Formally Specifying and Analyzing a Parallel Virtual Machine for Lazy Functional Languages Using Maude

Georgios Fourtounis[1,2]
Peter C. Ölveczky[2]
Nikolaos S. Papaspyrou[1]

[1]Software Engineering Laboratory, National Technical University of Athens
[2]Department of Informatics, University of Oslo

HLPP 2011
September 18, 2011

Outline

1. Lazy Languages and Dataflow
2. The Intensional Transformation and Eduction
3. Parallel Eduction with Distributed Warehouses
4. Prototyping with Maude
5. Conclusion

Outline

1. Lazy Languages and Dataflow
2. The Intensional Transformation and Eduction
3. Parallel Eduction with Distributed Warehouses
4. Prototyping with Maude
5. Conclusion

Dataflow Programming Languages

Dataflow Programming:

Dataflow programming is a programming paradigm in which data are processed while flowing through a network of processors. Dataflow was quite popular during the 1980s due to its implicitly parallel nature.

Dataflow Languages:

They are mostly functional in nature and they encourage stream processing. Examples: Val, Id, Lucid, GLU, etc.

Dataflow Machines:

Specialized parallel architectures intended to run dataflow languages (e.g. *The MIT Tagged-Token Machine*).

The Status of Dataflow

The 1990s:

Interest in Dataflow started to decline during the 1990s mainly due to the fact that Dataflow Architectures could not compete even with mainstream (von Neumann/sequential) architectures.

Today:

Interest in Dataflow started to revive lately due to the introduction of multi-core architectures.

The Next Day:

Google introduced Map-Reduce, a system that has similarities to Dataflow languages. A new generation of similar languages has started to develop (Dryad, Clustera, Hyrax, etc).

Dataflow and Laziness

Lazy languages are usually pure; data dependencies can then drive evaluation in a dataflow manner.

Dataflow as an Implementation Technique:

- pH/pHluid: parallelism through eager evaluation
- Rediflow: dataflow graph reduction
- Dataflow implementation using the **intensional transformation**

Outline

1. Lazy Languages and Dataflow
2. The Intensional Transformation and Eduction
3. Parallel Eduction with Distributed Warehouses
4. Prototyping with Maude
5. Conclusion

The Intensional Transformation

1. Yaghi (1984), Rondogiannis & Wadge (1997, 1999).
2. A transformation from a lazy functional language to a 0-order dataflow language where values are tagged by **contexts** (lists of natural numbers that encode the binding environment).
3. Originally for tagged-token dataflow, efficient implementation on stock hardware.

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

Example

Consider the following functional program:

```
result  =  f(4)+f(5)
f(x)    =  g(x+1)
g(y)    =  y
```

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

Example

Consider the following functional program:

```
result  =  f(4)+f(5)
f(x)    =  g(x+1)
g(y)    =  y
```

The output of the transformation is:

```
result  =  call0(f)+call1(f)
f       =  call0(g)
g       =  y
x       =  actuals(4,5)
y       =  actuals(x+1)
```

Semantics of the Operators

The technique is termed “intensional” because the target program contains operators that act on a hidden context — pretty much as in the case of certain temporal/intensional logics. The contexts in our case are lists of natural numbers.

$$\begin{aligned}(call_i(a))(w) &= a(i : w)\\(actuals(a_0, \ldots, a_{n-1}))(i : w) &= (a_i)(w)\end{aligned}$$

Evaluation of an intensional program is called **eduction**.

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

Example

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Evaluation of the target program:

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

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

Example

Evaluation of the target program:

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

```
result = call0(f)+call1(f)
f      = call0(g)
g      = y
x      = actuals(4,5)
y      = actuals(x+1)
```

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

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

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& EVAL(\texttt{result}, [\,]) \\
= \; & EVAL(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \; & EVAL(\texttt{call}_0\texttt{(f)}, [\,]) + EVAL(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \; & EVAL(\texttt{f}, [0]) + EVAL(\texttt{f}, [1]) \\
= \; & EVAL(\texttt{call}_0\texttt{(g)}, [0]) + EVAL(\texttt{call}_0\texttt{(g)}, [1]) \\
= \; & EVAL(\texttt{g}, [0,0]) + EVAL(\texttt{g}, [0,1]) \\
= \; & EVAL(\texttt{y}, [0,0]) + EVAL(\texttt{y}, [0,1]) \\
= \; & EVAL(\texttt{actuals(x+1)}, [0,0]) + EVAL(\texttt{actuals(x+1)}, [0,1]) \\
= \; & EVAL(\texttt{x+1}, [0]) + EVAL(\texttt{x+1}, [1]) \\
= \; & EVAL(\texttt{x}, [0]) + EVAL(\texttt{1}, [0]) + EVAL(\texttt{x}, [1]) + EVAL(\texttt{1}, [1]) \\
= \; & EVAL(\texttt{x}, [0]) + 1 + EVAL(\texttt{x}, [1]) + 1 \\
= \; & EVAL(\texttt{actuals(4, 5)}, [0]) + 1 + EVAL(\texttt{actuals(4, 5)}, [1]) + 1
\end{aligned}
$$

Example

Evaluation of the target program:

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

$$
\begin{aligned}
& EVAL(\texttt{result}, [\,]) \\
= \; & EVAL(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \; & EVAL(\texttt{call}_0\texttt{(f)}, [\,]) + EVAL(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \; & EVAL(\texttt{f}, [0]) + EVAL(\texttt{f}, [1]) \\
= \; & EVAL(\texttt{call}_0\texttt{(g)}, [0]) + EVAL(\texttt{call}_0\texttt{(g)}, [1]) \\
= \; & EVAL(\texttt{g}, [0, 0]) + EVAL(\texttt{g}, [0, 1]) \\
= \; & EVAL(\texttt{y}, [0, 0]) + EVAL(\texttt{y}, [0, 1]) \\
= \; & EVAL(\texttt{actuals(x+1)}, [0, 0]) + EVAL(\texttt{actuals(x+1)}, [0, 1]) \\
= \; & EVAL(\texttt{x+1}, [0]) + EVAL(\texttt{x+1}, [1]) \\
= \; & EVAL(\texttt{x}, [0]) + EVAL(\texttt{1}, [0]) + EVAL(\texttt{x}, [1]) + EVAL(\texttt{1}, [1]) \\
= \; & EVAL(\texttt{x}, [0]) + 1 + EVAL(\texttt{x}, [1]) + 1 \\
= \; & EVAL(\texttt{actuals(4, 5)}, [0]) + 1 + EVAL(\texttt{actuals(4, 5)}, [1]) + 1 \\
= \; & EVAL(\texttt{4}, [\,]) + 1 + EVAL(\texttt{5}, [\,]) + 1
\end{aligned}
$$

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& EVAL(\texttt{result}, [\,]) \\
= \; & EVAL(\texttt{call}_0\texttt{(f)+ call}_1\texttt{(f)}, [\,]) \\
= \; & EVAL(\texttt{call}_0\texttt{(f)}, [\,]) + EVAL(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \; & EVAL(\texttt{f}, [0]) + EVAL(\texttt{f}, [1]) \\
= \; & EVAL(\texttt{call}_0\texttt{(g)}, [0]) + EVAL(\texttt{call}_0\texttt{(g)}, [1]) \\
= \; & EVAL(\texttt{g}, [0,0]) + EVAL(\texttt{g}, [0,1]) \\
= \; & EVAL(\texttt{y}, [0,0]) + EVAL(\texttt{y}, [0,1]) \\
= \; & EVAL(\texttt{actuals(x+1)}, [0,0]) + EVAL(\texttt{actuals(x+1)}, [0,1]) \\
= \; & EVAL(\texttt{x+1}, [0]) + EVAL(\texttt{x+1}, [1]) \\
= \; & EVAL(\texttt{x}, [0]) + EVAL(\texttt{1}, [0]) + EVAL(\texttt{x}, [1]) + EVAL(\texttt{1}, [1]) \\
= \; & EVAL(\texttt{x}, [0]) + 1 + EVAL(\texttt{x}, [1]) + 1 \\
= \; & EVAL(\texttt{actuals(4, 5)}, [0]) + 1 + EVAL(\texttt{actuals(4, 5)}, [1]) + 1 \\
= \; & EVAL(\texttt{4}, [\,]) + 1 + EVAL(\texttt{5}, [\,]) + 1 \\
= \; & 4 + 1 + 5 + 1
\end{aligned}
$$

```
result = call_0(f)+call_1(f)
f      = call_0(g)
g      = y
x      = actuals(4, 5)
y      = actuals(x+1)
```

Example

Evaluation of the target program:

$$
\begin{aligned}
& EVAL(\mathtt{result}, [\,]) \\
= \; & EVAL(\mathtt{call}_0\mathtt{(f)+\ call}_1\mathtt{(f)}, [\,]) \\
= \; & EVAL(\mathtt{call}_0\mathtt{(f)}, [\,]) + EVAL(\mathtt{call}_1\mathtt{(f)}, [\,]) \\
= \; & EVAL(\mathtt{f}, [0]) + EVAL(\mathtt{f}, [1]) \\
= \; & EVAL(\mathtt{call}_0\mathtt{(g)}, [0]) + EVAL(\mathtt{call}_0\mathtt{(g)}, [1]) \\
= \; & EVAL(\mathtt{g}, [0, 0]) + EVAL(\mathtt{g}, [0, 1]) \\
= \; & EVAL(\mathtt{y}, [0, 0]) + EVAL(\mathtt{y}, [0, 1]) \\
= \; & EVAL(\mathtt{actuals(x+1)}, [0, 0]) + EVAL(\mathtt{actuals(x+1)}, [0, 1]) \\
= \; & EVAL(\mathtt{x+1}, [0]) + EVAL(\mathtt{x+1}, [1]) \\
= \; & EVAL(\mathtt{x}, [0]) + EVAL(\mathtt{1}, [0]) + EVAL(\mathtt{x}, [1]) + EVAL(\mathtt{1}, [1]) \\
= \; & EVAL(\mathtt{x}, [0]) + 1 + EVAL(\mathtt{x}, [1]) + 1 \\
= \; & EVAL(\mathtt{actuals(4,\ 5)}, [0]) + 1 + EVAL(\mathtt{actuals(4,\ 5)}, [1]) + 1 \\
= \; & EVAL(\mathtt{4}, [\,]) + 1 + EVAL(\mathtt{5}, [\,]) + 1 \\
= \; & 4 + 1 + 5 + 1 \\
= \; & 11
\end{aligned}
$$

Implementing Eduction with a Warehouse

Adding laziness

- To implement laziness, the result of evaluating each (`var`, `context`) is memoized in a **warehouse**.
- Warehouses are stores of thunks.
- A standard component of non-functional, intensional languages as well (GIPSY, TransLucid).

Outline

1. Lazy Languages and Dataflow
2. The Intensional Transformation and Eduction
3. Parallel Eduction with Distributed Warehouses
4. Prototyping with Maude
5. Conclusion

Parallelizing Evaluation

Design Choices

- Distribute evaluation using a shared-nothing approach.
- Use implicit parallelism (at every built-in strict operation); however we can easily support explicit parallelism annotations (**par** and **pseq**).

Parallelizing Evaluation

Design Choices

- Distribute evaluation using a shared-nothing approach.
- Use implicit parallelism (at every built-in strict operation); however we can easily support explicit parallelism annotations (**par** and **pseq**).

Style of Parallelism

- Evaluation of built-in operations (such as $+$, $-$, $\ldots$) is a candidate for **fork-join parallelism**.
- An expression is a process that spawns two child processes and waits for them to complete.

Parallelizing the Warehouse

Hitting the Warehouse in Parallel

- Entries should get locked so that they are evaluated once.
- A single warehouse becomes a bottleneck in parallel evaluation.

Parallelizing the Warehouse

Hitting the Warehouse in Parallel

- Entries should get locked so that they are evaluated once.
- A single warehouse becomes a bottleneck in parallel evaluation.

Warehouse Distribution

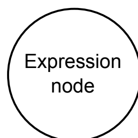
Memoized results are contained in distributed **warehouses**, processes that can be queried for the value of a variable in a context.

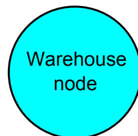
Overall Picture

The Processes of the Computation:

- **expression nodes**
  processes that are created dynamically during evaluation
- **warehouse nodes**
  processes managing memoized results, fixed number, created at program start

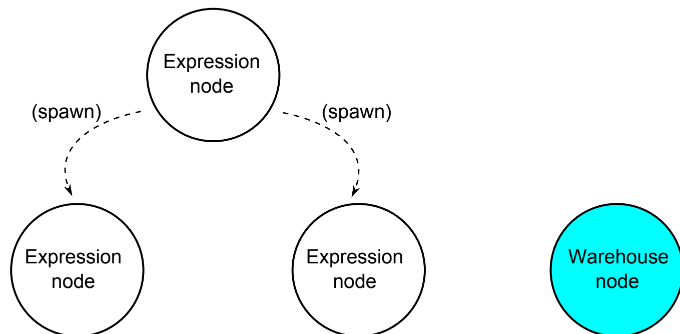
Example: Messages





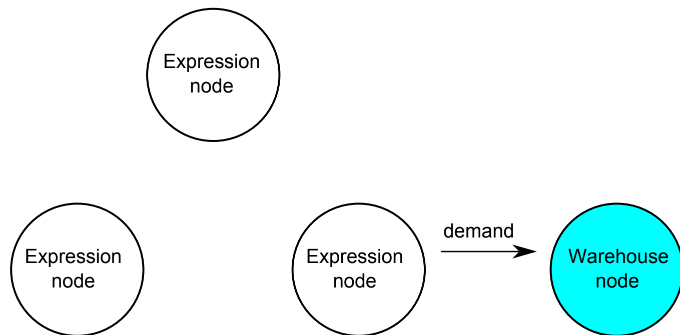
One expression node (that is a built-in operation) and one warehouse.

Example: Messages



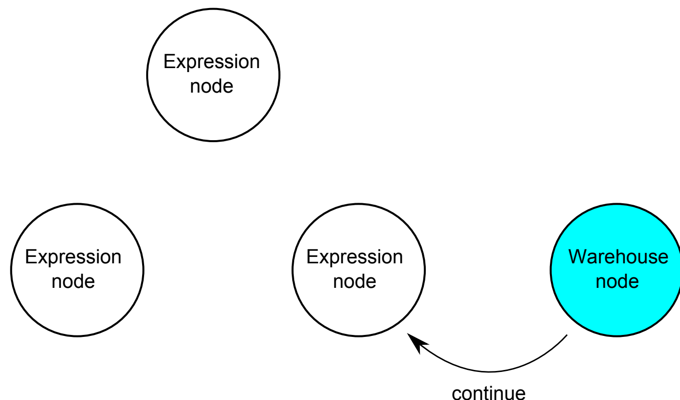
The expression node spawns two expression nodes to calculate its subexpressions.

Example: Messages



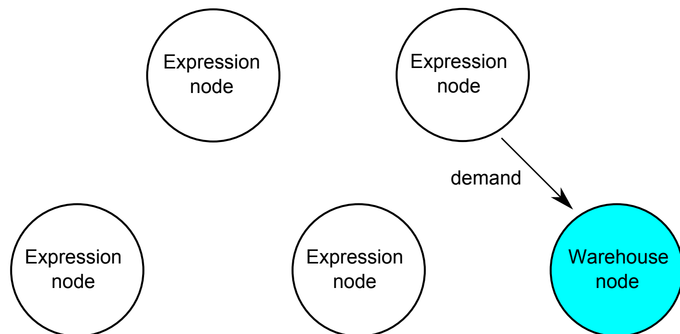
The right child demands a (variable, context) pair from the warehouse.

Example: Messages



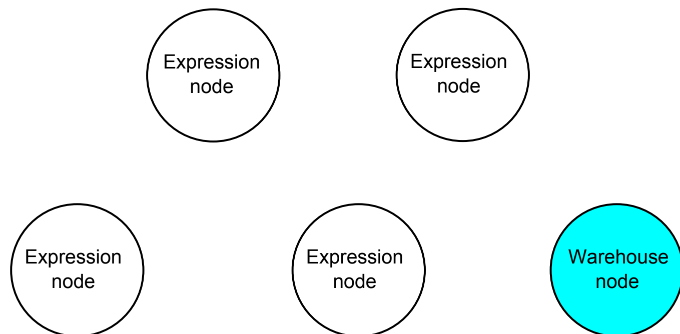
The warehouse node does not know anything about this demand, so it instructs the expression node to continue, in order to evaluate it.

Example: Messages



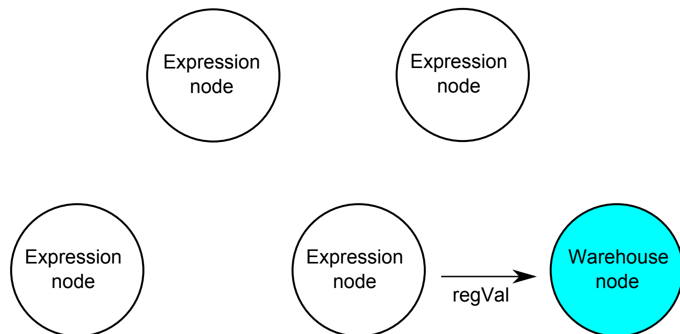
Some other expression node asks the warehouse for the same (variable, context).

Example: Messages



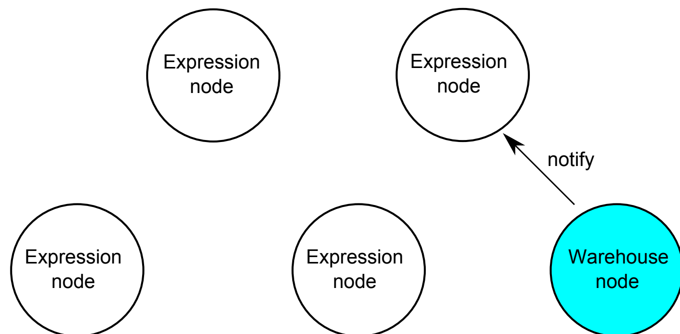
Since the result is already being computed, the warehouse node leaves the asking expression node in a blocked state.

Example: Messages



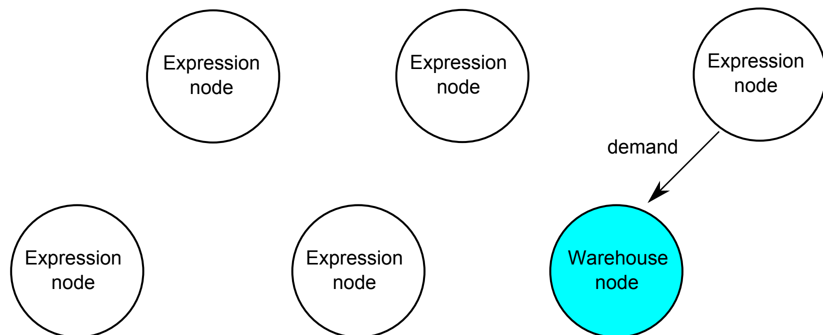
The computing expression node reaches a value and registers it in the warehouse.

Example: Messages



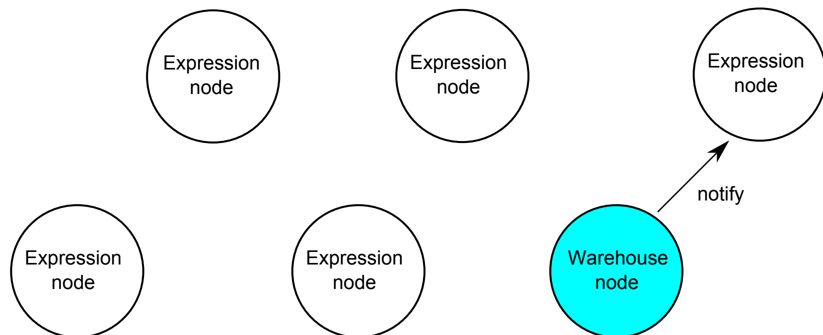
The warehouse node now sends a notification to the waiting expression node.

Example: Messages



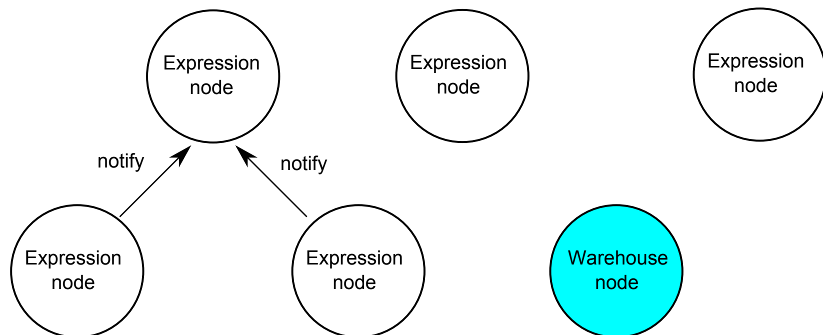
Some other expression node also asks the warehouse node for the same (variable, context).

Example: Messages



The warehouse node now has a value ready, so it sends back a notification with it.

Example: Messages



When the two expression nodes that were spawned become values, they notify their parent.

Warehouses

Demands for the same thunk (i.e. the same $(var, context)$)
should be sent to the same warehouse

Choosing the warehouse to ask is a **function** of the variable
and the context, it affects thunk distribution

Warehouses

Demands for the same thunk (i.e. the same $(var, context)$) should be sent to the same warehouse

Choosing the warehouse to ask is a **function** of the variable and the context, it affects thunk distribution

(if the choice is not a function but a relation that can assign more than one warehouse nodes, we have **recomputation**)

Our Thunk Distribution

All arguments to the same function call have the same context

We choose the warehouse according to the context length, a cheap way to put the arguments of the same call in the same warehouse

How to Handle Blocked Evaluation

Each warehouse node keeps a graph of all the dependencies between the expression nodes it knows:

1. Those computing a value needed by the warehouse
2. Those blocked on results pending from other expression nodes

Outline

1. Lazy Languages and Dataflow
2. The Intensional Transformation and Eduction
3. Parallel Eduction with Distributed Warehouses
4. Prototyping with Maude
5. Conclusion

Formalizing the Model

The Tool:

- We formalized and tested our model using Maude, a rewriting logic based tool.
- All nodes and messages are represented as terms that participate in concurrent rewrite rules.
- The resulting model can be simulated and instances of it can be exhaustively checked for correctness.

Our Experience:

- Easy to use tool and theory:
  `Maude>` *(search init =>! C:Configuration .)*
- Straightforward encoding of our model.
- Instance checking does not scale to realistic programs but we caught two errors in a previous version of the model.

The Prototype

The Maude model was directly transferred to an Erlang prototype.

The implicit parallelism of our approach is fine-grained; although Erlang is not an implementation language, its scheduler is robust enough to run interesting test programs.

Outline

1. Lazy Languages and Dataflow
2. The Intensional Transformation and Eduction
3. Parallel Eduction with Distributed Warehouses
4. Prototyping with Maude
5. Conclusion

Conclusion

What?

- A distributed model of lazy evaluation with message passing
- Supports explicit parallelism annotations

How?

- Intensional transformation and eduction
- Thunks and expressions as message-exchanging processes
- Configurable thunk distribution

What next?

- Faster implementation
- Explore issues of recomputation/redundancy/recovery

Thank you!