Implementing Non-Strict Functional Languages with the Generalized Intensional Transformation

Georgios Fourtounis
gfour@softlab.ntua.gr

National Technical University of Athens
School of Electrical and Computer Engineering
What this talk is about

An alternative technique for running Haskell programs using a dataflow formalism
Non-Strict Functional Programming Languages

Functional programming

- Programs are written in declarative style
- $\lambda$-calculus as foundation for semantics/syntax
- Higher-order: functions can take/return other functions

result = map inc [1, 5, 4, 2, 30]
inc a = a + 1
map f ls = case ls of
    [] -> []
    (x : xs) -> (f x) : (map f xs)
Functional programming

- Programs are written in declarative style
- $\lambda$-calculus as foundation for semantics/syntax
- Higher-order: functions can take/return other functions

```haskell
result  = map inc [1, 5, 4, 2, 30]
inc a   = a + 1
map f ls = case ls of
            []      -> []
            (x : xs) -> (f x) : (map f xs)
```
Non-Strict Functional Programming Languages

Non-strictness

- Expressions are not evaluated on the spot, but only \textit{when needed}
- Convenient for handling big/infinite data structures
- Code style becomes more declarative
- Strategies: call-by-name, call-by-need (lazy), etc.
- Examples: Haskell, Clean, R
- Strict languages also add non-strict constructs:
  - Lazy\textless T\textgreater in .NET (C\#, Visual Basic)
  - call-by-name parameters and lazy \texttt{val} in Scala
  - lazy futures in C++11
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 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, SISAL, 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, SISAL, 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
In the 1990s:

- Interest started to decline
- Dataflow architectures could not compete with mainstream uniprocessors (Moore’s law)
The Status of Dataflow

In the 1990s:
- Interest started to decline
- Dataflow architectures could not compete with mainstream uniprocessors (Moore’s law)

Today:
- Renewed interest
- Uniprocessors no longer follow Moore’s law for frequency
- Commodity parallel hardware on the rise
- A new generation of dataflow-esque languages/programming models: Dryad, Cluster, Hyrax, Map-Reduce, etc.
- Efficient implementation in mainstream multi-core architectures and reconfigurable hardware (FPGAs)
Alternative technique for implementing non-strict functional languages by transformation to dataflow programs


Some programming constructs (e.g. full higher-order functions, user-defined data types) were still not satisfactorily handled.
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>$f \ 3 + f \ 5$</td>
</tr>
<tr>
<td>$f \  x$</td>
<td>$g \ (x \times x)$</td>
</tr>
<tr>
<td>$g \  y$</td>
<td>$y + 2$</td>
</tr>
</tbody>
</table>
The Original Transformation Algorithm

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

Example

\[
\begin{align*}
\text{result} & = f \ 3 \ + \ f \ 5 \\
f \ x & = g \ (x*x) \\
g \ y & = y + 2
\end{align*}
\]

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

**Example**

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

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

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

<table>
<thead>
<tr>
<th>result</th>
<th>=</th>
<th>call₀(f) + call₁(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>=</td>
<td>call₀(g)</td>
</tr>
<tr>
<td>g</td>
<td>=</td>
<td>y+2</td>
</tr>
<tr>
<td>x</td>
<td>=</td>
<td>actuals(3, 5)</td>
</tr>
<tr>
<td>y</td>
<td>=</td>
<td>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 `actuals` clause, listing the actual parameters in the order of the calls
### 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
The Semantics of the Target language

Evaluation of expressions: $EVAL(e, w)$

- **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

$$EVAL(call_i(e), w) = EVAL(e, i : w)$$
$$EVAL(actuals(e_0, \ldots, e_{n-1}), i : w) = EVAL(e_i, w)$$
Example

Evaluation of the target program:

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

\[ EVAL(y) = \text{actuals}(x \times x) \]
Example

Evaluation of the target program:

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

Evaluation of the target program:

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

Evaluation of the target program:

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

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)
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)

\[
f = call_0(g), \quad g = y+2, \quad x = actuals(3,5), \quad y = actuals(x*x)
\]
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 = \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*x)
Example

Evaluation of the target program:

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

Evaluation of the target program:

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

Evaluation of the target program:

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

Evaluation of the target program:

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

\[ \text{EVAL(result, [ ])} \]
\[ = \text{EVAL(call}_0(f) + \text{call}_1(f), [ ])} \]
\[ = \text{EVAL(call}_0(f), [ ])} + \text{EVAL(call}_1(f), [ ])} \]
\[ = \text{EVAL(f, [0]) + EVAL(f, [1])} \]
\[ = \text{EVAL(call}_0(g), [0]) + \text{EVAL(call}_0(g), [1])} \]
\[ = \text{EVAL(g, [0, 0]) + EVAL(g, [0, 1])} \]
\[ = \text{EVAL(y, [0, 0]) + EVAL(2, [0, 0]) + EVAL(y, [0, 1]) + EVAL(2, [0, 1])} \]
\[ = \text{EVAL(actuals}(x \times x), [0, 0]) + 2 + \text{EVAL(actuals}(x \times x), [0, 1]) + 2 \]
\[ = \text{EVAL(x \times x, [0]) + 2 + EVAL(x \times x, [1]) + 2 \]
\[ = \text{EVAL(x, [0]) \times EVAL(x, [0]) + 2 + EVAL(x, [1]) \times EVAL(x, [1]) + 2 \]
\[ = \text{EVAL(actuals}(3, 5), [0]) \times EVAL(actuals}(3, 5), [0]) + 2 + \text{EVAL(actuals}(3, 5), [1]) \times EVAL(actuals}(3, 5), [1]) + 2 \]
Evaluation of the target program:

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

Evaluation of the target program:

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

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

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

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

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

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

\[
= \text{EVAL}(\text{actuals}(x\times x), [0, 0]) + 2 + \text{EVAL}(\text{actuals}(x\times x), [0, 1]) + 2
\]

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

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

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

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

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

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

\[
= 38
\]
Example

Evaluation of the target program:

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

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

\[
= \text{EVAL}(g, [0, 0]) + \text{EVAL}(g, [0, 1]) + \text{EVAL}(y, [0, 0]) + \text{EVAL}(2, [0, 0]) + \text{EVAL}(y, [0, 1]) + \text{EVAL}(2, [0, 1])
\]

\[
= \text{EVAL}(\text{actuals}(x \times x), [0, 0]) + 2 + \text{EVAL}(\text{actuals}(x \times x), [0, 1]) + 2 + \text{EVAL}(x, [0]) \times \text{EVAL}(x, [0]) + 2 + \text{EVAL}(x, [1]) \times \text{EVAL}(x, [1]) + 2
\]

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

\[
= 38
\]
Example

\[
\text{result} = f \ 3 \ + \ f \ 5 \\
\text{f} \ x = g \ (x \times x) \\
\text{g} \ y = y + 2
\]

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

1. **User-defined data structures:**

```haskell
data List = Nil | Cons Int List
length ls =
  case ls of
    Nil        → 0
    Cons x xs → 1 + length xs
```
The original intensional transformation lacks:

1. User-defined data structures:
   ```hs
   data List = Nil | Cons Int List
   length ls =
     case ls of
       Nil → 0
       Cons x xs → 1 + length xs
   ```

2. Partial application:
   ```hs
   result = double (add 1) 3
   double f x = f (f x)
   add a b = a + b
   ```
The original intensional transformation lacks:

1. **User-defined data structures:**
   
   ```haskell
data List = Nil | Cons Int List
length ls =
  case ls of
    Nil → 0
    Cons x xs → 1 + length xs
```

2. **Partial application:**
   
   ```haskell
result = double (add 1) 3
double f x = f (f x)
add a b = a + b
```

→ Problem (2) reduced to (1) with **defunctionalization**
Defunctionalization

- Transforms a higher-order program to an equivalent first-order one [Reynolds, 1972]
- Requirement: the language of the target program must support data types with different constructors and pattern matching
- Applicable to both typed and untyped settings
- Defunctionalization can support polymorphism and GADTs [Pottier & Gauthier, 2006]
Defunctionalization: Example

Example:

\[
\begin{align*}
\text{result} &= \text{double } (\text{add } 1) \ 3 \\
\text{double } f \ x &= f \ (f \ x) \\
\text{add } a \ b &= a + b
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Clos} &= \text{Add } \text{Int} \\
\text{result} &= \text{double } (\text{Add } 1) \ 3 \\
\text{double } f \ x &= \text{apply } f \ (\text{apply } f \ x) \\
\text{add } a \ b &= a + b \\
\text{apply } c \ z &= \text{case } c \ \text{of} \\
\quad &\quad \quad \quad \quad \text{Add } n \rightarrow \text{add } n \ z
\end{align*}
\]

Main ideas:

1. represent higher-order expressions (closures) with constructors of a new data type \text{Clos}

2. higher-order expressions are now applied to arguments through a special \text{apply()} function that does pattern matching
After defunctionalization, we now have to solve one problem:

**support user-defined data types with pattern matching**
Syntax of FOFL

- \( p \) ::= \( d \)
- \( d \) ::= \( f(v_0, \ldots, v_{n-1}) = e \)
- \( e \) ::= \( c(e_0, \ldots, e_{n-1}) \)
- \( f(e_0, \ldots, e_{n-1}) = e \)
- \( b \) ::= \( k(v_0, \ldots, v_{n-1}) \rightarrow e \)
- \( \text{program} \)
- \( \text{definition} \)
- \( \text{expression} \)
- \( \text{case clause} \)
- \( \text{inspections of data types} \)
- \( \text{constructor functions and naming of patterns} \)
- \( \text{distinct names for formal parameters} \)

\( f \) and \( v \) range over variables, \( c \) ranges over constructors, and \( n, m \geq 0 \)
Example: Sum of a list’s first two elements

Haskell:

\[
\begin{align*}
f \ l & = \text{case } l \ \text{of} \\
    \text{Nil} & \rightarrow 0 \\
    \text{Cons } x \ xs & \rightarrow \text{case } xs \ \text{of} \\
    \text{Nil} & \rightarrow x \\
    \text{Cons } y \ ys & \rightarrow x+y
\end{align*}
\]
Example: Sum of a list’s first two elements

Haskell:

\[ f \ l = \begin{cases} 
\text{Nil} & \rightarrow 0 \\
\text{Cons } x \ x s & \rightarrow \begin{cases} 
\text{Nil} & \rightarrow x \\
\text{Cons } y \ y s & \rightarrow x + y
\end{cases}
\end{cases} \]

FOFL:

\[ f(l) = \begin{cases} 
N \rightarrow 0; \\
\text{Cons}(h, t) & \rightarrow \begin{cases} 
N \rightarrow \#^1(h); \\
\text{Cons}(h, t) & \rightarrow +(\#^1(h), \#^0(h))
\end{cases}
\end{cases} \]
Syntax of NVIL

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

- **program**
- **definition**
- **expression**
- **constants and operators**
- **variables**
- **constructors**
- **inspection of data types**
- **case pattern expressions**
- **context switching**

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

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

A richer structure for contexts

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

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

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

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

```plaintext
nil = Nil
cons(h, t) = Cons(h, t)
reverse(zs) = aux(zs, nil)
```
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

```fo
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

**FOFL**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{nil}$</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>
</table>
Example: Reversing lists

**FOFL**

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

**NVIL**

\[
\begin{align*}
nil & = Nil \\
\text{cons} & = \text{Cons} \\
\text{reverse} & = \text{call}_{0}(\text{aux}) \\
\text{aux} & = \text{case } xs \text{ of} \\
& \quad \begin{cases} 
Nil & \rightarrow ys; \\
\text{Cons} & \rightarrow \text{call}_{1}(\text{aux}) 
\end{cases}
\end{align*}
\]
**Example: Reversing lists**

#### FOFL

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

#### NVIL

\[
\begin{align*}
\text{nil} & \quad = \quad \text{Nil} \\
\text{cons} & \quad = \quad \text{Cons} \\
\text{reverse} & \quad = \quad \text{call}_0(\text{aux}) \\
\text{aux} & \quad = \quad \text{case } xs \text{ of} \\
& \quad \quad \text{Nil} \rightarrow ys; \\
& \quad \quad \text{Cons} \rightarrow \text{call}_1(\text{aux}) \\
xs & \quad = \quad \text{actuals}(zs, \#^0(t)) \\
ys & \quad = \quad \text{actuals}(\text{nil}, \text{call}_0(\text{cons}))
\end{align*}
\]
Example: Reversing lists

### FOFL

<table>
<thead>
<tr>
<th>Symbol</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>
</tbody>
</table>
| $\text{aux}(xs, ys)$ | $\text{case } xs \text{ of}$
| | $\text{Nil} \to ys$;
| | $\text{Cons}(h, t) \to \text{aux} (^0(t), \text{cons} (^0(h), ys))$ |

### NVIL

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>$\text{Nil}$</td>
</tr>
<tr>
<td>$\text{cons} = \text{Cons}$</td>
<td></td>
</tr>
<tr>
<td>$h = \text{actuands}(^0(h))$</td>
<td></td>
</tr>
<tr>
<td>$t = \text{actuands}(ys)$</td>
<td></td>
</tr>
<tr>
<td>$\text{reverse} = \text{call}_0(\text{aux})$</td>
<td></td>
</tr>
</tbody>
</table>
| $\text{aux} = \text{case } xs \text{ of}$
| | $\text{Nil} \to ys$;
| | $\text{Cons} \to \text{call}_1(\text{aux})$ |
| $xs = \text{actuands}(zs, ^0(t))$ |
| $ys = \text{actuands}(\text{nil}, \text{call}_0(\text{cons}))$ |
Implementation Using a Warehouse

- Similar to other intensional techniques
- Uses a **context allocator** to represent contexts
- Interpreter prototype: https://github.com/gfour/gic
Implementation Using Lazy Activation Records

https://github.com/gfour/gic

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 Using Lazy Activation Records**

https://github.com/gfour/gic

<table>
<thead>
<tr>
<th>Key ideas:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- An efficient implementation of $EVAL_p(f, w)$ for each function $f$, written in C</td>
</tr>
<tr>
<td>- <strong>Lazy activation records</strong> for call-by-need semantics</td>
</tr>
<tr>
<td>- LARs store both function arguments and data objects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main difference from traditional implementation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- No <strong>closures</strong>: they are encoded in <strong>contexts</strong></td>
</tr>
</tbody>
</table>
Implementation Using Lazy Activation Records

https://github.com/gfour/gic

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
- Minimal sharing analysis to make some formals call-by-name
- Compact memory representation (on AMD64)
Lazy Activation Records

\[
f \ x \ y = \text{case } x \ of \\
  [] \rightarrow [1] \\
a:as \rightarrow [a + y]
\]
Compact Memory Representation

AMD64 pointers contain redundancy:

![Diagram showing AMD64 pointer structure with redundant 47th bit.](image)

- The 47th bit is redundant and can be ignored.
- The pointer body follows the 47th bit.
Compact Memory Representation

AMD64 pointers contain redundancy:

We use a variation of the **tagged pointers** technique
Thunks on AMD64

**Unevaluated thunk**

![Diagram of an unevaluated thunk showing the code pointer and constructor ID fields.](https://example.com/diagram)

- **Code pointer**: $0010$
- **Constructor ID**: $00$
- **Primitive value**: $3263$
### Benchmarks: Runtime

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ghc-7.6.3</td>
<td></td>
</tr>
<tr>
<td>gic/clang-3.3</td>
<td></td>
</tr>
<tr>
<td>gic/gcc-4.7.2</td>
<td></td>
</tr>
<tr>
<td>gic/icc-14.0.2</td>
<td></td>
</tr>
</tbody>
</table>

**Graph**

- **GHC-7.6.3** [-O3]
- **gic/clang-3.3** [-O3]
- **gic/gcc-4.7.2** [-O3]
- **gic/icc-14.0.2** [-fast]
## Benchmarks: Cache Behavior

<table>
<thead>
<tr>
<th></th>
<th>GHC</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I1</td>
<td>LLI</td>
<td>D1</td>
<td>LLd</td>
<td>LL</td>
<td>I1</td>
<td>LLI</td>
<td>D1</td>
<td>LLd</td>
<td>LL</td>
<td></td>
</tr>
<tr>
<td>ack</td>
<td>0.0</td>
<td>0.0</td>
<td>6.1</td>
<td>2.1</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
<td>10.8</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>church</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>collatz</td>
<td>0.0</td>
<td>0.0</td>
<td>0.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>digits_of_e1</td>
<td>0.0</td>
<td>0.0</td>
<td>3.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.3</td>
<td>0.8</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>fib</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>ntak</td>
<td>0.0</td>
<td>0.0</td>
<td>1.6</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.9</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>primes</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>queens</td>
<td>0.0</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.4</td>
<td>0.3</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>queens-num</td>
<td>0.0</td>
<td>0.0</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>quick-sort</td>
<td>0.0</td>
<td>0.0</td>
<td>4.2</td>
<td>0.5</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>8.2</td>
<td>1.7</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>0.0</td>
<td>0.0</td>
<td>8.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>15.0</td>
<td>3.6</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>tree-sort</td>
<td>0.0</td>
<td>0.0</td>
<td>7.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>7.9</td>
<td>1.6</td>
<td>0.4</td>
<td></td>
</tr>
</tbody>
</table>

**Figure**: Cache miss rates reported by Cachegrind (%). I1: first-level instruction cache. LLI: last-level instruction cache. D1: first-level data cache. LLd: last-level data cache. LL: last-level combined cache. Zeroes are shown as greyed out values.
The Need for Separate Compilation

Realistic compilers must be able to:

- **Efficiently recompile** big programs after source code changes
- Compile parts of programs to reusable libraries

Problem:
The generalized intensional transformation and defunctionalization have been given as whole-program transformations
- Modularity mechanism: Haskell-style modules
- Two-step process: separate compilation and linking
module Lib where
  high g x = g x
  h y = y + 1
  test = high h 1
  add a b = a + b

module Main where
import Lib (h :: Int->Int,
           high :: (Int->Int)->Int->Int,
           test :: Int,
           add :: Int->Int->Int )

result = f 10 + test ;
f a = a + high (add 1) +
     high dec 2
high g = g 10
dec x = x - 1
module Lib where
    Lib.high g x = g x
    Lib.h y = y + 1
    Lib.test = Lib.high Lib.h 1
    Lib.add a b = a + b

module Main where
    import Lib (Lib.h :: Int->Int,
                Lib.high :: (Int->Int)->Int->Int,
                Lib.test :: Int,
                Lib.add :: Int->Int->Int )
    Main.result = Main.f 10 + Lib.test ;
    Main.f a = a + Main.high (Lib.add 1) +
                Lib.high Main.dec 2
    Main.high g = g 10
    Main.dec x = x - 1
Modular Defunctionalization

Separate defunctionalization (HOFL→FOFL):

The module is defunctionalized:
- partial applications are replaced by constructor function calls
- keeps information about the module’s partial applications
  (defunctionalization interface, DFI)
- the apply() and constructor wrapper functions are not generated

Linking:

Missing constructor functions and apply() are generated by
reading all the DFIs
Separate intensional transformation (FOFL→NVIL):

May generate actuals for formals of other modules:
- Needs function signatures for external functions
- Intensional indices are qualified: \texttt{call(i)} becomes \texttt{call(Module, i)}
- Formals are qualified

Linking:
- May use defunctionalization’s linking step
- actuals of the same formals are merged
- Function definitions are just concatenated
**Modular Compilation to C**

<table>
<thead>
<tr>
<th>Separate compilation to C (NVIL→C):</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The NVIL code of the module is translated to C using LARs</td>
</tr>
<tr>
<td>- External symbols declared as <code>extern</code></td>
</tr>
<tr>
<td>- Generates object file <code>Module.o</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linking:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Uses the intensional linking step</td>
</tr>
<tr>
<td>- System linker (<code>ld</code>) links the object files</td>
</tr>
</tbody>
</table>
Conclusion

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

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

What next?

- Support more Haskell syntax in the front-end: type classes, pattern compilation, list comprehensions
- Evaluation as a GHC back-end
- Optimizations, e.g. strictness analysis, tail-call optimization
- Further investigation of the intensional transformation:
  - Machine-checked proof
  - Support for let, tail-recursion
- Possibilities for parallelization:
  - Work-in-progress: OpenMP-based prototype for shared-memory multicores
  - Hardware compilation for reconfigurable hardware
Thank you!
Publications

Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis.
The intensional transformation for functional languages with user-defined data types.

Georgios Fourtounis, Peter Csaba Ölveczky, and Nikolaos Papaspyrou.
Formally specifying and analyzing a parallel virtual machine for lazy functional languages using Maude.

Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis.
The generalized intensional transformation for implementing lazy functional languages.

Georgios Fourtounis and Nikolaos S. Papaspyrou.
Supporting separate compilation in a defunctionalizing compiler.

Georgios Fourtounis, Nikolaos Papaspyrou, and Panagiotis Theofilopoulos.
Modular polymorphic defunctionalization.
*Computer Science and Information Systems.*
Accepted for publication, to appear.

Georgios Fourtounis and Nikolaos Papaspyrou.
An efficient representation for lazy constructors using 64-bit pointers.
Accepted for presentation, to appear.