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
- λ-calculus as foundation for semantics/syntax
- Higher-order: functions can take/return other functions

Non-Strict Functional Programming Languages

Functional programming

- Programs are written in declarative style
- λ-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)
```

Non-Strict Functional Programming Languages

Non-strictness

- Expressions are not evaluated on the spot, but only **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<T> in .NET (C#, Visual Basic)
  - call-by-name parameters and `lazy val` in Scala
  - lazy futures in C++11

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

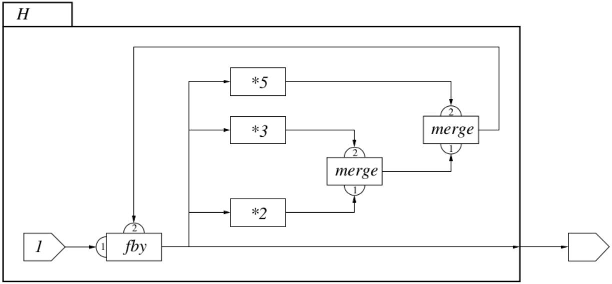


Figure from Joey Paquet's PhD thesis, "Intensional Scientific Programming" (1999)

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

The Status of Dataflow

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, Clustera, Hyrax, Map-Reduce, etc.
- Efficient implementation in mainstream **multi-core** architectures and reconfigurable hardware (FPGAs)

The Intensional Transformation

Alternative technique for implementing **non-strict functional languages** by transformation to dataflow programs

- [Yaghi, 1984] The intensional implementation technique for functional languages.
- [Arvind & Nikhil, 1990] The “coloring” technique for implementing functions on the MIT Dataflow Machine.
- [Rondogiannis & Wadge, 1997, 1999] A formalization of the intensional transformation and its extension for a class of higher-order programs.

Some programming constructs (e.g. full higher-order functions, user-defined data types) were 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

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

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

Step 1: for all functions f

- Replace the i-th call of f by $\texttt{call}_i(f)$
- 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   =  call_0(f)+call_1(f)
f        =  call_0(g)
g        =  y+2
```

Step 1: for all functions f

- Replace the i-th call of f by $\texttt{call}_i(f)$
- 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
```

$$\begin{array}{lll}\texttt{result} & = & \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)}\\ \texttt{f} & = & \texttt{call}_0\texttt{(g)}\\ \texttt{g} & = & \texttt{y+2}\end{array}$$

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

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: $EVAL(e, w)$

- **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: $\mathit{EVAL}(e, w)$

- **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`

$$\begin{aligned}
\mathit{EVAL}(\mathtt{call}_i(e), w) &= \mathit{EVAL}(e, i : w) \\
\mathit{EVAL}(\mathtt{actuals}(e_0, \ldots, e_{n-1}), i : w) &= \mathit{EVAL}(e_i, w)
\end{aligned}$$

```
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:

$$\mathit{EVAL}(\texttt{result},[\,])$$

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,])
\end{aligned}
$$

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$\begin{aligned} & \mathit{EVAL}(\texttt{result}, [\,]) \\ = \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\ = \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \end{aligned}$$

```
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:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result},[\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)},[\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)},[\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)},[\,]) \\
= \; & \mathit{EVAL}(\texttt{f},[0]) + \mathit{EVAL}(\texttt{f},[1])
\end{aligned}
$$

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1])
\end{aligned}
$$

```
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:

$$\begin{aligned} & \mathit{EVAL}(\texttt{result}, [\,]) \\ = \ & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\ = \ & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\ = \ & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\ = \ & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\ = \ & \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \end{aligned}$$

Example

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Evaluation of the target program:

$$\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\mathrm{g}, [0,0]) + \mathit{EVAL}(\mathrm{g}, [0,1]) \\
= \;& \mathit{EVAL}(\mathrm{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\mathrm{y}, [0,1]) + \mathit{EVAL}(2, [0,1])
\end{aligned}$$

Example

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Evaluation of the target program:

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

Example

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3,5)
y      = actuals(x*x)
```

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \; & \mathit{EVAL}(\mathrm{g}, [0,0]) + \mathit{EVAL}(\mathrm{g}, [0,1]) \\
= \; & \mathit{EVAL}(\mathrm{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\mathrm{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \; & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \; & \mathit{EVAL}(\mathrm{x*x}, [0]) + 2 + \mathit{EVAL}(\mathrm{x*x}, [1]) + 2
\end{aligned}
$$

```
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:

$$\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \; & \mathit{EVAL}(\mathrm{g}, [0,0]) + \mathit{EVAL}(\mathrm{g}, [0,1]) \\
= \; & \mathit{EVAL}(\mathrm{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\mathrm{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \; & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \; & \mathit{EVAL}(\mathrm{x*x}, [0]) + 2 + \mathit{EVAL}(\mathrm{x*x}, [1]) + 2 \\
= \; & \mathit{EVAL}(\mathrm{x}, [0]) * \mathit{EVAL}(\mathrm{x}, [0]) + 2 + \mathit{EVAL}(\mathrm{x}, [1]) * \mathit{EVAL}(\mathrm{x}, [1]) + 2
\end{aligned}$$

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \; & \mathit{EVAL}(\mathrm{g}, [0,0]) + \mathit{EVAL}(\mathrm{g}, [0,1]) \\
= \; & \mathit{EVAL}(\mathrm{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\mathrm{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \; & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \; & \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= \; & \mathit{EVAL}(\mathrm{x}, [0]) * \mathit{EVAL}(\mathrm{x}, [0]) + 2 + \mathit{EVAL}(\mathrm{x}, [1]) * \mathit{EVAL}(\mathrm{x}, [1]) + 2 \\
= \; & \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 + \\
& \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2
\end{aligned}
$$

Example

```
result = call₀(f)+call₁(f)
f      = call₀(g)
g      = y+2
x      = actuals(3, 5)
y      = actuals(x*x)
```

Evaluation of the target program:

$$\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \;& \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \;& \mathit{EVAL}(\mathrm{g}, [0,0]) + \mathit{EVAL}(\mathrm{g}, [0,1]) \\
= \;& \mathit{EVAL}(\mathrm{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\mathrm{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \;& \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\mathrm{x}, [0]) * \mathit{EVAL}(\mathrm{x}, [0]) + 2 + \mathit{EVAL}(\mathrm{x}, [1]) * \mathit{EVAL}(\mathrm{x}, [1]) + 2 \\
= \;& \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 + \\
& \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2 \\
= \;& \mathit{EVAL}(\mathsf{3}, [\,]) * \mathit{EVAL}(\mathsf{3}, [\,]) + 2 + \mathit{EVAL}(\mathsf{5}, [\,]) * \mathit{EVAL}(\mathsf{5}, [\,]) + 2
\end{aligned}$$

$$
\begin{aligned}
&\texttt{result} &&= \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)}\\
&\texttt{f} &&= \texttt{call}_0\texttt{(g)}\\
&\texttt{g} &&= \texttt{y+2}\\
&\texttt{x} &&= \texttt{actuals(3, 5)}\\
&\texttt{y} &&= \texttt{actuals(x*x)}
\end{aligned}
$$

Example

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,])\\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,])\\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,])\\
= \; & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1])\\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1])\\
= \; & \mathit{EVAL}(\texttt{g}, [0, 0]) + \mathit{EVAL}(\texttt{g}, [0, 1])\\
= \; & \mathit{EVAL}(\texttt{y}, [0, 0]) + \mathit{EVAL}(\texttt{2}, [0, 0]) + \mathit{EVAL}(\texttt{y}, [0, 1]) + \mathit{EVAL}(\texttt{2}, [0, 1])\\
= \; & \mathit{EVAL}(\texttt{actuals(x*x)}, [0, 0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0, 1]) + 2\\
= \; & \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2\\
= \; & \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2\\
= \; & \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) + 2 +\\
& \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) + 2\\
= \; & \mathit{EVAL}(\texttt{3}, [\,]) * \mathit{EVAL}(\texttt{3}, [\,]) + 2 + \mathit{EVAL}(\texttt{5}, [\,]) * \mathit{EVAL}(\texttt{5}, [\,]) + 2\\
= \; & 3 * 3 + 2 + 5 * 5 + 2
\end{aligned}
$$

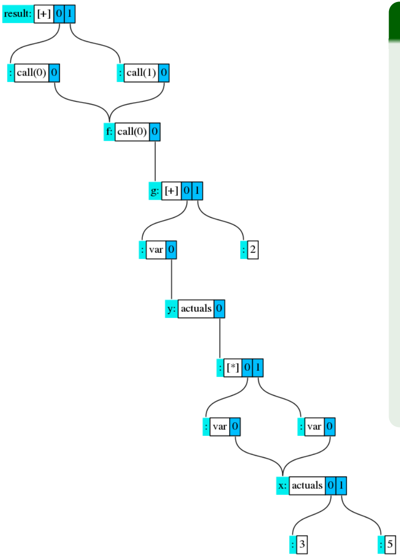
$$\begin{aligned}
&\texttt{result} &&= \texttt{call}_0\texttt{(f)+call}_1\texttt{(f)}\\
&\texttt{f} &&= \texttt{call}_0\texttt{(g)}\\
&\texttt{g} &&= \texttt{y+2}\\
&\texttt{x} &&= \texttt{actuals(3,5)}\\
&\texttt{y} &&= \texttt{actuals(x*x)}
\end{aligned}$$

Example

Evaluation of the target program:

$$\begin{aligned}
& \mathit{EVAL}(\texttt{result}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(f)}, [\,]) + \mathit{EVAL}(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \; & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= \; & \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [0]) + \mathit{EVAL}(\texttt{call}_0\texttt{(g)}, [1]) \\
= \; & \mathit{EVAL}(\texttt{g}, [0,0]) + \mathit{EVAL}(\texttt{g}, [0,1]) \\
= \; & \mathit{EVAL}(\texttt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\texttt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \; & \mathit{EVAL}(\texttt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= \; & \mathit{EVAL}(\texttt{x*x}, [0]) + 2 + \mathit{EVAL}(\texttt{x*x}, [1]) + 2 \\
= \; & \mathit{EVAL}(\texttt{x}, [0]) * \mathit{EVAL}(\texttt{x}, [0]) + 2 + \mathit{EVAL}(\texttt{x}, [1]) * \mathit{EVAL}(\texttt{x}, [1]) + 2 \\
= \; & \mathit{EVAL}(\texttt{actuals(3,5)}, [0]) * \mathit{EVAL}(\texttt{actuals(3,5)}, [0]) + 2 + \\
& \mathit{EVAL}(\texttt{actuals(3,5)}, [1]) * \mathit{EVAL}(\texttt{actuals(3,5)}, [1]) + 2 \\
= \; & \mathit{EVAL}(\texttt{3}, [\,]) * \mathit{EVAL}(\texttt{3}, [\,]) + 2 + \mathit{EVAL}(\texttt{5}, [\,]) * \mathit{EVAL}(\texttt{5}, [\,]) + 2 \\
= \; & 3 * 3 + 2 + 5 * 5 + 2 \\
= \; & 38
\end{aligned}$$

Dataflow Graph



Example

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

$$\begin{array}{lcl} \mathtt{result} & = & \mathtt{call}_0\mathtt{(f)+call}_1\mathtt{(f)} \\ \mathtt{f} & = & \mathtt{call}_0\mathtt{(g)} \\ \mathtt{g} & = & \mathtt{y+2} \\ \mathtt{x} & = & \mathtt{actuals(3,5)} \\ \mathtt{y} & = & \mathtt{actuals(x*x)} \end{array}$$

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

Missing Pieces

The original intensional transformation lacks:

1. User-defined data structures:

```
data List = Nil | Cons Int List
length ls =
  case ls of
    Nil       → 0
    Cons x xs → 1 + length xs
```

Missing Pieces

The original intensional transformation lacks:

1. User-defined data structures:

```
data List = Nil | Cons Int List
length ls =
  case ls of
    Nil       → 0
    Cons x xs → 1 + length xs
```

2. Partial application:

```
result     = double (add 1) 3
double f x = f (f x)
add a b    = a + b
```

Missing Pieces

The original intensional transformation lacks:

1. User-defined data structures:

```
data List = Nil | Cons Int List
length ls =
  case ls of
    Nil        → 0
    Cons x xs  → 1 + length xs
```

2. Partial application:

```
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:

```
result       = double (add 1) 3
double f x = f (f x)
add a b      = a + b
```

```
data Clos    = Add Int
result       = double (Add 1) 3
double f x = apply f (apply f x)
add a b      = a + b
apply c z    = case c of
                   Add n → add n z
```

Main ideas:

1. represent higher-order expressions (closures) with constructors of a new data type `Clos`
2. higher-order expressions are now applied to arguments through a special **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	$::=$	$d_0 \ldots d_n$	**program**
d	$::=$	$f(v_0, \ldots, v_{n-1}) = e$	**definition**
e	$::=$		**expression**
		$c(e_0, \ldots, e_{n-1})$	constants and operators
	$\mid$	$f(e_0, \ldots, e_{n-1})$	variables and functions
	$\mid$	$\kappa(e_0, \ldots, e_{n-1})$	constructors
	$\mid$	$\texttt{case}\ e\ \texttt{of}\ \{\ b_0\ ;\ \ldots\ ;\ b_n\ \}$	inspection of data types
	$\mid$	$\#^m(v)$	case pattern variables
b	$::=$	$\kappa(v_0, \ldots, v_{n-1}) \rightarrow e$	**case clause**

- f and v range over variables, c ranges over constants, κ 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:

```
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 = case l of
    Nil → 0
    Cons x xs → case xs of
                 Nil → x
                 Cons y ys → x+y
```

FOFL:

$$
\begin{array}{lcl}
f(l) & = & \mathsf{case}\ l\ \mathsf{of}\ \{ \\
 & & \quad \mathit{Nil} \rightarrow 0; \\
 & & \quad \mathit{Cons}(h,t) \ \rightarrow \mathsf{case}\ \#^{0}(t)\ \mathsf{of}\ \{ \\
 & & \qquad\qquad\qquad \mathit{Nil} \rightarrow \#^{1}(h); \\
 & & \qquad\qquad\qquad \mathit{Cons}(h,t) \rightarrow +(\#^{1}(h), \#^{0}(h)) \\
 & & \qquad\qquad\quad \} \\
 & & \}
\end{array}
$$

Syntax of NVIL

p	$::=$	$d_0 \ldots d_n$	**program**
d	$::=$	$f = e$	**definition**
e	$::=$		**expression**
		$c(e_0, \ldots, e_{n-1})$	constants and operators
	$\vert$	f	variables
	$\vert$	κ	constructors
	$\vert$	$\texttt{case}\ e\ \texttt{of}\ \{\ b_0\ \texttt{;}\ \ldots\ \texttt{;}\ b_n\ \}$	inspection of data types
	$\vert$	$\texttt{\#}^m(e)$	case pattern expressions
	$\vert$	$\texttt{call}_l(e)$	context switching
	$\vert$	$\texttt{actuals}(\langle e_l \rangle_{l \in I})$	context switching
b	$::=$	$\kappa \to e$	**case clause**

- Technicality: **labels** in contexts, instead of natural numbers

Semantics of NVIL

A richer structure for contexts

$$\begin{aligned} w &::= \bullet \mid \langle \ell, w, \mu \rangle \\ \mu &::= \bullet \mid w : \mu \end{aligned}$$

Semantics of NVIL

A richer structure for contexts

$$
\begin{aligned}
w &::= \bullet \mid \langle \ell, w, \mu \rangle \\
\mu &::= \bullet \mid w : \mu
\end{aligned}
$$

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

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

Example: Reversing lists (i)

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 (i)

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

$$\begin{aligned} nil &= Nil \\ cons(h,t) &= Cons(h,t) \end{aligned}$$

Example: Reversing lists (i)

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

$$\begin{aligned} nil &= Nil \\ cons(h,t) &= Cons(h,t) \\ reverse(zs) &= aux(zs, nil) \end{aligned}$$

Example: Reversing lists (i)

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

$$
\begin{array}{lcl}
nil & = & Nil \\
cons(h,t) & = & Cons(h,t) \\
reverse(zs) & = & aux(zs, nil) \\
aux(xs, ys) & = & \texttt{case}\ xs\ \texttt{of}\ \{ \\
 & & \quad Nil \rightarrow ys; \\
 & & \quad Cons(h,t) \ \rightarrow\ aux(\#^0(t), cons(\#^0(h), ys)) \\
 & & \}
\end{array}
$$

Example: Reversing lists (ii)

FOFL

$$\begin{array}{lcl} nil & = & Nil \\ cons(h,t) & = & Cons(h,t) \\ reverse(zs) & = & aux(zs, nil) \\ aux(xs, ys) & = & \texttt{case } xs \texttt{ of} \\ & & \quad Nil \rightarrow ys; \\ & & \quad Cons(h,t) \ \rightarrow \ aux(\texttt{\#}^0(t), cons(\texttt{\#}^0(h), ys)) \end{array}$$

NVIL

Example: Reversing lists (ii)

FOFL

$$\begin{array}{lcl}
nil & = & Nil \\
cons(h,t) & = & Cons(h,t) \\
reverse(zs) & = & aux(zs, nil) \\
aux(xs, ys) & = & \texttt{case}\ xs\ \texttt{of} \\
 & & \quad Nil \to ys; \\
 & & \quad Cons(h,t)\ \to\ aux(\#^0(t), cons(\#^0(h), ys))
\end{array}$$

NVIL

$$\begin{array}{lcl}
nil & = & Nil \\
cons & = & Cons
\end{array}$$

$$\begin{array}{lcl}
reverse & = & \texttt{call}_0(aux) \\
aux & = & \texttt{case}\ xs\ \texttt{of} \\
 & & \quad Nil \to ys; \\
 & & \quad Cons\ \to\ \texttt{call}_1(aux)
\end{array}$$

Example: Reversing lists (ii)

FOFL

$$\begin{array}{lcl} nil & = & Nil \\ cons(h,t) & = & Cons(h,t) \\ reverse(zs) & = & aux(zs, nil) \\ aux(xs, ys) & = & \texttt{case}\ xs\ \texttt{of} \\ & & \qquad Nil \to ys; \\ & & \qquad Cons(h,t)\ \to\ aux(\#^0(t), cons(\#^0(h), ys)) \end{array}$$

NVIL

$$\begin{array}{lcl|lcl} nil & = & Nil & reverse & = & \texttt{call}_0(aux) \\ cons & = & Cons & aux & = & \texttt{case}\ xs\ \texttt{of} \\ & & & & & \qquad Nil \to ys; \\ & & & & & \qquad Cons\ \to\ \texttt{call}_1(aux) \\ & & & xs & = & \texttt{actuals}(zs, \#^0(t)) \\ & & & ys & = & \texttt{actuals}(nil, \texttt{call}_0(cons)) \end{array}$$

Example: Reversing lists (ii)

FOFL

$$\begin{array}{lcl}
\mathit{nil} & = & \mathit{Nil} \\
\mathit{cons}(h,t) & = & \mathit{Cons}(h,t) \\
\mathit{reverse}(\mathit{zs}) & = & \mathit{aux}(\mathit{zs}, \mathit{nil}) \\
\mathit{aux}(\mathit{xs}, \mathit{ys}) & = & \texttt{case}\ \mathit{xs}\ \texttt{of} \\
 & & \qquad \mathit{Nil} \to \mathit{ys}; \\
 & & \qquad \mathit{Cons}(h,t) \ \to \ \mathit{aux}(\texttt{\#}^0(t), \mathit{cons}(\texttt{\#}^0(h), \mathit{ys}))
\end{array}$$

NVIL

$$\begin{array}{lcl}
\mathit{nil} & = & \mathit{Nil} \\
\mathit{cons} & = & \mathit{Cons} \\
h & = & \texttt{actuals}(\texttt{\#}^0(h)) \\
t & = & \texttt{actuals}(\mathit{ys})
\end{array}$$

$$\begin{array}{lcl}
\mathit{reverse} & = & \texttt{call}_0(\mathit{aux}) \\
\mathit{aux} & = & \texttt{case}\ \mathit{xs}\ \texttt{of} \\
 & & \qquad \mathit{Nil} \to \mathit{ys}; \\
 & & \qquad \mathit{Cons} \ \to \ \texttt{call}_1(\mathit{aux}) \\
\mathit{xs} & = & \texttt{actuals}(\mathit{zs}, \texttt{\#}^0(t)) \\
\mathit{ys} & = & \texttt{actuals}(\mathit{nil}, \texttt{call}_0(\mathit{cons}))
\end{array}$$

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 $\mathit{EVAL}_p(f, w)$ 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`

Key ideas:

- An efficient implementation of $\mathit{EVAL}_p(f, w)$ 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**

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:

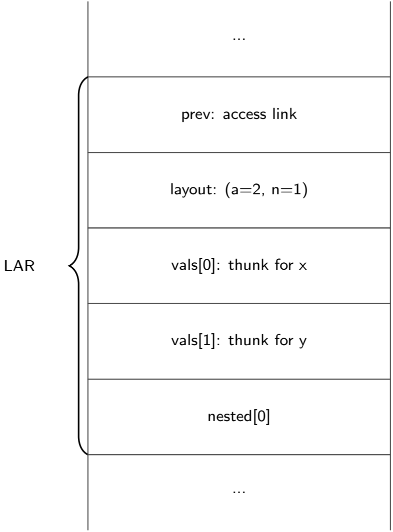
- No **closures**: they are encoded in **contexts**

Optimization:

- **Stack**- and **heap**-allocated LARs
- Minimal sharing analysis to make some formals call-by-name
- Compact memory representation (on AMD64)

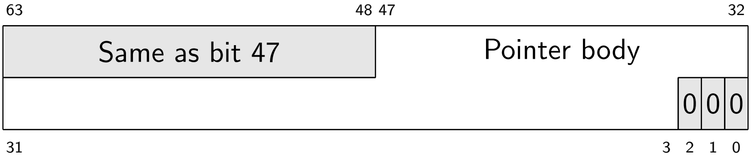
Lazy Activation Records

```
f x y = case x of
          []   -> [1]
          a:as -> [a + y]
```



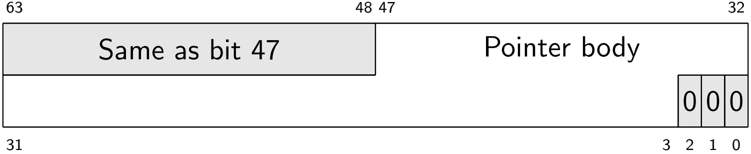
Compact Memory Representation

AMD64 pointers contain redundancy:



Compact Memory Representation

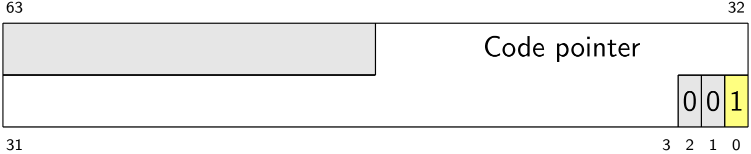
AMD64 pointers contain redundancy:



We use a variation of the **tagged pointers** technique

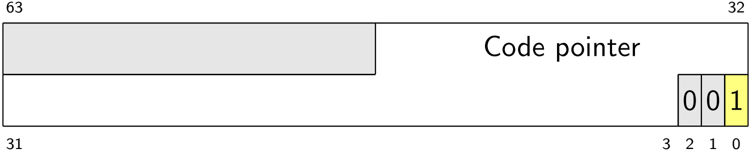
Thunks on AMD64

Unevaluated thunk

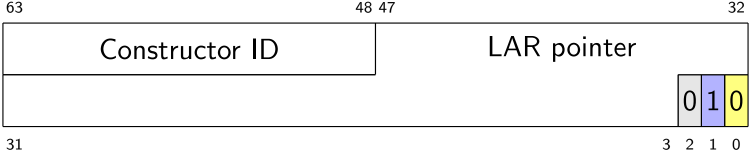


Thunks on AMD64

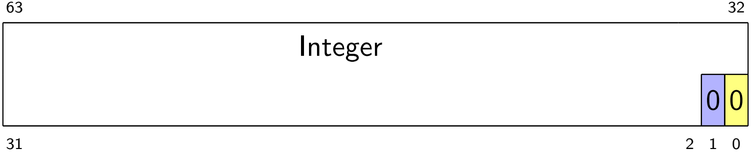
Unevaluated thunk



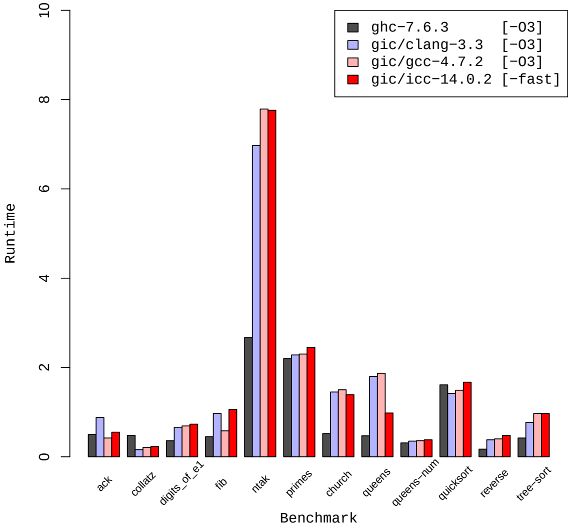
Lazy constructor



Primitive value



Benchmarks: Runtime



Benchmarks: Cache Behavior

	GHC					GIC				
	I1	LLi	D1	LLd	LL	I1	LLi	D1	LLd	LL
ack	0.0	0.0	6.1	2.1	0.3	0.0	0.0	10.8	0.0	0.0
church	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
collatz	0.0	0.0	0.7	0.0	0.0	0.0	0.0	0.1	0.1	0.0
digits_of_e1	0.0	0.0	3.2	0.0	0.0	0.0	0.0	1.3	0.8	0.2
fib	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ntak	0.0	0.0	1.6	0.0	0.0	0.0	0.0	1.0	0.9	0.2
primes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0
queens	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.4	0.3	0.0
queens-num	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
quick-sort	0.0	0.0	4.2	0.5	0.1	0.0	0.0	8.2	1.7	0.5
reverse	0.0	0.0	8.2	0.0	0.0	0.0	0.0	15.0	3.6	1.0
tree-sort	0.0	0.0	7.3	0.0	0.0	0.0	0.0	7.9	1.6	0.4

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

Modules

- Modularity mechanism: Haskell-style modules
- Two-step process: separate compilation and linking

Modules and Qualified Names

```
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
```

Modules and Qualified Names

```
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

Modular Intensional Transformation

Separate intensional transformation (FOFL→NVIL):

May generate `actuals` for formals of other modules:

- Needs function signatures for external functions
- Intensional indices are qualified: `call(i)` becomes `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

Separate compilation to C (NVIL→C):

- The NVIL code of the module is translated to C using LARs
- External symbols declared as `extern`
- Generates object file `Module.o`

Linking:

- Uses the intensional linking step
- System linker (`ld`) links the object files

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.
In *Proceedings of the 8th Panhellenic Logic Symposium*, pages 38–42, 2011.

Georgios Fourtounis, Peter Csaba Ölveczky, and Nikolaos Papaspyrou.
Formally specifying and analyzing a parallel virtual machine for lazy functional languages using Maude.
In *Proceedings of the 5th International Workshop on High-level Parallel Programming and Applications* (HLPP'11), pages 19–26, 2011.

Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis.
The generalized intensional transformation for implementing lazy functional languages.
In Konstantinos F. Sagonas, editor, *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages* (PADL '13), volume 7752 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2013.

Georgios Fourtounis and Nikolaos S. Papaspyrou.
Supporting separate compilation in a defunctionalizing compiler.
In José Paulo Leal, Ricardo Rocha, and Alberto Simões, editors, *2nd Symposium on Languages, Applications and Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 39–49, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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.
In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing* (FHPC'14), 2014.
Accepted for presentation, to appear.