

Implementing the Evaluation Transformer Model of Reduction on Parallel Machines*

G. L. Burn

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate
LONDON SW7 2BZ
United Kingdom.

Abstract

The evaluation transformer model of reduction generalises lazy evaluation in two ways: it can start the evaluation of expressions before their first use, and it can evaluate expressions further than weak head normal form. Moreover, the amount of evaluation required of an argument to a function may depend on the amount of evaluation required of the function application. It is a suitable candidate model for implementing lazy functional languages on parallel machines.

In this paper we explore the implementation of lazy functional languages on parallel machines, both shared and distributed memory architectures, using the evaluation transformer model of reduction. We will see that the same code can be produced for both styles of architecture, and the definition of the instruction set is virtually the same for each style. The essential difference is that a distributed memory architecture has one extra node type for non-local pointers, and instructions which involve the value of such nodes need their definitions extended to cover this new type of node.

To make our presentation accessible, we base our description on a variant of the well-known G-machine, an abstract machine for executing lazy functional programs.

1 Introduction

Lazy evaluation is a restrictive evaluation mechanism in two ways. Firstly, it always chooses to reduce a particular redex, usually the leftmost one, at each step in the execution of a program. Secondly, it only ever evaluates expressions to weak head normal form (WHNF).

The evaluation transformer model of reduction lifts both of these restrictions, and is therefore a suitable candidate model for implementing lazy functional languages on parallel machines. It introduces the concept of an *evaluator*, which specifies the amount of evaluation to do to an expression. With each argument to a function, we associate

*Research partially funded by ESPRIT Project 415: "Parallel Architectures and Languages for AIP - A VLSI-Directed Approach".

an *evaluation transformer*, a mapping from the amount of evaluation required of the function application to that which is allowed of the argument. Using this information, we may know that there are several expressions which must be evaluated, and that they need more evaluation than to WHNF.

This paper discusses compiling code for the evaluation transformer model of reduction for both shared and distributed memory architectures.

To make the work more accessible, we have described the compilation of lazy functional languages to abstract machines based on a variant of the G-machine [Joh83, Aug87a, Joh87], called the Spineless G-machine [BPR88].

The key features of the implementations are:

- They support the evaluation transformer model of reduction.
- An independent parallel process is created to evaluate an argument to a user-defined function when it is known that the expression must eventually be evaluated. No coordination is needed between the spawning and spawned processes.
- The code produced for shared and distributed memory architectures is the same.
- Slightly better code can be produced for a distributed memory architecture if we distinguish between the tasks created for arguments to user-defined and base functions (such as $+$). In the first case, all we know is that *some task will need the value in the future*, whilst in the second we know that the *current task will need the value*. If the spawned argument¹ to a base function lies on a remote processor, then the spawning instruction can set up a link to return the value, rather than waiting to fetch it when the current task actually needs the value.
- In a distributed memory architecture, the addition of an extra node type to represent non-local pointers simplifies the definition of a distributed memory machine significantly. This idea arose originally out of the design of a garbage collection algorithm in [Les89a].
- We make a proper account of the use of *context-free* and *context-sensitive* evaluation transformer information. Context-sensitive information takes into account the textual context of an expression, and generally gives more information about an argument to a function than the context-free evaluation transformers, which are valid in any context. We will see that context-free information can be used to capitalise on some of the information which becomes available at run-time.
- An expression only needs to be marked as *being evaluated* when its evaluation begins, and unmarked at the end.

At the time of doing the work, many of these features were novel; some similar ideas now appear in publications of work that was proceeding concurrently with ours — see [PCS89, AJ89, LKID89] for example.

¹We will see that the code produced for $+$ will spawn one of the argument expressions and have the process evaluating the application of $+$ try to evaluate the other itself. It is therefore possible that the spawned argument will be evaluated while the process evaluating the application of $+$ evaluates the other argument.

We have purposely ignored the issues of load balancing and restricting the number of parallel tasks produced by a machine. The first is lower-level than the concerns of this paper, but we are encouraged by the results of [ELJ86], which seem to indicate that any load balancing strategy will be reasonably close to an optimal strategy. Regarding the second, we believe that the most effective way of managing a parallel machine is to have two versions of the code for each function: one parallel and one sequential; when the number of tasks in the machine reaches a certain level, each processor switches to running the sequential code, switching back again when the load drops below a certain level, and so on. This idea seems to have also appeared in most other implementation projects, and is encouraged by the theoretical results on general purpose parallel architectures of [Val88].

This paper is strongly based on [Bur88b], which gave a complete definition of a shared memory parallel G-machine that implemented the evaluation transformer model. Unfortunately the description of the machine instructions in that paper is almost unreadable, pushing the register transfer notation beyond its usefulness. The machine definition was modified to a distributed memory architecture, and specified using a functional program in [LB89]. This latter paper also included a number of significant improvements over the definition of the original shared memory architecture. Since then our understanding has matured further, so this paper presents a simpler view of the important issues, without getting bogged down in the precise definition of the instruction set of a particular abstract machine. Those who are interested in more detail are referred to [LB89], but be warned that the instruction set we introduce and use in this paper is slightly different to the one appearing in that paper! Evaluation transformer information can also be used for generating code for sequential machines [Bur90, Bur91].

In the next section we give an introduction to the evaluation transformer model of reduction. Our examples will be in terms of functions over lists. To set the scene for discussing parallel implementations, we describe some of the features of the Spineless G-machine in Section 3. Section 4 describes the structures that are needed in shared and distributed memory architectures to support parallel graph reduction using the evaluation transformer model of reduction. They are specified using a functional language. In Section 5 we show how to extend the instruction set of the Spineless G-machine in order to support the evaluation model on parallel machines, and in Section 6 we show how to compile code for these machines. As our descriptions are in terms of abstract machines, they contain some inherent ‘inefficiencies’. Section 7 discusses two of them briefly, showing how the work of this paper can be used as the basis for a real implementation. Finally, we take the opportunity in Section 8 to set this work in the wider context of other implementations of lazy languages on parallel machines. In the Appendix we give the complete compilation rules for a simple combinator language.

2 The Evaluation Model

There are two broad classes of methods we may choose to obtain parallelism in the evaluation of lazy functional languages which have no explicit parallel constructs. A machine may employ *speculative parallel evaluation*, where any redex in a graph is a possible candidate for reduction, or it may use *conservative parallel evaluation*, where only expressions whose values are known to be needed are evaluated.

Speculative parallelism wastes machine resources by evaluating expressions which may eventually be discarded. For example, in the expression:

```
if condition then e1 else e2
```

the value of only one of `e1` and `e2` will be needed, depending on the truth of the `condition`, so any evaluation of the other expression is wasted work. The problem is compounded in languages which allow the writing of expressions denoting infinite computations, for such computations may try and consume infinite amounts of resources². This would imply the need to garbage collect infinite processes, which is a difficult problem we would like to avoid. Therefore, we have investigated a conservative parallel reduction model. Nevertheless, others have defined implementations using speculative evaluation, [Par91] for example. It remains to be seen whether the extra parallelism made available by speculative evaluation is needed.

The purpose of this section is to give an intuitive development of the evaluation transformer model of reduction. We will see that each argument of a function has an evaluation transformer associated with it. Given the amount of evaluation allowed of an application of the function, the evaluation transformer model says how much evaluation needs to be done to the argument expression (which may be no evaluation). This is useful for two reasons. Firstly, knowing that a function needs to evaluate an argument means that the evaluation order can be changed, either creating a parallel process to evaluate it, or generating code which will evaluate it straight away, saving the cost of building a closure. Secondly, knowing that an expression needs more evaluation than to WHNF allows us to make further optimisations, for example, increasing the granularity of a process, or saving the cost of building closures for substructures, or generating more parallelism in the case of a parallel data structure such as a binary tree.

We will develop the evaluation model using our intuitions about how certain functions behave. We note however that evaluation transformers can be determined by a semantically sound analysis technique, such as abstract interpretation [Bur87, Bur91] or projection analysis [WH87, Bur90], which can be implemented in a compiler [Hun89]. The evaluation transformers found using these analyses can be used in a conservative parallel evaluation model as they will only allow the evaluation of expressions which would eventually have been evaluated using lazy evaluation.

2.1 Evaluators

We know that in some function applications, the argument will need more reduction than just to WHNF. Suppose that the data type `list` has been declared as³:

```
list * ::= Nil | Cons * (list *)
```

and the function `length` defined by:

```
length Nil          = 0
length (Cons h t) = 1 + (length t)
```

²An infinite computation does not necessarily mean no result is produced. When one has structured data types, a computation may produce a finite or unbounded amount of output as well as proceeding forever.

³All example code fragments will be written in Miranda [Tur85, Tur86]. Miranda is a trademark of Research Software Ltd.

To evaluate an application of `length`, the whole of the structure of the argument list will need to be traversed, but the values which are the first argument to the constructor `Cons` will never have to be evaluated. The function `sum` however, also has to evaluate the first argument to the constructor `Cons` for each element in the list.

```
sum Nil          = 0
sum (Cons h t) = h + (sum t)
```

We will call the process of recursively evaluating the expressions which are in the recursive places in a data type definition, creating the *structure* of the expression. The process will only terminate if the structure of the object is finite.

We will say that we can evaluate an expression using a particular *evaluator*, and call an evaluator which evaluates expressions to WHNF ξ_1 , an evaluator which evaluates the structure of a data type ξ_2 , and an evaluator which evaluates the structure of a data type and every non-recursive element of the data type to WHNF ξ_3 . For completeness, the evaluator ξ_{NO} does no evaluation. There is an obvious generalisation of the concept of evaluators to the situation where more evaluation is done to substructures than that done by ξ_3 , and to other recursively defined data types.

Note that although ξ_2 and ξ_3 are defined in terms of evaluating the whole structure of an expression, they could be implemented so that they only evaluated a certain number of elements of the data structure at a time; when the evaluated part of the list has been consumed, the evaluation of the expression could be reawakened to evaluate some more.

2.2 Evaluation Transformers

Not only do some functions require more evaluation of their argument than to WHNF, but the amount of evaluation of an argument may depend on the amount of evaluation required of the application. Consider for example the function `append` defined by

```
append Nil ys      = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Normally only the first argument to `append` needs to be evaluated to WHNF because a result in WHNF is required. However, if one was to require that an application of `append` was to deliver the structure of a list, such as in the application `length (append e1 e2)`, then clearly it can only do this if the structure of both of its arguments are created.

For each argument of a function therefore, we have to determine an *evaluation transformer*. An evaluation transformer for an argument is a function which says which evaluator may be used for evaluating the argument, given the evaluator for the expression. For example, the evaluation transformers for `append` are given in Table 1. $APPEND_i$ is the evaluation transformer for the i th argument of `append`. If an application of `append` is being evaluated with the evaluator ξ_2 , then the first argument can be evaluated with the evaluator ($APPEND_1 \xi_2$) which, from the table, is ξ_2 . Similarly, the second argument can be evaluated with ξ_2 . Further examples can be found in [Bur87, Bur91].

E	$APPEND_1 E$	$APPEND_2 E$
ξ_{NO}	ξ_{NO}	ξ_{NO}
ξ_1	ξ_1	ξ_{NO}
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_3

Table 1: Evaluation transformers for `append`

2.3 The Evaluation Transformer Model of Reduction

Evaluation transformers say how much evaluation can be done to an argument expression in a function application, given the amount of evaluation that can be done to the application. In a sequential implementation, this information is used to evaluate the argument to that extent before applying the function, saving the cost of building a closure for the argument in the heap. We will use this information in a parallel implementation by creating a parallel process to evaluate the argument at the same time as the function application.

3 The Spineless G-machine as a Basis for Parallel Implementations

Until fairly recently, implementations of lazy functional languages have been described using abstract machines, see [Joh83, Aug87a, Joh87, FW86, FW87] for example. Now people have begun explaining implementations in terms of more conventional compiler technology, as can be seen in [BHY88, PS89, Tra89]. This shift has taken place because of an increased understanding of how implementations should work. Nevertheless, we have chosen to describe the use of evaluation transformer information in terms of a parallel abstract machine. Not only does this provide a suitable level of abstraction, but it makes the presentation more accessible to those familiar with the more traditional abstract machine-based implementations.

Our development will be based on the *Spineless G-machine* [BPR88]. The key observation that distinguishes this machine from its predecessor, the G-machine of Augustsson and Johnsson [Aug87a, Joh83, Joh87], is that graph representing an expression can only become shared in the special circumstance that the expression is named, for example by becoming bound to a formal parameter to a function, or the the variable in a `let` or `letrec`. For a sequential machine, this has the advantage that the updating of expressions can be associated with *sharing*, rather than with each reduction step. The root node of any shared graph can be marked as such, and only shared graphs need to be updated. When evaluating an expression, any sharing of nodes on the spine of the graph being built at each reduction step is either mediated through the root of the expression being reduced, because the original expression was shared, or because the function being applied is of the form:

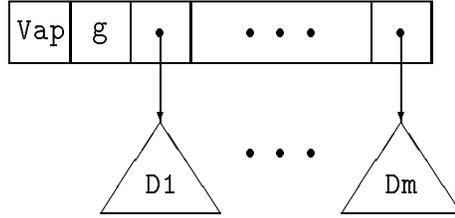


Figure 1: The VAP node representing the application (g D1 ... Dm)

telling how big it is. Whilst the original presentation of the Spineless G-machine was in terms of binary application nodes, this was only so that it could be easily compared with the classical G-machine. In this paper it is more convenient to use VAPs.

A compiler for the Spineless G-machine assumes that pointers to a complete set of argument expressions are available on the top of the stack at the entry to the code for a function. Function definitions are compiled using the \mathcal{F} -compilation rule:

$$\mathcal{F} \llbracket \mathbf{f} \ x_1 \ \dots \ x_n = \mathbf{g} \ D_1 \ \dots \ D_m \rrbracket = \mathcal{R} \llbracket \mathbf{g} \ D_1 \ \dots \ D_m \rrbracket [x_1 \mapsto n, \dots, x_n \mapsto 1] \ n$$

where the \mathcal{R} -compilation rule is used to compile the right-hand sides of function definitions. The third parameter to the \mathcal{R} -compilation rule records that the n arguments to \mathbf{f} are currently on the stack, and is incremented (decremented) as other items are pushed onto (popped from) the stack. Along with the third parameter, the second parameter maps the formal parameters of the function \mathbf{f} to the offsets into the stack where pointers to the graphs of the actual parameters can be found. The following invariant is maintained by the compiler:

If \mathcal{L} is some compilation rule, when beginning to compile the expression E using \mathcal{L} :

$$\mathcal{L} \llbracket E \rrbracket \ r \ n,$$

the index into the stack of the pointer to the actual parameter bound to the formal parameter x is given by:

$$(n - (r \ x)).$$

Note that this means that the stack is zero-indexed.

The code for the function \mathbf{f} creates graphs for each of the expressions which are arguments to \mathbf{g} (recall that it is implementing lazy evaluation), squeezes out the pointers to the arguments to \mathbf{f} by moving the m arguments of \mathbf{g} down n places, and then jumps to the code for \mathbf{g} if and only if there are enough arguments on the stack for the code for \mathbf{g} to be executed. This is expressed by the following compilation rule:

$$\mathcal{R} \llbracket \mathbf{g} \ D_1 \ \dots \ D_m \rrbracket \ r \ n = \mathcal{C} \llbracket D_m \rrbracket \ r \ n; \dots; \mathcal{C} \llbracket D_1 \rrbracket \ r \ (n+m-1); \text{SQUEEZE } m \ n; \text{PUSHFUN } \mathbf{g}; \text{ENTER}$$

The \mathcal{C} -rule compiles code to build the graph of an expression, and its details need not concern us further⁵. `PUSHFUN g` pushes a pointer to the code for \mathbf{g} onto the stack, and

⁵For those who are interested in its definition, the \mathcal{C} compilation rule for a simple combinator language is given in Section A.5 of the Appendix.

`ENTER` uses this to see if there are enough arguments on the stack in order to enter the code for `g`. If there are, then it removes the pointer and jumps to the code for `g`. Otherwise, it creates a new node containing the function application, and returns to the caller. This second case is orthogonal to the main issues of this paper and will be ignored in the subsequent development.

At this point it is worth noting that there are a number of possible optimisations of the above code. For example, if we know that `g` takes at most `m` arguments, then the code sequence `PUSHFUN g; ENTER` can be replaced by one which jumps directly to the code for `g`. Again, this type of optimisation is orthogonal to the work described in this paper. Our claim is not that we are developing the most efficient implementation, but using an abstract machine to show how evaluation transformer information can be used in compiling code for parallel implementations.

In the classical G-machine, the \mathcal{E} -compilation scheme was added to compile expressions which were known to need evaluating. For example, it can be used to compile applications of `+`, because `+` needs to evaluate its arguments before they can be added together:

$$\mathcal{R} \llbracket + \ D1 \ D2 \rrbracket \ r \ n = \mathcal{E} \llbracket D1 \rrbracket \ r \ n; \mathcal{E} \llbracket D2 \rrbracket \ r \ (n+1); \text{ADD}; \text{SQUEEZE } 1 \ n; \text{RETURN}$$

where the \mathcal{E} -rules generate code which evaluates an expression to WHNF and leaves a pointer to its graph on the top of the stack, and the instruction `ADD` adds the two integers together pointed at by the two top stack elements, and replaces the two pointers with a pointer to a node containing the result. As the result is in WHNF, the `RETURN` instruction is used to return to the code which caused the evaluation of this expression.

Either (or both) of `D1` and `D2` may themselves be arithmetic expressions, so we can define:

$$\mathcal{E} \llbracket + \ D1 \ D2 \rrbracket \ r \ n = \mathcal{E} \llbracket D1 \rrbracket \ r \ n; \mathcal{E} \llbracket D2 \rrbracket \ r \ (n+1); \text{ADD}.$$

If an argument to `+` is a general function application, then we have to define:

$$\mathcal{E} \llbracket g \ D1 \ \dots \ Dm \rrbracket \ r \ n = \mathcal{C} \llbracket g \ D1 \ \dots \ Dm \rrbracket \ r \ n; \text{EVAL}$$

where `EVAL` causes the evaluation of the expression pointed at by the top of stack to WHNF.

Another example of where we know that a particular expression needs to be evaluated is in compiling the conditional:

$$\mathcal{R} \llbracket \text{if } D1 \ D2 \ D3 \rrbracket \ r \ n = \mathcal{E} \llbracket D1 \rrbracket \ r \ n; \text{JFALSE } L1; \mathcal{R} \llbracket D2 \rrbracket \ r \ n; \text{LABEL } L1; \mathcal{R} \llbracket D3 \rrbracket \ r \ n$$

where the expression `D1` is evaluated, and then code for either `D2` or `D3` is executed, depending on whether or not `D1` evaluated to `true`.

The key feature of the evaluation transformer model of reduction is that it allows the evaluation of some argument expressions. If we were compiling code for a sequential machine, the evaluation transformer information could be incorporated into the code by adapting the definition of the \mathcal{E} -compilation scheme [Bur90, Bur91]. Instead, when we know an expression can be evaluated, we will build its graph, and generate a parallel task to evaluate it.

4 The Structure of a Parallel G-Machine

The data structures necessary for, and their use in the support of the evaluation transformer model of parallel graph reduction are developed and described in [BBKR89]. An attempt at a rational reconstruction of these was made in [Bur88a]. Unfortunately, some of the states of the nodes in the graph were coded up in a rather complex manner, and state information was kept on nodes which did not need it (integer nodes cannot have a task created to evaluate them for example). In keeping with our policy of using the abstract machine to explain our ideas, in the following discussion we will untangle the state information by adding some extra flags, and make the explanation simpler by only keeping state information on nodes that need it.

We will discuss the structure of a shared memory machine first, followed by that necessary for a distributed memory architecture. In order to make the presentation simpler, we will discuss an abstract machine which only supports the data types boolean, integer and list. Our description of the abstract machine will use fragments of Miranda code; a full description of a parallel, distributed memory abstract machine supporting the evaluation transformer model of reduction, which is very close to the one we will describe here, can be found in [LB89].

For those not familiar with Miranda,

```
state == (output, [processor], graph, task_pools, environment)
```

is a *type synonym* definition, saying that an object has type `state` if it is a tuple of five elements, which have the respective types `output`, `[processor]`, and so on, each of which is, or contains type synonyms itself, and will be defined subsequently in this paper. The special notation `[processor]` says that the second element of the tuple is a list of objects of type `processor`. We will also see declarations of the following form:

```
tagged_exp ::= BOOL bool | INT num          |
              NIL | CONS ev label label    |
              VAP ev [label]              |
              SVAP ev t_c t_e p_l [label]

ev  == num
t_c == bool
t_e == bool
p_l == [task_id]
```

This declares a *new* type, `tagged_exp`, which is like a mathematical sum-of-products, and is often called an *algebraic type*. An object of this type has the form of one of the alternatives which are separated by vertical bars. For example, two objects of this type are `(BOOL True)` and `(INT 5)`, where `BOOL` and `INT` are usually called *tags*, and tell us which part of the sum the data object comes from, and the arguments have the appropriate types (`True` is of type `bool` and `5` of type `num`). Further details of the Miranda language can be found in [Tur85, Tur86], and the book by Bird and Wadler uses a language pretty close to Miranda in its examples [BW88].

4.1 A Shared Memory Architecture

A shared memory architecture is one where the memory storing the graph and the task pools is shared by all of the processors. Its state is represented by the type `state`, and

consists of output generated so far, a list of processor states, the graph, the task pools, and an environment:

```
state == (output, [processor], graph, task_pools, environment)
```

The output of the machine contains the results of a program, and the environment provides a mapping between function names and their code; neither will concern us further in this paper. Note that the graph and task pools are part of the global state of the machine, and so are shared by all the processors. We now specify each of the other components in more detail.

4.1.1 Processors

A processor executes one task at a time. When it has either finished a task, or can proceed no further with its current task, then it obtains a new task from the task pool. The state of a processor is therefore just the state of the process it is currently executing.

```
processor == process_state
process_state == (thread, stack, dump)
thread == [gcode]
stack == [label]
dump == [(thread, stack)]
label == num
```

In the state of a process, the `thread` is the list of instructions (of type `gcode`) which must be executed by the process, the `stack` represents the stack of pointers, and the `dump` is used to record the state of the evaluation of an expression when a task recursively evaluates some subexpression⁶. A label can be thought of as being the address of a piece of graph.

We will not give a complete list of all of the instructions that are used in the parallel machine, but will discuss some specific ones in the text of the paper.

4.1.2 The Graph

The graph can be represented as a list of labelled, tagged expressions:

```
graph == [(label, tagged_exp)]
```

A tagged expression is one of the following:

```
tagged_exp ::= BOOL bool | INT num          |
              NIL | CONS ev label label    |
              VAP ev [label]               |
              SVAP ev t_c t_e p_l [label]

ev == num
t_c == bool
t_e == bool
p_l == [task_id]
```

⁶If a contiguous piece of store was being used for the stack of the process, then the dump would only need to store the stack pointer, rather than the whole stack. In a similar way, it would only have to store the current program counter rather than the list of instructions left for that process.

The first two alternatives store booleans and integers respectively, and the next two are for storing lists. A `CONS` node has an evaluator field (of type `ev`) to indicate how much evaluation has been or is being done to the expression which has the node as its root⁷. The Spineless G-machine distinguishes between expression graphs which are (possibly) shared, and those which are definitely not shared. Unshared expression graphs do not have to be updated. Two different node types are therefore used for storing VAPs: `VAP` and `SVAP`. VAPs are for storing expressions which do not need to be updated. They have an evaluator, which says how much evaluation must be done to the expression, and a list of labels, pointing to the function being applied and its list of arguments. An `SVAP` is used to represent the graph of an expression which may be shared (hence the `S`), and so must be updated when it has been evaluated to WHNF. The interpretations of the extra fields of a particular `SVAP`:

`SVAP e tc te pl ls`

are:

- `tc = true` if and only if a task has been created to evaluate the expression.
- `te = true` if and only if a task has started evaluating the expression.
- `pl` contains a list of identifiers of tasks waiting for the expression to be evaluated.

In a way, the flag `te` and the pending list are like an implementation of a semaphore [Dij65]. Initially it effectively has the value 1. When a task wants to evaluate the expression, then it does a `P` operation on the semaphore. Any further attempt to do a `P` operation on the node whilst it is being evaluated results in the task attempting it to be queued on the semaphore (that is, on the pending list). The number of `V` operations that need to be done when the evaluation of an expression has been completed depends on the execution profile of the program.

In a parallel graph reduction machine, the graph is the communications medium between processes⁸. This means that if a task is created to evaluate an expression, then the expression must be updated with the WHNF of the expression at the completion of the task, so that other tasks requiring the value of the expression can obtain the value. A `VAP` is used when there is only one pointer to the expression, and so the expression does not need to be updated. However, if a task is created to evaluate a `VAP`, the `VAP` must be changed to a `SVAP`, with all its attendant status information, to force the task which evaluates the expression to update the graph, so that the creator of the node can access its reduced value. Another way of saying this is that creating a task to evaluate an expression causes it to be shared.

4.1.3 Task Pools

```
task_pools == (active, blocked)
active     == task_pool
blocked   == task_pool
```

⁷We shall see in Section 6.2.2 that an application of `Cons` being evaluated with ξ_2 or ξ_3 will return the first `CONS` node, marked with the appropriate evaluator, and spawn processes to evaluate the subexpressions that need to be evaluated by that evaluator.

⁸Note that a node in the graph is usually more persistent than a process.

There are two task pools: when processes are created or they have become unblocked, they are put in the active task pool; a process becomes blocked if it needs the value of an expression being evaluated by some other task, whereupon it is put in the blocked task pool.

```
task_pool      == [(task_id, task_pool_entry)]
task_pool_entry == (vap_node, pending_count, update_node,
                    process_state)
vap_node       == label
task_id        == label
label          == num
pending_count  == num
update_node    == label
```

These task pools are represented by a list of `task_pool_entries` paired with their `task_ids`. The `vap_node` contains the label of the node which is to be evaluated by the task. When a task is created, the `pending_count` is set to zero. It is incremented if the task tries to evaluate an expression which some other task has started to evaluate. When a task has no more work it can do until it receives these values, it is put in the `blocked` task pool. The task is put back in the `active` task pool when its `pending_count` reaches zero again. The `update_node` is the label of the node that has to be updated with the weak head normal form (WHNF) value of the expression. The `process_state` is the state of the process at the point where it was last suspended.

4.2 A Distributed Memory Architecture

The key feature which distinguishes a distributed memory architecture from a shared memory one is that there is no global memory; the memory is distributed across the machine, with each processor having its own local store. To avoid bottle-necks in the machine, this means that the graph must be distributed over the local memories of the machine, but there must still be a global address space, and each processor must have its own local task pool. As we defined the state of a shared memory architecture, we will now do this for a distributed memory architecture.

The state of the machine can be represented by the type:

```
state == (output, [processor], [message], network)
```

where the graph and task pools have been removed from the global state of the machine. Instead we have a list of messages, and a representation of the communication network of the machine; the former represents the dynamic state of the network, the messages in transit, and the latter its static state, namely its topology and timing information. Again, the structure of the output will not concern us further, nor will we need to make any further comments concerning the communications network.

4.2.1 Processors

Conceptually, we may consider each processor to consist of a processor identifier, the instruction stream of the currently active task (`thread`), the local task pools, the local graph and the environment.

```
processor == (processor_id, thread, task_pools, graph, environment)
processor_id == num
```

In our distributed memory architecture, we distinguish between tasks which have started executing and those which have not; only tasks which have not begun executing are allowed to be migrated to other processors, to save the overhead of sending any state information with the migrated process. Therefore the distributed memory architecture has three task pools:

```
task_pools == (active, blocked, migratable)
active == task_pool
blocked == task_pool
migratable == task_pool
```

The first (`active`) contains those tasks that are executable and have been started or imported⁹, the second (`blocked`) contains tasks that have been suspended, and the third (`migratable`) contains tasks that may be exported, that is, those that are executable, but have not been started or imported.

We can make `label` entries globally unique by pairing them with the processor identifier (of type `processor_id`). The identifier of a processor is a `number`. There are several places, on the pending list for example, where two global addresses are kept, meaning that two copies of the processor identifier are kept. Clearly it is more space efficient, in a real implementation, to store one copy of the processor identifier and the two local identifiers of the task and graph labels.

```
global_task_id == (processor_id, task_id)
global_label == (processor_id, label)
```

We add one more type of node that can be in the graph, the node tagged with `OUTIND`, for *output indirection*. All pointers from one processor to another point to a local output indirection node, which points to the node on the remote processor. This makes explicit which pointers are local and which are global. All local pointers therefore only need contain a local label, and do not need the extra `processor_id` field. Furthermore, all the instructions that depend on looking at the objects in the graph do not have to have an explicit test on whether or not a pointer is local or global; they know by looking at the tag of the graph object. The use of `OUTINDs` has one further advantage, all local pointers to the global object can point to the same output indirection node. When the remote expression has been evaluated, and copied to the local processor, the value can overwrite the output indirection node so that all pointers that referred to the `OUTIND` node now share the reduced copy; the value of the expression only needs to be requested once. With this in mind, we have to extend the `tagged_exp` data type declaration to be:

⁹This condition prevents tasks being moved more than once.

```

tagged_exp ::= BOOL bool | INT num           |
            NIL | CONS ev label label      |
            VAP ev [label]                 |
            SVAP ev t_c t_e p_l [label]    |
            OUTIND ev t_r l_p_l global_label

ev      == num
t_c     == bool
t_e     == bool
t_r     == bool
p_l     == [(global_label, global_task_id)]
l_p_l   == [task_id]

```

Note that, besides a pointer to the remote node, an `OUTIND` needs some status information, which is used to cut down message traffic. Once a request has been made to evaluate a remote node, no further request need be sent. We will see later that not all commands to evaluate the remote expression actually require its value to be returned to the local processor, so the `t_r` field records whether or not its value has been requested. Local tasks which need the value of the remote expression are queued on the local pending list (`l_p_l`). The pending list of an `SVAP` node must now also contain the labels of the `OUTIND` nodes which have to receive a copy of the value of the expression when it has been reduced to WHNF.

5 Supporting the Evaluation Transformer Model on a Parallel Machine

Taking a sequential abstract machine, such as the Spineless G-machine, and extending it to support of the evaluation transformer model of reduction on a parallel machine consists of two orthogonal procedures:

- making the abstract machine parallel, and
- supporting different evaluators.

To make the abstract machine parallel, a number of new instructions have to be added to the machine, such as those which create tasks or select a new task to execute, instructions which cause the evaluation of an expression need to be modified to deal with the status information, and the data structures of the machine extended. We discussed the data structures in the previous section, and will discuss particular key instructions in this section. To support the evaluation transformer model of reduction, we need to show how different evaluators can be implemented, and how we can ensure that expressions are evaluated to the required extent.

5.1 Supporting Parallelism

5.1.1 Creating Tasks to Evaluate Expressions

Some base functions, such as `+` and `case`, require one or more of their arguments to be evaluated to WHNF *before* an application of the function can be reduced further.

This is different to the information that we have about user-defined functions, where an evaluation transformer may say that a particular argument can be evaluated, because its value is going to be needed *sometime in the future* (possibly by another task).

The **ISPAWN** instruction is used to create a task to evaluate an expression. Tasks are only created for unevaluated expressions (**SVAP** and **VAP** nodes), and only one task is created for an expression, all future **ISPAWN** instructions for that node are ignored. When the node being **ISPAWN**ed is an **OUTIND**, then a message is sent to the remote node to cause its evaluation. The **I** in **ISPAWN** comes from the fact that the instruction can be ignored without affecting the final answer from the program, but ignoring it will change the execution behaviour of the program.

In the case of an application of a base function such as **+**, we may decide to spawn a task for evaluating one of the argument expressions, and try to evaluate the other expression with the current task, saving the overhead of task switching. Furthermore, when the current task has finished evaluating its expression, it can check to see if the evaluation of the other argument has been started yet. If not, then it could evaluate that expression too¹⁰. A problem arises for a distributed memory architecture when one or more of the argument expressions are on a remote processor. In this case, we have to arrange for the remote graph to be evaluated and its value returned to the local processor (overwriting the **OUTIND** graph node). For a distributed memory architecture therefore, we need a special type of spawn instruction, which we have called **BSPAWN**, the **B** reminding us that it is to create a task as the result of an application of a base function, which requires its reduced value. If the node being **BSPAWN**ed is a local **VAP** or data object node, then the instruction behaves just as **ISPAWN** does. It is only when an **OUTIND** is **BSPAWN**ed that the action taken is different. In this case, the instruction sets up a linkage between the remote task and the local task and **OUTIND** node. The local task has its pending count incremented to indicate that it has requested the value of a remote node, and part of the mechanism of returning the value from the remote node is to decrement this pending count. As well, the task requested field of the **OUTIND** node is set to **true** to indicate that its value has been requested. When the remote expression has been evaluated, it must send the value back to overwrite the **OUTIND** node. After the first **BSPAWN** of an **OUTIND**, all future requests are queued on the local pending list of the **OUTIND** node, cutting down message traffic. Note that the current task is not suspended when it executes a **BSPAWN** instruction, but only when it comes to the point in the code where the value of the expression is needed (see the discussion of the **EVAL** instruction). Ignoring a **BSPAWN** instruction will cause deadlock.

5.1.2 Evaluating an Expression

The **EVAL** instruction evaluates the graph pointed to by the top of the stack. Its action in a shared memory machine is defined in Figure 2 by the function **eval**, which takes one state of the machine and returns the state after the execution of the instruction (where we assume that the processor executing the **EVAL** instruction is at the front of the list of processors)¹¹. We assume that a number of auxiliary functions have been defined:

¹⁰This idea grew up during the time when we were holding regular meetings with Simon Peyton Jones and his GRIP team. It has already been reported elsewhere, in [PCS89, Bur88b] for example.

¹¹Miranda has lists as a built-in data type, with **' :** as an infix **Cons**, so that **Cons h t** is written **h : t**, and the empty list is **[]** rather than **Nil**. It also allows a shorthand notation for finite lists, so that the list: **Cons a (Cons b (Cons c Nil))** can be written: **[a, b, c]**.

```

eval (o, (c, l:s, d):ps, g, tp)
  = (o, (c, l:s, d):ps, g, tp), if isBOOL te \ / isINT te
                                \ / isNIL te \ / isCONS te
  = eval_vap te (o, (c, l:s, d):ps, g, tp), if isVAP te
  = eval_svap te (o, (c, l:s, d):ps, g, tp), if isSVAP te
  where
    te = lookup_graph l g
    eval_vap (VAP e ls) (o, (c, l:s, d):ps, g, tp)
      = (o, ([LOAD], ls ++ s, d), g, tp)
    eval_svap (SVAP e tc te pl ls) (o, (c, s, d):ps, g, tp)
      = (o, ([LOAD], ls, (UPDATE:c,s):d):ps, g', tp), if ~te & ~tc
      = (o, ([LOAD], ls, (UPDATE:c,s):d):ps, g', tp'), if ~te & tc
      = (o, (BLOCK:c, s, d):ps, g, tp), if te
    where
      g' = set_te True l g
      tp' = remove_task l tp

```

Figure 2: Definition of the EVAL instruction for a shared memory machine

- `isBOOL`, `isINT`, `isNIL`, `isCONS`, `isVAP` and `isSVAP` return true if and only if their argument respectively has tag `BOOL`, `INT`, `CONS`, `VAP` or `SVAP`.
- `set_te b l g` sets the task-executing flag of the node with label `l` in the graph `g` to `b`;
- `remove_task l tp` removes the task created for the node with label `l` from the task pool `tp`; and
- `lookup_graph l g` returns the tagged expression which has the label `l` in the graph `g`.

We can now explain the action of `EVAL` in more detail:

- If the expression is already evaluated, that is, it is a boolean or an integer, or a list node, then the instruction has no effect.
- If the expression is a `VAP`, then the current task must have the unique pointer to it. This means that the expression does not have to be updated when it has been evaluated, so all the pointers can just be loaded onto the stack (so the stack becomes `(ls ++ s)`, popping the pointer to the `VAP` node off the stack). The `LOAD` instruction tests to see if the top of stack points to a function or another `VAP` node of some description. In the former case it effectively does an `ENTER`, whilst in the latter case it acts a bit like `EVAL`.
- If the expression is a `SVAP`, then there are three cases to consider:

- If no task is evaluating the expression already, and no task has been created to evaluate it, then the current task may evaluate it, after setting the task executing flag to `True`.
- If a task has been created to evaluate the expression, but no task has actually started evaluating it, then the entry in the task pool for the expression can be removed (to save scheduling a redundant task later on), and the current task can evaluate the expression.
- If a task is already evaluating the expression, the current task must be suspended, awaiting the value of the expression. It is put on the pending list of the task, and its pending count is incremented by one. These actions are performed by the `BLOCK` instruction.

In a distributed memory machine, the corresponding actions have to be performed. For the case where the expression to be evaluated is an `OUTIND`, the actions performed are identical to those performed with the `BSPAWN` of an `OUTIND`, except that current task must be suspended; an `EVAL` instruction says that the current task cannot proceed any further until it actually has the value of the expression.

5.1.3 The Life Cycle of a Task

When a task is created to evaluate an expression, its initial code sequence is :

```
EVAL; REMOVE; GETTASK
```

Recall that an expression which is being evaluated by a task is effectively shared (between the task which is evaluating the expression and any other which may require its value), and so the tag of the root node of its graph will initially be `SVAP`. The `EVAL` instruction will cause the expression to be evaluated and updated with the evaluated result, and all the tasks waiting for its value to be informed that it has been evaluated (and sent the value on a distributed memory architecture if necessary). The `REMOVE` instruction removes the current task from the task pool, and `GETTASK` obtains a new task from the task pool, if there is one. On a distributed memory architecture the load distribution strategy decides what to do when a processor has no task to execute, whilst on a shared memory architecture the processor could busy wait until a task becomes available, or do a local garbage collection.

5.2 Supporting Evaluation Transformers

The two key points about the evaluation transformer model of reduction are that:

- we may know that an expression has to be evaluated, and
- the expression may need more evaluation than to `WHNF`.

We will see in the next section that knowing that an expression needs to be evaluated means that we can construct its graph and spawn a parallel process to evaluate it. In this section we will consider how to force expressions be evaluated to more than `WHNF`.

Because the amount of evaluation that is required of an argument expression depends on that required of a function application, we will generate a number of different *versions*

E	$CONS_1 E$	$CONS_2 E$	$TAIL E$
ξ_{NO}	ξ_{NO}	ξ_{NO}	ξ_{NO}
ξ_1	ξ_{NO}	ξ_{NO}	ξ_1
ξ_2	ξ_{NO}	ξ_2	ξ_2
ξ_3	ξ_1	ξ_3	ξ_2

Table 2: Evaluation Transformers for `cons` and `tail`

of the code for each function, one for each evaluator that can evaluate an application of a function. Suppose that we are generating the code for the function defined by:

```
f x1 ... xn = g D1 ... Dm
```

when an application of it is to be evaluated with the evaluator ξ . Firstly note that the evaluator which is evaluating the application of \mathbf{f} is the one which then has to evaluate the application of \mathbf{g} . This is indicated in the code by giving the `ENTER` instruction an evaluator as an argument, which says which version of the code for \mathbf{g} to choose; `ENTER ξ` says to choose the ξ version. Secondly, the evaluation transformers for \mathbf{g} say how much evaluation can be performed to each of the \mathbf{D}_i in the application of \mathbf{g} , so these can be used to create parallel tasks to evaluate some of the \mathbf{D}_i , the ones which need evaluating, to the extent given by the evaluation transformers¹².

So far the evaluator information has only been used to say how much evaluation can be done to an expression, and to pass it to the (tail-recursive) calls of other functions. It is in the compilation of applications of the constructors of a data type that we cause expressions to be evaluated to the extent given by the evaluators. We can get a clue of how this can be done in general by considering the evaluation transformers given for `cons` in Table 2. For example, if an application of `cons` is to be evaluated with ξ_3 , then the evaluation transformers say that its first argument (its head) can be evaluated to WHNF (with ξ_1), and its tail with ξ_3 . In implementing this we have a choice. Either the current task can lock out all other tasks from accessing the value of the expression until it has been completely evaluated to the required extent, or it can create a `CONS` node, make it available for other processes, and arrange for the required evaluation of the subexpressions to be done. The second option is probably more sensible in a parallel implementation. The compilation of `cons` will be discussed further in Section 6.2.2.

When an expression graph is shared, it is possible that it will require further evaluation than was known when it was created. For example, applications of the function:

```
f x ys = if x=0 then sumlist ys else length ys
```

can only guarantee to need to evaluate their second argument with ξ_2 . However, if the first argument in an application reduces to 0, then the second argument could be evaluated with ξ_3 . We would therefore like to be allowed to change the evaluator field at run-time.

¹²Note that tasks for evaluating the required arguments of \mathbf{f} would have been created by the function which called \mathbf{f} .

```

updateev evltr (o, (c, l:s, d):ps, g, tp)
  = updateev_vap te (o, (c, l:s, d):ps, g, tp), if isVAP te
  = updateev_svap te (o, (c, l:s, d):ps, g, tp), if isSVAP te
  = updateev_cons te (o, (c, l:s, d):ps, g, tp), if isCONS te
  = (o, (c, l:s, d):ps, g, tp), otherwise
  where
    updateev_vap (VAP e ls) (o, (c, l:s, d):ps, g, tp)
      = (o, (c, l:s, d):ps, g', tp)
    updateev_svap (SVAP e tc te pl ls) (o, (c, l:s, d):ps, g, tp)
      = (o, (c, l:s, d):ps, g', tp)
    updateev_cons (CONS e l1 l2) (o, (c, l:s, d):ps, g, tp)
      = (o, (c, l:s, d):ps, g, tp), if evltr<=e
      = (o, (c', l2:l:s, d):ps, g, tp), if (evltr=2) & (evltr>e)
      = (o, (c'', l1:l2:l:s, d):ps, g, tp), if (evltr=3) & (evltr>e)
    where
      g' = update_evltr (max [evltr, e]) l g
      c' = UPDATEEV 2:ISPAWN:POP 1:c
      c'' = UPDATEEV 1:ISPAWN:POP 1:UPDATEEV 3:ISPAWN:POP 1:c

```

Figure 3: Definition of the instruction UPDATEEV ξ for a shared memory architecture

The UPDATEEV ξ (for update evaluator with evaluator ξ) instruction does just this. Since the only types of node which can have more evaluation than to WHNF are CONS nodes and SVAPs and VAPs containing expressions of type list, UPDATEEV instructions will only be generated for these nodes. The actions of this instruction on a shared memory architecture are defined by the function `updateev` in Figure 3¹³, where again we assume we are dealing with the processor at the front of the list of processors. The function call:

$$(\text{update_evltr } (\text{max } [\text{evltr}, e]) \text{ l } g)$$

stores the maximum of the evaluators `evltr` and `e` on the node with label `l` in the graph `g`. There are three cases for what the instruction does:

- If the expression is a SVAP or VAP node, then it just records in the evaluator field of the node the maximum of the evaluator which is an argument to the UPDATEEV instruction and the evaluator currently in the evaluator field.
- If the expression is a CONS node, and the new amount of evaluation is less than that already recorded, then the instruction has no effect. Otherwise, it stores the new evaluator on the node, and causes the required evaluation of the two subgraphs of the CONS node. Storing the evaluator on the CONS node in this way means that future request for at most that much evaluation do not have to do any work.
- Otherwise it does nothing.

¹³In the definition of `updateev`, we give the evaluator number to the instruction UPDATEEV.

We note that it is possible for the evaluator to be changed on a SVAP (but not a VAP) node whilst it is being evaluated, so part of the task ending procedure needs to make sure that enough evaluation has been done to the reduced expression.

A complete specification of a distributed memory architecture that supports the evaluation transformer model of reduction can be found in [LB89].

6 Compilation

In Section 3 we discussed three different compilation rules that are used in the Spineless G-machine:

- \mathcal{R} used to compile the right-hand side of function definitions. The code produced will leave a pointer on the top of the stack to an expression in WHNF;
- \mathcal{E} used to generate code that will evaluate an argument expression and leave a pointer to its WHNF on the top of the stack; and
- \mathcal{C} used to generate code which will build the graph of an expression.

In this section we introduce one further compilation rule:

- \mathcal{P} which will construct the graph of an expression and spawn a parallel process to evaluate it.

The evaluation transformer model also allows us to specify how much evaluation can be done to an expression. This is most easily expressed by passing an evaluator as an argument to each of the compilation rules, except \mathcal{C} , which is used when no evaluation can be done to an expression.

The evaluation transformers motivated in Section 2 have been called *context-free* evaluation transformers [Bur87, Bur91], because they tell us information about how a function is guaranteed to use its arguments, *no matter what the other arguments are*. For example, the context-free evaluation transformer for the first argument of the function `append`, $APPEND_1$, given in Table 1, tells us that whenever an application of `append` is being evaluated with ξ_3 , then the first argument can be evaluated with ξ_3 , no matter what expression the second argument is bound to.

Sometimes we can find out more information about an argument to a function in a particular textual application context. Typically this happens with higher-order functions, the function `apply` for example:

```
apply f x = f x
```

The context-free evaluation transformer for the second argument of `apply` says that no evaluation can be done to the second argument, because `apply` may be applied to a function which ignores its argument. However, in the application:

```
apply (+ e1) e2,
```

we know that `e2` has to be evaluated; by taking into account the textual context of the expression `e2`, we have been able to find out more information than is available from the context-free evaluation transformer for `apply`.

Both sorts of evaluation transformer information are useful; the first because the application context of an argument expression may mean that more evaluation can be done to the argument than that allowed by the context-free evaluation transformers, and the second to use some of the information that becomes available at run-time. In the previous section we saw an example of how we may discover at run-time that an expression needs more evaluation than was originally thought. Context-free evaluation transformer information is useful for this because knowing that more evaluation is required of a function application may mean that some more evaluation is allowed of the argument expressions in the application when the evaluation of the application begins. An extreme example of this is the function defined by:

$$f \ x_1 \ \dots \ x_n = x_i \ D_1 \ \dots \ D_m$$

where the function being applied is not known until run-time, and so no evaluation will have been done on any of the D_i . When the function becomes known, its context-free evaluation transformer information can be used to force the evaluation of some of the D_i . These issues are discussed extensively in [Bur91, Chapter 6].

We assume that the program to be compiled comes with both context-free and context sensitive evaluation transformer information. As in Section 2, we will denote the context-free evaluation transformer for the i th argument of a function by writing the name of the function in upper-case letters, and subscripting it with i . Thus F_i is the context-free evaluation transformer for the i th argument of the function f . The context-sensitive evaluation transformer information is provided in the form of annotations on every function application in the program, for example:

$$g \ \{ET_1\} \ D_1 \ \dots \ \{ET_m\} \ D_m$$

where $\{ET_i\}$ is the evaluation transformer for D_i , the i th argument expression.

We will now discuss the compilation of user-defined functions and some of the base functions.

6.1 Compiling User-Defined Function Definitions

Suppose that we are compiling code for the ξ -version of a function. The code has two entry points:

- **ENTRY** $\xi \ 1$ which creates processes to evaluate the arguments to the function being applied, given that the application has to be evaluated with ξ . It is used when the evaluation of an expression begins, and the expression is an application of this function¹⁴, as more evaluation may have been requested of the application than when it was first created. It is the entry point chosen by the **LOAD** instruction, and
- **ENTRY** $\xi \ 2$ used when a tail call is made to the code for the function, and is analogous to the code produced for the Spineless G-machine; it is the entry point chosen by the **ENTER** ξ instruction.

¹⁴It is also used in evaluating an application of a function of the form:

$$f \ x_1 \ \dots \ x_n = x_i \ D_1 \ \dots \ D_m.$$

When the function bound to x_i becomes known, and if the expression is being evaluated with ξ , entry point **ENTRY** $\xi \ 1$ is used for the function, so that some of the D_i can be evaluated.

The code for the function defined by:

$$\mathbf{f} \ x_1 \ \dots \ x_n = \mathbf{E}$$

therefore has the following form:

$$\begin{aligned} & \text{ENTRY } \xi \ 1; \text{ PUSH } (n-1); \text{ UPDATEEV } (F_n \ \xi); \text{ ISPAWN}; \text{ POP } 1; \dots; \text{ PUSH } 0; \\ & \text{UPDATEEV } (F_1 \ \xi); \text{ ISPAWN}; \text{ POP } 1; \text{ ENTRY } \xi \ 2; \mathcal{R} \ \xi \llbracket \mathbf{E} \rrbracket [x_1 \mapsto n, \dots, x_n \mapsto 1] \ n \end{aligned}$$

where `PUSH n` pushes a copy of the pointer at stack index `n` onto the stack and `POP n` pops `n` items from the stack. The code between the two entry points is called the *prelude*, and spawns off tasks for any argument expressions which are allowed by the context-free evaluation transformers for `f`. After that it falls through to code which is obtained by compiling the body of the function. Note that for any `j` such that $(F_j \ \xi) = \xi_{NO}$, the sequence of four instructions:

$$\text{PUSH } (j-1); \text{ UPDATEEV } (F_j \ \xi); \text{ ISPAWN}; \text{ POP } 1;$$

are omitted, as it is not safe to do any evaluation of the argument expression.

We now turn our attention to defining

$$\mathcal{R} \ \xi \llbracket \mathbf{E} \rrbracket [x_1 \mapsto n, \dots, x_n \mapsto 1] \ n$$

where `E` is an application of a user-defined function. The code for the Spineless G-machine for the function defined by:

$$\mathbf{f} \ x_1 \ \dots \ x_n = \mathbf{g} \ D_1 \ \dots \ D_m$$

was given to be:

$$\begin{aligned} \mathcal{R} \llbracket \mathbf{g} \ D_1 \ \dots \ D_m \rrbracket \ r \ n &= \mathcal{C} \llbracket D_m \rrbracket \ r \ n; \dots; \mathcal{C} \llbracket D_1 \rrbracket \ r \ (n+m-1); \\ & \text{SQUEEZE } m \ n; \text{ PUSHFUN } \mathbf{g}; \text{ ENTER} \end{aligned}$$

in Section 3. In compiling the ξ version of the definition:

$$\mathbf{f} \ x_1 \ \dots \ x_n = \mathbf{g} \ D_1 \ \{\text{ET}_1\} \ \dots \ \{\text{ET}_m\} \ D_m$$

we want to spawn a process to evaluate `Di` if the evaluation transformer `ETi` allows it. To aid in defining the compilation rules, we define the following function:

$$\mathcal{A} \ \xi = \begin{cases} \mathcal{C} & \text{if } \xi = \xi_{NO} \\ \mathcal{P} \ \xi & \text{otherwise} \end{cases}$$

The function \mathcal{A} tests to see if its argument is the evaluator ξ_{NO} or not. If it is, then it says that an expression has to be compiled with \mathcal{C} , which will build the graph of the expression, otherwise it can be compiled with $\mathcal{P} \ \xi$, which will construct the graph of the expression and spawn a parallel process to evaluate it with ξ . Now we can define:

$$\begin{aligned} & \mathcal{R} \ \xi \llbracket \mathbf{g} \ \{\text{ET}_1\} \ D_1 \ \dots \ \{\text{ET}_m\} \ D_m \rrbracket \ r \ n \\ &= \mathcal{A} \ (\text{ET}_m \ \xi) \llbracket D_m \rrbracket \ r \ n; \dots; \mathcal{A} \ (\text{ET}_1 \ \xi) \llbracket D_1 \rrbracket \ r \ (n+m-1); \\ & \text{SQUEEZE } m \ n; \text{ PUSHFUN } \mathbf{g}; \text{ ENTER } \xi \end{aligned}$$

In the case of the function application:

\mathbf{x}_i {ET1} D1 ... {ETm} Dm

the rule is the same except that the ENTER ξ instruction is replaced by a LOAD instruction, which forces the evaluation of the expression bound to \mathbf{x}_i , and then continues with the evaluation of the application. Note that all of the {ETi} in this case will say that no evaluation is allowed of any of the argument expressions, and so the compilation rule could be simplified to:

$$\mathcal{R} \xi \llbracket \mathbf{x}_i \text{ {ET1} D1 ... {ETm} Dm} \rrbracket r n = \mathcal{C} \llbracket Dm \rrbracket r n; \dots; \mathcal{C} \llbracket D1 \rrbracket r (n+m-1); \\ \text{SQUEEZE } m \ n; \text{ PUSH } ((n+m)-(r \ \mathbf{x})); \text{ LOAD}$$

The code produced by the \mathcal{P} -rule for a general function application is:

$$\mathcal{P} \xi \llbracket g \text{ {ET1} D1 ... {ETm} Dm} \rrbracket r n \\ = \mathcal{A} (\text{ETm } \xi) \llbracket Dm \rrbracket r n; \dots; \mathcal{A} (\text{ET1 } \xi) \llbracket D1 \rrbracket r (n+m-1); \text{SQUEEZE } m \ n; \\ \text{PUSHFUN } g; \text{ STORE } (m+1) \ \xi; \text{ ISPAWN}$$

where the STORE (m+1) ξ instruction creates a new SVAP node in the heap, containing the top (m+1) pointers from the stack, with its evaluator field set to ξ , its task created and task evaluating fields set to false, and the pending list set to empty¹⁵. Similar code is produced when g is replaced by \mathbf{x}_i .

Note that the \mathcal{P} compilation rule propagates the evaluation information inwards, just as we saw the \mathcal{E} -rule did in Section 3.

6.2 Compiling Applications of Some Base Functions

An important feature of the G-machine is that it tries to evaluate the strict arguments to base functions in-line. When a task reduces an expression to an application of a base function, the application cannot be performed until the strict arguments of the function have been evaluated. Rather than building the graph for all of the arguments, ISPAWNING them, and then suspending the task, we generate code which will try and evaluate one of the strict argument expressions in-line, which will in general remove some of the overhead of graph building and task switching. For the conditional, the first argument is evaluated in-line and either the second or the third argument, depending on the condition, will continue to be evaluated by the current task.

6.2.1 Arithmetic and Boolean Base Functions

We saw in Section 3 that the code produced for the Spineless G-machine for an application of $+$ was:

$$\mathcal{R} \llbracket + \ D1 \ D2 \rrbracket r n = \mathcal{E} \llbracket D1 \rrbracket r n; \mathcal{E} \llbracket D2 \rrbracket r (n+1); \text{ ADD}; \text{ SQUEEZE } 1 \ n; \text{ RETURN}$$

In Section 5 we saw that on a parallel machine the code for $+$ may spawn a task to evaluate one of the arguments and try to make the task evaluating the application of $+$ evaluate

¹⁵Note that this assumes that all expressions may be shared. If some sharing analysis was available, then there could be two different store instructions, creating a SVAP or a VAP depending on whether or not the expression was shared. The STORE instruction is also used by the \mathcal{C} compilation scheme for building graphs of expressions

the other. Moreover, if the evaluation of the spawned argument has not commenced by the time the other argument has been evaluated, that expression too should be evaluated by the same task. For a shared memory machine, this is expressed by the compilation rule:

$$\mathcal{R} \xi_1 \llbracket + D1 D2 \rrbracket r n = \mathcal{P} \xi_1 \llbracket D1 \rrbracket r n; \mathcal{E} \xi_1 \llbracket D2 \rrbracket r (n+1); \text{PUSH } 1; \text{EVAL}; \text{ADD} \\ \text{SQUEEZE } 1 (n+1); \text{RETURN}$$

where the \mathcal{E} -rule is used to compile the expression which is to be evaluated by the process evaluating the application of $+$. In this rule we have arbitrarily decided to spawn a parallel process for the first argument expression. A good heuristic for choosing which argument expression to spawn is that if one of the D_i is a formal parameter, and the other an application of a function, then the expression bound to the parameter should be spawned and the application evaluated by the process evaluating the expression $(+ D1 D2)$. This is because some evaluation is guaranteed to be possible of the application, but the expression bound to the parameter may already be being evaluated by another process. A more complex strategy may allow for the run-time testing of expressions in the heap to see if the expression bound to a parameter is being evaluated or not, and make the decision of which expression to evaluate at run-time. In a distributed memory machine the **ISPAWN** instruction generated by the \mathcal{P} -scheme should be replaced by a **BSPAWN** in case $D1$ is a formal parameter bound to an expression on a remote processor. Whilst this is not necessary for the correctness of the implementation, as the **EVAL** would cause the evaluation of the remote expression and the return of its value, using the **BSPAWN** instruction may allow the remote value to be returned sooner, and perhaps even before the **EVAL** instruction is executed, so the process will not be descheduled.

If one of the argument expressions to $+$ is a simple arithmetic expression, then it is probably more efficient for the task evaluating the application try to evaluate both argument expressions.

Similar compilation rules can be defined for all of the other binary arithmetic and boolean functions.

6.2.2 Cons

The compilation rules for an application of **Cons** are:

$$\mathcal{R} \xi_1 \llbracket \text{Cons } D1 D2 \rrbracket r n = \mathcal{C} \llbracket D2 \rrbracket r n; \mathcal{C} \llbracket D1 \rrbracket r (n+1); \text{CONS } 1; \\ \text{SQUEEZE } 1 n; \text{RETURN}$$

$$\mathcal{R} \xi_2 \llbracket \text{Cons } D1 D2 \rrbracket r n = \mathcal{P} \xi_2 \llbracket D2 \rrbracket r n; \mathcal{C} \llbracket D1 \rrbracket r (n+1); \text{CONS } 2; \\ \text{SQUEEZE } 1 n; \text{RETURN}$$

$$\mathcal{R} \xi_3 \llbracket \text{Cons } D1 D2 \rrbracket r n = \mathcal{P} \xi_3 \llbracket D2 \rrbracket r n; \mathcal{P} \xi_1 \llbracket D1 \rrbracket r (n+1); \text{CONS } 3; \\ \text{SQUEEZE } 1 n; \text{RETURN}$$

The argument to the **CONS** instruction is used to set the evaluator field of the **CONS** node, and tells how much evaluation has been requested of its subgraphs (its head and tail).

6.2.3 The Conditional

For the conditional:

if E1 E2 E3

it is pointless creating a parallel task to evaluate **E1** because no more work can be done by the task evaluating the application of **if** until **E1** has been evaluated. Therefore, the compilation rule for **if** is:

$$\mathcal{R} \xi \llbracket \text{if } D1 \ D2 \ D3 \rrbracket r \ n = \mathcal{E} \ \xi_1 \llbracket E1 \rrbracket r \ n; \text{JFALSE } L1; \mathcal{R} \ \xi \llbracket E2 \rrbracket r \ n; \\ \text{LABEL } L1; \mathcal{R} \ \xi \llbracket E3 \rrbracket r \ n$$

Note that the expression **E1** is evaluated with ξ_1 , that is, to WHNF, and that the evaluator which evaluates whichever of **E2** and **E3** is chosen is the same as that which is evaluating the application of **if**.

A similar sort of thing is done when **case** is used to compile pattern matching; the discriminating expression is first evaluated as far as it needs to be, and then the evaluator evaluating the application is the one which evaluates the chosen expression.

How should the compilation rule:

$$\mathcal{P} \xi \llbracket \text{if } D1 \ D2 \ D3 \rrbracket r \ n$$

be defined? Clearly it is a waste of time and memory to build both **D2** and **D3** because only one will be needed. Instead, if **y1** to **yk** are the free variables in $(\text{if } D1 \ D2 \ D3)$, then we can define a new function **h**:

$$h \ z1 \ \dots \ zk = \text{if } D1 \ D2 \ D3,$$

replace the expression $(\text{if } D1 \ D2 \ D3)$ with the application $(h \ y1 \ \dots \ yk)$, and compile it using $\mathcal{P} \xi$. This will then have the desired behaviour.

6.2.4 Tail

The function **tail** returns its argument list minus its first element. It could be defined by¹⁶:

$$\text{tail } (\text{Cons } E1 \ E2) = E2 \\ \text{tail } \text{Nil} = \text{error "Tried to take tail of an empty list"}$$

The compilation of the function **tail** shows two further points when generating code for the evaluation transformer model. Applications of **tail** are compiled using the following compilation rule in the Spineless G-machine:

$$\mathcal{R} \llbracket \text{tail } E \rrbracket r \ n = \mathcal{E} \llbracket E \rrbracket r \ n; \text{TAIL}; \text{EVAL}; \text{SQUEEZE } 1 \ n; \text{RETURN}.$$

It generates code which first evaluates the expression **E**. If the pointer on the top of the stack points to a **CONS** node, then the **TAIL** instruction then replaces it with a pointer to the tail of the list, otherwise an error stop occurs. Finally, the **EVAL** instruction evaluates the tail to WHNF.

Consider the evaluation transformers for the function **tail** given in Table 2, and suppose we are compiling an application of **tail** which is to be evaluated with ξ_3 . Using

¹⁶In Miranda, the function **error** causes an error stop in the execution of a program, printing the string which is its argument.

the Spineless G-machine code as a basis, we could generate the following code for a parallel G-machine:

$$\mathcal{R} \xi_3 \llbracket \text{tail } E \rrbracket r n = \mathcal{E} \xi_2 \llbracket E \rrbracket r n; \text{TAIL}; \text{EVAL}.$$

We have used the \mathcal{E} -scheme to compile the application of E because there is no point in starting a parallel process to evaluate the argument expression as the process evaluating the application of `tail` cannot do any more work until the E has been evaluated to WHNF. As we have noted before in the compilation of applications of `Cons`, compiling E with the evaluator ξ_2 means that the `CONS` node will be made available as soon as possible, and a parallel process will have been created to evaluate the tail (with ξ_2). This is as good as we can do by just using the straight evaluation transformer information. However, we know that when the tail of the list has been obtained, seeing as the result of the application `tail E` is to be evaluated with ξ_3 , then the tail of the list should be allowed to be evaluated with ξ_3 . Therefore we can generate the following code, which causes more evaluation of the argument to `tail`:

$$\mathcal{R} \xi_3 \llbracket \text{tail } E \rrbracket r n = \mathcal{E} \xi_2 \llbracket E \rrbracket r n; \text{TAIL}; \text{UPDATEEV } \xi_3; \text{EVAL}.$$

At this stage we can make an important observation:

In most works on making implementations more efficient, we use information like evaluation transformers to evaluate expressions as much as possible, as early as possible. If we were compiling code for a sequential implementation, the above sequence of code would evaluate the whole of the structure of the list E , creating graphs for each of the elements of the list, before the rest of the code was executed (see [Bur91, Section 6.4.1]). Later, the `UPDATEEV` instruction would crawl over the structure, evaluating each of the elements of the list. However, this probably does not save much time and space; the principal difference between ξ_2 and ξ_3 for a sequential machine is that the latter saves the cost of having to build graphs for the elements in a list. *Therefore, there seems to be a principle of judiciously delaying the evaluation of an expression until we know the maximum amount of evaluation that is required of it.*

In the case of an application of `tail`, we could generate code which only evaluated its argument to WHNF, took the tail, and only then caused the evaluation of the result with ξ_3 :

$$\mathcal{R} \xi_3 \llbracket \text{tail } E \rrbracket r n = \mathcal{E} \xi_1 \llbracket E \rrbracket r n; \text{TAIL}; \text{UPDATEEV } \xi_3; \text{EVAL}.$$

The utility of this can only be established by experimentation.

6.2.5 Let and Letrec

Many functional languages allow the use `let` and `letrec` (or `where` and `whererec`) to allow the programmer to define local subfunctions. A typical syntax might be:

```
let x = e1 in e2
letrec d in e
```

where the first one (non-recursively) binds the expression $e1$ to all the free occurrences of x in $e2$, and in the second d is a number of recursively defined expressions, which are

bound to names that can be used in the expression e . The value of the first expression is the value of e_2 , and e is the value of the second. In compiling these expressions, the evaluator which is used to evaluate e_2 or e is the evaluator which is evaluating the whole expression. If the evaluation transformer information in e_2 or e indicated that either e_1 or some of the expressions bound in d could be evaluated with a particular evaluator, then they could be compiled with either of the \mathcal{E} - or \mathcal{P} -schemes with that evaluator passed as an argument.

6.3 Peephole Optimisations

There are many optimisations that can be made to the code produced by the compilation rules in this section. For example, there is no point in spawning a process to evaluate an integer node whose address has just been pushed onto the stack. These are fairly easy to implement as a peephole optimisation phase in the compiler.

6.4 When Livelock Becomes Deadlock

The usual slogan for using program analysis techniques to change the evaluation order of lazy functional programs is: “If it is going to fail to produce any output, then it does not matter how it fails”. There is a curious thing that can happen when we use this in a simple way in a parallel implementation: programs which would have proceeded forever, not producing any output, may deadlock instead. Consider the two definitions:

$$\begin{aligned} a &= b + 1 \\ b &= a + 1. \end{aligned}$$

Normally any attempt to evaluate either a or b will result in a non-terminating computation, producing no output. Suppose that in a parallel machine we have the scenario that two processes begin to execute at the same time, one to evaluate a , and the other to evaluate b . Deadlock would then occur, for neither can be evaluated until the other has been evaluated. It is unclear whether this is a problem or not.

6.5 A Complete Compiler for a Simple Combinator Language

A complete compiler for a simple combinator language can be found in the appendix.

7 Making a Real Implementation

To emphasise that we have used an abstract machine in order to explain our ideas, in this section we point out a couple of issues that must be tackled in order to build a real implementation.

7.1 Stacks in the State of a Process

Both the shared and distributed memory architectures described in Section 4 include the stack being used to evaluate an expression as part of the state of a process. A real implementation needs to make process switching fast, and so an efficient way of switching stack frames needs to be devised. Independently Lester [Les89b] and Augustsson and

Johnsson [AJ89] proposed that the VAP node that is created for an expression should be used as the stack frame to evaluate the expression. When a subexpression needs to be evaluated, a link is established between it and the expression currently being evaluated, and the VAP for the subexpression becomes the new centre of computation. After the evaluation of the subexpression, a return is made to the parent expression via the link. This mechanism allows the saving of the state of the stack of a process to be very fast, just recording the pointer to the VAP node currently being evaluated.

7.2 Memory Allocation and Garbage Collection

In a distributed memory architecture, a memory allocation and garbage collection strategy needs to be designed so that each processor can act fairly independently of the others for most of the time. Bevan [Bev87] and Watson and Watson [WW87] independently designed an elegant reference counting algorithm for distributed memory architectures. The problem with a reference counting algorithm for local references is that it is more expensive than one dividing the heap into two semi-spaces and copying live data from one to another when the current semi-space becomes full [Har88]. Lester designed an algorithm which uses reference counting for interprocessor pointers, and the semi-space allocation/copying collector for local references [Les89a]. Externally referenced nodes in the local heap cannot be moved by the garbage collector, because the pointers held on other processes (in `OUTIND` nodes) would thereby be invalidated. One way of ensuring this is to add one extra type of node into a distributed memory architecture, an *input indirection*, and have all external references point to an input indirection. An input indirection then points to the current position of the graph node in the heap. During garbage collection, input indirections are not moved, but their contents need to be changed in order to point to the new location of the graph node.

8 Relationship With Other Work

A number of early papers about parallel G-machines were published, for example, [Aug87b, NP86, RRM⁺87, RRH⁺87]. As we have already observed in Section 3, these papers did not use the observations about sharing of expressions which were the basis for designing the Spineless G-machine, so they had heavy overheads in terms of locking and unlocking graph nodes during reduction steps. Most later papers, such as [PS89, AJ89] do not suffer from this problem

Most papers describe their implementation in terms of *serial combinators* [HG85]. Serial combinators are generated from lambda-lifted programs by lifting out the concurrent substructure. For example, if the function `g` is strict in its first and third arguments, then the function definition

$$f \ x1 \ \dots \ xn = g \ D1 \ \dots \ Dm$$

could be transformed to :

$$f \ x1 \ \dots \ xn = \text{spawn } v1 = D1 \ \text{and } v2 = D3 \\ \text{in} \\ g \ v1 \ D2 \ v3 \ D4 \ \dots \ Dm$$

where the `spawn` construct is compiled in a similar manner to a `let` except that, when the expressions bound to the `vi` are constructed, they are `ISPAWNed`. The method can be applied to recursively lift out any concurrent substructure from `D1` and `D3`.

We have shown that it is not necessary to use the program transformation to serial combinators before generating parallel code, as the top-level spawns are caused by the \mathcal{R} -scheme using the \mathcal{P} -scheme, and the concurrent substructure is spawned using the \mathcal{P} scheme. This has a number of advantages. Firstly, we do not need complex rules about where the `spawn` construct can appear in the right-hand side of a function definition. Secondly, in the evaluation transformer model, the amount of evaluation of a subexpression depends on the amount of evaluation of the whole expression. This is also sometimes the difference between whether or not a parallel task is created at all. Serial combinators have to be modified to have some sort of conditional spawn, see [LKID89] for example. Most importantly, serial combinators blur the distinction between context-free and context-sensitive evaluation transformers. Implicit in the definition of serial combinators is the use of context-free information when building the body of a function. The papers [HBP86, HBP88] were the first to introduce the idea of having a prelude for a function for the context-free information and using the context-sensitive information when building the body of the function. Unfortunately the (SKI-combinator) code produced in that paper spawned needed arguments twice, once when the body of the application was being built, and again in the prelude. In this paper we have used pairs of entry points for a function so that argument are not spawned twice for each function application. In Section 6.1 we showed the appropriate entry point to use. We note that [LKID89] also has the concept of a number of entry points.

The paper [RRM⁺87] introduces two new node types in the graph, `STRICT m` and `SPAWN m`. A `STRICT m` node indicates that there are `m` vertebrae on the spine above the node which can be evaluated in-line. In a similar manner, the `SPAWN` node says that there are `m` expressions which can be `ISPAWNed`. They are generated when there are some strict or `ISPAWN` subcomponents of an expression which is being constructed but not evaluated or `ISPAWNed` immediately (i.e. they are subexpressions of an expression being compiled using \mathcal{C}). We note firstly that the effect of the `SPAWN` node is captured by the prelude of `ISPAWNs` in the code we generate in this paper (Section 6.1), and so the special node is not necessary. Secondly, there is no advantage in putting in a `STRICT` node. In-line evaluation of an expression only saves time because the expression can be compiled using the \mathcal{E} -scheme, and hence less graph is built in its evaluation. Once the graph has been built, as is the case here, there is no point in evaluating the expressions in-line, and hence no need for the `STRICT` node. We can, however, achieve the desired effect by doing a source to source translation on the text of the program, as is discussed in [BPR88]. The second paper by the authors [RRH⁺87] seems to include these ideas.

The work of [LKID89] is closest to ours, being a distributed memory architecture which supports the evaluation transformer model of reduction. It has been simulated in `occam`, whereas we first specified an implementation using a functional language [LB89], and then used this as a basis for a transputer machine code implementation [KLB91].

9 Conclusion

We have shown how to compile code for shared and distributed memory architectures in order to support the evaluation transformer model of reduction. Adding one extra node type to those present in a shared memory machine, to store pointers to non-local objects, enables us to distinguish very cleanly between the two types of machine. Although originally introduced by Lester when designing a garbage collection algorithm [Les89a], it has enabled us to design a compiler which works for both shared and distributed memory architectures. Furthermore, these output indirection nodes have a number of other benefits. For example, they provide somewhere for a copy of the reduced value of the non-local object to be stored, and mean that only one copy of the remote value needs to be fetched, no matter how many objects on the local processor point to it.

Our discussion of different evaluators has been in terms of lists, and only three particular evaluators for that type. The essential feature of the evaluation transformer model of reduction is that of needing to evaluate an expression to a specified extent. Different data types will have different evaluators, and there may be other sensible evaluators for the list type, but the concepts introduced in this paper can easily be adapted to different evaluators; in Section 5.2 we saw how the evaluation transformers for the constructors of a type told us how the various evaluators could be implemented.

In this paper we have used the evaluation transformer information to try to do as much evaluation as possible. This has meant that we keep an evaluator on graph nodes representing function applications, and we choose the version of the code to use at run-time, because at run-time we may find that an expression needs more evaluation than could be determined at compile-time. Code could instead be produced which chose the version at compile-time. The problem with this is that any expression compiled with the \mathcal{C} -scheme would only ever get evaluated to WHNF, as we could not guarantee at compile-time that it needs more evaluation if it was ever evaluated. A discussion of compile-time and run-time choice of version can be found in [Bur90, Bur91]. We note in passing that projection analysis gives us more information in this respect than abstract interpretation, as sometimes it will say that it is not known if an expression will be evaluated, but if it ever is, then it needs a certain amount of evaluation [WH87, Bur90]. Nevertheless, it is only by using some run-time information that we are able to take into account that an expression needs more evaluation than we are able to determine at compile-time.

10 Acknowledgements

Most of this work was completed whilst working at the GEC Hirst Research Centre in Wembley, Middx, UK. Its development owes a lot to the many valuable discussions I was able to have with David Lester and John Robson, my colleagues at GEC, during that period. We were also fortunate in being able to take part in the invigorating GRIP technical meetings, led by Simon Peyton Jones. Inevitably some of the ideas developed in this paper bear resemblance to some of the issues discussed in his papers, for they were being developed at the same time and discussed in the same forum. It was also helpful to discuss our ideas with our colleagues in ESPRIT Project 415, and interesting to see how our work interacted with the development of implementations of the other language styles.

References

- [AJ89] L. Augustsson and T. Johnsson. The $\langle \nu, G \rangle$ -machine: An abstract machine for parallel graph reduction. In D.B. MacQueen, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 202–213. ACM, 11–13 September 1989.
- [Aug87a] L. Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.
- [Aug87b] L. Augustsson. A parallel G-machine. Technical Report PMG53, Department of Computer Science, Chalmers University of Technology, S-412 96 Goteborg, Sweden, 1987.
- [BBKR89] D.I. Bevan, G.L. Burn, R.J. Karia, and J.D. Robson. Design principles of a distributed memory architecture for parallel graph reduction. *The Computer Journal*, 32(5):461–469, October 1989.
- [Bev87] D.I. Bevan. Distributed garbage collection using reference counting. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, volume 2, pages 176–187, Eindhoven, The Netherlands, June 1987. Springer-Verlag LNCS 259.
- [BHY88] A. Bloss, P. Hudak, and J. Young. Code optimisations for lazy evaluation. *Lisp and Symbolic Computation: An International Journal*, 1(2):147–164, 1988.
- [BPR88] G.L. Burn, S.L. Peyton Jones, and J.D. Robson. The spineless G-machine. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 244–258, Snowbird, Utah, 25–27 July 1988.
- [Bur87] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Imperial College, University of London, March 1987.
- [Bur88a] G.L. Burn. Developing a distributed memory architecture for parallel graph reduction. In *Proceedings of CONPAR 88*, pages 53–62, Manchester, United Kingdom, 12–16 September 1988. Cambridge University Press.
- [Bur88b] G.L. Burn. A shared memory parallel G-machine based on the evaluation transformer model of computation. In *Proceedings of the Workshop on the Implementation of Lazy Functional Languages*, pages 301–330, Aspenäs, Göteborg, Sweden, 5–8 September 1988.
- [Bur90] G.L. Burn. Using projection analysis in compiling lazy functional programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 227–241, Nice, France, 27–29 June 1990.

- [Bur91] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman in association with MIT Press, 1991. 238pp.
- [BW88] R. Bird and P.L. Wadler. *An Introduction to Functional Programming*. Prentice-Hall Series in Computer Science. Prentice-Hall International (UK) Ltd., Hemel Hempstead, Hertfordshire, England, 1988.
- [Dij65] E.W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in F. Genuys (ed.), *Programming Languages*, Academic Press, New York, 1968, pp. 43–112).
- [ELJ86] D.L. Eager, E.D. Lazowska, and Zahorjan J. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.
- [FW86] J. Fairbairn and S.C. Wray. Code generation techniques for functional languages. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, pages 94–104, Cambridge, Massachusetts, USA, 1986.
- [FW87] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 34–45. Springer-Verlag LNCS 274, September 1987.
- [Har88] P.H. Hartel. *Performance Analysis of Storage Management in Combinator Graph Reduction*. PhD thesis, Computing Science Department, University of Amsterdam, 1988.
- [HBP86] C.L. Hankin, G.L. Burn, and S.L. Peyton Jones. A safe approach to parallel combinator reduction (extended abstract). In *Proceedings ESOP 86 (European Symposium on Programming)*, pages 99–110, Saabrücken, Federal Republic of Germany, March 1986. Springer-Verlag LNCS 213.
- [HBP88] C.L. Hankin, G.L. Burn, and S.L. Peyton Jones. A safe approach to parallel combinator reduction. *Theoretical Computer Science*, 56:17–36, 1988.
- [HG85] P. Hudak and B. Goldberg. Serial combinators: “optimal” grains of parallelism. In J.-P. Jouannaud, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 382–399. Springer-Verlag LNCS 201, September 1985.
- [Hun89] S. Hunt. Frontiers and open sets in abstract interpretation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, 11–13 September 1989.
- [Joh83] T. Johnsson. The G-machine. An abstract machine for graph reduction. In *Declarative Programming Workshop*, pages 1–20, University College London, April 1983.

- [Joh87] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.
- [KLB91] H. Kingdon, D.R. Lester, and G.L. Burn. A transputer-based HDG-machine. *The Computer Journal*, (4):290–301, August 1991.
- [LB89] D.R. Lester and G.L. Burn. An executable specification of the HDG-Machine. In *Workshop on Massive Parallelism: Hardware, Programming and Applications*, Amalfi, Italy, 9–15 October 1989. Academic Press. To appear in 1991.
- [Les89a] D.R. Lester. An efficient distributed garbage collection algorithm. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE Parallel Architectures and Languages Europe*, volume 1, pages 207–223. Springer-Verlag, 12–16 June 1989.
- [Les89b] D.R. Lester. Stacklessness: Compiling recursion for a distributed architecture. In *Conference on Functional Programming Languages and Computer Architecture*, pages 116–128, London, U.K., 11–13 September 1989. ACM.
- [LKID89] R. Loogen, H. Kuchen, K. Indermark, and W. Damm. Distributed implementation of programmed graph reduction. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Proceedings of PARLE 89*, volume 1, pages 136–157, Eindhoven, The Netherlands, 12–16 June 1989. SPRINGER-Verlag LNCS 365.
- [NP86] E. Nocker and R. Plasmeyer. Combinator reduction on a parallel G-machine. Technical report, Dept of Computer Science, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, 1986.
- [Par91] A. Partridge. *Dynamic Aspects of Distributed Graph Reduction*. PhD thesis, Department of Electrical Engineering and Computer Science, University of Tasmania, 1991. In preparation.
- [PCS89] S.L. Peyton Jones, C. Clack, and J. Salkild. High-performance parallel graph reduction. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Proceedings of PARLE 89*, volume 1, pages 193–206, Eindhoven, The Netherlands, 12–16 June 1989. Springer-Verlag LNCS 365.
- [Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.
- [PS89] S.L. Peyton Jones and J. Salkild. The Spineless Tagless G-Machine. In D. B. MacQueen, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 184–201. ACM, 11–13 September 1989.
- [RRH⁺87] M. Raber, T. Remmel, E. Hoffman, D. Maurer, F. Muller, H.-G. Oberhauser, and R. Wilhelm. Compiled graph reduction on a processor network. Technical report, Fachbereich 10, Universität des Saarlandes, Im Stadtwald, 6600 Saarbrücken, 1987.

- [RRM⁺87] M. Raber, T. Rimmel, D. Maurer, F. Muller, H.-G. Oberhauser, and R. Wilhelm. A concept for a parallel G-machine. Report SFB 124-C1, Fachbereich 10, Universitat des Saarlandes, Im Stadtwald, 6600 Saarbrucken, 1987.
- [Tra89] K.R. Traub. *Sequential Implementation of Lenient Programming Languages*. PhD thesis, Laboratory of Computer Science, MIT, September 1989. MIT/LCS/TR-417.
- [Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 1–16. Springer-Verlag LNCS 201, September 1985.
- [Tur86] D.A. Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.
- [Val88] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, Amsterdam, The Netherlands, 1988.
- [WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 385–407. Springer-Verlag LNCS 274, September 1987.
- [WW87] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, volume 2, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer-Verlag LNCS 259.

A Compilation Rules

In this appendix we give the complete compilation rules for a simple combinator language supporting the types boolean, integer and list. As we saw in Section 6.2.1, the only real difference between the code for a shared memory machine and that for a distributed memory one is in the code generated for applications of strict base functions (such as $\+$) with at least two arguments. For these functions, where an argument is **ISPAWN**ed in a shared memory architecture, it must be **BSPAWN**ed in a distributed memory machine. Therefore we will omit the compilation rules for such functions from this appendix, referring the reader to Section 6.2.1 instead. This has the advantage that all of the following compilation rules can be used for both types of architecture.

The abstract compiler which generates parallel G-code is divided into four compilation schemes. When using each particular scheme, the source program fragment should be matched with each rule of the scheme in turn. The type of an object appearing in the compilation rules can always be deduced from the letter representing it, as shown in Table 3.

As in Section 6.1, we define \mathcal{A} by:

$$\mathcal{A} \xi = \begin{cases} \mathcal{C} & \text{if } \xi = \xi_{NO} \\ \mathcal{P} \xi & \text{otherwise} \end{cases}$$

r	is an ‘environment’ indicating where variables reside on the stack
n	is the current stack depth
x	is a variable
b	is a boolean constant
m	is ≥ 0
i	is an integer constant
f	is a function
l	is a new and unique label
D	is an arbitrary expression

Table 3: The meanings of variables used in the compiler

A.1 Scheme \mathcal{F} (Function Definition)

This generates code for an entire function definition.

$$\mathcal{F} \llbracket f \ x_1 \ \dots \ x_m = D \rrbracket = \mathcal{R} \llbracket D \rrbracket [x_1 \mapsto m, \dots, x_m \mapsto 1] \ m$$

At the point of entry for a function, it is guaranteed that its first argument is pointed to by the top of stack, the second argument by the next-to-top element, and so on, so that the m th argument is $m-1$ elements from the top of the stack.

A.2 Scheme \mathcal{R} (Return Value)

$\mathcal{R} \ \xi \llbracket D \rrbracket \ r \ n$ generates code to evaluate D with ξ , push it onto the stack, and return from a function. The code will make the value of the expression available once the WHNF of it has been evaluated, even if further evaluation of it is taking place.

$$\begin{aligned} \mathcal{R} \ \xi \llbracket i \rrbracket \ r \ n &= \text{PUSHINT } i; \text{ SQUEEZE } 1 \ n; \text{ RETURN} \\ \mathcal{R} \ \xi \llbracket b \rrbracket \ r \ n &= \text{PUSHBOOL } b; \text{ SQUEEZE } 1 \ n; \text{ RETURN} \\ \mathcal{R} \ \xi \llbracket \text{Nil} \rrbracket \ r \ n &= \text{PUSHNIL}; \text{ SQUEEZE } 1 \ n; \text{ RETURN} \\ \mathcal{R} \ \xi \llbracket \text{Cons } D_1 \ D_2 \rrbracket \ r \ n &= \mathcal{P} \ \xi \llbracket \text{Cons } D_1 \ D_2 \rrbracket \ r \ n; \text{ SQUEEZE } 1 \ n; \text{ RETURN} \\ \mathcal{R} \ \xi \llbracket \text{head } D \rrbracket \ r \ n &= \mathcal{E} \ \xi \llbracket \text{head } D \rrbracket \ r \ n; \text{ SQUEEZE } 1 \ n; \text{ RETURN} \\ \mathcal{R} \ \xi \llbracket \text{tail } D \rrbracket \ r \ n &= \mathcal{E} \ \xi \llbracket \text{tail } D \rrbracket \ r \ n; \text{ SQUEEZE } 1 \ n; \text{ RETURN} \\ \mathcal{R} \ \xi \llbracket \text{if } D_1 \ D_2 \ D_3 \rrbracket \ r \ n &= \mathcal{E} \ \xi_1 \llbracket D_1 \rrbracket \ r \ n; \text{ JFALSE } l; \\ &\quad \mathcal{R} \ \xi \llbracket D_2 \rrbracket \ r \ n; \text{ LABEL } l; \mathcal{R} \ \xi \llbracket D_3 \rrbracket \ r \ n \\ \mathcal{R} \ \xi \llbracket f \ \{ET_1\} \ D_1 \ \dots \ \{ET_m\} \ D_m \rrbracket \ r \ n &= \mathcal{A} \ (ET_m \ \xi) \llbracket D_m \rrbracket \ r \ n; \dots; \\ &\quad \mathcal{A} \ (ET_1 \ \xi) \llbracket D_1 \rrbracket \ r \ (n+m-1); \\ &\quad \text{SQUEEZE } m \ n; \text{ PUSHFUN } f; \text{ ENTER } \xi \\ \mathcal{R} \ \xi \llbracket x \ \{ET_1\} \ D_1 \ \dots; \ \{ET_m\} \ D_m \rrbracket \ r \ n &= \mathcal{C} \llbracket D_m \rrbracket \ r \ n; \dots \mathcal{C} \llbracket D_1 \rrbracket \ r \ (n+m-1); \\ &\quad \text{PUSH } ((n+m)-(r \ x)); \text{ SQUEEZE } (m+1) \ n; \\ &\quad \text{LOAD} \\ \mathcal{R} \ \xi \llbracket \text{let } x=e_1 \ \text{in } e_2 \rrbracket \ r \ n &= \mathcal{C} \llbracket e_1 \rrbracket \ r \ n; \mathcal{R} \ \xi \llbracket e_2 \rrbracket \ r[x \mapsto (n+1)] \ (n+1) \end{aligned}$$

A.3 Scheme \mathcal{P} (Parallel Task)

$\mathcal{P} \ \xi \llbracket D \rrbracket \ r \ n$ generates code to construct the graph of D , leaving a pointer to it onto the stack, and spawning a parallel task to have it evaluated with ξ .

Note that in the rules for `head` and `tail`, there is no point in creating a parallel process for the argument expression. Therefore the graph of the whole application is created. The code for `head` and `tail` will then force the evaluation of their argument.

$$\begin{aligned}
\mathcal{P} \xi \llbracket i \rrbracket r n &= \text{PUSHINT } i \\
\mathcal{P} \xi \llbracket b \rrbracket r n &= \text{PUSHBOOL } b \\
\mathcal{P} \xi \llbracket \text{Nil} \rrbracket r n &= \text{PUSHNIL} \\
\mathcal{P} \xi_1 \llbracket \text{Cons } D1 \ D2 \rrbracket r n &= \mathcal{C} \llbracket D2 \rrbracket r n; \mathcal{C} \llbracket D1 \rrbracket r (n+1); \text{CONS } 1 \\
\mathcal{P} \xi_2 \llbracket \text{Cons } D1 \ D2 \rrbracket r n &= \mathcal{P} \xi_2 \llbracket D2 \rrbracket r n; \mathcal{C} \llbracket D1 \rrbracket r (n+1); \text{CONS } 2 \\
\mathcal{P} \xi_3 \llbracket \text{Cons } D1 \ D2 \rrbracket r n &= \mathcal{P} \xi_3 \llbracket D2 \rrbracket r n; \mathcal{P} \xi_1 \llbracket D1 \rrbracket r (n+1); \text{CONS } 3 \\
\mathcal{P} \xi_1 \llbracket \text{head } D \rrbracket r n &= \mathcal{C} \llbracket D \rrbracket r n; \text{PUSHFUN head}; \text{STORE } 2 \ \xi_1; \text{ISPAWN} \\
\mathcal{P} \xi \llbracket \text{tail } D \rrbracket r n &= \mathcal{C} \llbracket D \rrbracket r n; \text{PUSHFUN tail}; \text{STORE } 2 \ \xi; \text{ISPAWN} \\
\mathcal{P} \xi \llbracket \text{if } D1 \ D2 \ D3 \rrbracket r n &— \textit{see Section 6.2.3} \\
\mathcal{P} \xi \llbracket f \ \{\text{ET1}\} \ D1 \ \dots \ \{\text{ETm}\} \ Dm \rrbracket r n &= \mathcal{A} (\text{ETm } \xi) \llbracket Dm \rrbracket r n; \dots; \\
&\quad \mathcal{A} (\text{ET1 } \xi) \llbracket D1 \rrbracket r (n+m-1); \\
&\quad \text{PUSHFUN } f; \text{STORE } (m+1) \ \xi; \text{ISPAWN} \\
\mathcal{P} \xi \llbracket x \ \{\text{ET1}\} \ D1 \ \dots; \ \{\text{ETm}\} \ Dm \rrbracket r n &= \mathcal{C} \llbracket Dm \rrbracket r n; \dots \mathcal{C} \llbracket D1 \rrbracket r (n+m-1); \\
&\quad \text{PUSH } ((n+m)-(r \ x)); \text{STORE } (m+1) \ \xi; \\
&\quad \text{ISPAWN} \\
\mathcal{P} \xi \llbracket \text{let } x=e1 \ \text{in } e2 \rrbracket r n &= \mathcal{C} \llbracket e1 \rrbracket r n; \mathcal{P} \xi \llbracket e2 \rrbracket r[x \mapsto (n+1)] (n+1); \\
&\quad \text{SQUEEZE } 1 \ 1
\end{aligned}$$

A.4 Scheme \mathcal{E} (Evaluate)

$\mathcal{E} \xi \llbracket D \rrbracket r n$ generates code which evaluates D with evaluator ξ , and leaves a pointer to this value on top of the stack. The code will make the value of the expression available once the WHNF of it has been evaluated, even if further evaluation of it is taking place.

The \mathcal{E} -scheme is used when it is known that an expression needs to be evaluated, but it is not worth creating a parallel process to evaluate it, for one of two reasons:

- the expression is an argument to a strict base function of one argument, `head` for example, or the argument of a base function that must be evaluated before any further evaluation of an application of the function can be made, `if` for example; or
- some complexity analysis has been done and it has been determined that the work required to evaluate the expression does not warrant creating a parallel task for it.

In the first case, although a parallel task is not created to evaluate the whole expression, saving the the cost of suspending the current process, it may be worthwhile spawning subtasks to evaluate some of its subexpressions. Therefore, the \mathcal{E} -scheme spawns parallel subtasks where sensible. In the second case, we assume that there has been some pass of the compiler which also marks all the subexpressions of the expression being compiled with the \mathcal{E} -scheme, so that no parallel tasks will be created. The following compilation rules do not take evaluation complexity into account, but it should be clear how this could be done.

$$\begin{aligned}
\mathcal{E} \xi \llbracket i \rrbracket r n &= \text{PUSHINT } i \\
\mathcal{E} \xi \llbracket b \rrbracket r n &= \text{PUSHBOOL } b \\
\mathcal{E} \xi \llbracket \text{Nil} \rrbracket r n &= \text{PUSHNIL} \\
\mathcal{E} \xi \llbracket x \rrbracket r n &= \text{PUSH } (n-(r \ x)); \text{EVAL} \\
\mathcal{E} \xi \llbracket \text{Cons } D1 \ D2 \rrbracket r n &= \mathcal{P} \xi \llbracket \text{Cons } D1 \ D2 \rrbracket r n
\end{aligned}$$

$$\begin{aligned}
\mathcal{E} \xi_1 \llbracket \text{head } D \rrbracket r \ n &= \mathcal{E} \xi_1 \llbracket D \rrbracket r \ n; \text{HEAD}; \text{EVAL} \\
\mathcal{E} \xi_1 \llbracket \text{tail } D \rrbracket r \ n &= \mathcal{E} \xi_1 \llbracket D \rrbracket r \ n; \text{TAIL}; \text{EVAL} \\
\mathcal{E} \xi_2 \llbracket \text{tail } D \rrbracket r \ n &= \mathcal{E} \xi_2 \llbracket D \rrbracket r \ n; \text{TAIL}; \text{EVAL} \\
\mathcal{E} \xi_3 \llbracket \text{tail } D \rrbracket r \ n &= \mathcal{E} \xi_2 \llbracket D \rrbracket r \ n; \text{UPDATEEV } \xi_3 \text{ TAIL}; \text{EVAL} \\
\mathcal{E} \xi \llbracket \text{if } D1 \ D2 \ D3 \rrbracket r \ n &= \mathcal{E} \xi_1 \llbracket D1 \rrbracket r \ n; \text{JFALSE } l1; \mathcal{E} \xi \llbracket D2 \rrbracket r \ n; \text{JMP } l2; \\
&\quad \text{LABEL } l1; \mathcal{E} \xi \llbracket D3 \rrbracket r \ n; \text{LABEL } l2 \\
\mathcal{E} \xi \llbracket \text{f } \{ETm\} \ D1 \ \dots \{ET1\} \ Dm \rrbracket r \ n &= \mathcal{A} (ETm \ \xi) \llbracket Dm \rrbracket r \ n; \dots; \\
&\quad \mathcal{A} (ET1 \ \xi) \llbracket D1 \rrbracket r \ (n+m-1); \\
&\quad \text{PUSHFUN } f; \text{CALL } (m+1) \\
\mathcal{E} \xi \llbracket \text{x } \{ETm\} \ D1 \ \dots \{ET1\} \ Dm \rrbracket r \ n &= \mathcal{C} \llbracket Dm \rrbracket r \ n; \dots \mathcal{C} \llbracket D1 \rrbracket r \ (n+m-1); \\
&\quad \text{PUSH } ((n+m)-(r \ x)); \text{CALL } (m+1) \\
\mathcal{E} \xi \llbracket \text{let } x=e1 \ \text{in } e2 \rrbracket r \ n &= \mathcal{C} \llbracket e1 \rrbracket r \ n; \mathcal{E} \xi \llbracket e2 \rrbracket r[x \mapsto (n+1)] \ (n+1); \\
&\quad \text{SQUEEZE } 1 \ 1
\end{aligned}$$

A.5 Scheme \mathcal{C} (Construct Graph)

$\mathcal{C} \llbracket D \rrbracket r \ n$ generates code which constructs the graph of D , and leaves a pointer to this graph on the stack.

When a graph is created in the heap, then we mark it set its evaluator field to be ξ_1 , as it will require evaluation at least to WHNF if it is ever evaluated¹⁷.

$$\begin{aligned}
\mathcal{C} \llbracket i \rrbracket r \ n &= \text{PUSHINT } i \\
\mathcal{C} \llbracket b \rrbracket r \ n &= \text{PUSHBOOL } b \\
\mathcal{C} \llbracket \text{Nil} \rrbracket r \ n &= \text{PUSHNIL} \\
\mathcal{C} \llbracket f \rrbracket r \ n &= \text{PUSHFUN } f \\
\mathcal{C} \llbracket x \rrbracket r \ n &= \text{PUSH } (n-(r \ x)) \\
\mathcal{C} \llbracket \text{Cons } D1 \ D2 \rrbracket r \ n &= \mathcal{C} \llbracket D2 \rrbracket r \ n; \mathcal{C} \llbracket D1 \rrbracket r \ (n+1); \text{CONS } 1 \\
\mathcal{C} \llbracket D1 \ \dots Dm \rrbracket r \ n &= \mathcal{C} \llbracket Dm \rrbracket r \ n; \dots \mathcal{C} \llbracket D1 \rrbracket r \ (n+m-1); \text{STORE } m \ \xi_1
\end{aligned}$$

¹⁷This is different to the convention made in [Bur88b, LB89], where the evaluator field is always set to ξ_{NO} , and only changed when a task has been created to evaluate the expression.