The Generalized Intensional Transformation for Implementing Lazy Functional Languages

Georgios Fourtounis[1] Nikolaos Papaspyrou[1] Panos Rondogiannis[2]

[1]National Technical University of Athens
School of Electrical and Computer Engineering

[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 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

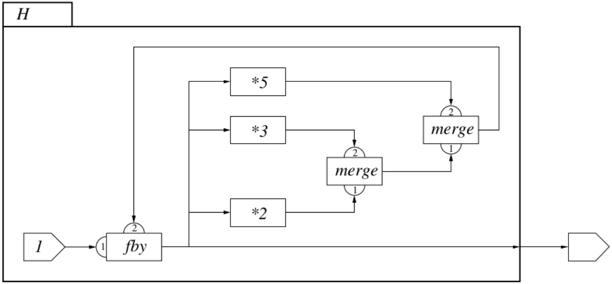


Figure from Joey Paquet's PhD thesis, "Intensional Scientific Programming" (1999)

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, 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 **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) 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

```
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 ith 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	=	$\mathtt{call}_0$(f)+$\mathtt{call}_1$(f)
f	=	$\mathtt{call}_0$(g)
g	=	y+2

Step 1: for all functions f

- Replace the ith call of f by $\mathtt{call}_i(f)$
- 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 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`	`result = call`$_0$`(f)+call`$_1$`(f)`
`f x = g (x*x)`	`f = call`$_0$`(g)`
`g y = y+2`	`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:

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

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)
```

```
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:

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

Example

Evaluation of the target program:

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

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

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

Example

Evaluation of the target program:

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

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

Example

Evaluation of the target program:

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

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

Example

Evaluation of the target program:

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

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

Example

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

Evaluation of the target program:

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

Example

Evaluation of the target program:

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

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

Example

Evaluation of the target program:

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

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

Example

Evaluation of the target program:

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

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

Example

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

Evaluation of the target program:

$$
\begin{aligned}
& \mathit{EVAL}(\mathtt{result}, [\,]) \\
= \; & \mathit{EVAL}(\mathtt{call_0(f)+\ call_1(f)}, [\,]) \\
= \; & \mathit{EVAL}(\mathtt{call_0(f)}, [\,]) + \mathit{EVAL}(\mathtt{call_1(f)}, [\,]) \\
= \; & \mathit{EVAL}(\mathtt{f}, [0]) + \mathit{EVAL}(\mathtt{f}, [1]) \\
= \; & \mathit{EVAL}(\mathtt{call_0(g)}, [0]) + \mathit{EVAL}(\mathtt{call_0(g)}, [1]) \\
= \; & \mathit{EVAL}(\mathtt{g}, [0,0]) + \mathit{EVAL}(\mathtt{g}, [0,1]) \\
= \; & \mathit{EVAL}(\mathtt{y}, [0,0]) + \mathit{EVAL}(2, [0,0]) + \mathit{EVAL}(\mathtt{y}, [0,1]) + \mathit{EVAL}(2, [0,1]) \\
= \; & \mathit{EVAL}(\mathtt{actuals(x*x)}, [0,0]) + 2 + \mathit{EVAL}(\mathtt{actuals(x*x)}, [0,1]) + 2 \\
= \; & \mathit{EVAL}(\mathtt{x*x}, [0]) + 2 + \mathit{EVAL}(\mathtt{x*x}, [1]) + 2 \\
= \; & \mathit{EVAL}(\mathtt{x}, [0]) * \mathit{EVAL}(\mathtt{x}, [0]) + 2 + \mathit{EVAL}(\mathtt{x}, [1]) * \mathit{EVAL}(\mathtt{x}, [1]) + 2 \\
= \; & \mathit{EVAL}(\mathtt{actuals(3, 5)}, [0]) * \mathit{EVAL}(\mathtt{actuals(3, 5)}, [0]) + 2 + \\
& \mathit{EVAL}(\mathtt{actuals(3, 5)}, [1]) * \mathit{EVAL}(\mathtt{actuals(3, 5)}, [1]) + 2 \\
= \; & \mathit{EVAL}(\mathtt{3}, [\,]) * \mathit{EVAL}(\mathtt{3}, [\,]) + 2 + \mathit{EVAL}(\mathtt{5}, [\,]) * \mathit{EVAL}(\mathtt{5}, [\,]) + 2 \\
= \; & 3 * 3 + 2 + 5 * 5 + 2
\end{aligned}
$$

Example

Evaluation of the target program:

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

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

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

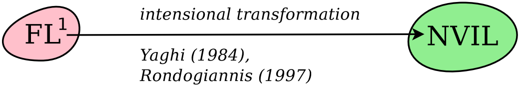
Evaluation order: from call-by-name to call-by-need

- 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

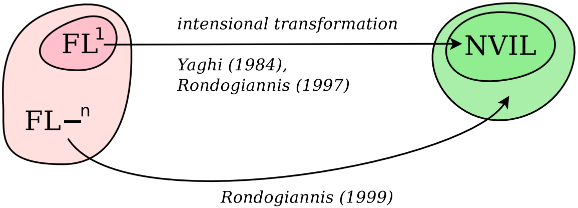
The New Intensional Transformation



Original intensional transformation

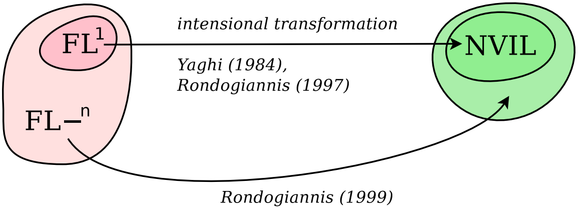
- FL^1: first-order functional language
- NVIL: zero-order intensional language

The New Intensional Transformation



Higher-order intensional transformation

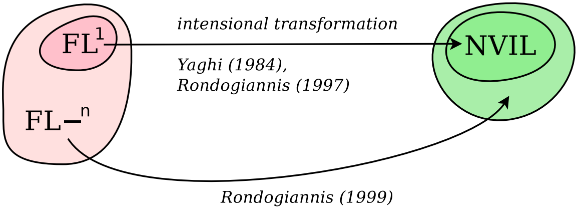
The New Intensional Transformation



Higher-order intensional transformation

- Missing: partial application (closures + currying)

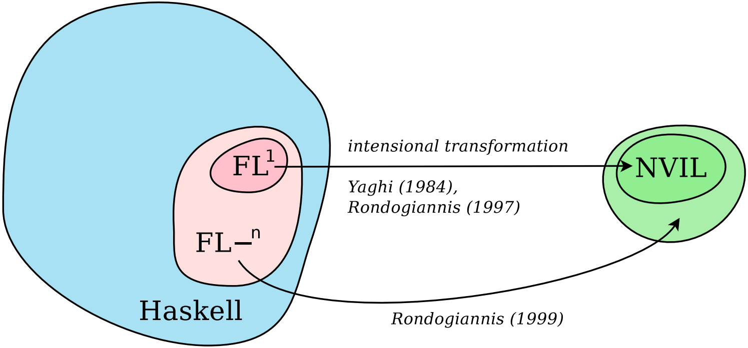
The New Intensional Transformation



Higher-order intensional transformation

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

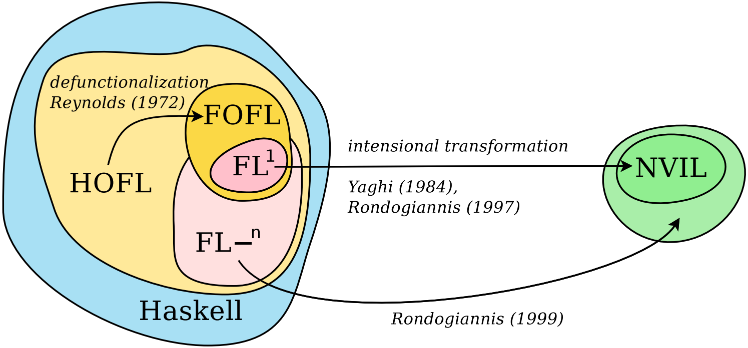
The New Intensional Transformation



Higher-order intensional transformation

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

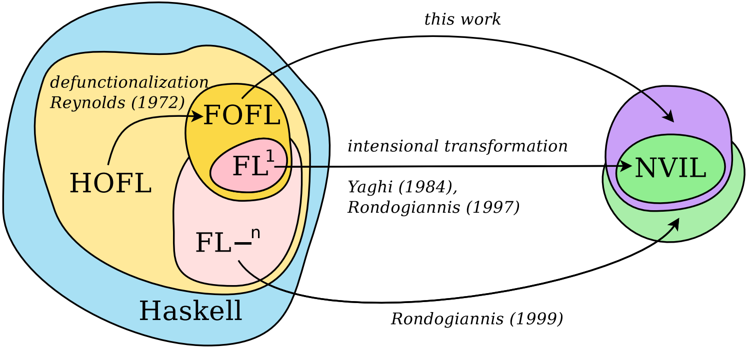
The New Intensional Transformation



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

Syntax of FOFL

p	$::=$	$d_0 \dots d_n$	**program**
d	$::=$	$f(v_0, \dots, v_{n-1}) = e$	**definition**
e	$::=$		**expression**
		$c(e_0, \dots, e_{n-1})$	constants and operators
	$\mid$	$f(e_0, \dots, e_{n-1})$	variables and functions
	$\mid$	$\kappa(e_0, \dots, e_{n-1})$	constructors
	$\mid$	$\texttt{case}\ e\ \texttt{of}\ \{\ b_0\ ;\ \dots ;\ b_n\ \}$	inspection of data types
	$\mid$	$\#^m(v)$	case pattern variables
b	$::=$	$\kappa(v_0, \dots, v_{n-1}) \to e$	**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:

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

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
	$\vert$	$\texttt{case}\ e\ \texttt{of}\ \{\ b_0\ ;\ \ldots\ ;\ b_n\ \}$	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	$::=$	$\kappa \rightarrow e$	**case clause**

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

Semantics of NVIL

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 $\langle \kappa, w \rangle$

$$\mathit{EVAL}_p(c(e_0, \ \ldots, \ e_{n-1}), w) \ = \ c(\mathit{EVAL}_p(e_0, w), \ldots, \mathit{EVAL}_p(e_{n-1}, w))$$

$$\mathit{EVAL}_p(f, w) \ = \ \mathit{EVAL}_p(\mathit{body}(f, p), w)$$

$$\mathit{EVAL}_p(\kappa, w) \ = \ \langle \kappa, w \rangle$$

$$\mathit{EVAL}_p(\texttt{case } e \texttt{ of } \{\kappa_0 \to e_0; \ \ldots; \ \kappa_n \to e_n\}, \langle \ell, w, \mu \rangle) \ = \\ \mathit{EVAL}_p(e_i, \langle \ell, w, w' : \mu \rangle) \quad \text{if } \ \mathit{EVAL}_p(e, \langle \ell, w, \mu \rangle) = \langle \kappa_i, w' \rangle$$

$$\mathit{EVAL}_p(\#^m(e), \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e, \mu_m)$$

$$\mathit{EVAL}_p(\texttt{call}_\ell(e), w) \ = \ \mathit{EVAL}_p(e, \langle \ell, w, \bullet \rangle)$$

$$\mathit{EVAL}_p(\texttt{actuals}(\langle e_\ell \rangle_{\ell \in I}), \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e_\ell, w)$$

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

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

Example: Reversing lists (ii)

FOFL

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

NVIL

Example: Reversing lists (ii)

FOFL

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

NVIL

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

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

Example: Reversing lists (ii)

FOFL

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

NVIL

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

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

Example: Reversing lists (ii)

FOFL

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

NVIL

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

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

Implementation

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

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

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**

Implementation

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

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**

Optimization:

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

Benchmarks

Program	GIC	GIC-llvm	GHC7	GHC6	NHC	UHC	JHC
ack	2.47	1.25	0.62	0.48	6.18	40.03	0.05
church	3.55	2.09	0.61	0.55	11.58	68.37	0.17
collatz	0.69	0.41	1.07	2.66	84.28	46.90	0.16
digits_of_e1	2.30	2.09	0.77	1.74	60.71	75.29	–[1]
fast-reverse	3.03	1.95	1.74	1.82	1.35	9.41	–[2]
fib	1.35	1.12	0.50	0.51	10.43	55.55	0.17
naive-reverse	3.02	2.87	0.49	0.42	0.79	3.56	0.75
ntak	8.62	5.87	2.91	3.65	154.74	91.95	7.18
primes	2.55	1.58	2.19	2.30	172.45	173.81	0.73
queens-num	0.33	0.23	0.31	0.33	21.16	12.43	0.14
queens	3.92	3.24	0.44	0.48	27.17	123.98	0.82
quick-sort	3.18	2.77	1.92	1.90	1.51	5.42	8.58
tree-sort	2.19	1.97	0.39	0.33	0.91	6.58	0.72
GMR[3]	1.38	1.00	0.51	0.57	7.28	18.49	0.33

[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**

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

```
result  = inc (add 1) 2 + inc sq 3
inc f x = f (x+1)
add a b = a+b
sq z    = z*z
```

First-order, defunctionalized

```
result  = inc (Fadd 1) 2 + inc Fsq 3
inc f x = apply f (x+1)
add a b = a+b
sq z    = z*z

data Func = Fadd Int | Fsq

apply cl d = case cl of
               Fadd c -> add c d
               Fsq -> sq d
```