SERIE B - INFORMATIK

Improving Parallel Implementations of Lazy Functional Languages Using Evaluation Transformers

Matthias Horn

B 95-15
November, 1995

Abstract

This report outlines a parallel abstract machine for the implementation of a lazy functional core language. The evaluation transformer model of reduction [1] is used to control the evaluation process. An evaluator space containing parameterised evaluators for structured and polymorphic types is introduced. Using this framework it is possible to handle runtime information about evaluators for arbitrary structured types as well as evaluation transformers for functions over polymorphic types.

Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin
Takustraße 9
D-14109 Berlin
`horn@inf.fu-berlin.de` `http://www.inf.fu-berlin.de/~horn/`

1 Introduction

Because of their referential transparency, pure functional languages are especially well suited for implementation on parallel machines. Since they do not allow side effects, subexpressions can be evaluated in arbitrary order or in parallel. Lazy functional languages with lazy evaluation retain this property as well. But with these languages a subexpression will not be evaluated until its result is necessary for another computation. Therefore it is not possible to decide if a subexpression is ever evaluated. There is only one situation where this decision can be made - when a function is called which is known to evaluate its argument in any case. A lot of work has been done to find as many of these arguments as possible (strictness analysis). Until now the information obtained by this analysis has not proved to be able to produce enough independently evaluable expressions to saturate large multiprocessors. One reason for this is the fact that strictness only means that evaluation to weak head normal form is necessary. But expressions are much more complex and often they are evaluated further than to weak head normal form. The evaluation transformer analysis [2, 17, 18] addresses this topic. It reasons about the amount of evaluation an argument will go through, if the function is evaluated with a particular amount of evaluation. If for instance a call to a function which counts elements of lists is evaluated, the complete list structure will be evaluated, but no element needs any evaluation. The argument for a function adding all elements of number lists will be evaluated completely, i.e. evaluation to normal form. The evaluation transformer analysis[1] seems to produce many more computable expressions than simple strictness analysis.

Some work to implement lazy functional languages using evaluation transformers has already been done [1, 6, 8, 10, 12]. But all these approaches mainly deal with evaluators for simple structured types like lists and trees. The problems arising when using evaluators for general structured types are not addressed. One reason for this is the absence of practicable analysis methods for this general case. [17] deals with evaluation transformer analysis for structured types, but it restricts the types which can be analysed. Further work on practicable analysis methods is on the way [18].

This report defines a universally valid evaluator space containing evaluators for general structured types as well as polymorphic types, and a parallel abstract machine which implements evaluation transformers on distributed memory parallel computers like Fujitsu AP1000 and Cray T3D. As compiler input, a core language program annotated with type and evaluation transformer information for each function is assumed.

Section 2 defines the evaluator space for structured types. In section 3 the representation of evaluators at runtime is introduced. Section 4 outlines the abstract machine model used to implement the language.

2 The Evaluator Space

The amount of evaluation an expression can go through is mainly determined by its type. Expressions of atomic types like numbers can only be evaluated to normal form, which is equal to weak head normal form in this case. Another evaluator which can be applied to all expressions is “no evaluation”. Expressions of structured types can go through many different kinds of evaluation.

[1] There is another terminology: evaluators are called contexts, evaluation transformer analysis is called context analysis and evaluation transformers are called context maps.

2.1 Polymorphic Types

Since in modern functional programming polymorphic types play an important role, it is necessary to pay special attention to this feature. Therefore an evaluator space containing parameterised evaluators for polymorphic types is defined in this report. Functions over polymorphic types can have transformers for parameterised evaluators.

An example for the problem addressed here is the function `reverse` for polymorphic lists. Some of the evaluators of type `[α]`[2] are for instance:

- no evaluation
- evaluate to weak head normal form
- evaluate the complete list structure and do not evaluate any element
- evaluate the complete list structure and evaluate each element using evaluator *a*

One of the evaluation transformers of `reverse : [α] -> [α]` is:

> "If the application of `reverse` is evaluated to weak head normal form, then the complete structure of its argument will be evaluated, but no element needs to be evaluated. "

Another evaluation transformer of `reverse : [α] -> [α]` is:

> "If the application is evaluated to complete structure, and each element is evaluated using evaluator *a*, then the argument will be evaluated to the same degree."

The evaluator *a* belongs to type α. The transformer is valid for all types α and all evaluators *a*.
Because it is desirable to compile a function over polymorphic types only once, the types used for α during runtime cannot be determined at compile time. In the case of separate compilation they are not even known. By introducing parameterised evaluators, the resulting code is able to split the evaluator at runtime into parts for each argument type and can propagate them to the appropriate subexpressions.

2.2 Runtime Updating of Evaluators

Often the function called at runtime is not known at compile time. An example for this case is the function `select`.

During the execution of `app` the evaluator valid for `list` is not known until `expr` has

```
> select = if expr then
>            reverse
>          else
>            λx.[]
> app list = select list
```

been evaluated. In particular it is not known at compile time. There is no evaluation transformer information for `app` available. The only safe evaluation which can be applied to

[2] In the remainder Greek letters like α,β and τ are used for type variables and Latin letters like a,b and c are used for evaluator variables.

`list` is “no evaluation”. But since `reverse` has an evaluation transformer, the reduction process is able to initiate the evaluation of `list` to complete structure as soon as the expression `reverse list` has been built. The evaluator will be determined at runtime.
The expression used as argument `list` can be used in other expressions as well. So it might have already obtained an evaluator. In this case it is necessary to “merge” two evaluators to obtain a new one which performs as much evaluation as both evaluators one after the other would do.

2.3 Evaluators

The evaluator ξ_{NO}[3], which does nothing, can be applied to all types. Atomic types like numbers and structured types, containing only parameterless constructors, allow only one additional evaluator $\xi_{NF} = \xi_{WHNF}$ which evaluates a term to its normal form. Evaluators for structured types are defined recursively over the structure of the type. A structured type has some polymorphic type arguments and consists of some alternatives having a constructor and zero or more type expressions as arguments. Evaluators for a given structured type are defined by a tuple of evaluators - one for each argument type of the constructors. For recursive type definitions the evaluators can also be defined recursively. Each evaluator except ξ_{NO} implies the evaluation to weak head normal form.

The example for the following explanations is a tree parameterised by types for values contained in leaves and nodes.

```
> data Tree α β = Leaf α
>               | Node (Tree α β) β (Tree α β)
```

Some evaluators might be:

- No evaluation

  ξ_{NO}
- Evaluation to weak head normal form and a given evaluation *a* to the argument if the term is a leaf and evaluation *b* to the second argument if the term is a node, no recursive evaluation

  $\xi_{HEAD}\ a\ b = \{Leaf(a), Node(\xi_{NO}, b, \xi_{NO})\}$[4]
- The evaluation to weak head normal form ξ_{WHNF} can be obtained by parameterising ξ_{HEAD} with ξ_{NO} for both subtypes.
- Evaluation of the complete structure and a given evaluation *a* to the values of all leaves and evaluation *b* to the values of all nodes

  $\xi_{TREE}\ a\ b = \{Leaf(a), Node(\xi_{TREE}\ a\ b, b, \xi_{TREE}\ a\ b)\}$
- The evaluation which evaluates the complete structure and no value can be obtained by parameterising ξ_{TREE} with ξ_{NO} for both subtypes.

[3] The letter ξ for evaluators was introduced by [2] because it looks like the letter E.
[4] evaluate to WHNF, apply the appropriate evaluators to the arguments of the resulting constructor

- Evaluation of an alternating path through the tree starting left and a given evaluation *b* to the values of all nodes on the path, a given evaluation *a* to the value of the leaf at the end of the path

  $\xi_{PL}\ a\ b = \{Leaf(a), Node(\xi_{PR}\ a\ b, b, \xi_{NO})\}$
- Evaluation of an alternating path through the tree starting right and a given evaluation *b* to the values of all nodes on the path, a given evaluation *a* to the value of the leaf at the end of the path

  $\xi_{PR}\ a\ b = \{Leaf(a), Node(\xi_{NO}, b, \xi_{PL}\ a\ b)\}$

2.4 Combining Evaluators

Two evaluators can be combined to one evaluator which performs exactly as much evaluation as the two evaluators would do one after the other. The combination operator is denoted by $\cup_\xi$.

$$\xi_1 = \{Leaf(a_1), Node(b_1,c_1,d_1)\}$$
$$\xi_2 = \{Leaf(a_2), Node(b_2,c_2,d_2)\}$$
$$\Rightarrow$$
$$\xi_1 \cup_\xi^{(Tree\,\alpha\,\beta)} \xi_2 = \{Leaf(a_1 \cup_\xi^{\alpha} a_2), Node(b_1 \cup_\xi^{(Tree\,a\,b)} b_2, c_1 \cup_\xi^{\beta} c_2, d_1 \cup_\xi^{(Tree\,\alpha\,\beta)} d_2)\}$$

The combination of evaluators must not depend on the parameters of the combined operators.

The combination operation $\cup_\xi^\tau$ for evaluators of type τ satisfies the rules:

(1) $a^\tau \cup_\xi^\tau a^\tau = a^\tau$

(2) $a^\tau \cup_\xi^\tau b^\tau = b^\tau \cup_\xi^\tau a^\tau$

(3) $(a^\tau \cup_\xi^\tau b^\tau) \cup_\xi^\tau c^\tau = a^\tau \cup_\xi^\tau (b^\tau \cup_\xi^\tau c^\tau)$

Examples:

$$(\xi_{TREE}\ a\ b) \cup_\xi^{(Tree\,\alpha\,\beta)} (\xi_{TREE}\ \xi_{NO}\ \xi_{NO}) = (\xi_{TREE}\ a\ b)$$

$$(\xi_{HEAD}\ a_1\ b_1) \cup_\xi^{(Tree\,\alpha\,\beta)} (\xi_{TREE}\ a_2\ b_2) = (\xi_{HTREE}\ a_1\ b_1\ a_2\ b_2)$$

with: $\xi_{HTREE}\ a_1\ b_1\ a_2\ b_2 = \{Leaf(a_1), Node(\xi_{TREE}\ a_2\ b_2, b_1, \xi_{TREE}\ a_2\ b_2)\}$

$$(\xi_{TREE}\ a_1\ b_1) \cup_\xi^{(Tree\,\alpha\,\beta)} (\xi_{PL}\ a_2\ b_2) = (\xi_{SPL}\ a_1\ b_1\ a_2\ b_2)$$

with:

$$\xi_{TPL}\ a_1\ b_1\ a_2\ b_2 = \{Leaf(a_1 \cup_\xi^{\alpha} a_2), Node((\xi_{TPR}\ a_1\ b_1\ a_2\ b_2), b_1 \cup_\xi^{\beta} b_2, (\xi_{TREE}\ a_1\ b_1))\}$$
$$\xi_{TPR}\ a_1\ b_1\ a_2\ b_2 = \{Leaf(a_1 \cup_\xi^{\alpha} a_2), Node((\xi_{TREE}\ a_1\ b_1), b_1 \cup_\xi^{\beta} b_2, (\xi_{TPL}\ a_1\ b_1\ a_2\ b_2))\}$$

The set of evaluators for a particular type τ found in the source program is finite. The set of evaluators which can be constructed by combining an arbitrary number of evaluators from the source program forms the evaluator space E^τ for type τ. The properties of $\cup_\xi^\tau$ ensure the finiteness of E^τ. The set of evaluators found in the source program is a system of generators for E^τ. Its existence is sufficient for the implementation proposed in this report. For a more efficient implementation it is desirable to find an orthogonal basis for E^τ. Until now it is not completely clear how such a basis can be constructed. In the remaining parts of this report the set of evaluators which generates the evaluator space is called evaluator basis although it is only proved to be a system of generators.

3 Evaluator Representation

The input for compilation described in this report is a core language program annotated by evaluation transformer information. Each function has a transformer which maps evaluators for the function application to evaluators for arguments. The finite set of evaluators for each type found in the source program is collected, completed with respect to $\cup_\xi$ and the evaluator basis is computed.

3.1 Evaluator Identifiers

During runtime an evaluator is represented by an identifier. The main idea is to choose this identifier in such a way that the combination of two evaluators is simply a binary OR operation. This trick leads to a very efficient implementation of dynamic evaluator changes.

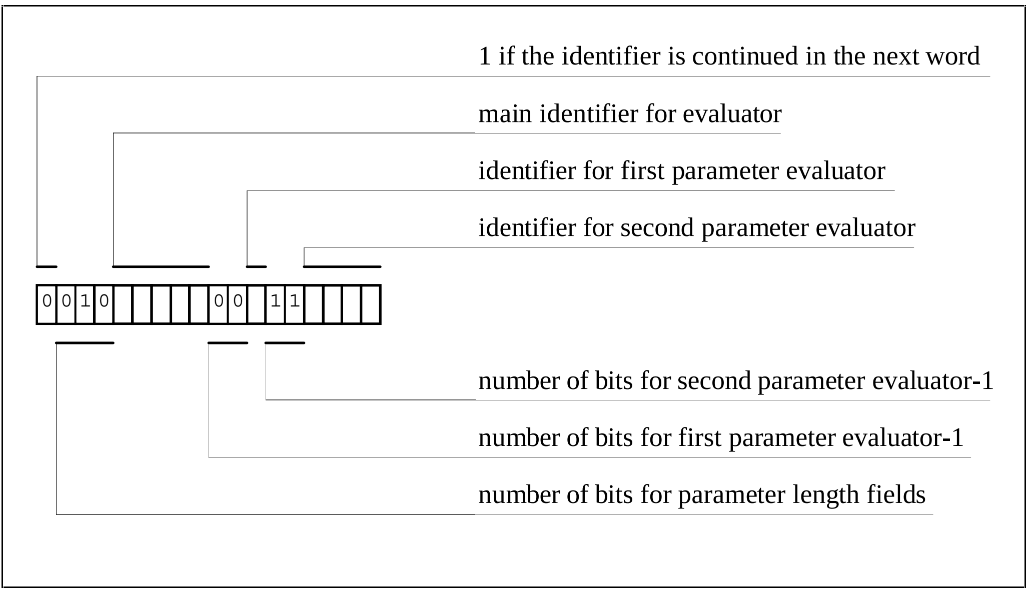
1 if the identifier is continued in the next word
main identifier for evaluator
identifier for first parameter evaluator
identifier for second parameter evaluator
0 0 1 0 0 0 1 1
number of bits for second parameter evaluator-1
number of bits for first parameter evaluator-1
number of bits for parameter length fields

Figure 1: Evaluator Identifier

The first bit in each identifier indicates whether this identifier is continued in the next machine word[5]. Each of the evaluators in the evaluator basis corresponds to a bit in the main identifier starting at the fifth bit. The evaluator ξ_{WHNF} which plays a special role corresponds to the first bit of these. Because all other evaluators can be defined as a combination of basis evaluators, their identifiers are composed from the basis identifiers by binary OR.

The bit for ξ_{WHNF} is set to 1 in all evaluator identifiers except ξ_{NO}. The main identifier is followed by identifiers for the parameters. Because an evaluator can be combined from all evaluators contained in the evaluator basis, it possibly uses all parameters of these evaluat-

[5] The word size depends on the target processor.

ors. Therefore the identifier has as many parameters as all basis evaluators together. The lengths of the parameter identifiers are not known for polymorphic types, hence each parameter is preceded by its length. The number of bits used for the parameter length is determined once and stored in the second to fourth bit of the complete identifier. If all parameter types are atomic, the length information is omitted. The length of the main identifier need not be stored since it is only used in contexts where the type the expression delivers is known. The first bit is used to determine the length of the complete identifier in situations where the type is unknown. Because the length fields in all evaluators for a particular type are equal, they will not be changed during binary OR operations[6].

3.2 Examples

Expressions of atomic types like numbers or functions can only be evaluated using $\xi_{WHNF} = \xi_{NF}$ or ξ_{NO}. The evaluation basis for such types only contains ξ_{WHNF}. The main identifier needs only one bit. There are no arguments, therefore it is not necessary to store the number of bits for the argument length fields. The following identifiers are assigned to the evaluators:

ξ_{NO} =	`0.0`[7]
ξ_{WHNF} =	`0.1`

Expressions of list types can be evaluated using the evaluators:

ξ_{NO}
$\xi_{WHNF}\ a = \{Nil, Cons\ a\ \xi_{NO}\}$
$\xi_{SPINE}\ a = \{Nil, Cons\ a\ (\xi_{SPINE}\ a)\}$

Possibly there are more sophisticated evaluators, but in this example the source program is assumed to contain only these three. The evaluator basis contains ξ_{WHNF} and ξ_{SPINE}. Since each of them has one argument, the complete identifier needs two arguments. The main identifier `10` is assigned to ξ_{WHNF}. The evaluator ξ_{SPINE} corresponds to the main identifier `11`. The first bit for ξ_{WHNF} is set in identifiers for all types except for ξ_{NO}.
If the list is parameterised with atomic types, the following identifiers are assigned to the evaluators:

ξ_{NO} =	`0.000.00.0.0`
$\xi_{WHNF}\ \xi_{NO}$ =	`0.000.10.0.0`
$\xi_{SPINE}\ \xi_{NO}$ =	`0.000.11.0.0`
$\xi_{WHNF}\ \xi_{WHNF}$ =	`0.000.10.1.0`
$\xi_{SPINE}\ \xi_{WHNF}$ =	`0.000.11.0.1`

This example uncovers a problem of the chosen evaluator basis. The identifiers `0.000.11.0.1` and `0.000.11.1.1` describe the same evaluator. It would be better to take

$$\xi_{TSPINE}\ a = \quad \{Nil, Cons(\xi_{NO}, \xi_{WHNF}\ a \cup_{\xi} \xi_{TSPINE}\ a)\}$$

as the second basis evaluator.

[6] This is also valid for instances of polymorphic types, because only evaluators for the same instance of a polymorphic type are combined.
[7] To improve readability the identifier parts are separated by dots.

If the list is parameterised with a list of atomic types the following identifiers are assigned to the evaluators:

$\xi_{NO} =$ `0.011.00.110.0000000.110.0000000`[8]

$\xi_{SPINE}\ (\xi_{WHNF}\ \xi_{WHNF}) =$ `0.011.11.110.0000000.110.0001010`

$\xi_{WHNF}\ (\xi_{SPINE}\ \xi_{NO}) =$ `0.011.10.110.0001100.110.0000000`

$\xi_{WHNF}\ (\xi_{SPINE}\ \xi_{NO}) \cup_{\xi} \xi_{SPINE}\ (\xi_{WHNF}\ \xi_{WHNF}) =$ `0.011.11.110.0001100.110.0001010`

The type `Tree` defined in section 2.3 has four evaluators in the evaluator basis:

$\xi_{HEAD}\ a\ b = \{Leaf(a), Node(\xi_{NO}, b, \xi_{NO})\}$

$\xi_{TREE}\ a\ b = \{Leaf(a), Node(\xi_{TREE}\ a\ b, b, \xi_{TREE}\ a\ b)\}$

$\xi_{PL}\ a\ b = \{Leaf(a), Node(\xi_{PR}\ a\ b, b, \xi_{NO})\}$

$\xi_{PR}\ a\ b = \{Leaf(a), Node(\xi_{NO}, b, \xi_{PL}\ a\ b)\}$

Thus the main identifier is four bits long. Each of the evaluators has two arguments. The complete identifier needs eight arguments. If it is parameterised with atomic types the following identifiers are assigned to the evaluators:

$\xi_{HEAD}\ a\ b =$ `0.000.1000.a.b.0.0.0.0.0.0`

$\xi_{TREE}\ a\ b =$ `0.000.1100.0.0.a.b.0.0.0.0`

$\xi_{PL}\ a\ b =$ `0.000.1010.0.0.0.0.a.b.0.0`

$\xi_{PR}\ a\ b =$ `0.000.1001.0.0.0.0.0.0.a.b`

The letters `a` and `b` are replaced by `1` if the corresponding subexpression must be evaluated and otherwise by `0`.

4 The Abstract Machine

The abstract machine described in this section is spineless and almost tagless. The activation records controlling the reduction process are placed in graph nodes. This makes handling of tasks across different processors much easier. There is no special scheduling processor. All processors are involved in load balancing, synchronisation, etc. This enables the machine to be scalable even on large networks of processors. Because shared memory multiprocessor machines are not truly scalable, the parallel machine underlying the abstract model is assumed to have a distributed memory architecture.

The parallel abstract machine has a number of processing elements. Each element contains a processor and local memory. The access times to local memory are assumed to be much shorter than accesses to other processing elements. At every moment each processor executes different instructions on different data (MIMD - multiple instruction multiple data). It is not practicable to load each processor on machines with many processing elements with its own program. Therefore all processors are assumed to run the same program (SPMD - single program multiple data). The parallel abstract machine model does not commit the kind of communication between processing elements. There are only a few more abstract communication instructions like “transfer a graph node from processing element x to processing element y”. The actual implementation is shifted into the runtime system. This enables the machine to be implemented on different types of distributed memory computers as efficiently as possible.

[8] The size of machine words is assumed to be at least 32 bit. If it were 16 bit, the identifier would be split into the words `1.011.00.110.0000000` and `0.110.0000000`. The first bit indicating whether the id is continued in the next word, is contained in each word.

The abstract machine works on three different types of data. The graph nodes are used to represent expressions still to be evaluated. They build a possibly cyclic graph. There is one start node which represents the main function call of the functional program. The second data structure is the reducible nodes queue which holds references to graph nodes reducible at the moment. The third data structure represents function tables. These tables contain pointers to functions which handle different events. Graph nodes always refer to a function table. The tables are statically allocated at program start-up time.

4.1 Graph Nodes

The expressions still to be evaluated are represented as a graph.

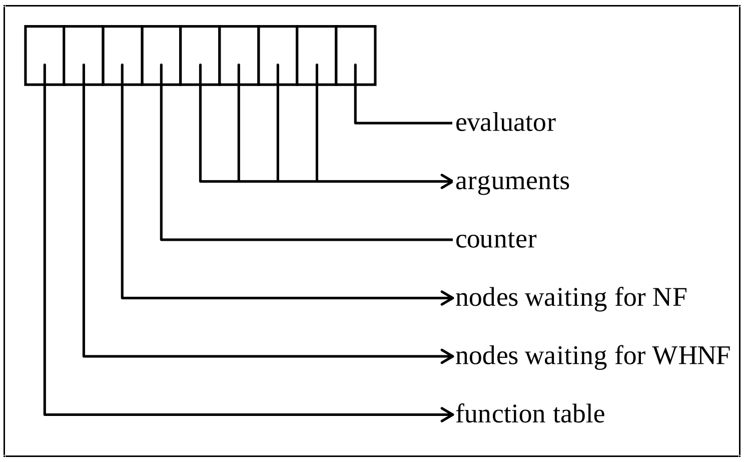

Figure 2: Ordinary Graph Node

An ordinary graph node consists of several parts:

- a pointer to a function table
- a pointer to a list of graph nodes waiting for the weak head normal form of this graph node
- a pointer to a list of graph nodes waiting for the normal form of this graph node
- a counter used to count the subexpression not yet evaluated completely
- references to other graph nodes, one for each argument
- an entry which holds the current evaluator identifier

Because the number of arguments and the size of evaluators vary, the size of graph nodes is not fixed. Their size cannot even be computed at compile time. The function table is a statically created array of function pointers to handle the different events which can happen to a graph node. The references to arguments point to graph nodes. The evaluator is stored at the end of the graph node. Thus it is not necessary to know the evaluator size to access the arguments.

Indirection graph nodes are used to refer to another graph node. They are necessary if a

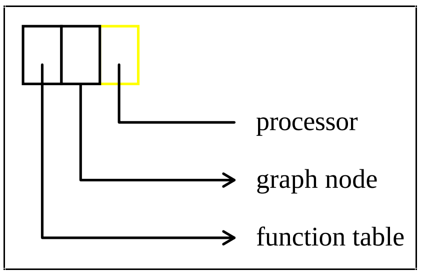

Figure 3: Indirection Graph Node

graph node has to be updated with a constructor or function application requiring more space than before. The node is updated with an indirection node containing a reference to a new graph node. Because these nodes are always located on the same processing element, they do not contain a processor field. If a graph node is located at another processor, an indirection node points to it. Only these indirection nodes contain a processor field. In order to avoid multiple references from one processor to the same node at another processor, all references to this node are routed through the same indirection node.

Graph nodes are allocated dynamically and may become unused without notification. Therefore a garbage collector is necessary. In other implementations of functional programming languages many different techniques are used. Generational garbage collection seems to be the most successful technique [15]. Although they have several drawbacks, simple algorithms as weighted reference counting have proved to work satisfactorily [9, 11]. Recent work also promises good properties for mark and scan algorithms [4, 19].

4.2 Reducible Nodes Queue

The reducible nodes queue (RNQ) is the main control structure of the abstract machine described here. Each processor has its own queue. It holds references to graph nodes in local memory still to be reduced. The main execution loop of the machine takes one node from the queue and performs the necessary steps for its reduction. These steps may lead to other reduction tasks. A reducible node is appended to the RNQ if the processor has the possibility to continue evaluation in more than one direction. This can happen in two cases:

- A function application has an evaluation transformer which initiates evaluation for arguments. All the nodes which have received an evaluator are appended to the RNQ. The processor continues the normal reduction process.
- A function has more than one strict argument. The first arguments are appended to the RNQ. The last one is evaluated by the processor itself.

The reduction process for a program is started by creating a node for the main function and appending it to the reducible nodes queue. The reduction stops if this main node has been evaluated to normal form.
If a processor needs to evaluate a node located on another processing element, it informs the other one. The exact technique used to notify other processors depends on the underlying architecture. On the other processing element the task will be appended to the RNQ. The reducible nodes queue always points to nodes on the same processing element. This is commonly called “owner computes rule”. If a processor runs out of work, i.e. its RNQ is empty and the main node has not yet reached normal form, it queries other processors for work. Another processor which has enough reducible nodes transfers at least one node.

Then the evaluation process continues until again a processor runs out of work. The technique, not to transfer tasks to other processors before the other processor has run out of work, is also called lazy task creation [13]. If all processors are busy, no transfers take place.
The reducible nodes queue is used from both ends. The local processor always uses the one, while reducible nodes transferred to other processing elements are taken from the other. This hopefully leads to transfers of large tasks. Evaluation requests from other processors are put at the “local” end of the queue. In order to prevent the other processing element from waiting they should be served as fast as possible.
The structure of the RNQ entries depends on the underlying architecture. On machines with a message based communication system like Fujitsu AP1000 the reducible nodes queue only holds pointers to graph nodes. On virtual shared memory machines like Cray T3D, where communication is done by accessing the memory of another processing element, the RNQ also contains additional data which determines the action to be performed. This is necessary if a particular communication cannot be done by the initiating processing element alone. Then an action is put into the RNQ and the other processor will execute it when it accesses the reducible nodes queue the next time. In this case the RNQ partly works like a message queue.

4.3 Graph Node Function Table

There are five events a graph node must deal with:

`evaluate`:	The node has to be evaluated with a particular evaluator.
`evaluateWHNF`:	The node has to be evaluated with a particular evaluator. It has to notify the father node if weak head normal form is reached.
`evaluateNF`:	The node has to be evaluated to normal form. It has to notify the father node if the evaluation has been completed.
`activate`:	The node has just been removed from the RNQ and has to be reduced.
`childEvaluated`:	A subnode (child) has been evaluated to (weak head) normal form.

A function table consists of a couple of function pointers. For each of the six events an entry exists. These entries point to functions which handle events. Only these five events are possible for all types of graph nodes. So it is not necessary to look at the type of a node before “jumping” to the appropriate message handler. This has already been used in many implementations of functional programming languages, for instance the spineless tagless G-machine [14], the Three Instructions Machine [16] and the jump-machine [3].

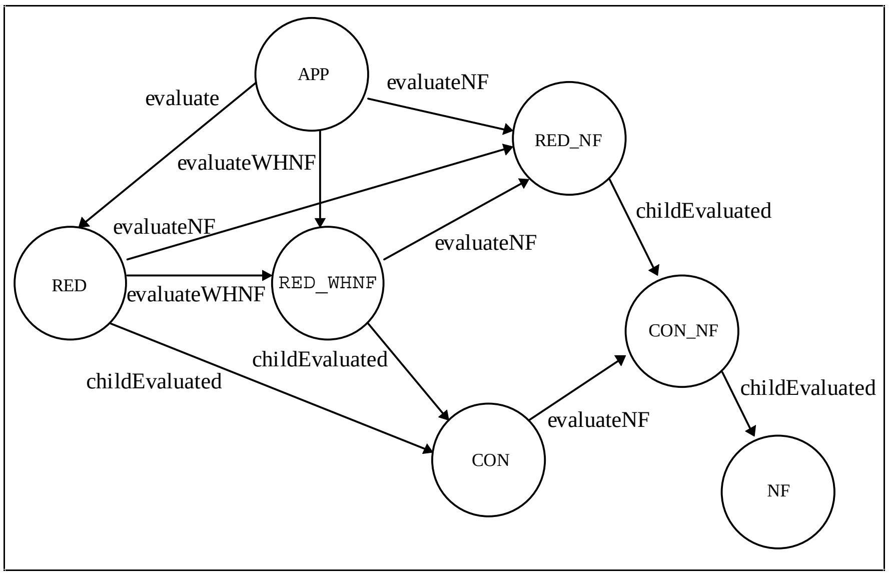

Figure 4: States for Ordinary Graph Nodes

A graph node can take seven different states. A node reacts in each of these states very differently to a particular event. Therefore one function table corresponds to each state. For each function four function tables and for each constructor three function tables exist. State transitions are performed by replacing the function table pointer. All events, their meaning and the actions performed as response to each event are listed below:

State `APP`:

The node represents a function application. It is not yet clear if it will ever be evaluated. No evaluator is assigned to the node. The lists of fathers waiting for (weak head) normal form are empty. The counter field is undefined.

- Event: `evaluate`; arguments: `evaluator`
  - assign `evaluator` to graph node
  - call the appropriate evaluation transformer
  - change state to `RED_WHNF`
  - insert the node into the RNQ or call `activate`[9]
- Event: `evaluateWHNF`; arguments: `evaluator, father`
  - assign `evaluator` to graph node
  - call the appropriate evaluation transformer
  - insert `father` into the list of fathers waiting for weak head normal form
  - change state to `RED_WHNF`
  - insert the node into the RNQ or call `activate`

[9] This depends on whether there are other possible reduction paths to continue. If there is only this one, the node is activated immediately without using the reducible nodes queue.

- Event: `evaluateNF`; arguments: `evaluator, father`
  - assign `evaluator` to graph node
  - call the appropriate evaluation transformer
  - insert `father` into the list of fathers waiting for normal form
  - change state to `RED_NF`
  - insert the node into the RNQ or call `activate`

The events `activate` and `childEvaluated` do not occur.

State `RED`:

The node represents a function application which must be evaluated at least to weak head normal form. It already has an evaluator. The lists of fathers waiting for (weak head) normal form are empty. There are two different cases for a node in state `RED`. Prior to the call of `activate`, the node is contained in the reducible nodes queue of its processing element and the counter field is undefined. Afterwards it is not contained in the RNQ and the counter field holds the number of arguments not in weak head normal form but necessary for the reduction of the function application.

- Event: `evaluate`; arguments: `evaluator`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - call the appropriate evaluation transformer
- Event: `evaluateWHNF`; arguments: `evaluator, father`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - call the appropriate evaluation transformer
  - insert `father` into the list of fathers waiting for weak head normal form
  - change state to `RED_WHNF`
- Event: `evaluateNF`; arguments: `evaluator, father`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - call the appropriate evaluation transformer
  - insert `father` into the list of fathers waiting for normal form
  - change state to `RED_NF`
- Event: `activate`
  - store number of strict arguments in the counter field
  - initiate the evaluation of strict arguments by calling their `evaluateWHNF`-handlers
- Event: `childEvaluated`
  - decrease the counter
  - if the counter reaches zero reduce the graph node (i.e. update)

State `RED_WHNF`:

The node represents a function application which must be evaluated at least to weak head normal form. It already has an evaluator. The list of fathers waiting for weak head normal form contains at least one reference. The list of fathers waiting for normal form is empty. There are two different cases for a node in state `RED_WHNF`. Prior to the call of `activate`, the node is contained in the reducible nodes queue of its processing element and the counter field is undefined. Afterwards it is not contained in the RNQ and the counter field holds the number of arguments not in weak head normal form but necessary for the reduction of the function application.

- Event: `evaluate`; arguments: `evaluator`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - call the appropriate evaluation transformer
- Event: `evaluateWHNF`; arguments: `evaluator, father`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - call the appropriate evaluation transformer
  - insert `father` into the list of fathers waiting for weak head normal form
- Event: `evaluateNF`; arguments: `evaluator, father`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - call the appropriate evaluation transformer
  - insert `father` into the list of fathers waiting for normal form
  - change state to `RED_NF`
- Event: `activate`
  - store number of strict arguments in the counter field
  - initiate the evaluation of strict arguments by calling its `evaluateWHNF`-handler
- Event: `childEvaluated`
  - decrease the counter
  - if the counter reaches zero reduce the graph node (i.e. update)
  - if the node is updated to weak head normal form, call the `childEvalu-ated`-handler for all fathers in the appropriate list and change state to `CON_NF`.

State `RED_NF`:

The node represents a function application which must be evaluated to normal form. This state is very similar to RED_WHNF, but the list of fathers waiting for weak head normal form is possibly empty. The list of fathers waiting for the normal form contains at least one reference. The evaluator assigned to the node is ξ_{NF}. Therefore any new evaluator would not change it.

- Event: `evaluate`; arguments: `evaluator`
  - do nothing
- Event: `evaluateWHNF`; arguments: `evaluator, father`
  - insert `father` into the list of fathers waiting for weak head normal form
- Event: `evaluateNF`; arguments: `evaluator, father`
  - insert `father` into the list of fathers waiting for normal form
- Event: `activate`
  - store number of strict arguments in the graph node
  - initiate the evaluation of strict arguments by sending `evaluateWHNF`
- Event: `childEvaluated`
  - decrease the counter
  - if the counter is zero, reduce the graph node
  - update the graph node

  - if the node is updated to weak head normal form, call the `childEvaluated`-handler for all fathers in the appropriate list and change state to `CON_NF`.

State `CON`:

The node represents a constructor expression. The lists of fathers waiting for (weak head) normal form are empty. The counter field is undefined.

- Event: `evaluate`; arguments: `evaluator`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - send evaluator parameters to the subexpressions
- Event: `evaluateWHNF`; arguments: `evaluator, father`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - send evaluator parameters to the subexpressions
  - notify `father` by calling its `childEvaluated`-handler
- Event: `evaluateNF`; arguments: `evaluator, father`
  - combine `evaluator` with the evaluator already assigned to the graph node
  - send evaluator parameters to the subexpressions
  - insert `father` into the list of fathers waiting for normal form
  - change state to `CON_NF`
  - call the `activate`-handler

The events `activate` and `childEvaluated` do not occur.

State `CON_NF`:

The node represents a constructor expression and must be evaluated to normal form. The list of fathers waiting for weak head normal form is empty. The list of fathers waiting for normal form contains at least one reference. There are two different cases for a node in state `CON_NF`. Prior to the call of `activate`, the node is contained in the reducible nodes queue of its processing element and the counter field is undefined. Afterwards it is not contained in the RNQ and the counter field holds the number of arguments not yet in normal form.

- Event: `evaluate`; arguments: `evaluator`
  - do nothing
- Event: `evaluateWHNF`; arguments: `evaluator, father`
  - notify `father` by sending `childEvaluated`
- Event: `evaluateNF`; arguments: `evaluator, father`
  - insert `father` into the list of fathers waiting for normal form
- Event: `activate`
  - store number of subexpression in the graph node
  - call the `evaluateNF`-handlers for all subexpressions
- Event: `childEvaluated`
  - decrease the counter
  - if the counter is zero, change state to `NF` and call the `childEvaluated`-handlers of all fathers contained in the appropriate list.

State `NF`:

The node represents a constructor expression and has reached normal form. The lists of fathers waiting for (weak head) normal form are empty. The counter field is undefined.

- Event: `evaluate`; arguments: `evaluator`
  - do nothing
- Event: `evaluateWHNF`; arguments: `evaluator, father`
  - notify `father` by calling its `childEvaluated`-handler
- Event: `evaluateNF`; arguments: `evaluator, father`
  - notify `father` by calling its `childEvaluated`-handler

The events `activate` and `childEvaluated` do not occur.

State `IND_L`:

The node is an indirection to a graph node on the same processor. All events are propagated to the node the indirection points to. The event `activate` does not occur.

State `IND_R`:

The node is an indirection to a graph node on another processor. All events are propagated to the node the indirection points to. The event `activate` does not occur.

4.4 Communication Instructions

There are several situations in the parallel abstract machine where communication between processing elements is necessary.

- One of the four events `evaluate`, `evaluateWHNF`, `evaluateNF`, `childEvaluated` is propagated to another processing element. `activate` events are never sent to other processing elements.
- A processing element has run out of work and queries another element for work.
- As response to such a request a reducible graph node is transferred to another processing element.

The communication between processing elements on machines with message based communication is done by active messages [5, 3]. This means each message contains a pointer to a message handler function which does the necessary things to respond to the message. The processing element which receives a message only has to call this function. It is not necessary to look after a message id. The SPMD model where all processors run the same program makes this technique possible.

On virtual shared memory architectures like Cray T3D the possibility of accessing the memory of other processors directly without interrupting the other processor promises much more efficient communication. This advantage can be used for work requests and node transfers in a rather straightforward way. The processing element which runs out of work reads the reducible nodes queue of the other element. If there are enough nodes the processing element reads a graph node, creates a copy of it in its own memory and overwrites the source with an indirection. All other messages depend on the graph node state. It is not yet completely clear how to use direct memory access in these cases. The first idea is to introduce more sophisticated entries in the reducible nodes queue. If each entry contains an action, the receiving processing element could call the appropriate event handler as soon as it accesses the RNQ the next time.

5 Conclusion and Further Work

This report outlines a technique to implement lazy functional languages on distributed memory parallel machines using the evaluation transformer model of evaluation. It introduces for the first time the possibility of handling evaluators even for structured types. But a lot of work still has to be done.

- The theoretical foundation has to be worked out. In particular the construction of an evaluator basis should be investigated.
- Each graph node maintains lists of father nodes which must be notified if a (weak head) normal form is reached. Another possibility to carry out this notification is to let the father check the reduction state of its subnodes regularly. Some work [7] promises a better performance for this polling technique.
- On modern microprocessors pipelining and caching try to minimise memory accesses during program execution. The extensive use of function tables may lead to poor performance on these processors since jumps may cause pipeline breaks. It should be investigated whether some kind of tagging nodes instead of general taglessness performs better.
- The implementation proposed in this report may not be the most efficient for all possible functional programs. It has to be investigated which kinds of programs will be supported best.
- Some implementation experiments have already been done. A simple prototype has been implemented, but the main work is still to be carried out.

6 References

[1] Geoffrey L. Burn; *Implementing the evaluation transformer model of reduction on parallel machines*; In: Journal of Functional Programming, 1(3): 329-366 1991

[2] Geoffrey L. Burn; *Lazy functional languages: Abstract interpretation and compilation*; Research monographs in parallel and distributed computing MIT Press 1991

[3] Manuel M.T. Chakravarty; *A Self-Scheduling, Non-Blocking, Parallel Abstract Machine for Lazy Functional Languages*; In: Proceedings of the 6th International Workshop on the Implementation of Functional Languages, 1994

[4] Charles L. A. Clarke, David V. Mason; *Compacting Garbage Collection can be Fast and Simple*; In: Software-Practice and Experience, 1995

[5] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser; *Active Messages: a mechanism for integrated communication and computation*; In: Proceedings of the 19th international Symposium on Computer Architecture, ACM Press 1992

[6] Sigbjørn Finne, Geoffrey L. Burn; *Assessing the Evaluation Transformer Model of Reduction on the Spineless G-machine*; In: Functional Programming & Computer Architecture, ACM, June 93

[7] John H.G. van Groningen; *Some implementation aspects of Concurrent Clean on distributed memory architectures*; In: Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages, 1992

[8] Denis B. Howe, Geoffrey L. Burn; *Experiments with strict STG code*; In: Proc. of the 4th international workshop on the parallel implementation of functional languages, 1992

[9] Richard E. Jones, Rafael D.Lins; *Cyclic Weighted Reference Counting without Delay*; Technical report 28-92, Computing Laboratory, University of Kent, Nov 92

[10] Hugh Kingdon, David Lester, Geoffrey L. Burn; *The HDG-Machine: A Highly Distributed Graph Reducer for a Transputer Network*; In: The Computer Journal 34(4): 290-301 Aug. 1991

[11] Herbert Kuchen; *Parallele Implementierung einer funktionalen Programmiersprache auf einem OCCAM-Transputer-System unter besonderer Berücksichtigung applikativer Datenstrukturen*; Dissertation RWTH Aachen 1989

[12] Sava Mintchev; *Using Strictness Information in the STG-machine*; In: Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Programming Languagues, 1992

[13] Eric Mohr, David A. Kranz, Robert H. Halstead Jr.; *Lazy task creation: A technique for increasing the granularity of parallel programs*; IEEE Transactions on Parallel and Distributed Systems, 1990

[14] Simon L. Peyton Jones; *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5*; In: Journal of Functional Programming , July 92

[15] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, Phil Wadler; *The Glasgow Haskell compiler: a technical overview*; In: Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference, 1993

[16] Simon L. Peyton Jones, David Lester; *Implementing Functional Languages*; Prentice Hall 1992

[17] Julian Richard Seward; *Towards a Strictness Analyser for haskell: Putting Theory into Practice*; Technical Report University of Manchester (UMCS-92-2-2)

[18] Julian Richard Seward; *Abstract Interpretation of Functional Languages: A Quantitative Assessment*; PhD. Thesis University of Manchester August 1995

[19] Benjamin Zorn; *The Effect of Garbage Collection on Cache Performance*; University of Colorado at Boulder (CU-CS-528-91)