Improved Integration of Multithreading into the STGM

Matthias Horn

B 96-07
July, 1996

Abstract

A variant of the Spineless Tagless G-Machine (STGM) which contains explicit support for multithreading is introduced in [1]. The main design decisions are the separation of demand for evaluation from case selection and the introduction of an abstract notion of thread boundaries and thread synchronisation. This report proposes an alternative solution which does not separate demand from selection. Instead, case selections are extended by additional alternatives which handle the appearance of long latency operations. The overhead, necessary to control multithreading, is reduced and sequentially evaluated parts of a program are more efficient.

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

1 Introduction

In [1] a variant of the Spineless Tagless G-Machine (STGM) containing explicit support for multithreading is introduced. The main design decisions are the separation of demand for evaluation from case selection and the introduction of an abstract notion of thread boundaries and thread synchronisation. The new language element *letpar* defines value bindings which can be evaluated independently. All *case* expressions with only one alternative[1] are substituted by *letpar* expressions. Nested *letpar* expressions with only one binding are joined to expressions with multiple bindings as far as possible. While executing a program a *letpar* expression creates a synchronisation frame containing a counter for the number of not yet evaluated bindings and space for the result of each binding. If the evaluation of a value binding is blocked by a long latency operation, another value binding can be evaluated first. The results for completely evaluated bindings are stored in the synchronisation frame. As soon as all bindings of a particular *letpar* are evaluated, the body of this *letpar* can be evaluated. In the case of sequential evaluation, the maintenance of synchronisation frames is additional work compared with the original STGM.

In order to achieve high speedups for parallel programs, it is desirable that each processor does a lot of uninterrupted work between long latency operations. These sequential parts of the program execution should be computed as fast as in the original STGM.

This report proposes an alternative integration of multithreading into the STGM, which does not separate demand from selection. Instead, case selections are extended by additional alternatives which handle the occurrence of long latency operations. On the machine proposed here, sequential execution is almost as fast as on the original STGM. Only two additional pointers into the execution stack have to be maintained. Synchronisation frames are replaced by extensions of continuation frames, but these extended continuation frames have to be maintained only when long latency operations have occurred.

2 Abstract Machine Language

2.1 Extended Language

The abstract machine used here contains five new closure types, a new constructor *Delay* and *letrem* expressions.

The constructor *Delay* is considered to be a member of each structured type. It is returned by a computation if this computation is delayed by a long latency operation.

Similar to [1] *letrem* expressions are used to define bindings which can be transmitted to other processing elements. It defines tasks. Bindings defined in *letrem* must not have any arguments and are marked as updateable. A dynamic load balancing algorithm using lazy task creation [3] chooses at runtime a processing element which evaluates the closures created by the *letrem* to weak head normal form.

The five new closure types behave as follows:

1. A closure containing <u>*fetch pe adr*</u> represents a closure located on processing element *pe* at address *adr*. If a *fetch* closure is entered, a *Fetch*

[1] Since strictness information is transformed in single-alternative-cases, this case is very common.

message is sent to the other processing element. This message initiates the evaluation of the demanded closure. The computed weak head normal form will be sent back. In order to prevent a processing element from fetching a closure twice, *fetch* closures are changed into *delayR* closures after the message is sent.

2. A closure containing *delayR w* represents a delayed evaluation which has already been started on another processing element. The list *w* contains addresses of all evaluations which depend on this value. If a *delayR* closure is entered, a *Delay* is returned to the topmost continuation frame.
3. A closure containing *delay n a s f w* represents an evaluation which was delayed because *n* other necessary values are delayed by long latency operations. *s* and *f* are stack portions which will be pushed onto the argument and frame stacks in order to continue the evaluation. The evaluation is continued by entering closure *a*. The list *w* contains addresses of all evaluations which depend on this value.
4. A closure containing *reactivate a s f* represents an evaluation which was delayed, but can now be continued. *a*, *s* and *f* have the same meaning as in the *delay* closure.
5. A closure containing *start e s f* represents the value of expression *e*. If a *start* closure is entered it removes itself from the task queue, pushes *s* onto the argument stack, *f* onto the frame stack and evaluates *e*.

2.2 Definitions

The evaluation of an STG-expression can often be started, although not all free variables of this expression are bound to proper values. Nested case expressions are a good example for this situation:

```
case ea of
  va -> case eb of
          vb -> e
```

If the variable `va` is not used in `eb`, `eb` can be evaluated even if the evaluation of `ea` is delayed by a long latency operation. The evaluation need not be stopped until the delayed value is used in `e`. Delayed evaluations are represented by *delay* and *delayR* closures. Since unboxed values cannot be represented by closures, they are handled in a special way.

The property that the evaluation of an expression must be stopped if an unboxed variable is not yet bound properly, leads to the definition of immediately necessary variables. The set of immediately necessary variables contains all unboxed variables which must be bound to their real values before the evaluation of an expression can be started. If any immediately necessary variable is delayed by a long latency operation, the evaluation of the expression has to be delayed as well.

The set of immediately necessary variables $IN_E(e)$ of STG expression e is defined inductively:

(1) if $e = v \wedge v$ unboxed, then $IN_E(e) = \{v\}$;
(2) if e = case e' of ... , then $IN_E(e) = IN_E(e')$;
(3) if $e = f\ e_1 \ldots e_n$, then $IN_E(e) = IN_E(f) \cup (\cup IN_B(e_i))$;

(4) if e = let(rec/par) $B_1,..,B_n$ in e', then $IN_E(e) = (\cup IN_B(B_i)) \cup IN_E(e')$
where $IN_B(vs \backslash\pi\ xs \rightarrow e) =$
$\{ v \mid v \in vs \wedge v \text{ unboxed} \}$

In the remaining parts of this report an abbreviated notation for closures with no arguments is used:

$\langle \underline{f\ a\ b\ c\ ...} \rangle$

is an abbreviation for

$(\{va,vb,vc,...\} \backslash n\ \{\} \rightarrow f\ va\ vb\ vc\ ...)\ \{a,b,c,...\}$

where va, vb, vc,... are fresh variables.

2.3 Program Transformation

The semantics of STG language defines only one situation, where the evaluation of an expression is started. Only expressions whose results are used to choose an alternative in a case expression are evaluated in STG. This leads to the idea that long latency operations can be handled in case expressions as well. Since only built-in functions deliver unboxed values, case expressions over unboxed types cannot be delayed. They need not be transformed. All other case expressions which contain only one alternative are extended by an additional *Delay* alternative, which is entered if the value computed in the case is delayed.

If a case expression has only one default alternative, the evaluation can simply be continued:

```
case ec of                        case ec of
  default -> e;     ---->           default -> e;
                                    Delay d -> e;
```

If a case expression has only one default alternative which uses the result, the evaluation can be continued:

```
case ec of                        case ec of
  v -> e;           ---->           v -> e;
                                    Delay v -> e;
```

Since case expressions over unboxed types are not transformed, `v` cannot be a member of IN(`e`). The variable v is bound to a delay closure which delays the evaluation of e if it is entered. Since other immediately necessary variables could be delayed, they have to be checked prior to the evaluation of `e`.

If a case expression has only one constructor alternative and the unboxed components are not immediately necessary, the evaluation can be continued:

```
case ec of                        case ec of
  C v1...vn -> e;   ---->          C v1...vn -> e;
                                   Delay d v1...vn -> e;
```

When the delay alternative is entered, the v_i are bound to special delay closures which contain a select for the i-th component. Since all strictness and unboxing in STG is expressed using case expressions with only one alternative, this case in particular appears very often.

All other case expressions, especially those with more than one alternative, are not extended, so they propagate delays to the surrounding case expression.

3 Operational Semantics

The abstract machine used here, consists of an arbitrary but fixed number of identical processing elements. The abstract state has seven components for each processing element:

1. the *code (C)*, which takes one of the forms *Eval, Enter, ReturnCons, ReturnInt*, *ReturnDelay, StartTask* or *Wait*;
2. the *argument stack (A)*, which contains addresses of closures and values used as arguments;
3. the *frame stack (F)* which contains continuation, update and remote update frames;
   Continuation and update frames are similar to the STGM. A remote update frame contains the address of a closure on another processing element, which needs to be overwritten with a result.
4. the *task queue (T)*, which contains executable tasks;
   A task is represented by a closure address. It is activated by entering this closure to which the address points. An executable task is appended to the front end, while tasks sent to other processing elements are taken from the back end. This hopefully leads to larger exported tasks.
5. the *heap (H)*, which maps addresses to closures;
6. the *global environment (G)*, which maps globally defined symbols to heap addresses of top level closures;
7. the *message queue (M)*, which contains messages received by the processing element;
   The messages *QueryTask* and *PutTask* are used to provide work for idle processing elements. *Fetch* and *Return* are used to transfer closures needed for a computation between processing elements.

Each transition rule of the operational semantics contains only state components which are used or modified in this rule. Each component mentioned in a rule is indexed by its processing element id. One rule affects at most two processing elements.

A parallel transition step is defined as a set of applications of rules which contains at least one rule and in which no processing element is affected by more than one rule.

3.1 The initial state

The initial state of the machine described here is almost identical to the state in the original STGM. The task and message queues are initialised as empty. With the exception of code all processing elements are initialised to the same state. Processing element 0 starts the evaluation of the main function.

C_0	A_0	F_0	T_0	H_0	G_0	M_0
$\sigma =$	$\{ g_i \rightarrow a_i \}$					
$h_{init} =$	$\{ a_i \rightarrow (vs_i \backslash\pi_i\ xs_i \rightarrow e_i)\ (\sigma\ vs_i) \}$					
Eval(main{ }){ }	{ }	{ }	{ }	h_{init}	σ	{ }

Rule 1: Initial state for processing element 0

All other processing elements start by looking for work.

C_p	A_p	F_p	T_p	H_p	G_p	M_p
$p \neq 0$						
$\sigma =$ $\{ g_i \rightarrow a_i \}$						
$h_{init} =$ $\{ a_i \rightarrow (vs_i \backslash\pi_i\ xs_i \rightarrow e_i)\ (\sigma\ vs_i) \}$						
StartTask	{ }	{ }	{ }	h_{init}	σ	{ }

Rule 2: Initial state for other processing elements

3.2 Activate Tasks

The task queue holds addresses of closures. Tasks taken from the task queue are activated by entering the closure.

C_p	A_p	F_p	T_p
StartTask	{ }	{ }	a : t
Enter a	{ }	{ }	a : t

Rule 3: Activate a task from the task queue

If the code form *StartTask* is executed while the task queue is empty, a *QueryTask* message is sent to another processing element.

C_p	A_p	F_p	T_p	T_q
StartTask	{ }	{ }	{ }	m
q is computed by a load balancing algorithm				
Wait	{ }	{ }	{ }	m ++ (QueryTask p)

Rule 4: Query for work

3.3 Application

The rule for function application is unchanged. A function application is implemented by pushing the arguments onto the argument stack and entering the function closure.

C_p	A_p	G_p
Eval (f xs) ρ	as	σ
val[2] $\rho\ \sigma$ f = a		
Enter a	(val $\rho\ \sigma$ xs) ++ as	σ

Rule 5: Application

If a non-updateable lambda form is entered and enough arguments are supplied on the stack, the body of this lambda form can be evaluated. Free variables found in the closure and arguments found on the stack are copied into the local environment and the evaluation continues with the body. To ensure that all messages from other processing elements are processed first, the message queue has to be empty before entering a closure.

C_p	A_p	H_p	M_p
Enter a	ws_a++as	h	{ }
$h(a) = (vs \backslash n\ xs \rightarrow e)\ ws_f$			
$length(ws_a) = length(xs)$			
$\rho = [vs \rightarrow ws_f , xs \rightarrow ws_a]$			
Eval e ρ	as	h	{ }

Rule 6: Enter saturated non-updateable closure

The case of partial application is detected if the number of arguments on the argument stack is less than the number needed by the closure. In this case the topmost frame is

[2] Defined in [4] on page 34

always an update or remote update frame. The local or remote closure to which this frame points is updated by a newly created closure representing the partial application.

C_p	A_p	F_p	H_p	M_p
Enter a	ws_a	$(a_u,as):fs$	h	{ }
$h(a) = (vs \backslash n\ xs \to e)\ ws_f$ $length(ws_a) < length(xs)$ $length(xs_1) = length(ws_a)$ $h_u = h[a_u \to (f:xs_1 \backslash n \{\} \to f\ xs_1)\ a:ws_a]$				
Enter a	ws_a ++ as	fs	h_u	{ }

Rule 7: Enter partially applied closure and process update frame

Since a remote update frame is always the first element on the stack, in this case the processing element has to look for work next.

C_p	A_p	F_p	M_p	M_q
Enter a	ws_a	$(q,a_u):\{\}$	{ }	m
$h(a) = (vs \backslash n\ xs \to e)\ ws_f$ $length(ws_a) < length(xs)$ $length(xs_1) = length(ws_a)$ $a' = (f:xs_1 \backslash n \{\} \to f\ xs_1)\ a:ws_a$				
StartTask	{ }	{ }	{ }	m ++ (Return a_u p a')

Rule 8: Enter partially applied closure and process remote update frame

If an updateable lambda form is entered, prior to the evaluation an update frame has to be pushed onto the stack. This update frame contains the address of the closure to be updated and the current argument stack. Since updateable lambda forms must not have any arguments, they are always saturated.

C_p	A_p	F_p	M_p
Enter a	as	fs	{ }
$h(a) = (vs \backslash u \{\} \to e)\ ws_f$ $\rho = [vs \to ws_f]$			
Eval e ρ	{ }	(a,as):fs	{ }

Rule 9: Enter updateable closure

3.4 Let Expressions

A `let` expression builds closures in the heap and evaluates the body expression using the newly created closures.

C_p	H_p
Eval (let x_i=... in e) ρ	h
$\rho' = \rho\ [x_i \to a_i]$ $h' = h\ [a_i \to (vs_i \backslash \pi_i\ xs_i \to e_i)\ (\rho_{rhs}\ vs_i)]$ $\rho_{rhs} = \rho$	
Eval e ρ'	h'

Rule 10: Evaluate let(rec)

The rule for *letrec* is almost identical, except that ρ_{rhs} is defined to be ρ' instead of ρ. *letrem* builds closures in a similar way, but created closures are appended to the top of the task queue.

C_p	A_p	T_p	H_p
Eval (letrem x_i=... in e) ρ	as	t	h
ρ' = p[$x_i \rightarrow a_i$]			
h'= h[$a_i \rightarrow$ (vs_i \u { } → (start e_i { } { })) (ρ_{rhs} vs_i)]			
ρ_{rhs}= ρ			
Eval e ρ'	as	a_i : t	h'

Rule 11: Evaluate letrem

Since shared closures can be activated by *Enter* or *StartTask*, it is necessary to wrap the expression into the built-in function *start* which removes the closure from the task queue when it is activated.

C_p	A_p	F_p	T_p	H_p	M_p
Enter a	as	fs	t_1 ++ [a] ++ t_2	h	{ }
h(a) = (vs \u { } → (start e s f)) vsf					
ρ = [vs → ws_f , xs → ws_a]					
Eval e ρ	s	f:(a,as):fs	t_1 ++ t_2	h	{ }

Rule 12: Enter closure registered in the task queue

3.5 Evaluating case

Similar to the original STGM the evaluation of a case expression pushes a continuation frame onto the stack. It stores the current local environment ρ, the alternatives which continue the evaluation and the argument stack. Continuation frames for case expressions, which are outermost in their binding, contain a list of currently delayed unboxed variables together with addresses of delay closures which caused the delay. This list is initialised to empty.

C_p	A_p	F_p	H_p
Eval (case e of alts) ρ	as	fs	h
frame = (alts,as,ρ,{ }), if this case is the outermost case expression of a binding			
= (alts,as,ρ), otherwise			
Eval e ρ	{ }	frame:fs	h

Rule 13: Evaluate case

3.6 Delayed evaluation

If a closure which refers to another processing element is entered, a long latency action is triggered. In this case a *Fetch* message is sent to this processing element and the local element tries to execute another part of the program. In order to prevent the current processing element from sending a second evaluation message, the fetch closure is changed into a *delayR* closure. Finally a *Delay* is returned.

C_p	H_p	M_q
Enter a	h	m
h(a) = ⟨fetch q ra⟩		
h' = h[a → ⟨delayR { }⟩]		
ReturnDelay a	h'	m ++ (Fetch p a ra)

Rule 14: Enter remote closure

If a delayed value is entered, the topmost continuation frame which contains a *Delay* alternative is entered. The current evaluation environment containing

- the values on the argument and frame stacks
- the alternatives and
- the address of the closure which caused the delay

is stored in a new delay closure. The environment of the case expression is restored and the expression of the delay alternative is evaluated. The newly created delay closure is registered as a dependent closure in the closure which caused the delay. Prior to the evaluation of the body all unboxed variables immediately necessary for this evaluation have to be checked as to whether they are delayed. Two cases for *Delay* alternatives are distinguished.

In the first case the delayed value is used as a whole.

C_p	A_p	F_p	H_p
ReturnDelay a	as	f ++ $cf{:}f_1 \ldots f_k{:}(\_,\_,d_u){:}fs$	h
$cf = (\ldots., \text{Delay } d \rightarrow e, \ldots.., as', \rho)$			
$h' = h[d' \rightarrow \langle \underline{\text{delay } n\ a\ as\ (f \text{ ++ } \{cf\})\ \{\}} \rangle\ ]$			
$\quad [a \rightarrow \text{register\_dep\_delay}^3\ d'\ h(a)]$			
if $IN(e) \cap (\text{map fst } d_u) = \varnothing$			
$\quad n = 1$; code = Eval e $\rho[d \rightarrow d']$; $h'' = h'$			
else			
$\quad n = 1 + \#(IN(e) \cap (\text{map fst } d_u))$; code = ReturnDelay d'			
$\quad h'' = h'\ [d_i \rightarrow \text{register\_dep\_delay}^3\ d'\ h(d_i)]$		for all $(v_i,d_i) \in d_u$	
		where $v_i \in (IN(e) \cap (\text{map fst } d_u))$	
code	as'	$f_1 \ldots f_k{:}(\_,\_,d_u){:}fs$	h''

Rule 15: Return delay to continuation frame (general case)

In the second case the delayed value is split into its components. New *delay* closures for delayed components are created. In order to select the correct component a continuation frame is added to the saved frame stack portion. Delayed unboxed variables are recorded in the continuation frame of the outermost case expression of the current closure. The program transformation described in 2.3 ensures that there is no $v_i \in IN(e)$, but other immediately necessary variables can be delayed at this point. They have to be checked in a similar way as in Rule 15.

C_p	A_p	F_p	H_i
ReturnDelay a	as	f ++ $cf{:}f_1 \ldots f_k{:}(\_,\_,d_u){:}fs$	h
$cf = (\ldots, \text{Delay } d\ v_1 \ldots v_n \rightarrow e, as', \rho)$			
$u = \{ (v_i,a) \mid v_i \text{ unboxed } \}$			
if $v_i \in u$			
$\quad w_i = 0,$			
$\quad h_i = h_{i-1}$			
else			
$\quad w_i = d_i$			
$\quad h_i = h_{i-1}\ [d_i \rightarrow \langle \underline{\text{delay } n\ a\ as\ ((\_\ v_1 \ldots v_k \rightarrow v_i), \{\}){:}f \text{ ++ } \{cf\})\ \{\}} \rangle\ ]$			
$\qquad [a \rightarrow \text{register\_dep\_delay}^4\ d_i\ h(a)]$			
if $IN(e) \cap (\text{map fst } d_u) = \varnothing$			
$\quad n = 1$			
$\quad$ code = Eval e $\rho[v_i \rightarrow w_i]$			
$\quad d_u' = d_u$ ++ u			

[3] defined on page 11

```
        h' = h_n
    else
        n = 1 + #(IN(e) ∩ (map fst d_u))
        d_u' = d_u
        code = ReturnDelay d'
            where
                h'= h_n [d' → ⟨delay n a as (f ++ {cf}) {}⟩ ]
                        [a  → register_dep_delay[4] d' h(a)]
                        [d_k  → register_dep_delay[4] d' h(d_k)]  for all (v_k,d_k) ∈ d_u
                                                                   where v_k ∈ (IN(e) ∩ (map fst d_u))
```

code	as'	$f_1 \ldots f_k$:(_,_,d_u'):fs	h_n

Rule 16: Return delay to continuation frame (with selection of components)

If no continuation frame which contains a *Delay* alternative is on the stack, the whole task has to be suspended until delayed values arrive. Since the frame stack contains at least one update or remote update frame both stacks are stored in a delay closure and the processing element next looks for work.

C_p	A_p	F	H_p
ReturnDelay a	as	fs	h
h'= h [d → ⟨delay 1 a as fs {}⟩] [a → register_dep_delay d h(a)]			
StartTask	{}	{}	h'

Rule 17: Return delay to remote update frame

A new dependent closure is registered by adding its address to the closure on which it depends:

```
register_dep_delay d ⟨delay n a s f w⟩ = ⟨delay n a s f d:w⟩
register_dep_delay d ⟨delayR w⟩ = ⟨delayR d:w⟩
```

If a *delay* or *delayR* closure is entered, a *Delay* is returned to the topmost continuation frame.

C_i	H_i
Enter d	h
h(d)= ⟨delay n a s w⟩	
or h(d)= ⟨delayR w⟩	
ReturnDelay d	h

Rule 18: Enter delay closure

Delay closures are overwritten with *reactivate* or *start* as soon as all delayed values on which they depend have arrived (see Rule 36 and Rule 37). In order to complete the delayed computation the environment saved while creating the *delay* closure is restored and the closure which caused the delay is entered.

C_p	A_p	F_p
Enter a	as	fs
h(a) = ⟨reactivate d s f⟩		
Enter d	s ++ as	f ++ fs

Rule 19: Enter reactivate closure

[4] defined on page 11

3.7 Evaluation of Constructors and Basic Values

The evaluation of constructors and basic values is similar to the original STGM.

C_p
Eval k ρ
ReturnInt k

Rule 20: Evaluate integer constant

C_p	G_p
Eval (c xs) ρ	σ
ws= val ρ σ xs	
ReturnCon c ws	σ

Rule 21: Evaluate constructor expression

C_p
Eval v {} ρ
ρ(v) = k
ReturnInt k

Rule 22: Evaluate variable bound to an integer

3.8 Returning Constructors

If the evaluation has ended up with a constructor, the result is used to select an alternative in a surrounding case expression or to return an answer to another processing element. Possibly some closures have to be updated with the computed weak head normal form before entering an alternative or sending a result. If any update has to be carried out, an update frame is on top of the frame stack. The closure to which the update frame points, is replaced by a closure containing the computed result. After removing the update frame from the frame stack, the constructor is returned to the next frame on the stack.

C_p	A_p	F_p	H_p
ReturnCon c ws	{}	$(a_u$,as):fs	h
length(vs) = length(ws)			
h_u= h[a_u → (vs \n {} → c vs) ws]			
ReturnCon c ws	as	fs	h_u

Rule 23: Return constructor to update frame

When all update frames are removed from the frame stack, the *ReturnCon* finds either a continuation or remote update frame on top of the frame stack.

Continuation frames contain all alternatives and the environment which were pushed while evaluating the corresponding case expression. *ReturnCon* has to choose the right alternative, remove the continuation frame from the frame stack and start evaluating the alternative. Prior to the evaluation of any alternative, a check for delayed unboxed values which are immediately necessary for the evaluation of this alternative is performed. Unboxed values delayed at this moment are found in the continuation frame of the outermost case expression in the current closure. If any unboxed value is delayed at this moment, a delay closure has to be built.

C_p	A_p	F_p
ReturnCon c ws	{}	cf:f_1...f_k:(_,_,d_u)
cf= (...c vs → e...,as,ρ)		
code = Eval e ρ[vs → ws],	if IN(e) ∩ (map fst d_u) = ∅	
= ReturnDelay d,	otherwise	
where		
n = #(IN(e) ∩ (map fst d_u))		
h' = h [a → (vs \n {} → c vs) ws]		
[d → ⟨delay n a {} {cf} {}⟩]		
[v_i → register_dep_delay[5] d_i h(v_i)]	for all (v_i,d_i) ∈ d_u	
	where v_i ∈ (IN(e) ∩ (map fst d_u))	
code	as	f_1...f_k:(_,_,d_u)

Rule 24: Return constructor to continuation frame (constructor alternative)

If no alternative matches, the default alternative is chosen. In this case it is not necessary to bind variables in the local environment.

C_p	A_p	F_p
ReturnCon c ws	{}	cf:f_1...f_k:(_,_, d_u)
cf= (...default → e,as,ρ)		
code = Eval e ρ,	if IN(e) ∩ (map fst d_u) = ∅	
= ReturnDelay d,	otherwise	
where		
similar to Rule 24		
code	as	f_1...f_k:(_,_, d_u)

Rule 25: Return constructor to continuation frame (default alternative)

If the default alternative is a variable alternative, a new closure has to be built in the heap. The variable is bound to this new closure.

C_p	A_p	F_p	H_p
ReturnCon c ws	{}	cf:f_1...f_k:(_,_,d_u)	h
cf= (...v → e,as,ρ)			
code = Eval e ρ[v → a'],	if IN(e) ∩ (map fst d_u) = ∅		
where			
h'= h[a' → (vs \n {} → c vs) ws]			
= ReturnDelay d,	otherwise		
where			
similar to the Rule 24			
code	as	f_1...f_k:(_,_, d_u)	h'

Rule 26: Return constructor to continuation frame (variable alternative)

A remote update frame contains the remote address (processing element id and address) of the closure which has to be overwritten with the weak head normal form just computed. A *Return* Message is sent to this processing element. Since a remote update frame is always the first element on the stack, the processing element has to look for work next.

C_p	F_p	M_q
ReturnCon c ws	(q,ra):{}	m
StartTask	{}	m ++ (Return ra p (vs \n {} → c vs) ws)

Rule 27: Return constructor to remote update frame

[5] defined on page 11

3.9 Termination

The machine terminates when either an *Enter* or *ReturnCon* finds an empty stack after processing an update frame.

3.10 Returning Basic Values

Basic values are returned in a similar way. The only difference is that neither update nor remote update frames occur. If delayed immediately necessary values occur, a closure containing a boxed integer and a delay closure are built in the heap.

C_p	A_p	F_p
ReturnInt z	{ }	$cf{:}a_1 \ldots a_k{:}(\_,\_,d_u)$

$cf = (\ldots z \rightarrow e \ldots, as, \rho)$

code = Eval e ρ, if $IN(e) \cap (\text{map fst } d_u) = \varnothing$
= ReturnDelay d, otherwise
where
$n = \#(IN(e) \cap (\text{map fst } d_u))$
$h' = h\,[a \rightarrow \langle \underline{I\#\ z} \rangle\,]$
$[d \rightarrow \langle \underline{\text{delay } 1\ a\ \{\}\ \{(I\#\ z \rightarrow e,\rho)\}\ \{\}} \rangle\,]$
$[v_i \rightarrow \text{register\_dep\_delay}^5\ d_i\ h(v_i)]$ for all $(v_i,d_i) \in d_u$
where $v_i \in (IN(e) \cap (\text{map fst } d_u))$

code	as	$a_1 \ldots a_k{:}(\_,\_,d_u)$

Rule 28: Return integer to continuation frame (constant alternative)

C_p	A_p	F_p
ReturnInt z	{ }	$cf{:}f_1 \ldots f_k{:}(\_,\_,d_u){:}fs$

$cf = (\ldots \text{default} \rightarrow e, as, \rho)$

code = Eval e ρ, if $IN(e) \cap (\text{map fst } d_u) = \varnothing$
= ReturnDelay d, otherwise
where
similar to Rule 28

code	as	$f_1 \ldots f_k{:}(\_,\_,d_u){:}fs$

Rule 29: Return integer to continuation frame (default alternative)

C_p	A_p	F_p	H_p
ReturnInt z	{ }	$cf{:}f_1 \ldots f_k{:}(\_,\_,d_u){:}fs$	h

$cf = (\ldots v \rightarrow e, \rho)$

code = Eval e $\rho[v \rightarrow z]$, if $IN(e) \cap (\text{map fst } d_u) = \varnothing$
= ReturnDelay d, otherwise
where
similar to Rule 28

code	as	$f_1 \ldots f_k{:}(\_,\_,d_u){:}fs$	h'

Rule 30: Return integer to continuation frame (variable alternative)

3.11 Evaluation of Basic Operations

Operations on unboxed values can be performed without pushing a continuation frame, since all arguments are already evaluated.

C_p
Eval (case $(x_1 \oplus x_2)$ of ...k $\rightarrow$ e... ρ
$k = x_1 \oplus x_2$ code = Eval e ρ, if $IN(e) \cap (\text{map fst } d_u) = \emptyset$ = ReturnDelay d, otherwise where $n = \#(IN(e) \cap (\text{map fst } d_u))$ $h' = h\,[a \rightarrow \langle \underline{I\#\ x_1 \oplus x_2} \rangle]$ $[d \rightarrow \langle \underline{\text{delay 1 a } \{\}\ \{(I\#\ k \rightarrow e,\rho)\}\ \{\}} \rangle]$ $[v_i \rightarrow$ register_dep_delay[5] $d_i\ h(v_i)]$ for all $(v_i,d_i) \in d_u$ where $v_i \in (IN(e) \cap (\text{map fst } d_u))$
code

Rule 31: Return result of basic operation to continuation frame (constant alternative)

C_p
Eval (case $(x_1 \oplus x_2)$ of ...default $\rightarrow$ e... ρ
code = Eval e ρ, if $IN(e) \cap (\text{map fst } d_u) = \emptyset$ = ReturnDelay d, otherwise where similar to the rule above
code

Rule 32: Return result of basic operation to continuation frame (default alternative)

C_p
Eval (case $(x_1 \oplus x_2)$ of ...x $\rightarrow$ e... ρ
code = Eval e $\rho[x \rightarrow (x_1 \oplus x_2)]$, if $IN(e) \cap (\text{map fst } d_u) = \emptyset$ = ReturnDelay d, otherwise where similar to the rule above
code

Rule 33: Return result of basic operation to continuation frame (variable alternative)

3.12 Processing Messages

A processing element has to process messages periodically. For simplicity the message queue is always examined before an *Enter* is performed or if the code form *Wait* is executed.

If a *Fetch* message arrives, the closure which has been demanded is appended to the front end of the task queue, so it will be evaluated the next time the task queue is looked up[6].

C_p	T_p	H_p	M_p
Enter a	t	h	(Fetch q ra a') ++ m
h(a') = (vs \u { } → e) ws h'= h [a' → (s:f:vs \u { } → start e s f) { }:(q,ra):ws]			
Enter a	a' ++ t	h'	m

Rule 34: Process *Fetch* message before entering a closure

[6] Demand messages for closures already in weak head normal form could be optimised by returning the value immediately.

If the code form *Wait* has been executed, the demand which has arrived can be executed immediately.

C_p	A_p	T_p	H_p	M_p
Wait	{ }	{ }	h	(Fetch q ra a') ++ m
h(a') = (vs \u { } → e) ws				
h'= h [a' → (s:f:vs \u { } → start e s f) { }:(q,ra):ws]				
StartTask	{ }	{ a'}	h'	m

Rule 35: Process *Fetch* message while waiting

A processing element which receives a result overwrites the corresponding fetch closure with the weak head normal form received.

C_p	T_p	H_p	M_p
Enter a'	t	h	(Return d j (c ws)) ++ m
c = (ws \n as → e)			
h(d) = ⟨delayR w⟩			
ws' = map ins_fetch ws			
where			
ins_fetch v = ⟨fetch j v⟩,			if v boxed
ins_fetch v= v,			otherwise
h' = h[d → c ws']			
(t',h'') = trans_reactivate h' w			
Enter a'	t' ++ t	h''	m

Rule 36: Process *Return* Message before entering a closure

Boxed constructor arguments which point into another processing element are replaced by *fetch* closures. In order to avoid multiple references to another processing element, this should be optimised in future developments.

C_p	T_p	H_p	M_p
Wait	{ }	h	(Return d j ((ws \u as → e) ws) ++ m
c = (ws \n as → e)			
h(d) = ⟨delayR w⟩			
ws' = map ins_fetch ws			
where			
ins_fetch v = ⟨fetch j v⟩,			if v boxed
ins_fetch v= v,			otherwise
h' = h[d → c ws']			
(t,h'') = trans_reactivate h' w			
StartTask	t	h''	m

Rule 37: Process *Return* Message while waiting

When a delayed value has arrived, all dependent *delay* closures are transformed into *reactivate* or *start* closures. This operation is propagated over the whole directed acyclic graph of dependent closures. A delay closure which does not have dependent closures[7] is appended to the top of the task queue and replaced by *start*. All other delay closures are replaced by *reactivate*.

[7] Delay closures with no dependent closures always have an update or remote update frame at the bottom of their saved stack portion.

The function trans_reactivate describes this operation:

```
trans_reactivate h { } = ({ },h)
trans_reactivate h d:ds = (t',h'')
            where
                  h(d) = ⟨delay n a s f w⟩
                  (t,h') = trans_reactivate h w++ds
                  if n = 1
                    if w = { }
                       t' = a:t
                       h'' = h'[d → ⟨start a s f⟩ ]
                     else
                       t'= t
                       h'' = h'[d → ⟨reactivate a s f⟩ ]
                  else
                    h'' = h'[d → ⟨delay n-1 a s f w⟩ ]
```

If a *QueryTask* is received by a processing element which has executable tasks available, a task taken from the backend of the task queue is sent to the other processing element.

C_p	T_p	M_p	M_q
Enter a	t ++ ({ },{ },c_q)	{(QueryTask q)} ++ m_p	m_q
Enter a	t	m_p	m_q ++ (PutTask p h(c_q))

Rule 38: Process *QueryTask* message before entering a closure

If the processing element itself is waiting for work, the *QueryTask* message is propagated to another element.

C_p	T_p	M_p	M_q
Wait	{ }	(QueryTask j) ++ m_p	m_q
q is computed by a load balancing algorithm			
Wait	{ }	m_p	m_q ++ (QueryTask j)

Rule 39: Process *QueryTask* message while waiting

Tasks received by *PutTask* are registered in the task queue.

C_p	T_p	H_p	M_p
Enter a	t	h	(PutTask j (s,(ws \u { } → e))) ++ m
ws' = map ins_fetch ws where ins_fetch v = ⟨fetch j v⟩, if v boxed ins_fetch v= v, otherwise h' = h[a' →(ws \n { } → e) ws']			
Enter a	(s,a'):t	h'	m

Rule 40: Process *PutTask* message before entering a closure

If the processing element is waiting when the *PutTask* message arrives, it starts the task which has arrived.

C_p	T_p	H_p	M_p
Wait	{ }	h	(PutTask j (s,(ws \u { } → e))) ++ m
ws' = map ins_fetch ws where ins_fetch v = ⟨fetch j v⟩, if v boxed ins_fetch v= v, otherwise h' = h[a' →(ws \n { } → e) ws']			
StartTask	{ (s,a') }	h'	m

Rule 41: Process *PutTask* message while waiting

4 Conclusion and Further Work

In the Spineless Tagless G-machine, evaluation of an expression is started only if the value of this expression is used in a case expression. A long latency operation can be handled in an additional (*Delay*) alternative of the case expression, which caused the evaluation of the delayed expression. If such a *Delay* alternative is entered, an alternative evaluation (another thread), which does not need the delayed value, can be carried out first. Since sequentially computed expressions never use the *Delay* alternatives, evaluations without long latency operations can be almost as fast as in the original STGM. This report introduces an abstract machine which uses *Delay* alternatives for the integration of multithreading into the STGM.

The abstract semantics presented here can be implemented efficiently. Only two additional pointers into the frame stack have to be maintained. One pointer refers to the topmost continuation frame which contains a *Delay* alternative. The other refers to the topmost extended continuation frame which contains information about delayed unboxed values. Extended continuation frames have to be maintained only if a long latency operation has occurred, but they have to be examined on each entry of an alternative. This seems to be expensive, but the information which unboxed values are delayed can be represented by a word which contains a bit for each possibly delayed value. Using this representation, the check for delayed unboxed values is only a bitwise AND and a comparison with zero. Furthermore, if no long latency operation has occurred, this effort can be avoided by the code duplication trick used in the original STGM to optimise updates.

In the future the ideas developed in this report will be combined with efficient runtime representation of evaluators proposed in [2]. Evaluators and evaluation transformers will be used to identify larger tasks for parallel evaluation.

5 Rule Index

6 References

[1] Manuel M.T.Chakravarty; *Integrating Multithreading into the Spineless Tagless G-machine*; In: Proceedings of the 1995 Glasgow Workshop on Functional Programming, Springer Verlag, 1996

[2] Matthias Horn; *Improving Parallel Implementations of Lazy Functional Languages Using Evaluation Transformers*; In: Proceedings of the 5th International Workshop on Functional and Logic Programming, Rauischholzhausen 1996

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

[4] Simon L. Peyton Jones; *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5*; In: Journal of Functional Programming 2(2) (April 1992), pp 127-202