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

Georgios Fourtounis
School of Electrical and Computer Eng.
National Technical University of Athens
gfour@softlab.ntua.gr

Peter Csaba Ölveczky
Department of Informatics
University of Oslo
peterol@ifi.uio.no

Nikolaos Papaspyrou
School of Electrical and Computer Eng.
National Technical University of Athens
nickie@softlab.ntua.gr

Abstract

Pure lazy functional languages are a promising programming paradigm for harvesting massive parallelism, as their abstraction features and lack of side effects support the development of modular programs without unneeded serialization. We give a new formal message passing semantics for implicitly parallel execution of a lazy functional programming language, based on the intensional transformation that converts programs in functional style to a form that can be executed in a dataflow paradigm. We use rewriting logic to define the semantics of our parallel virtual machine and we use the Maude tool to formally analyze our model. We also briefly discuss a prototype parallel implementation of our model in Erlang.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; Model checking; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Data-flow languages; F.1.2 [*Computation by Abstract Devices*]: Modes of Computation—Parallelism and concurrency

General Terms Declarative parallel programming languages: semantics and implementation

Keywords Dataflow, formal analysis, intensional transformation, lazy functional programming languages, Maude, parallelism, rewriting logic.

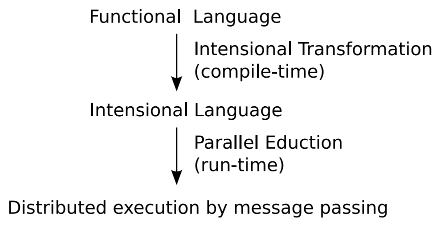
1. Introduction

Pure lazy functional languages have been proposed as a way to harvest the underlying parallelism of programs: their mathematical compositional semantics and lack of side effects suggest an elegant way of distributing computations with little effort from the programmer, while laziness avoids unneeded computations. Parallelism in lazy languages has therefore been an active topic of research (see, e.g., [11, 17, 18, 23, 29, 31]), usually in connection with the two established methods for implementing lazy languages: graph reduction and graph rewriting.

Parallel lazy evaluation is, however, not without its problems: laziness complicates reasoning about programs and their control flow, while the bookkeeping needed to avoid recomputations may affect performance. These problems have been addressed by many successful systems, including Eden [17], GpH [30], and Data Parallel Haskell [15], which all depend on the programmer to provide clues about what can be parallelized (explicit parallelism). Automatically inferring parallelism from the user's program (implicit parallelism) is an active field of research [13], but seems to be a more difficult target, as parallel lazy evaluation has a cost model that is difficult to analyze automatically.

To capture the behavior of parallel lazy evaluation, different semantics have been given for the parallel execution of lazy functional languages [3, 4, 9, 14], focusing on different ways of declaring parallelism (explicit vs. implicit), and using different models for distributing computations. Continuing this line of research, we give a new formal semantics for implicitly parallel execution of lazy functional programming languages, based on message passing for distributing computation without shared memory. In this paradigm, we can express lazy evaluation of expressions in terms of messages exchanged between evaluating expressions and suspended computations (*thunks*), treating both as processes.

Our semantics is not only of theoretical interest; it should be suitable as a basis for parallel implementation since it is based on the intensional transformation [26, 27], a technique for implementing higher-order lazy functional languages as dataflow, whose execution model (*eduction*) has proved to be efficient in sequential architectures [5]. The figure below summarizes our methodology.



Our semantics is formally defined in rewriting logic [19], using the Maude prototyping and verification platform [6], which supports the executable definition of distributed computations and provides model checking tools with which we analyze the behaviors of the model. Furthermore, the message passing style of the semantics made it very easy to move from the Maude model to a prototype parallel implementation in Erlang [1].

Section 2 introduces our source functional language, the intensional transformation that transforms such programs into dataflow programs, and rewriting logic and the Maude tool. Section 3 informally describes our parallel virtual machine, which is formalized in rewriting logic in Section 4. We formally analyze our model in Section 5 and discuss in Section 6 how we defined a parallel prototype implementation in Erlang from this model. We also discuss related work in Section 7 and give some final remarks in Section 8.

2. Preliminaries

This section briefly discusses the source lazy functional programming language FOFL; the intermediate language NVIL, which is the language for which we actually provide a parallel execution model, and the transformation from FOFL to NVIL; and the Maude formalism and tool that is used to formally define and analyze our parallel execution environment.

2.1 The Language FOFL

The source language FOFL is a first-order lazy functional programming language [26], whose syntax is given below:

$$\begin{array}{lll} \mathit{Program} & := & \mathit{Def}_0, \ldots, \mathit{Def}_n \\ \mathit{Def} & := & f(v_0, \ldots, v_{n-1}) = \mathit{Expr} \\ \mathit{Expr} & := & v \mid n \mid f(\mathit{Expr}_0, \ldots, \mathit{Expr}_{n-1}) \\ & \mid & \mathit{Op}\ (\mathit{Expr}_0, \ldots, \mathit{Expr}_m) \\ \mathit{Op} & := & + \mid - \mid * \mid / \mid \leq \mid == \mid \textbf{if} \mid \ldots \end{array}$$

where v, v_i are variable names, $n \geq 0$, and $m > 0$.

FOFL is the first-order subset of the higher-order FL language in the work of Rondogiannis [27], but is equally expressive, since all FL programs can be translated to FOFL using the technique of defunctionalization [25]. Therefore, we only treat first-order programs, assuming that higher-order ones have been defunctionalized to first-order ones. Using FOFL, we can then accept all first-order as well as all higher-order lazy functional programs without partial applications (see [27] for details about these restrictions). Compared to the higher-order, multidimensional model defined by Rondogiannis, we prefer this combination of a first-order language and defunctionalization, as it simplifies our parallel model without losing expressiveness.

The Fibonacci function can be written in FOFL as follows:

```
result = fib(3)
fib(x) = if (x < 2) then 1
         else fib(x - 1) + fib(x - 2)
```

By convention, the final result of the program is always contained in a variable named `result`. We use infix notation for operators, whenever appropriate, and the more intuitive syntax **if** e **then** e_1 **else** e_2 for conditional expressions.

2.2 The Language NVIL and the Intensional Transformation

NVIL [26] is the language of the programs resulting from the transformation and the one whose parallel execution we model. Programs in NVIL are made of nullary variable definitions; i.e., there are no functions, while expressions of the language are accompanied during execution by a context, represented as a stack.

The syntax of NVIL is defined as follows:

$$\begin{array}{lll} \mathit{Program} & := & \mathit{Def}_0, \ldots, \mathit{Def}_n \\ \mathit{Def} & := & v = \mathit{Expr} \\ \mathit{Expr} & := & v \mid n \mid \mathit{Op}\ (\mathit{Expr}_0, \ldots, \mathit{Expr}_m) \\ & \mid & \textbf{call}_n(\mathit{Expr}) \mid \textbf{actuals}(\mathit{Expr}_0, \ldots, \mathit{Expr}_n) \\ \mathit{Op} & := & + \mid - \mid * \mid / \mid \leq \mid == \mid \textbf{if} \mid \ldots \end{array}$$

where v is a variable name and $n \geq 0$.

A program consists of a list of definitions, each defining an expression as the body of a unique variable v. An expression can be a variable, a number, a **call/actuals** expression, or an application of a built-in operator. We provide the convenient notation *false* and *true* for the truth values 0 and 1.

The execution uses a stack as the context: $\textbf{call}_j(e)$ pushes the value j to the context stack and continues to evaluate e in the new context; its inverse, $\textbf{actuals}(e_0, \ldots, e_n)$, pops the head i of the context and evaluates e_i in the new context.

$$\begin{array}{l} \mathit{eval} :: (\mathit{Expr}, \mathit{Context}, \mathit{Prog}) \to \mathit{Value} \\ \mathit{eval}(n, \mathit{ctxt}, p) = n \\ \mathit{eval}(\textbf{call}_i(e), \mathit{ctxt}, p) = \mathit{eval}(e, i : \mathit{ctxt}, p) \\ \mathit{eval}(\textbf{actuals}(e_0, \ldots, e_n), j : \mathit{ctxt}, p) = \mathit{eval}(e_j, \mathit{ctxt}, p) \\ \mathit{eval}(v, \mathit{ctxt}, p) = \mathit{eval}(\mathit{lookup}(v, p), \mathit{ctxt}, p) \\ \mathit{eval}(e_1 + e_2, \mathit{ctxt}, p) = \mathit{eval}(e_1, \mathit{ctxt}, p) + \mathit{eval}(e_2, \mathit{ctxt}, p) \\ \quad \text{(similar rules for other built-in operators)} \\ \mathit{eval}(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2, \mathit{ctxt}, p) = \mathit{eval}(e_1, \mathit{ctxt}, p) \\ \quad \text{if } \mathit{eval}(e_0, \mathit{ctxt}, p) = 1 \\ \mathit{eval}(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2, \mathit{ctxt}, p) = \mathit{eval}(e_2, \mathit{ctxt}, p) \\ \quad \text{if } \mathit{eval}(e_0, \mathit{ctxt}, p) = 0 \\ \quad \text{where } 0 \leq i \text{ and } 0 \leq j \leq n. \\ \\ \mathit{lookup} :: (\mathit{Var}, \mathit{Prog}) \to \mathit{Expr} \\ \mathit{lookup}(v, (\ldots, (v = e), \ldots)) = e \end{array}$$

Figure 1. An interpreter for NVIL.

An interpreter for NVIL is shown in Figure 1, following the established demand-driven model of intensional languages, *eduction*. It is defined as a function *eval*, which takes an expression, a context, and a program, and produces a value, which is the result of executing the program. Since the NVIL program is immutable during execution and is only used for variable definition lookup, the parameters of evaluation that change during runtime are the expression and the context — a pair for which we use the notation $\langle \mathit{expr}, \mathit{context} \rangle$.

We use the metavariable n to represent numbers, and v to represent variable names. The function *lookup* returns the definition body of a variable. We write $x : s$ for the stack obtained by pushing element x to the stack s.

The intensional transformation converts programs written in the lazy first-order functional language *FOFL* to the zero-order dataflow language *NVIL*. The transformation itself and its associated execution model (*eduction*) are described in Rondogiannis' work [26]. We introduce here the basic ideas of the transformation by means of an example. Our work, however, does not focus on the transformation itself, but on a parallel model of execution for its target language.

An example of the transformation is shown below:

```
result = f(2)
f(x) = if (x ≤ 1) then 1 else x * f(x-1)
```

$$\Downarrow$$

$$\begin{array}{l} \mathit{result} = \textbf{call}_0(f) \\ f = \textbf{if}\ (x \leq 1)\ \textbf{then}\ 1\ \textbf{else}\ x * \textbf{call}_1(f) \\ x = \textbf{actuals}(2, x - 1) \end{array}$$

The transformation essentially lowers all functions of a program to variable definitions and uses stacks to pass their arguments around, with the **call** and **actuals** operators being the "push" and "pop" operations.

We give a step by step description of how the NVIL code of this example runs (we use the notation [ ] to denote the empty context):

1. Initially the value of *result* is demanded:
   $\mathit{eval}(\mathit{result},\ [\,]) \to \mathit{eval}(\textbf{call}_0(f),\ [\,]) \to \mathit{eval}(f,\ [0])$
   $\to \mathit{eval}(\textbf{if}\ (x \leq 1)\ \textbf{then}\ 1\ \textbf{else}\ x * \textbf{call}_1(f),\ [0])$
2. At this point, the **if** expression must check its $(x \leq 1)$ subexpression, so it evaluates it separately in the current context:
   $\mathit{eval}(x \leq 1,\ [0])$
3. The last expression is a constant operator that has two subexpressions: x and 1. They are evaluated as new tasks as follows:

- $\mathit{eval}(x,\ [0]) \to \mathit{eval}(\textbf{actuals}_0(2, x-1),\ [0])$
  $\to \mathit{eval}(2,\ [\,]) \to 2$
- $\mathit{eval}(1,\ [0]) \to 1$, a value

4. The $x \leq 1$ expression consumes the two values computed above to become $2 \leq 1$, which equals *false*.
5. The **if** expression uses the evaluated condition and becomes:
   $\mathit{eval}(\textbf{if}\ \mathit{false}\ \textbf{then}\ 1\ \textbf{else}\ x\ *\ \textbf{call}_1(f),\ [0])$
   which evaluates to the branch $x\ *\ \textbf{call}_1(f)$.
6. As before, this expression is a constant operator that creates two new subexpressions:
   - $\mathit{eval}(x,\ [0]) \to \ldots \to 2$ (as before)
   - $\mathit{eval}(\textbf{call}_1(f),\ [0]) \to \mathit{eval}(f,\ [1\ 0]) \to$
     $\mathit{eval}(\textbf{if}\ (x \leq 1)\ \textbf{then}\ 1\ \textbf{else}\ x\ *\ \textbf{call}_1(f),\ [1\ 0])$
7. Now x can be shown to be 1 in the new context and the branch containing 1 is returned. This results in the former multiplication giving $2 * 1 = 2$, which is the result of the whole program.

2.3 Rewriting Logic and Maude

Rewriting logic [19] extends algebraic specifications to concurrent systems, and has proved suitable for specifying distributed systems in an object-oriented style.

In rewriting logic, the static parts of a system (functions, data types, etc.) are defined as an algebraic equational specification; i.e., we declare sorts, subsorts, and function symbols, and equations are used to define the functions. The transitions of a system are specified by labeled rewrite rules of the form $l\ :\ t \longrightarrow t'$ **if** cond, where t and t' are *terms*, l is a rule label, and cond is a (possibly empty) conjunction of equalities. Such a rule specifies a local transition from an instance of the term t to the corresponding instance of the term t', provided that the condition cond is satisfied.

Maude [6] is a language and high-performance tool for specifying, simulating, and model checking rewriting logic theories. The Maude syntax is fairly intuitive (see [6] for details). A function f with arguments of sorts $s_1 \ldots s_n$ and value of sort s is declared by `op` $f : s_1 \ldots s_n$ `->` s. Equations are written `eq` t `=` t', and `ceq` t `=` t' `if` cond for conditional equations. Unconditional and conditional rewrite rules are written, respectively, `rl [`l`] :` t `=>` t' and `crl [`l`] :` t `=>` t' `if` cond. Variables are declared with the keywords `var` and `vars`.

A *class* declaration `class` C `|` $\mathit{att}_1 : s_1, \ldots, \mathit{att}_n : s_n$ declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term `<` $O : C \mid \mathit{att}_1 : \mathit{val}_1, \ldots, \mathit{att}_n : \mathit{val}_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent object-oriented system, a state is a term of sort `Configuration`. It has the structure of a *multiset* of objects and *messages*. Multiset union for configurations is denoted by an associaive and commutative juxtaposition operator (empty syntax), so that rewriting is *multiset rewriting* supported in Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] :
  m(O,w)
  < O : C | a1 : x, a2 : O', a3 : z >
=>
  < O : C | a1 : x + w, a2 : O', a3 : z >
  m'(O') .
```

defines a parameterized family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions change the attribute `a1` of the object `O` and send a new message `m'(O')`. "Irrelevant" attributes (such as `a3`) need not be mentioned in a rule.

A Maude specification is *executable* under reasonable conditions, and the tool offers a variety of formal analysis methods. The *rewrite* command (`rew [`n`]` t_0 `.`) simulates *one* behavior of the system *up to* n rewrite steps from initial state t_0. The *search* command uses breadth-first search to search for states that match a *pattern* and satisfy a given *condition* and that can be reached from the initial state. For example, the search command that searches for all states reachable from t_0 that cannot be further rewritten and that match $\mathit{pattern}$ is written (`search` t_0 `=>!` $\mathit{pattern}$ `.`). When the state space reachable from the initial state is finite, Maude's *linear temporal logic model checker* [6] can check whether each behavior from the initial state satisfies a linear temporal logic formula.

3. Parallelizing Eduction

The semantics of NVIL is a good basis for executing NVIL programs but does not take advantage of distribution in the case of parallel architectures and does not memoize computed values to avoid recomputation. Therefore, we have designed a parallel Eduction Virtual Machine (EVM) to execute intensional NVIL programs.

In this section we describe the parallelism potential of NVIL programs, a distributed mechanism that guides eduction and avoids recomputation (the *warehouse*), and our use of message passing.

3.1 NVIL Parallelism

Lazy evaluation in functional programming languages is inherently sequential: the next symbol to reduce is always the head of the current expression; function arguments are never evaluated on call but later, on demand, during evaluation. On the other hand, strictness gives room for parallel execution: if we have an operation that contains many expressions that will all be needed, we can evaluate them in parallel.

NVIL exposes strictness in a simple way: most built-in operators (such as $+$ and $-$, but not **if**) are strict by definition. These operators are therefore the source of parallelism in the language, since their operands can be evaluated independently in parallel.

3.2 The Warehouse

A common problem of pure lazy functional languages is sharing: the same code may be called multiple times but should only be computed once.

For NVIL, this is illustrated by the program in the right-hand side below, which is the result of transforming the FOFL program in the left-hand side. The subexpression a is used twice:

`result = double(a)`	$\mathit{result} = \textbf{call}_0(\mathit{double})$
`double(x) = x + x`	$\mathit{double} = x + x$
`a = ...`	$a = \ldots$
	$x = \textbf{actuals}(a)$

Part of its execution is shown in the following figure:

$$\begin{tikzcd}
& \langle \mathit{result}, [\,] \rangle \arrow[d, dash] & \\
& \langle \textbf{call}_0(\mathit{double}), [\,] \rangle \arrow[d, dash] & \\
& \langle \mathit{double}, [0] \rangle \arrow[d, dash] & \\
& + \arrow[dl, dash] \arrow[dr, dash] & \\
\underline{\langle x, [0] \rangle} & & \underline{\langle x, [0] \rangle}
\end{tikzcd}$$

Since the transformation exposes the flow of data in the program, shared expressions are seen as the same variable in the same context. In the example above, $\langle x, [0]\rangle$ appears twice during evaluation.

To solve the problem of repeated computations, eduction uses a *warehouse*, which stores already computed values. NVIL programs use the warehouse to demand values for $\langle \mathit{variable}, \mathit{context}\rangle$ pairs and avoid recomputation.

In sequential implementations of intensional languages there is one warehouse which records all computed values. This model is not suitable for a parallel implementation: a single warehouse would become a bottleneck, should many evaluating expressions hit it concurrently. The solution is to use more than one warehouse to distribute the demands. An evaluating expression may ask different warehouses for different $\langle \mathit{variable}, \mathit{context}\rangle$ pairs. A demand for a specific $\langle \mathit{variable}, \mathit{context}\rangle$ pair will always be sent to the same warehouse; the warehouse choice is a function[1] from the variable and the context to a warehouse, therefore no computation can appear in more than one warehouse. It may happen that different nodes ask for the same expression. To avoid recomputation, only one of them should then be allowed to be evaluated, while all the others should wait for it.

The status of an expression node can change during its lifetime: initially it is running, but it will be blocked by the warehouse when demanding the value of some variable in a context. If the warehouse has an already computed value, or if it waits for it from some other node, it will eventually send the value to the asking node. If the warehouse knows nothing about this value, the node continues running, until it becomes a value, or it reaches another demand (and communicates for that demand with the warehouse again).

3.3 The Message Passing Model

To express distributed evaluation, we represent expression nodes and warehouses as processes that communicate through message passing. Expression nodes may spawn new expression nodes and wait for them to send a notification that they finished with a value; they may also communicate with the warehouse, in order to get a value for a $\langle \mathit{variable}, \mathit{context}\rangle$ pair.

The advantage of using message passing is that it unifies the two different ways that eduction can be parallelized: the "fork-join" parallelism that results from the NVIL evaluation tree, together with the safe concurrent access to the distributed warehouse that memoizes results.

4. Formalization in Rewriting Logic

In this section we explain how we have formalized our parallel eduction model in Maude in an object-oriented style. Intensional expressions (*expression nodes*) and warehouses (*warehouse nodes*) are modeled as objects that communicate by message passing.

4.1 Representing NVIL Programs

NVIL expressions are encoded in Maude as follows: a program variable v is represented by the Maude term `$ "`v`"`; a number n by `#` n; a built-in operator $\underline{Op}$ applied to some arguments $E_1, E_2, \ldots$ by `cOp("`Op`",`$\overline{E_1}$ `:` $\overline{E_2}$ `: ...)`, where $\overline{E_i}$ is the Maude representation of E_i; and the intensional operators $\mathbf{call}_j(E)$ and $\mathbf{actuals}(E_1, E_2, \ldots)$ by `call(`$j, \overline{E}$`)` and `actuals(`$\overline{E_1}$ `:` $\overline{E_2}$ `: ...)`. A list of NVIL expressions is represented by either the empty list `snil`, or an expression E followed by another list L ($\overline{E}$: $\overline{L}$).

```
sorts Variable Expr ExprList .
subsort Variable < Expr < ExprList .
```

[1] Our model can also be extended to express redundancy in call-by-name evaluation, if warehouse choice is non-deterministic.

```
op enil : -> ExprList [ctor] .
op _:_  : Expr ExprList -> ExprList [ctor right id: enil] .

op $_      : String -> Variable [ctor] .
op #_      : Int -> Expr [ctor] .
op cOp     : String ExprList -> Expr [ctor] .
op call    : Nat Expr -> Expr [ctor] .
op actuals : ExprList -> Expr [ctor] .
```

Each variable definition $v = E$ of the program is represented by the term `def("`v`",`$\overline{E}$`)`:

```
sort Def .
op def : String Expr -> Def [ctor] .
```

The NVIL definition $x = \mathbf{actuals}(2, x - 1)$ is represented by `def("x", actuals(# 2 : cOp("-", $ "x" : # 1)))`.

A context is a stack, with `snil` the empty stack and `::` the push operation. We represent $\langle \mathit{expr}, \mathit{context}\rangle$ pairs as:

```
sort ExprContext .
op <_;_> : Expr Stack -> ExprContext [ctor] .
```

For example, $\langle v, [1, 0]\rangle$ is represented in Maude by the term `< $ "v" ; (1 :: 0) >`.

4.2 Expression Nodes

For the set of operators that we have used, the tree formed during execution has a maximum branch factor of 2 and we can name nodes accordingly: in the case of an expression node `EN` that has one sub-node, that child is given the name `(EN \ left)`; in the case of two sub-nodes, these are given the names `(EN \ left)` and `(EN \ right)`.

Using the Maude sort `Oid`, for object names, the tree nodes are defined as instances of the following class `Node`:

```
sort NodeStatus .
ops running blocked : -> NodeStatus [ctor] .

class Node | status : NodeStatus,  expr   : ExprContext,
             wp     : Oids,        whs    : Oids,
             prog   : List{Def} .
```

The class `Node` describes expression nodes, which have the following attributes: a scheduling `status` (`running` or `blocked`), the expression currently being evaluated (`expr`), a list of known warehouses to send demands to (`whs`), a list of warehouses to update after evaluation (`wp`) and the read-only program text (`prog`).

4.3 Warehouse Nodes

The distributed warehouse is represented by a set of warehouse nodes, small warehouses that have no knowledge of each other and communicate only with expression nodes that demand values for $\langle \mathit{variable}, \mathit{context}\rangle$ pairs.

Each warehouse node holds a table of `slots` that correspond to the demands it receives. Each slot of this table is a tuple `<` $\overline{\mathit{variable}}$ `;` $\overline{\mathit{context}}$ `;` *entry* `>` which represents what the warehouse knows for a $\langle \mathit{variable}, \mathit{context}\rangle$ pair: if there is a computed value i for it, *entry* is a term `data(`i`)`; if, on the other hand, its computation is still pending from another expression node en, it is `pending(`en`)`. These slots ensure that only one expression node will compute a given $\langle \mathit{variable}, \mathit{context}\rangle$: the first such demand to the warehouse creates a pending slot and notifies the asking expression node to continue with this new computation to reach a value; in the meantime, demands from other expression nodes will block on that pending slot.

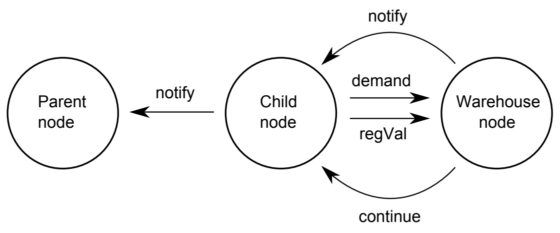


Figure 2. The messages that nodes can exchange.

```
sorts Slot CacheStatus Slots .
subsort Slot < Slots .

op data    : Int -> CacheStatus [ctor] .
op pending : Oid -> CacheStatus [ctor] .
op <_;_;_> : String Stack CacheStatus -> Slot [ctor] .
```

When the expression node that continued to do the computation informs the warehouse that it has a value, all the expression nodes that had to block on that computation must unblock and be notified of this value. To keep this information of blocked nodes that depend on computations of other nodes, the warehouse has to keep another table to record these dependencies. This is table `pdep` ("pending dependencies") and contains pairs `pDep(`en_1, en_2`)`, which state that the result of expression node en_1 should be communicated to expression node en_2 when the former finishes evaluation.

```
sorts PendingDep PendingDeps .
subsort PendingDep < PendingDeps .
op pDep : Oid Oid -> PendingDep .
```

Warehouse nodes are defined as instances of class `WHouse`:

```
class WHouse | slots : Slots, pend : PendingDeps .
```

4.4 Messages

A node en sends a message `demand(`$wh, en, var, ctxt$`)`[2] to a warehouse wh for the value of variable var in context $ctxt$. The warehouse can either reply that it knows the value and it is i (message `notify(`en, wh, i`)`), or tell the node to continue evaluation, as it does not know anything about the demand (message `continue(`en, wh`)`). If the warehouse has a slot that the computation is pending somewhere else, it leaves the node waiting, until it gets a value for it. Warehouse wh is updated when an expression node en that was instructed to `continue`, eventually becomes a value i and sends back a `regVal(`wh, en, i`)` message. Finally, a child node sends a `notify` message, with its value, to its parent when it finishes.

The messages exchanged between nodes are shown in Figure 2.

```
msg demand   : Oid Oid String Stack -> Msg .
msg notify   : Oid Oid Int -> Msg .
msg continue : Oid Oid -> Msg .
msg regVal   : Oid Oid Int -> Msg .
```

4.5 Parallel Execution Rules

The rewrite rules defining parallel eduction are given below, grouped according to their functions: warehouse interaction, process internal state change, and spawn-merge parallelism.

[2] By convention, the first argument of a message is the recipient, the second is the sender, and any remaining ones are the message payload.

We use the following variables in our Maude model:

```
vars VAR OP : String .          var CTXT : Stack .
var N : Nat .                   vars I I1 I2 : Int .
var SLOTS : Slots .             var PDEPS : PendingDeps .
var DEFS : List{Def} .          vars WPEND WS : Oids .
vars E0 E1 E2 : Expr .          var EL : ExprList .
vars EN EN1 EN2 WH : Oid .      var BR : Branch .
```

Interaction with the Warehouse. The following rewrite rules define the communication between the expression nodes and the warehouse nodes.

A `running` node whose expression is an NVIL variable asks the warehouse for a value and blocks:

```
rl [DEMAND_ID] :
  < EN : Node | status : running,
                expr : < $ VAR ; CTXT >, whs : WS >
=>
  < EN : Node | status : blocked >
  demand(choose(VAR, CTXT, WS), EN, VAR, CTXT) .
```

Since a demand for a specific $\langle var, ctxt\rangle$ pair will always be sent to the same warehouse, the warehouse choice (`choose`) can be any function: its implementation does not affect correctness, only performance.

If the warehouse has a value for the demand, it returns that value:

```
rl [UPDATE_FROM_W] :
  demand(WH, EN, VAR, CTXT)
  < WH : WHouse | slots : < VAR ; CTXT ; data(I) > SLOTS >
=>
  < WH : WHouse | >
  notify(EN, WH, I) .
```

The notification message informs expression node `EN` that execution has finished and a value is ready to be used.

If a warehouse is asked for a value of an unknown pair, it will create a pending slot for this new demand and send a message to the asking node to continue evaluation:

```
crl [CREATE_PENDING_SLOT] :
  demand(WH, EN, VAR, CTXT)
  < WH : WHouse | slots : SLOTS >
=>
  < WH : WHouse | slots : < VAR ; CTXT ; pending(EN) >
                          SLOTS >
  continue(EN, WH)
if not VAR ; CTXT in SLOTS .
```

However, if the value demanded is being computed somewhere else, the warehouse will find a pending slot for it. In this case, the demanding expression node will be blocked until the warehouse gets a computed value for its demand. To remember to unblock the node in that case, the warehouse also records this dependency in its `pend` list:

```
rl [CREATE_BLOCKED_SLOT] :
  demand(WH, EN, VAR, CTXT)
  < WH : WHouse | slots : < VAR ; CTXT ; pending(EN1) >
                          SLOTS,
                  pend : PDEPS >
=>
  < WH : WHouse | pend : pDep(EN1, EN) PDEPS > .
```

A `running` node that eventually becomes a value will send a message to the warehouses that wait for it (in set `wp`), in order to register that value; it will also notify its parent:

```
rl [NUM_FINISH] :
  < EN \ BR : Node | status : running,
                     expr : < # I ; CTXT >,
                     wp : WPEND >
=>
  regval-wh(WPEND, EN \ BR, I) notify(EN, EN \ BR, I) .
```

The `regval-wh` function creates a `regVal` message to each warehouse that had to wait for the node's computation. The expression node is then deleted (in a real implementation, the process exits).

A blocked expression node may receive two different messages from the warehouse: either to continue evaluation, or to terminate with a value. A `continue` message from a warehouse lets the node continue evaluating a variable by looking it up in the program:

```
rl [CONTINUE] :
  continue(EN, WH)
  < EN : Node | status : blocked, wp : WPEND,
                expr : < $ VAR ; CTXT >, prog : DEFS >
=>
  < EN : Node | status : running,
                expr : < lookup(VAR, DEFS) ; CTXT >,
                wp : (WH WPEND) > .
```

The node must add `WH` to its `wp` list ("warehouse-pending"), to remember to notify it when the node becomes a value.

A blocked node that gets a notification from the warehouse, for the value it has been waiting for, becomes that value (the empty context `snil` is used as an optimization):

```
rl [TERMINATE_WITH_VALUE] :
  notify(EN, WH, I)
  < EN : Node | status : blocked,
                expr : < $ VAR ; CTXT > >
=>
  < EN : Node | status : running,
                expr : < # I ; snil > > .
```

Finally, a warehouse may receive a message for value registration and must update the pending slots for that node (the `updWH` operation):

```
rl [UPD_PENDING] :
  regVal(WH, EN, I)
  < WH : WHouse | slots : SLOTS, pend : PDEPS >
=>
  < WH : WHouse | slots : updWH(EN, I, SLOTS),
                  pend : delete-waiting(PDEPS, EN) >
  notify-waiting(PDEPS, WH, EN, I) .
```

`PDEPS` contains information about other nodes that `EN` has kept waiting, and is then used by `notify-waiting` to send them notifications that they should unblock and receive value `I`. The `delete-waiting` operation updates the warehouse to forget about these nodes.

Internal Process State. The intensional operators **call** and **actuals** have their standard push-pop semantics. No messages are sent or received, only the internal state of the process changes. The `::` operator represents a given stack as a "head" element and a "tail" stack and `nthExpr(EL, N)` chooses the `N`-th expression from the expression list `EL`:

```
rl [CALL] :
  < EN : Node | status : running,
                expr : < call(N, E0) ; CTXT > >
=>
  < EN : Node | expr : < E0 ; (N :: CTXT) > > .

rl [ACTUALS] :
  < EN : Node | status : running,
                expr : < actuals(EL) ; (N :: CTXT) > >
=>
  < EN : Node | expr : < nthExpr(EL, N) ; CTXT > > .
```

Spawn/Merge Operations. A running built-in operator node *spawns* new running nodes for its subexpressions and blocks. For instance, the rule for the plus operator is as follows (the rules for the other strict operators are similar):

```
rl [SPAWN_PLUS] :
  < EN : Node | status : running,
                expr : < cOp("+", E1 : E2) ; CTXT >,
                whs : WS, prog : DEFS >
=>
  < EN : Node | status : blocked,
                expr : < cOp("+", E1 : E2) ; CTXT > >
  < (EN \ left) : Node | status : running, prog : DEFS,
                         expr : < E1 ; CTXT >,
                         wp : none, whs : WS >
  < (EN \ right) : Node | status : running, prog : DEFS,
                          expr : < E2 ; CTXT >,
                          wp : none, whs : WS > .
```

If all children of a blocked built-in operation node have replied, the node can continue. For instance, for the addition operator, two messages must exist for the parent node `EN`:

```
rl [MERGE_PLUS] :
  notify(EN, (EN \ left), I1)
  notify(EN, (EN \ right), I2)
  < EN : Node | status : blocked,
                expr : < cOp("+", E1 : E2) ; CTXT > >
=>
  < EN : Node | status : running,
                expr : < # (I1 + I2) ; snil > > .
```

5. Formal Analysis

This section explains how we have used Maude to automatically analyze all possible behaviors of our model of the parallel virtual machine for given programs. In particular, we are interested in the following properties:

1. Parallel evaluation should always produce a unique and correct result, if any.
2. No deadlocks should occur during program execution.
3. Finishing with the correct value is not enough: our warehouse nodes must also be in a consistent state, without any pending slots or pending dependencies. Such problems may not affect correctness but lead to space leaks and thus should never occur.

We check these properties for the FOFL program in Section 2.1 (the `fib(3)` function), whose NVIL counterpart is:

$$\begin{aligned}&result = \mathbf{call}_0(\mathit{fib})\\ &\mathit{fib} = \mathbf{if}\ x < 2\ \mathbf{then}\ 1\ \mathbf{else}\ \mathbf{call}_1(\mathit{fib}) + \mathbf{call}_2(\mathit{fib})\\ &x = \mathbf{actuals}(3, x-1, x-2)\end{aligned}$$

We can check all three properties by searching for all reachable states that cannot be further rewritten (notice that any state matches the variable `C` of sort `Configuration`):

```
Maude> (search init =>! C:Configuration .)

Solution 1
C:Configuration -->
 < whouse(0) : WHouse | pend : none, slots :
     < "fib" ; 1 :: 0 :: snil ; data(2) >
```

```
      < "fib" ; 2 :: 0 :: snil ; data(1) >
      < "x" ; 1 :: 0 :: snil ; data(2) >
      < "x" ; 2 :: 0 :: snil ; data(1) >
      < "result" ; snil ; data(3) > >
 < whouse(1) : WHouse | pend : none, slots :
      < "fib" ; 0 :: snil ; data(3) >
      < "fib" ; 1 :: 1 :: 0 :: snil ; data(1) >
      < "fib" ; 2 :: 1 :: 0 :: snil ; data(1) >
      < "x" ; 0 :: snil ; data(3) >
      < "x" ; 1 :: 1 :: 0 :: snil ; data(1) >
      < "x" ; 2 :: 1 :: 0 :: snil ; data(0) > >

No more solutions.
```

Property 1 is satisfied, since only one final state is found; the unique result is 8, as expected. No other states were reached, therefore no deadlocked states are reachable. For the last property, we inspect the values of the `slots` and `pend` attributes of the warehouses: no pending slots or dependencies exist in the final state.

The table below shows the execution time for the search command for some programs and numbers of warehouses:

Program	Number of warehouses			
	2	3	4	10
`fib(3)`	73 sec	76 sec	76 sec	88 sec
`ack(1,3)`	123 sec	147 sec	124 sec	131 sec
`fact(6)`	131 sec	133 sec	136 sec	152 sec

Since Maude's model checker is fast, these execution times expose the large number of different behaviors possible in our model, even for seemingly simple example programs. The number of warehouses used does not affect significantly the execution time of our search.

It is also worth mentioning that even in a relatively simple model like ours, model checking uncovered two errors in a previous version of the model.

6. Parallel Implementation

We have used Erlang [1], a mature concurrent functional programming language with an excellent parallel runtime system, to implement our parallel eduction model.

Since the Maude model is defined in a message passing style, and since Erlang is based on message passing between processes, implementation in Erlang turned out to be very easy. However, we identified the following issues:

- The rewrite rules are divided into two groups: (a) rules that consume a message and send another message (`UPDATE_FROM_W`, `CREATE_PENDING_SLOT`, `UPD_PENDING`) or change the internal state of a process (`CREATE_BLOCKED_SLOT`, `CONTINUE`, `TERMINATE_WITH_VALUE`, `MERGE`-rules), and (b) rules that are activated according to the current internal state of a proccess (`DEMAND_ID`, `NUM_FINISH`, `CALL`, `ACTUALS`, `SPAWN`-rules). Rules in the first group are easy to transfer directly as receive clauses in Erlang; those in the second group contain program logic and their implementation does not follow automatically from the Maude model.
- An important implementation choice is how to choose the warehouse to use every time a node reaches an identifier in a context; this is the hard problem of cache locality. A simple function that we used was to choose a warehouse according to the length of the context, which resulted in arguments to the same function call of the initial program residing in the same warehouse node.
- For efficient data storage and manipulation in the warehouses, we used the Erlang ETS tables.

This made our prototype easy to implement, based on the rewriting logic model, and still, useful for real-world tests.[3]

7. Related Work

We discuss related work on semantics and implementations of parallel lazy functional languages, dataflow, and intensional programming.

Semantics of Parallel Lazy Functional Languages. An operational semantics for parallel lazy evaluation is given by Baker-Finch *et al.* [3] for the graph reduction-based GpH. A distributed operational semantics for a parallel functional language is given by Hidalgo-Herrero and Ortega-Mallén [14] for the explicit parallel programming system Eden. It also expresses communication between parts of the program with message passing at a higher level, but it is not fully lazy, permitting speculative evaluation.

Another view on the relationship between rewriting and lazy functional languages implementations is explored by Plasmeijer and v. Eekelen [23] for graph rewriting systems; however, they treat explicit parallelism, whereas we describe implicit parallelism.

Our warehouse is a distributed associative memory with block-on-read semantics; another one is tuple spaces [10], used e.g., by Peterson *et al.* [20] for Haskell programming. Tuple spaces are similar to our distributed warehouse, in the sense that both form a middleware of the runtime system that controls parallelism. They differ however in their communication model: ours is based on message passing, while tuple spaces are a standalone, alternative solution to communication between parts of the program.

Lazy Functional and Dataflow Languages. The distributed warehouse that we have described shares some fundamental ideas with traditional implementations of graph reduction [16, §2.4.1], our blocking being similar to the "black hole" technique [18] (which however relies on polling instead of message passing).

The GUM parallel implementation for Haskell is based on message passing [29], using messages to move pieces of the program graph between parts of the program, since it is based on graph reduction. By contrast, our parallel eduction moves identifiers, contexts and values between the expression and the warehouse nodes, expressing a distributed dataflow view on the functional program.

Flanagan and Nikhil [7] describe the dataflow implementation of pHluid. Its distributed caching model is very different from the warehouse, as it is not based on eduction.

Intensional Languages. The TransLucid scientific intensional language has a multithreaded implementation [24], the TVM (TransLucid Virtual Machine). The TVM shares basic principles with our virtual machine but depends on explicit parallelism and has a centralized warehouse.

Warehouses are frequently used in implementations of eduction. Operational semantics may be found in the work of Plaice *et al.* [21, 22, 24] for centralized warehouses. A distributed model for eduction is proposed in the work of Vassev and Paquet [32], related to the implementation of the General Intensional Programming System (GIPSY) [12]. In its latest implementation (which — to the best of our knowledge — is not publicly available), GIPSY supports a distributed warehouse (a collection of "demand migration systems") implemented using Java technologies for distributed software. However, GIPSY targets multi-machine distributed environments, which is a quite different domain from what our work is concerned with (the semantics and implementation of lazy functional languages), with different performance and scalability requirements.

[3] Our Maude formalization and the Erlang prototype are both available at `http://www.softlab.ntua.gr/~gfour/lazy_par_vm.zip`

8. Concluding Remarks

We have given a formal semantics for a parallel model of a lazy functional language using rewriting logic. The semantics is based on the intensional transformation, which converts programs in the original language to programs in NVIL, a zero-order dataflow language with intensional operators. Parallelization makes use of distributed memoization, whose implementation is based on message passing. We have built a prototype implementation of a parallel virtual machine implementing this semantics in Erlang. We have also formally analyzed our model in Maude.

NVIL was originally conceived to express tagged-token demand-driven dataflow. With our rewriting semantics, execution of NVIL programs is expressed using message passing. The performance of our implementation can be much improved, as the computation model described here exposes maximum parallelism that needs fine-tuning.

Since the warehouse represents lazy activation records that may be built during execution [5], our semantics can also model distributed lazy activation records, with each argument entry being an Oz-like dataflow synchronization variable [28], or an I-structure as in Id [2]. As the use of lazy activation records has been shown to perform significantly better than memoized eduction for the implementation of NVIL [5], we plan to further investigate this possibility to implement efficiently distributed lazy activation records.

We also intend to use the same technique to implement an extension of the original intensional transformation, which handles a higher-order lazy functional programming language, with partial application and user-defined datatypes [8]. We are also working on abstraction techniques for verifying not only instances of our model in Maude, but the general case.

Acknowledgment. G. Fourtounis was supported by the EEA FM EL0086 NTUA Mobility and Scholarship Program.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[2] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39:300–318, 1990. doi: 10.1109/12.48862.

[3] C. Baker-Finch, D. J. King, and P. Trinder. An operational semantics for parallel lazy evaluation. *SIGPLAN Not.*, 35:162–173, 2000. doi: 10.1145/357766.351256.

[4] G. Boudol. Some chemical abstract machines. In *A Decade of Concurrency Reflections and Perspectives*, volume 803 of *LNCS*. Springer, 1994.

[5] A. Charalambidis, A. Grivas, N. S. Papaspyrou, and P. Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, 2(1):123–141, 2008. doi: 10.1007/s11786-008-0047-5.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude*, volume 4350 of *LNCS*, 2007. Springer.

[7] C. Flanagan and R. S. Nikhil. pHluid: The Design of a Parallel Functional Language Implementation on Workstations. In *Proc. ICFP'96*. ACM, 1996.

[8] G. Fourtounis, N. Papaspyrou, and P. Rondogiannis. The Intensional Transformation for Functional Languages with User-Defined Data Types. In *Proc. PLS8*, 2011. (to appear).

[9] F. Gava and F. Loulergue. Verifying Functional Bulk Synchronous Parallel Programs Using the Coq System. *TPHOLS03 Emerging Trends*, (2003-02), 2003.

[10] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, 1985. doi: 10.1145/2363.2433.

[11] K. Hammond and G. Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 2000.

[12] B. Han, S. A. Mokhov, and J. Paquet. Advances in the Design and Implementation of a Multi-tier Architecture in the GIPSY Environment with Java. In *Proc. SERA'10*. IEEE, 2010. doi: 10.1109/SERA.2010.40.

[13] T. Harris and S. Singh. Feedback directed implicit parallelism. *SIGPLAN Not.*, 42:251–264, 2007. doi: 10.1145/1291220.1291192.

[14] M. Hidalgo-Herrero and Y. Ortega-Mallén. A Distributed Operational Semantics for a Parallel Functional Language. In *Proc. SFP'00*. Intellect Books, 2000.

[15] S. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *Proc. FSTTCS'08*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008. doi: 10.4230/LIPIcs.FSTTCS.2008.1769.

[16] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, 1998.

[17] R. Loogen, Y. Ortega-Mallén, and R. Peña Marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15:431–475, 2005. doi: 10.1017/S0956796805005526.

[18] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *Proc. ICFP'09*. ACM, 2009. doi: 10.1145/1596550.1596563.

[19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96:73–155, 1992. doi: 10.1016/0304-3975(92)90182-F.

[20] J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In *PADL'00*, volume 1753 of *LNCS*. Springer, 2000.

[21] J. Plaice. Multidimensional Lucid: Design, Semantics and Implementation. In *Proc. DCW'00*, volume 1830 of *LNCS*. Springer, 2000.

[22] J. Plaice, B. Mancilla, G. Ditu, and W. W. Wadge. Sequential demand-driven evaluation of Eager TransLucid. *COMPSAC'08*, 2008. doi: 10.1109/COMPSAC.2008.191.

[23] R. Plasmeijer and M. v. Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[24] T. Rahilly and J. Plaice. A multithreaded implementation for TransLucid. In *COMPSAC*. IEEE, 2008.

[25] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*. ACM, 1972.

[26] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *J. Funct. Program.*, 7:73–101, 1997. doi: 10.1017/S0956796897002633.

[27] P. Rondogiannis and W. W. Wadge. Higher-order functional languages and intensional logic. *J. Funct. Program.*, 9(5):527–564, 1999. doi: 10.1017/S0956796899003445.

[28] P. v. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[29] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proc. PLDI'96*, 1996.

[30] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Funct. Program.*, 8(1):23–60, 1998. doi: 10.1017/S0956796897002967.

[31] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed Haskells. *J. Funct. Program.*, 12:469–510, 2002. doi: 10.1017/S0956796802004343.

[32] E. Vassev and J. Paquet. A general architecture for demand migration in a demand-driven execution engine in a heterogeneous and distributed environment. In *Proc. CNSR'05*, 2005. doi: 10.1109/CNSR.2005.9.