_0(g), [1]) = EVAL(g, [0, 0]) + EVAL(g, [0, 1]) = EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1]) = EVAL(actuals(x*x), [0, 0]) + 2 + EVAL(actuals(x*x), [0, 1]) + 2 = EVAL(x*x, [0]) + 2 + EVAL(x*x, [1]) + 2 = EVAL(x, [0]) \times EVAL(x, [0]) + 2 + EVAL(x, [1]) \times EVAL(x, [1]) + 2 = EVAL(actuals(3, 5), [0]) \times EVAL(actuals(3, 5), [0]) + 2 + EVAL(actuals(3, 5), [1]) \times EVAL(actuals(3, 5), [1]) + 2 = 3 \times 3 + 2 + 5 \times 5 + 2 = 38
\]
Example

Evaluation of the target program:

\[
\text{EVAL}(\text{result}, []) = \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), [])
\]

result = call_0(f)+call_1(f)
f = call_0(g)
g = y+2
x = actuals(3, 5)
y = actuals(x*x)
Example

Evaluation of the target program:

\[
EVAL(\text{result},[]) = EVAL(\text{call}_0(f) + \text{call}_1(f),[]) = EVAL(\text{call}_0(f),[]) + EVAL(\text{call}_1(f),[])
\]
Example

Evaluation of the target program:

\[ \text{EVAL}(\text{result}, []) = \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), []) = \text{EVAL}(\text{call}_0(f), []) + \text{EVAL}(\text{call}_1(f), []) = \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \]

\[
\begin{align*}
\text{result} & = \text{call}_0(f) + \text{call}_1(f) \\
f & = \text{call}_0(g) \\
g & = y + 2 \\
x & = \text{actuals}(3, 5) \\
y & = \text{actuals}(x \times x)
\end{align*}
\]
Example

Evaluation of the target program:

\[
EVAL(result, []) = EVAL(call_0(f) + call_1(f), [])
\]

\[
= EVAL(call_0(f), []) + EVAL(call_1(f), [])
\]

\[
= EVAL(f, [0]) + EVAL(f, [1])
\]

\[
= EVAL(call_0(g), [0]) + EVAL(call_0(g), [1])
\]

result = call_0(f) + call_1(f)
f = call_0(g)
g = y + 2
x = actuals(3, 5)
y = actuals(x * x)
Example

Evaluation of the target program:

\[
\text{EVAL}(\text{result}, []) \\
= \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), []) \\
= \text{EVAL}(\text{call}_0(f), []) + \text{EVAL}(\text{call}_1(f), []) \\
= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \\
= \text{EVAL}(\text{call}_0(g), [0]) + \text{EVAL}(\text{call}_0(g), [1]) \\
= \text{EVAL}(g, [0, 0]) + \text{EVAL}(g, [0, 1])
\]

result = call_0(f) + call_1(f)  
\[
\begin{align*}
\text{f} &= \text{call}_0(g) \\
\text{g} &= y + 2 \\
\text{x} &= \text{actuals}(3, 5) \\
\text{y} &= \text{actuals}(x \times x)
\end{align*}
\]
Example

Evaluation of the target program:

\[
EVAL(\text{result}, \text{[]} )
= EVAL(\text{call}_0(f) + \text{call}_1(f), \text{[]})
= EVAL(\text{call}_0(f), \text{[]}) + EVAL(\text{call}_1(f), \text{[]})
= EVAL(f, \text{[0]}) + EVAL(f, \text{[1]})
= EVAL(\text{call}_0(g), \text{[0]}) + EVAL(\text{call}_0(g), \text{[1]})
= EVAL(g, \text{[0, 0]}) + EVAL(g, \text{[0, 1]})
= EVAL(y, \text{[0, 0]}) + EVAL(2, \text{[0, 0]}) + EVAL(y, \text{[0, 1]}) + EVAL(2, \text{[0, 1]})
\]

\[
\text{result} = \text{call}_0(f) + \text{call}_1(f)
\text{f} = \text{call}_0(g)
\text{g} = y + 2
\text{x} = \text{actuals}(3, 5)
\text{y} = \text{actuals}(x \times x)
\]
Example

Evaluation of the target program:

\[
\begin{align*}
\text{EVAL}(\text{result}, []) &= \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), []) \\
&= \text{EVAL}(\text{call}_0(f), []) + \text{EVAL}(\text{call}_1(f), []) \\
&= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \\
&= \text{EVAL}(\text{call}_0(g), [0]) + \text{EVAL}(\text{call}_0(g), [1]) \\
&= \text{EVAL}(g, [0, 0]) + \text{EVAL}(g, [0, 1]) \\
&= \text{EVAL}(y, [0, 0]) + \text{EVAL}(2, [0, 0]) + \text{EVAL}(y, [0, 1]) + \text{EVAL}(2, [0, 1]) \\
&= \text{EVAL}(\text{actuals}(x \times x), [0, 0]) + 2 + \text{EVAL}(\text{actuals}(x \times x), [0, 1]) + 2
\end{align*}
\]
Example

Evaluation of the target program:

\[
\begin{align*}
EVAL(\text{result},[]) &= EVAL(\text{call}_0(f) + \text{call}_1(f),[]) \\
&= EVAL(\text{call}_0(f),[]) + EVAL(\text{call}_1(f),[]) \\
&= EVAL(f,[0]) + EVAL(f,[1]) \\
&= EVAL(\text{call}_0(g),[0]) + EVAL(\text{call}_0(g),[1]) \\
&= EVAL(g,[0,0]) + EVAL(g,[0,1]) \\
&= EVAL(y,[0,0]) + EVAL(2,[0,0]) + EVAL(y,[0,1]) + EVAL(2,[0,1]) \\
&= EVAL(\text{actuals}(x*x),[0,0]) + 2 + EVAL(\text{actuals}(x*x),[0,1]) + 2 \\
&= EVAL(x*x,[0]) + 2 + EVAL(x*x,[1]) + 2
\end{align*}
\]

result = call\textsubscript{0}(f)+call\textsubscript{1}(f)
f = call\textsubscript{0}(g)
g = y+2
x = actuals(3,5)
y = actuals(x*x)
Example

Evaluation of the target program:

\[
\begin{align*}
EVAL(& \text{result, } []) \\
= & \quad EVAL(\text{call}_0(f) + \text{call}_1(f), []) \\
= & \quad EVAL(\text{call}_0(f), []) + EVAL(\text{call}_1(f), []) \\
= & \quad EVAL(f, [0]) + EVAL(f, [1]) \\
= & \quad EVAL(\text{call}_0(g), [0]) + EVAL(\text{call}_0(g), [1]) \\
= & \quad EVAL(g, [0, 0]) + EVAL(g, [0, 1]) \\
= & \quad EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1]) \\
= & \quad EVAL(\text{actuals}(x\times x), [0, 0]) + 2 + EVAL(\text{actuals}(x\times x), [0, 1]) + 2 \\
= & \quad EVAL(x\times x, [0]) + 2 + EVAL(x\times x, [1]) + 2 \\
= & \quad EVAL(x, [0]) \times EVAL(x, [0]) + 2 + EVAL(x, [1]) \times EVAL(x, [1]) + 2
\end{align*}
\]
Example

Evaluation of the target program:

\[
\begin{align*}
\text{result} &= \text{call}_0(f) + \text{call}_1(f) \\
f &= \text{call}_0(g) \\
g &= y + 2 \\
x &= \text{actuals}(3, 5) \\
y &= \text{actuals}(x \times x)
\end{align*}
\]
Evaluation of the target program:

\[
\begin{align*}
    EVAL(\text{result},[]) &= EVAL(\text{call}_0(f)+\text{call}_1(f),[]) \\
    &= EVAL(\text{call}_0(f),[]) + EVAL(\text{call}_1(f),[]) \\
    &= EVAL(f,[0]) + EVAL(f,[1]) \\
    &= EVAL(\text{call}_0(g),[0]) + EVAL(\text{call}_0(g),[1]) \\
    &= EVAL(g,[0,0]) + EVAL(g,[0,1]) \\
    &= EVAL(y,[0,0]) + EVAL(2,[0,0]) + EVAL(y,[0,1]) + EVAL(2,[0,1]) \\
    &= EVAL(\text{actuals}(x\times x),[0,0]) + 2 + EVAL(\text{actuals}(x\times x),[0,1]) + 2 \\
    &= EVAL(x\times x,[0]) + 2 + EVAL(x\times x,[1]) + 2 \\
    &= EVAL(x,[0]) \times EVAL(x,[0]) + 2 + EVAL(x,[1]) \times EVAL(x,[1]) + 2 \\
    &= EVAL(\text{actuals}(3,5),[0]) \times EVAL(\text{actuals}(3,5),[0]) + 2 + \\
        EVAL(\text{actuals}(3,5),[1]) \times EVAL(\text{actuals}(3,5),[1]) + 2 \\
    &= EVAL(3,[ ]) \times EVAL(3,[ ]) + 2 + EVAL(5,[ ]) \times EVAL(5,[ ]) + 2
\end{align*}
\]
Example

Evaluation of the target program:

\[
EVAL(result, []) = EVAL(call_0(f) + call_1(f), [])
\]

\[
= EVAL(call_0(f), []) + EVAL(call_1(f), [])
\]

\[
= EVAL(f, [0]) + EVAL(f, [1])
\]

\[
= EVAL(call_0(g), [0]) + EVAL(call_0(g), [1])
\]

\[
= EVAL(g, [0, 0]) + EVAL(g, [0, 1])
\]

\[
= EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1])
\]

\[
= EVAL(actuals(x*x), [0, 0]) + 2 + EVAL(actuals(x*x), [0, 1]) + 2
\]

\[
= EVAL(x*x, [0]) + 2 + EVAL(x*x, [1]) + 2
\]

\[
= EVAL(x, [0]) \times EVAL(x, [0]) + 2 + EVAL(x, [1]) \times EVAL(x, [1]) + 2
\]

\[
= EVAL(actuals(3, 5), [0]) \times EVAL(actuals(3, 5), [0]) + 2 + EVAL(actuals(3, 5), [1]) \times EVAL(actuals(3, 5), [1]) + 2
\]

\[
= EVAL(3, []) \times EVAL(3, []) + 2 + EVAL(5, []) \times EVAL(5, []) + 2
\]

\[
= 3 \times 3 + 2 + 5 \times 5 + 2
\]

\[
= 38
\]
Example

Evaluation of the target program:

\[
\begin{align*}
\text{result} &= \text{call}_0(f) + \text{call}_1(f) \\
f &= \text{call}_0(g) \\
g &= y + 2 \\
x &= \text{actuals}(3, 5) \\
y &= \text{actuals}(x \times x)
\end{align*}
\]

\[
\begin{align*}
\text{EVAL}(\text{result}, []) &= \text{EVAL}(\text{call}_0(f) + \text{call}_1(f), []) \\
&= \text{EVAL}(\text{call}_0(f), []) + \text{EVAL}(\text{call}_1(f), []) \\
&= \text{EVAL}(f, [0]) + \text{EVAL}(f, [1]) \\
&= \text{EVAL}(\text{call}_0(g), [0]) + \text{EVAL}(\text{call}_0(g), [1]) \\
&= \text{EVAL}(g, [0, 0]) + \text{EVAL}(g, [0, 1]) \\
&= \text{EVAL}(y, [0, 0]) + \text{EVAL}(2, [0, 0]) + \text{EVAL}(y, [0, 1]) + \text{EVAL}(2, [0, 1]) \\
&= \text{EVAL}(\text{actuals}(x \times x), [0, 0]) + 2 + \text{EVAL}(\text{actuals}(x \times x), [0, 1]) + 2 \\
&= \text{EVAL}(x \times x, [0]) + 2 + \text{EVAL}(x \times x, [1]) + 2 \\
&= \text{EVAL}(x, [0]) \times \text{EVAL}(x, [0]) + 2 + \text{EVAL}(x, [1]) \times \text{EVAL}(x, [1]) + 2 \\
&= \text{EVAL}(\text{actuals}(3, 5), [0]) \times \text{EVAL}(\text{actuals}(3, 5), [0]) + 2 + \text{EVAL}(\text{actuals}(3, 5), [1]) \times \text{EVAL}(\text{actuals}(3, 5), [1]) + 2 \\
&= \text{EVAL}(3, []) \times \text{EVAL}(3, []) + 2 + \text{EVAL}(5, []) \times \text{EVAL}(5, []) + 2 \\
&= 3 \times 3 + 2 + 5 \times 5 + 2 \\
&= 38
\end{align*}
\]
Implementation Issues

Evaluation order: from call-by-name to call-by-need

- Use a **warehouse** to store already computed values
- The warehouse contains triples \((x, w, v)\)
- **Hash-consing** for efficient context comparison
Implementation Issues

Evaluation order: from call-by-name to call-by-need

- Use a **warehouse** to store already computed values
- The warehouse contains triples \((x, w, v)\)
- **Hash-consing** for efficient context comparison

A more efficient memoization: LARs

- **Lazy Activation Record**: corresponds to a context and memoizes a function’s actual parameters
- [Charalambidis, Grivas, Papaspyrou & Rondogiannis, 2008]
  A **stack-based** implementation for a language with a restricted class of higher-order functions
The New Intensional Transformation

Original intensional transformation

- FL$^1$: first-order functional language
- NVIL: zero-order intensional language

Yaghi (1984), Rondogiannis (1997)
The New Intensional Transformation

Higher-order intensional transformation

$\text{FL}^{1}$\quad intensional transformation\quad Yaghi (1984), Rondogiannis (1997)

$\text{FL}^{-n}$\quad Rondogiannis (1999)

NVIL
Higher-order intensional transformation

- Missing: partial application (closures + currying)
The New Intensional Transformation

Higher-order intensional transformation

- Missing: partial application (closures + currying)
- Missing: user defined data types

G. Fourtounis, N. Papaspyrou, P. Rondogiannis

Yaghi (1984), Rondogiannis (1997)

Rondogiannis (1999)
The New Intensional Transformation

Higher-order intensional transformation

- Missing: partial application (closures + currying)
- Missing: user defined data types
Defunctionalization to the rescue

- **FOFL**: first-order functional language, with **data types**
- **HOFL**: higher-order functional language, with **data types** and with **partial application**
The New Intensional Transformation

This work: the missing link

- Similar to the original intensional transformation
- With **data types** in the source and target languages

- defunctionalization
  - Reynolds (1972)

- HOFL

- FOFL

- FL\(^1\)

- FL\(_n\)

- Haskell

- NVIL


- Rondogiannis (1999)

- this work

G. Fourtounis, N. Papaspyrou, P. Rondogiannis
Syntax of FOFL

\[\begin{align*}
p & ::= d_0 \ldots d_n & \text{program} \\
d & ::= f(v_0, \ldots, v_{n-1}) = e & \text{definition} \\
e & ::= \\
& \quad c(e_0, \ldots, e_{n-1}) & \text{expression} \\
& \quad | f(e_0, \ldots, e_{n-1}) \\
& \quad | \kappa(e_0, \ldots, e_{n-1}) \\
& \quad | \text{case } e \text{ of } \{ b_0 ; \ldots ; b_n \} \\
& \quad | \#^m(v) \\
b & ::= \kappa(v_0, \ldots, v_{n-1}) \rightarrow e & \text{case clause}
\end{align*}\]

- \(f\) and \(v\) range over variables, \(c\) ranges over constants, \(\kappa\) ranges over constructors, and \(n, m \geq 0\)
- distinct names for formal parameters
- constructor functions and naming of patterns
Example: Sum of a list’s first two elements

Haskell:

```haskell
f l = case l of
    Nil → 0
    Cons x xs → case xs of
        Nil → x
        Cons y ys → x+y
```
Example: Sum of a list’s first two elements

**Haskell:**

\[
f \ l = \text{case } \ l \ \text{of}
\]
\[
\text{Nil } \rightarrow \ 0
\]
\[
\text{Cons } x \ \text{xs} \rightarrow \text{case } \ \text{xs} \ \text{of}
\]
\[
\text{Nil } \rightarrow \ x
\]
\[
\text{Cons } y \ \text{ys} \rightarrow x + y
\]

**FOFL:**

\[
f(l) = \text{case } l \ \text{of } \{
\]
\[
\text{Nil } \rightarrow 0;
\]
\[
\text{Cons}(h, t) \rightarrow \text{case } \#^0(t) \ \text{of } \{
\]
\[
\text{Nil} \rightarrow \#^1(h);
\]
\[
\text{Cons}(h, t) \rightarrow + (\#^1(h), \#^0(h))
\]
\[
\}
\]
Syntax of NVIL

\[ p ::= d_0 \ldots d_n \]
\[ d ::= f = e \]
\[ e ::= \]
\[ c(e_0, \ldots, e_{n-1}) \]
\[ f \]
\[ \kappa \]
\[ \text{case } e \text{ of } \{ b_0 ; \ldots ; b_n \} \]
\[ \#^m(e) \]
\[ \text{call}_l(e) \]
\[ \text{actuals}(\langle e_l \rangle_{l \in I}) \]
\[ b ::= \kappa \rightarrow e \]

**program**

**definition**

**expression**

- constants and operators
- variables
- constructors
- inspection of data types
- case pattern expressions
- context switching
- context switching

**case clause**

● Technicality: **labels** in contexts, instead of natural numbers
A richer structure for contexts

\[ w ::= \bullet \mid \langle \ell, w, \mu \rangle \]

\[ \mu ::= \bullet \mid w : \mu \]  

(similar to lists with backpointers)
Semantics of NVIL

A richer structure for contexts

\[ w ::= \bullet | \langle \ell, w, \mu \rangle \]
\[ \mu ::= \bullet | w : \mu \]  
(similar to lists with backpointers)

Evaluation function: returns ground value or \( \langle \kappa, w \rangle \)

\[
\begin{align*}
EVAL_p(c(e_0, \ldots, e_{n-1}), w) &= c(EVAL_p(e_0, w), \ldots, EVAL_p(e_{n-1}, w)) \\
EVAL_p(f, w) &= EVAL_p(body(f, p), w) \\
EVAL_p(\kappa, w) &= \langle \kappa, w \rangle \\
EVAL_p(case\ e \ of \ \{ \kappa_0 \rightarrow e_0; \ \ldots; \ \kappa_n \rightarrow e_n \}, \langle \ell, w, \mu \rangle) &= \begin{cases} \\
EVAL_p(e_i, \langle \ell, w, w' : \mu \rangle) & \text{if } EVAL_p(e, \langle \ell, w, \mu \rangle) = \langle \kappa_i, w' \rangle \\
\end{cases} \\
EVAL_p(#^m(e), \langle \ell, w, \mu \rangle) &= EVAL_p(e, \mu_m) \\
EVAL_p(call_\ell(e), w) &= EVAL_p(e, \langle \ell, w, \bullet \rangle) \\
EVAL_p(actuals(\langle e_\ell \rangle_{\ell \in I}), \langle \ell, w, \mu \rangle) &= EVAL_p(e_\ell, w)
\end{align*}
\]
Example: Reversing lists

Haskell

```haskell
data List = Nil | Cons Int List
reverse xs = aux xs Nil
aux xs ys = case xs of
    Nil -> ys
    Cons h t -> aux t (Cons h ys)
```

FOFL

nil = Nil
cons (h, t) = Cons (h, t)
reverse (zs) = aux (zs, nil)
aux (xs, ys) = case xs of
    Nil -> ys
    Cons h t -> aux (t, cons (h, ys))
```
Example: Reversing lists

Haskell

```
data List = Nil | Cons Int List
reverse xs = aux xs Nil
aux xs ys = case xs of
               Nil -> ys
               Cons h t -> aux t (Cons h ys)
```

FOFL

```
nil = Nil
cons(h, t) = Cons(h, t)
```
Example: Reversing lists

Haskell

```haskell
data List = Nil | Cons Int List
reverse xs = aux xs Nil
aux xs ys = case xs of
               Nil -> ys
               Cons h t -> aux t (Cons h ys)
```

FOFL

```
nil = Nil
cons(h, t) = Cons(h, t)
reverse(zs) = aux(zs, nil)
```
Example: Reversing lists

Haskell

data List = Nil | Cons Int List
reverse xs = aux xs Nil
aux xs ys = case xs of
  Nil -> ys
  Cons h t -> aux t (Cons h ys)

FOFL

nil = Nil
cons(h, t) = Cons(h, t)
reverse(zs) = aux(zs, nil)
aux(xs, ys) = case xs of {
  Nil -> ys;
  Cons(h, t) -> aux(#0(t), cons(#0(h), ys))
}
Example: Reversing lists

<table>
<thead>
<tr>
<th>FOFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( nil ) = \text{Nil}</td>
</tr>
<tr>
<td>( \text{cons}(h, t) ) = \text{Cons}(h, t)</td>
</tr>
<tr>
<td>( \text{reverse}(zs) ) = \text{aux}(zs, nil)</td>
</tr>
<tr>
<td>( \text{aux}(xs, ys) ) = \text{case } xs \text{ of}</td>
</tr>
<tr>
<td>\hspace{1cm} Nil \rightarrow ys;</td>
</tr>
<tr>
<td>\hspace{1cm} \text{Cons}(h, t) \rightarrow \text{aux}(#^0(t), \text{cons}(#^0(h), ys))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVIL</th>
</tr>
</thead>
</table>
**Example: Reversing lists (ii)**

### FOFL

\[
\begin{align*}
nil & = Nil \\
\text{cons}(h, t) & = \text{Cons}(h, t) \\
\text{reverse}(zs) & = \text{aux}(zs, nil) \\
\text{aux}(xs, ys) & = \text{case } xs \text{ of} \\
& \quad \text{Nil} \rightarrow ys; \\
& \quad \text{Cons}(h, t) \rightarrow \text{aux}(\#^0(t), \text{cons}(\#^0(h), ys))
\end{align*}
\]

### NVIL

\[
\begin{align*}
nil & = Nil \\
\text{cons} & = \text{Cons} \\
\text{reverse} & = \text{call}_0(\text{aux}) \\
\text{aux} & = \text{case } xs \text{ of} \\
& \quad \text{Nil} \rightarrow ys; \\
& \quad \text{Cons} \rightarrow \text{call}_1(\text{aux})
\end{align*}
\]
### Example: Reversing lists (ii)

**FOFL**

<table>
<thead>
<tr>
<th>expression</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>$\text{Nil}$</td>
</tr>
<tr>
<td>$\text{cons}(h, t)$</td>
<td>$\text{Cons}(h, t)$</td>
</tr>
<tr>
<td>$\text{reverse}(zs)$</td>
<td>$\text{aux}(zs, \text{nil})$</td>
</tr>
<tr>
<td>$\text{aux}(xs, ys)$</td>
<td>$\text{case } xs \text{ of}$</td>
</tr>
<tr>
<td></td>
<td>$\text{Nil} \rightarrow ys;$</td>
</tr>
<tr>
<td></td>
<td>$\text{Cons}(h, t) \rightarrow \text{aux}(#^0(t), \text{cons}(#^0(h), ys))$</td>
</tr>
</tbody>
</table>

**NVIL**

<table>
<thead>
<tr>
<th>expression</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>$\text{Nil}$</td>
</tr>
<tr>
<td>$\text{cons}$</td>
<td>$\text{Cons}$</td>
</tr>
<tr>
<td>$\text{reverse}$</td>
<td>$\text{call}_0(\text{aux})$</td>
</tr>
<tr>
<td>$\text{aux}$</td>
<td>$\text{case } xs \text{ of}$</td>
</tr>
<tr>
<td></td>
<td>$\text{Nil} \rightarrow ys;$</td>
</tr>
<tr>
<td></td>
<td>$\text{Cons} \rightarrow \text{call}_1(\text{aux})$</td>
</tr>
<tr>
<td>$xs$</td>
<td>$\text{actuals}(zs, #^0(t))$</td>
</tr>
<tr>
<td>$ys$</td>
<td>$\text{actuals}(\text{nil}, \text{call}_0(\text{cons}))$</td>
</tr>
</tbody>
</table>
Example: Reversing lists (ii)

### FOFL

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>$\text{Nil}$</td>
</tr>
<tr>
<td>$\text{cons}(h, t)$</td>
<td>$\text{Cons}(h, t)$</td>
</tr>
<tr>
<td>$\text{reverse}(zs)$</td>
<td>$\text{aux}(zs, \text{nil})$</td>
</tr>
<tr>
<td>$\text{aux}(xs, ys)$</td>
<td>\texttt{case xs of} $\begin{align*} \text{Nil} &amp; \rightarrow ys; \ \text{Cons}(h, t) &amp; \rightarrow aux(#^0(t), \text{cons(#^0(h), ys)}) \end{align*}$</td>
</tr>
</tbody>
</table>

### NVIL

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>$\text{Nil}$</td>
</tr>
<tr>
<td>cons</td>
<td>$\text{Cons}$</td>
</tr>
<tr>
<td>$h$</td>
<td>$\text{actuals(#^0(h))}$</td>
</tr>
<tr>
<td>$t$</td>
<td>$\text{actuals(ys)}$</td>
</tr>
<tr>
<td>reverse</td>
<td>$\text{call}_0(aux)$</td>
</tr>
<tr>
<td>aux</td>
<td>\texttt{case xs of} $\begin{align*} \text{Nil} &amp; \rightarrow ys; \ \text{Cons} &amp; \rightarrow \text{call}_1(aux) \end{align*}$</td>
</tr>
<tr>
<td>$xs$</td>
<td>$\text{actuals}(zs, #^0(t))$</td>
</tr>
<tr>
<td>$ys$</td>
<td>$\text{actuals}(\text{nil}, \text{call}_0(\text{cons}))$</td>
</tr>
</tbody>
</table>
 Implementation

http://www.softlab.ntua.gr/~gfour/dftoic/

Key ideas:

- An efficient implementation of $EVAL_p(f, w)$ for each function $f$, written in C
- **Lazy activation records** for call-by-need semantics
- LARs store both function arguments and data objects
Implementation

http://www.softlab.ntua.gr/~gfour/dftoic/

Key ideas:
- An efficient implementation of $EVAL_p(f, w)$ for each function $f$, written in C
- Lazy activation records for call-by-need semantics
- LARs store both function arguments and data objects

Main difference from traditional implementation:
- No closures: they are encoded in contexts
http://www.softlab.ntua.gr/~gfour/dftoic/

**Key ideas:**
- An efficient implementation of $EVAL_p(f, w)$ for each function $f$, written in C
- Lazy activation records for call-by-need semantics
- LARs store both function arguments and data objects

**Main difference from traditional implementation:**
- No closures: they are encoded in contexts

**Optimization:**
- Stack- and heap-allocated LARs
- Aiming to turn our implementation to a back-end for GHC
## Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>GIC</th>
<th>GIC-llvm</th>
<th>GHC7</th>
<th>GHC6</th>
<th>NHC</th>
<th>UHC</th>
<th>JHC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ack</td>
<td>2.47</td>
<td>1.25</td>
<td>0.62</td>
<td>0.48</td>
<td>6.18</td>
<td>40.03</td>
<td>0.05</td>
</tr>
<tr>
<td>church</td>
<td>3.55</td>
<td>2.09</td>
<td>0.61</td>
<td>0.55</td>
<td>11.58</td>
<td>68.37</td>
<td>0.17</td>
</tr>
<tr>
<td>collatz</td>
<td>0.69</td>
<td>0.41</td>
<td>1.07</td>
<td>2.66</td>
<td>84.28</td>
<td>46.90</td>
<td>0.16</td>
</tr>
<tr>
<td>digits_of_e1</td>
<td>2.30</td>
<td>2.09</td>
<td>0.77</td>
<td>1.74</td>
<td>60.71</td>
<td>75.29</td>
<td>0.17</td>
</tr>
<tr>
<td>fast-reverse</td>
<td>3.03</td>
<td>1.95</td>
<td>1.74</td>
<td>1.82</td>
<td>1.35</td>
<td>9.41</td>
<td>–2</td>
</tr>
<tr>
<td>fib</td>
<td>1.35</td>
<td>1.12</td>
<td>0.50</td>
<td>0.51</td>
<td>10.43</td>
<td>55.55</td>
<td>0.17</td>
</tr>
<tr>
<td>naive-reverse</td>
<td>3.02</td>
<td>2.87</td>
<td>0.49</td>
<td>0.42</td>
<td>0.79</td>
<td>3.56</td>
<td>0.75</td>
</tr>
<tr>
<td>ntak</td>
<td>8.62</td>
<td>5.87</td>
<td>2.91</td>
<td>3.65</td>
<td>154.74</td>
<td>91.95</td>
<td>7.18</td>
</tr>
<tr>
<td>primes</td>
<td>2.55</td>
<td>1.58</td>
<td>2.19</td>
<td>2.30</td>
<td>172.45</td>
<td>173.81</td>
<td>0.73</td>
</tr>
<tr>
<td>queens-num</td>
<td>0.33</td>
<td>0.23</td>
<td>0.31</td>
<td>0.33</td>
<td>21.16</td>
<td>12.43</td>
<td>0.14</td>
</tr>
<tr>
<td>queens</td>
<td>3.92</td>
<td>3.24</td>
<td>0.44</td>
<td>0.48</td>
<td>27.17</td>
<td>123.98</td>
<td>0.82</td>
</tr>
<tr>
<td>quick-sort</td>
<td>3.18</td>
<td>2.77</td>
<td>1.92</td>
<td>1.90</td>
<td>1.51</td>
<td>5.42</td>
<td>8.58</td>
</tr>
<tr>
<td>tree-sort</td>
<td>2.19</td>
<td>1.97</td>
<td>0.39</td>
<td>0.33</td>
<td>0.91</td>
<td>6.58</td>
<td>0.72</td>
</tr>
<tr>
<td>GMR$^3$</td>
<td>1.38</td>
<td>1.00</td>
<td>0.51</td>
<td>0.57</td>
<td>7.28</td>
<td>18.49</td>
<td>0.33</td>
</tr>
</tbody>
</table>

1. jhc compilation error, 2. jhc runtime error.
3. Geometric mean of the ratios, compared to GIC-llvm.
Conclusion

What?
- An alternative way to implement higher-order lazy functional languages

How?
- Defunctionalization
- First-order **intensional transformation** with source and target languages extended with user-defined data types

What next?
- Support full Haskell: polymorphism
- Support for separate compilation
- Optimizations, better garbage collection for LARs
- Possibilities for parallelization
Conclusion

What?
- An alternative way to implement higher-order lazy functional languages

How?
- Defunctionalization
- First-order intensional transformation with source and target languages extended with user-defined data types

What next?
- Support full Haskell: polymorphism
- Support for separate compilation
- Optimizations, better garbage collection for LARs
- Possibilities for parallelization
Example: Defunctionalization

Higher-order

\[
\text{result} = \text{inc} \ (\text{add} \ 1) \ 2 + \text{inc} \ \text{sq} \ 3 \\
\text{inc} \ f \ x = f \ (x+1) \\
\text{add} \ a \ b = a+b \\
\text{sq} \ z = z*z 
\]

First-order, defunctionalized

\[
\text{result} = \text{inc} \ (\text{Fadd} \ 1) \ 2 + \text{inc} \ \text{Fsq} \ 3 \\
\text{inc} \ f \ x = \text{apply} \ f \ (x+1) \\
\text{add} \ a \ b = a+b \\
\text{sq} \ z = z*z \\
\text{data} \ \text{Func} = \text{Fadd} \ \text{Int} \mid \text{Fsq} \\
\text{apply} \ \text{cl} \ d = \text{case} \ \text{cl} \ \text{of} \\
\quad \text{Fadd} \ c \rightarrow \text{add} \ c \ d \\
\quad \text{Fsq} \rightarrow \text{sq} \ d 
\]