PASTEL

A Parallel Stack-based Implementation of Eager Functional Programs with Lazy Data Structures

Guido Hogen and Rita Loogen[*]
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany

Extended Abstract

1 Introduction

In purely sequential implementations of functional programming languages a *runtime stack* is used for the control of the evaluation. Function calls lead to the allocation of control information (function blocks) on top of the stack. When the function call terminates, this information is removed and control is given back to the calling environment. In addition, a graph or heap structure is used for the representation of data structures, functional values (closures) and expressions not evaluated yet (suspensions), see e. g. [Johnsson 84], [Peyton Jones 87], [Koopman et al. 90], [Peyton Jones 91].

For parallel implementations, the efficient realisation of the runtime stack is one of the most important problems. The use of a global contiguous array of function blocks is no longer advisable, because each parallel task or process needs its own stack structure. Furthermore a suspension mechanism must be realized to keep the state of tasks which are waiting for the result of some parallel subtask. This leads to the idea of allocating runtime structures in a graph or heap instead of on a stack [Goldberg, Hudak 86]. A main advantage of the graph organisation of runtime structures is the *decentralisation* which is especially suited for distributed systems, as the bottleneck of a global central stack is avoided. Furthermore context switches can simply be realized by redirecting the pointer to the currently active task. Heap *allocation* can be done as efficiently as stack allocation by the incrementation of a heap pointer [Appel 87]. A disadvantage is however the increased organisational overhead for the *deallocation* (garbage collection) of runtime information.

In the parallel abstract machine (PAM), which underlies the distributed implementation of [Loogen et al. 89], each (non tail recursive) function call leads to the generation of a new function block (called task node) within the runtime graph. This function block contains all the information and substructures that are necessary for the evaluation of the corresponding function call. A parallel evaluation can simply be achieved by the migration of such a function block (task node) to another processor element.

Thus, the runtime stack of each process is realized as a linked list of function blocks. Spawning of parallel processes yields a tree structure of such linked lists, which represents a so-called *cactus stack*. An analogous organisation of runtime structures has been adopted in the $\langle \nu, G \rangle$-machine [Augustsson, Johnsson 89] and the HDG-machine [Kingdon et al. 91]. The dynamic extension of function blocks in case of tail recursive function calls can be avoided by the use of the stacklessness analysis of [Lester 89], which allows to create task nodes large enough to evaluate all tail recursive calls that may occur in a reduction sequence.

[*]email: {ghogen, rita}@zeus.informatik.rwth-aachen.de

The runtime experiments of the PAM show acceptable speed-up results due to this simple organisation of parallelism [Loogen et al.89]. But the runtimes of purely sequential programs (on a single processor element) are not satisfying compared with purely sequential implementations. The main reason for this loss of performance is the "linked list"-realization of the stack which does not support the sequential evaluations within processes in an optimal way.

[Goldberg, Hudak 86] already pointed out that the graph allocation of function blocks should be avoided for purely sequential functions which do not spawn parallel subtasks and may not be suspended due to parallel subtasks. They proposed a hybrid system which assumes an analysis and decomposition of the program into sequential and non-sequential functions. For sequential functions an ordinary stack-based evaluation is arranged. For non-sequential function calls the more expensive allocation of runtime information in the heap must be chosen to support parallel evaluations.

In this paper we present an alternative realization of the runtime structures which tends to support parallel and sequential evaluations equally well. Instead of representing the cactus stack as a linked tree of function blocks, we allocate function blocks, within each processor element of the parallel system, on a single *meshed stack*, which is organized as a contiguous array, but shared by several processes. The graph is only used for the representation of parallel processes, structured data, closures and suspensions. Evaluation is controlled by the stack. During the sequential evaluation of processes the meshed stack is used as in conventional implementations. Function blocks are allocated on top of it, when function calls are executed, and removed, when function calls terminate. The interesting questions are of course, what happens when a parallel process is generated and when a function evaluation must be suspended due to some missing result of a parallel subprocess.

When a parallel process is spawned, a graph node, which serves as a placeholder for the result of the process, is generated and a process message containing the description of the process and the address of the result node is created. This message is first of all stored in a queue. It may be sent to other processor elements when these ask for work. The placeholder for the result of the parallel process can be seen as a graph representation of the migrated evaluation. For this reason we call it an indirection node.

The suspension of evaluations is of course more involved, as it has to enable the context switch to another process. When a function call is suspended, the current value of the stack pointer (which indicates the active function block) is saved in the graph representation of the parallel process for which the current evaluation has to wait. The function blocks are kept on the stack until the result of the parallel process is available and the evaluation can continue. In the meantime, the processor element can proceed with the evaluation of another parallel process, whose function blocks will simply be allocated on top of the function blocks of the suspended evaluation. If this evaluation will eventually suspend and the previously suspended evaluation is reactivated, newly created function blocks of this evaluation will simply be allocated on top of the function blocks of the previous evaluation. Thus we will get a mixture of function blocks belonging to different parallel processes on the stack, which is the reason for calling it a meshed stack.

But, what happens when function calls terminate and the corresponding blocks can be deallocated. The deallocation of blocks on top of the stack is of course no problem. In the current version of our implementation, inner function blocks which can be removed are only marked as garbage. They are collected, i.e. removed, as soon as there is no valid, i.e. active or suspended function block on top of them.

The meshed stack allows an efficient parallel implementation of functional languages, which supports sequential evaluations as in a purely sequential environment. Note that the graph structure is only used for the organisation of parallelism.

Another reason for the poor performance of functional languages is the overhead which is caused by the lazy evaluation strategy. Lazy evaluation is of course one of the main characteristic features of modern functional languages [Hudak 89]. Its importance is basically due to the possibility to work with potentially infinite data structures. But note that infinite data structures can also be supported by the use of an eager strategy which evaluates however expressions only up to weak head normal form (WHNF). The advantage of such an eager strategy to WHNF is that suspensions will only be necessary for the representation of components of data structures and not, as with a purely lazy strategy, for all function arguments. The detection of implicit parallelism is essentially simplified. It can be reduced to the detection whether functions are going to evaluate structured arguments even to normal form [Sekar et al. 91].

The paper is organized as follows. In the next section, we introduce a simple source language which contains already parallelism information. Section 3 presents some details of the parallel abstract machine. The compilation of functional programs into abstract machine code is explained in section 4. After some comments on the status of the implementation, we give some hints at future work. Note that this paper is preliminary, as the status of the implementation does not allow us to draw final conclusions up to now.

2 The Source Language

For simplicity, we consider in this paper simple first order, monomorphically typed supercombinator systems as the functional source language. We assume that the programs are already annotated with parallelization informations in form of special *letpar*-constructs. These annotations could have been introduced by transforming a high level functional language unsing a parallelizing compiler [Hogen et al. 92]. Note that, in contrast to a lazy strategy, an eager approach makes a parallelizing compiler much easier to develop.

Parallel execution only happens at combinator level. Instead of going into detail, we give two small examples of programs, showing the main concepts: The first function generates the (infinite) list of all fibonacci numbers, whereas the second one computes the n-th fibonacci number in the classical way.

```
TYPE list = NIL | CONS OF int * list;

FUN fiblist: int * int -> list;
    x y -> CONS(x, fiblist y (x+y));

FUN fib: int -> int;
    x -> IF x<3 THEN 1
                ELSE LETPAR
                         y = fib (x-1);
                         z = fib (x-2);
                     IN y+z;
```

3 Structure of the Parallel Machine

3.1 General structure

The parallel implementation is based on a parallel abstract machine. It consists of a finite number of identical processor elements, each with a local storage. An interconnection network enables the processor elements to communicate by exchanging messages. Each processor element consists of two independent processing units, which communicate via exchanging messages in a local shared memory.

- The *communication unit* handles the organization of the parallel aspects of the reduction process.
- The *reduction unit* executes the parallelized functional program by code-directed stack-based reduction.

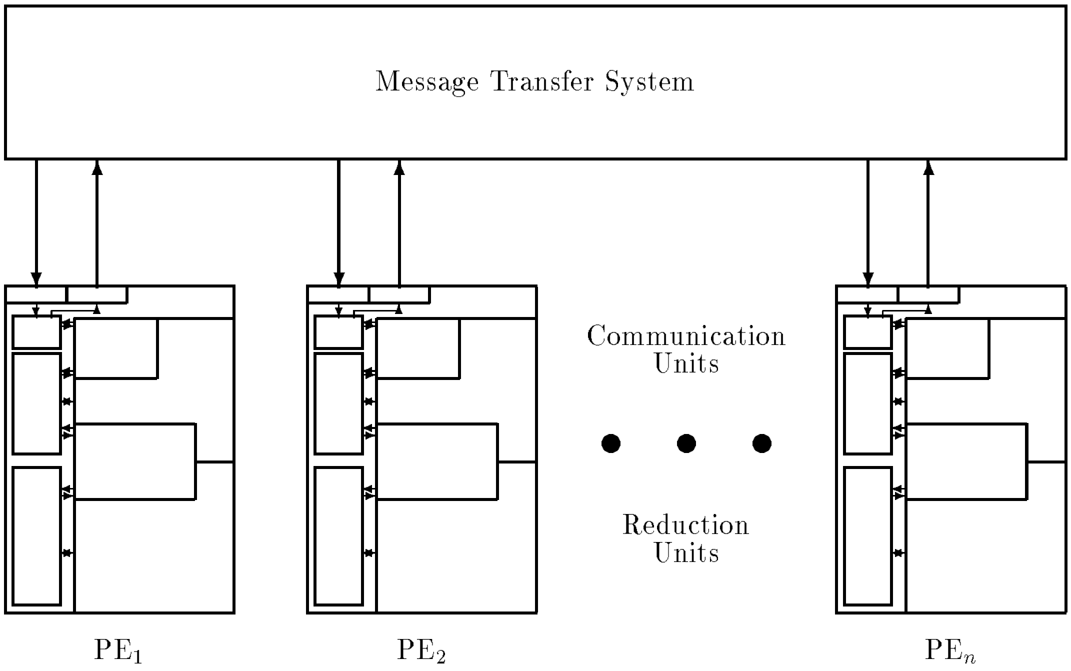

Figure 1: Global Structure of the machine

The main item of the communication unit is the distribution of parallel tasks and the passing of messages from and to the reduction unit.
The main structures of the reduction unit are

- the *function stack* component to manage the function blocks of the combinator calls,
- the *data stack* component to hold the local data stacks for each combinator call,

For clarity of presentation we distinguish two stacks, which are both organized as meshed stacks. For each function call there is exactly one function block and a corresponding local data stack. The maximal block sizes are estimated during compile time using a stacklessness analysis similar to [Lester 89].

- the *heap* component to represent (structured) data (terminal nodes) and placeholders for results of parallel processes (indirection nodes),
- a *ready queue* which holds the tasks which have been suspended, but now are ready to be re-activated, and
- the *program store* for the machine code to be executed.

The execution of a task is controlled by the *program counter* which points to the next program instruction to be executed, the *active task pointer* which points to the active function block on the function stack and the *active datastack pointer* which points to the accompanying local data stack.

3.2 Instruction set

The instruction set of the machine consists of the following commands:

LIT v loads basic value v onto the active data stack.

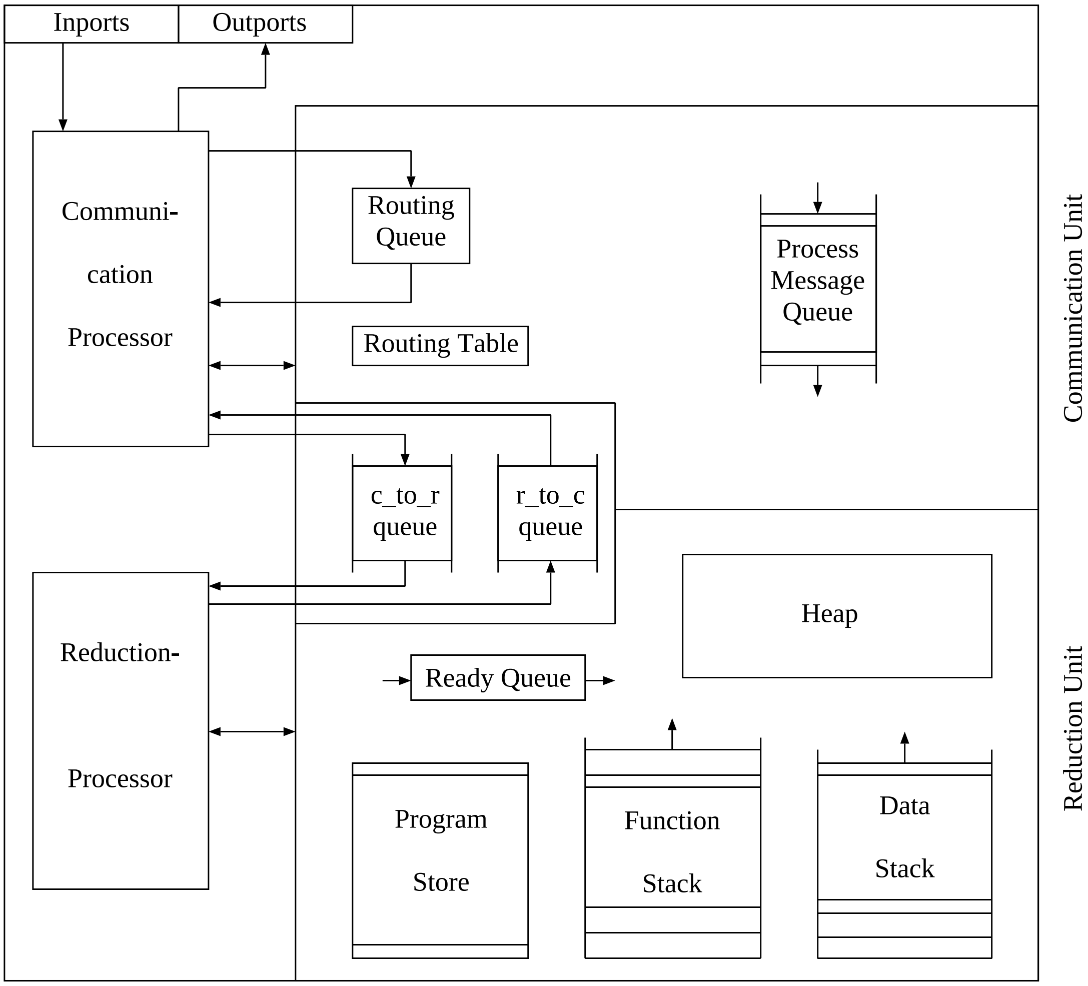

Figure 2: Global Structure of a Processor Unit

EXEC f executes the basic operation f on the active data stack.

UNBOX unboxes the boxed representation of a basic value. This case arises when the task yielding this result was executed in parallel (see below).

LOAD i loads the i-th argument from the active function block on top of the active data stack. STORE i behaves analogously.

JMP l**, JPF** l are (conditional) jumps to program address l.

CNODE $(c,k,i_1,\ldots,i_k)$ creates on the heap a representation of a constructor application of the k-ary constructor c. Since we want to handle infinite data structures, the components should not be evaluated. So for each component we have to compile a separate code-piece for evaluation. The starting addresses for these code-pieces are stored in the information i_j. For the delayed evaluation of a component we also have to remember the actual environment, i.e. the values of the variables in the actual function block, and the processor number, because all references in the environment are local to that processor element. So we create an extra *environment node* with a copy of the values of the variables and add the corresponding pointer and the processor number to the constructor application node.

At this point some optimisations can be added. If a component of an application is a basic constant, we don't have to create a separate code-piece to evaluate that constant.

Instead we can add this information to the constructor application node, e.g. i_j = <*VAL, v*>. When a component consists only of a variable identifier of basic type which wasn't introduced by a *letpar*, we can look up the value in the appropriate environment. For example, if we refer to variable x_m, we can give the information i_j = <*LOOKUP, m*>.

TEST c tests, if the pointer on top of the active data stack refers to a constructor node labelled with c.

PROJ i expects on top of the data stack a pointer to a constructor application and returns the evaluated i-th component. In order to do this, one has to check the information entry of the constructor application. A couple of different situations can arise: If one finds a VAL-entry, the value can be loaded directly. If there is a LOOKUP-entry and the local processor number matches the processor number entry of the constructor application, the value can be loaded directly from the environment. Otherwise a memory request has to be sent to the referred processor element. If one finds a CODE-entry the corresponding code-piece has to be executed. If the constructor application was created locally, the environment is also available, so the function call to the code-piece can be executed immediately. Otherwise a computation request has to be sent to the processor element that built the application, and the active task has to be suspended until the result is available.

CALL (ca, n, k, $size$) implements a local combinator application call. It creates a new function block on top of the function stack, moves n entries from the data stack to the function stack, reserves k cells for local variables, creates a new data stack of size $size$, writes the value of the program counter as return address into the new function block and activates the new function block (program address ca).

TASK (ca, n, k, $size$) creates a combinator call that can be executed in parallel. The actual parameters and status informations are sent to the communication unit. In the heap component, a placeholder for the result is created. The computation of the actual task proceeds, until there is a demand for the result.

WAIT i checks if argument i is evaluated. If this is not the case, the active task is suspended until the result is available. Therefore the stack addresses of the task are added to a list of waiting tasks which is attached to the placeholder. When the result is delivered, the list is moved to the ready queue.

RETURN returns the address of the result (a datastructure or a higher order function) to the calling task and marks the actual function block and data stack as garbage. Due to parallel execution, the calling task can be local (so one can put the pointer onto the calling data stack and continue computing the calling task) or global (here the result is sent to the placeholder on the other processor element).

RETVAL is similar to RETURN, only the value is returned directly in the local case. In the global case the value has to be boxed before sending and unboxed on demand.

3.3 Message handling

If there is no actual task, the reduction unit looks up the ready queue for further tasks which are ready for activation. If the queue is also empty, a work request is sent to the communication unit. If necessary, the communication unit will propagate the work request to some neighbour processor element. What is left is the explanation of the behaviour of the reduction unit concerning incoming messages:

- A *task message* represents an external ready-to-run task. A new function block is created and the stack pointer is added to the ready queue. One cannot start the task directly (although demanded by a work request) because in the time between the work request and the task message, a result could have been delivered which triggers the activation of dormant tasks. Also, due to different scheduling strategies, on demand several tasks could be delivered.

- A *memory request message* forces the reduction unit to look up a basic value in the desired environment and to send the value in form of an answer message back to the inquiring processor.
- A *computation message* starts a computation of a desired value on another (known) processor element (see comments on the PROJ-instruction).
- Finally, *answer messages* overwrite the placeholder nodes and activate the waiting tasks. If the result needs more space than the placeholder offers, a local indirection node has to be used.

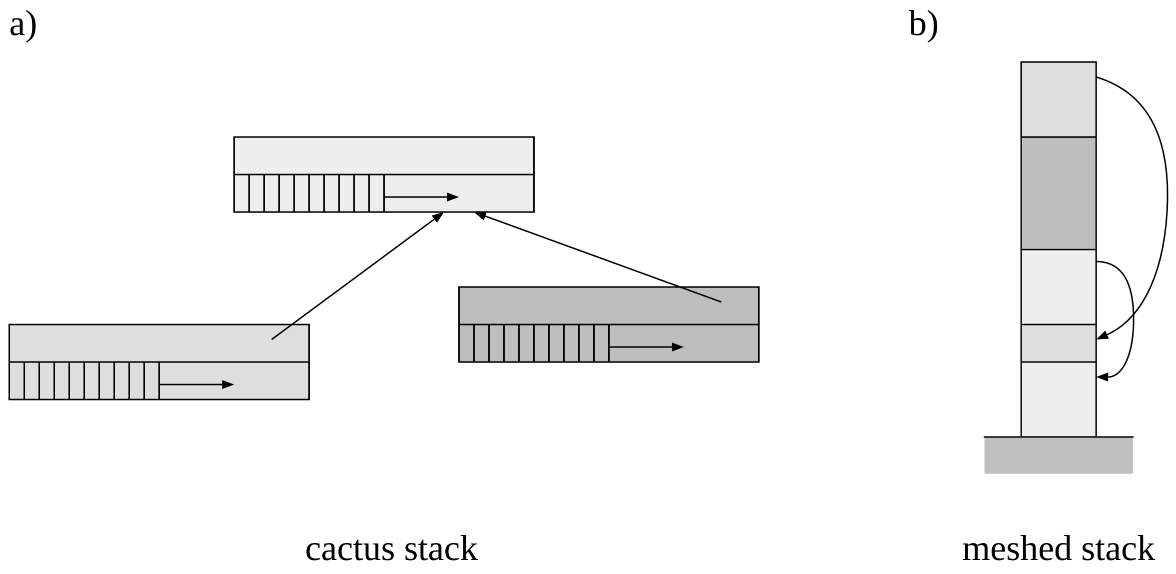

Figure 3: Comparison of Cactus Stack and Meshed Stack

3.4 Organisation of the function stack

Subsequent combinator calls cause the placement of function blocks on the function stack. These blocks are linked. When a function call is finished, the calling block is activated and the actual block is marked as garbage.

When evaluating in parallel, it could be the case that the actual task has to be suspended due to a missing result. Inbetween another new task can be activated causing a new function block on top of the stack. This, and subsequent function blocks caused by this new task, are independent. So the stack can be divided into regions of connected function blocks. If the new task is also suspended and the old task is reactivated, possible combinator calls yield new blocks on top of the stack. So we get a mesh of independent blocks. This structure represents a special realization of the cactus stack. In Figure 3 a cactus stack and its representation in a meshed stack is shown. In the cactus stack each node represents one parallel task including sequential function calls. So each node represents a number of function blocks. If we look at the meshed stack, we find these function blocks intermixed with blocks of other parallel tasks[1].

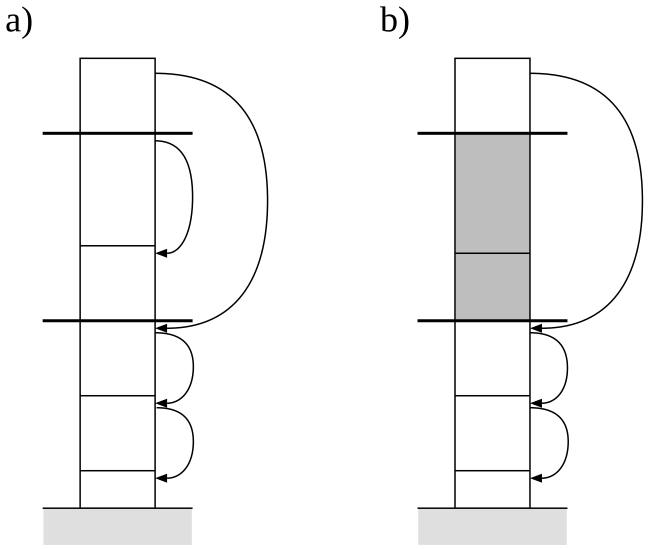

Figure 4: Function Stack Structure

The meshed structure of the function stack has two consequences:

- The active function block need not be on the top of the stack.
- One can only collect garbage from the top-of-stack pointer until one finds a non-garbage block. Inbetween the stack, there may be still some "holes" left (see Figure 4). This special structure can be subject to special garbage collection or stack allocation techniques, which are however out of the scope of this paper.

4 Program compilation

The compilation of the functional source language to machine code is quite standard, so we will restrict the description to the interesting cases, namely the compilation schemes of a constructor application, a combinator application, the scheme for the *letpar*-construct and the scheme for variables.

4.1 The scheme for constructor applications

$$
\begin{array}{lrl}
\multicolumn{3}{l}{\textsc{Trans}\ [\![c(e_1,\ldots,e_k)]\!]} \\
:= & & \text{JMP}\ l_{k+1}; \\
 & l_1: & \textsc{Init}\ [\![e_1]\!] \\
 & & \vdots \\
 & l_k: & \textsc{Init}\ [\![e_k]\!] \\
 & l_{k+1}: & \text{CNODE}\ (c, k, \textsc{Info}[\![e_1]\!](l_1) : \ldots : \textsc{Info}[\![e_k]\!](l_k));
\end{array}
$$

if c is a k-ary constructor

where

[1]Although executed on the same processor element, the three parallel tasks are not linked. After finishing a task, the result is sent to the home processor, which is coincidentally the same processor.

$$
\begin{array}{lll}
\textsc{Init}\ [\![v]\!] & := \varepsilon & \text{if } v \text{ is a constant} \\
\textsc{Init}\ [\![y]\!] & := \varepsilon & \text{if } y \text{ is a variable of basic type, not introduced by } \textbf{letpar} \\
\textsc{Init}\ [\![e]\!] & := \textsc{Trans}[\![e]\!] & \\
 & \quad \text{RETURN;} & \text{else}
\end{array}
$$

and

$$
\begin{array}{lll}
\textsc{Info}\ [\![v]\!](j) := \langle \text{VAL}, v \rangle & & \text{if } v \text{ is a constant} \\
\textsc{Info}\ [\![y]\!](j) := \langle \text{LOOKUP}, i \rangle & & \text{if } y \text{ is a variable of basic type, not introduced by } \textbf{letpar} \\
\textsc{Info}\ [\![e]\!](j) := \langle \text{CODE}, j, \mathit{dataspace}(e) \rangle & & \text{else}
\end{array}
$$

The representation for the constructor application is allocated immediately. For the optimized cases of the delayed component evaluation, all necessary information is stored in the application node, otherwise code sequences for evaluating have to be added. In the constructor node the program address and the maximum data stack size – which can be estimated at compile time – is stored. Note that the entries for the environment and the processor number are only available at runtime.

4.2 The scheme for combinator applications

$$
\begin{array}{ll}
\textsc{Trans}\ [\![F(e_1, \ldots, e_k)]\!] & \\
\quad := \textsc{Trans}\ [\![e_1]\!] & \\
\qquad \vdots & \\
\qquad \textsc{Trans}\ [\![e_k]\!] & \\
\qquad \text{CALL } (\mathit{ca}(F), k, \mathit{locspace}(F), \mathit{dataspace}(F)); & \text{if } \mathit{rank}(F) = k
\end{array}
$$

First, the arguments are evaluated, then an immediate jump to the evaluation sequence for the combinator application follows. The functions *ca*, *locspace* and *dataspace* yield the code address, the maximum storage demand for local variables and the maximum data stack size, respectively.

4.3 The scheme for the letpar-construct

$$
\begin{array}{l}
\textsc{Trans}\ [\![\textbf{letpar}\ y_1 = \tilde{F}_1(e_{11}, \ldots, e_{1l_1}) \\
\qquad\quad \textbf{and} \quad \ldots \\
\qquad\quad \textbf{and} \quad y_p = \tilde{F}_p(e_{p1}, \ldots, e_{pl_p})\ \textbf{in}\ e]\!] \\
:= \mathit{pcode}(\tilde{F}_1(e_{11}, \ldots, e_{1l_1})) \\
\quad \text{STO } m+1; \\
\qquad \vdots \\
\quad \mathit{pcode}(\tilde{F}_p(e_{p1}, \ldots, e_{pl_p})) \\
\quad \text{STO } m+p; \\
\quad \textsc{Trans}\ [\![e]\!]
\end{array}
$$

$$
\text{where } \mathit{pcode}(\tilde{F}_j(e_{j1}, \ldots, e_{jl_j})) := \left\{
\begin{array}{l}
\textsc{Trans}[\![e_{j1}]\!] \\
\qquad \vdots \\
\textsc{Trans}[\![e_{jl_j}]\!] \\
\text{TASK } (\mathit{ca}(\tilde{F}_j), l_j, \mathit{locspace}(\tilde{F}_j), \mathit{dataspace}(\tilde{F}_j));
\end{array}
\right.
$$

When reaching a **letpar**-construct, parallel tasks for all combinator calls on the right hand side are created (of course, after evaluating the arguments). Placeholder for the results are stored to the newly introduced local variables. The number of used variables, m, is passed as a parameter of the TRANS-scheme (which is omitted here for reasons of simplicity).

4.4 The scheme for variables

$$
\textsc{Trans}\,[\![y]\!] := \begin{cases} \text{WAIT } i; & \\ \text{LOAD } i; & \text{if } y \text{ is a variable of non-basic type, introduced using } \textbf{letpar} \\ & \\ \text{WAIT } i; & \\ \text{LOAD } i; & \\ \text{UNBOX}; & \text{if } y \text{ is a variable of basic type, introduced using } \textbf{letpar} \\ & \\ \text{LOAD } i; & \text{else} \end{cases}
$$

We have to distinguish between variables introduced by the **letpar**-construct and others (formal parameters and local variables introduced by **let**). The latter are always available evaluated and can be accessed directly. The result of the first may not be available, so one has to wait and suspend the actual evaluation. After that, the value can be loaded. Due to the parallel execution, basic values are stored boxed, so they have to be unboxed before continuing.

5 State of the implementation

The implementation of the machine on a transputer system under Helios C is in progress. It is too early to present detailed runtime measurements, but first results on simple programs are promising.

6 Future Work

We presented a basic framework of the machine which gives some opportunities for future work. Garbage collection / memory allocation is an interesting field, due to the special structure of the function and data stack component. All garbage blocks can be easily detected by following the dynamic links of the function blocks. It is also planned to investigate more sophisticated techniques for handling data structures. A demand analysis (e.g. via abstract interpretations) could yield annotations for constructors indicating if a full evaluation is demanded. So only those data structures are evaluated lazily which are used lazily.

Furthermore we want to compare this implementation to a graph-based version and to an implementation on a shared-memory multi-processor system.

References

[Appel 87] A.W. Appel: *Garbage Collection can be faster than stack allocation*, Information Processing Letters, 21(4):275–279, 1987.

[Augustsson, Johnsson 89] L.Augustsson, Th.Johnsson: *Parallel Graph Reduction with the $\langle\nu, G\rangle$-Machine*, Conference on Functional Programming Languages and Computer Architecture (FPCA), Addison-Wesley 1989.

[Goldberg, Hudak 86] B.Goldberg, P.Hudak: *ALFALFA — Distributed Graph Reduction on a Hypercube Multiprocessor*, Workshop on Graph Reduction, Lecture Notes in Computer Science 279, Springer Verlag 1986.

[Hogen et al. 92] G.Hogen, A.Kindler, R.Loogen: *Automatic Parallelization of Lazy Functional Programs*, 4th European Symposium on Programming (ESOP '92), LNCS 582, Springer Verlag 1992.

[Hudak 89] . Hudak: *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys, Vol. 21, No. 2, September 1989, 359–411.

[Johnsson 84] Th.Johnsson: *Efficient Compilation of Lazy Evaluation*, SIGPLAN Notices Vol. 19, No. 6, June 1984.

[Kingdon et al. 91] H.Kingdon, D.Lester, G.Burn: *An Executable Specification of the HDG-Machine*, The Computer Journal, Vol. 34, No. 4, 1991.

[Koopman et al. 90] P. Koopman, M. van Eekelen, E. Nöcker, J. Smeetsers, R. Plasmeijer: *The ABC-machine: A Sequential Stack-based Machine for Graph Rewriting*, Internal Report, University of Nijmegen 1990.

[Lester 89] D.Lester: *Stacklessness: Compiling Recursion for a Distributed Architecture*, Conference on Functional Programming Languages and Computer Architecture (FPCA), Addison-Wesley 1989.

[Loogen et al. 89] R.Loogen, H.Kuchen, K.Indermark, W.Damm: *Distributed Implementation of Programmed Graph Reduction*, Conference on Parallel Architectures and Languages Europe (PARLE) 1989, Lecture Notes in Computer Science 365, Springer Verlag 1989.

[Peyton-Jones 87] S.L.Peyton-Jones: *The Implementation of Functional Programming Languages*, Prentice Hall 1987.

[Peyton-Jones 91] S.L.Peyton-Jones: *The Spineless tagless G-machine: Second Attempt*, Proceedings of the Workshop on Parallel Implementation of Functional Languages, Technical Report CSTR 91–07, University of Southhampton 1991.

[Sekar et al. 91] .C.Sekar, Sh. Pawagi, I.V.Ramakrishnan: *Small Domains Spell Fast Strictness Analysis*, ACM Conference on Principles of Programming Languages (POPL) 91.