The ν–STG machine : a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture Draft version: comments welcomed

Suntae Hwang and David Rushall

Department of Computer Science,
Victoria University of Manchester,
Oxford Road, Manchester M13 9PL, UK.
hwangs@uk.ac.man.cs drushall@uk.ac.man.cs

Abstract. This paper describes the ν–STG machine, a parallelized Spineless Tagless Graph Reduction(STG) machine in a distributed memory architecture. In the ν–STG machine, a stack for each task is distributed by allocating a *context frame* for each *tail–call sequence* on the *heap*. Two sparking mechanisms, *BSpark* and *ISpark*, are supported to introduce parallelism, which are similar to those in the HDG machine. The *BSpark* cannot be ignored and should be synchronized with the parent task to prevent it being blocked and resumed too frequently, while *ISpark* may be ignored. Even though a variable is sparked by *BSpark* or *ISpark*, it may be evaluated *in–line* by the parent task. Creating a new task to evaluate a *boxed* variable according to the spark annotation is delayed until it is really needed. A message passing mechanism is used to distribute jobs and synchronize them in a machine. A *lazy task creation mechanism* is supported to exploit parallelism in *unboxed* arithmetic expression by *boxing up* on demand. A simple printing mechanism is added to print outputs clearly.

1 Introduction

In the STG machine, each *closure* including data values is represented uniformly, and scrutinized only by entering it[Peyton Jones]. Because there is no tag for any closure, code for each closure should be built at compile time and its label be placed in the entry point of an *info table*. In the *info table*, there are two more entry points for garbage collection, which are entered at a garbage collection phase.

To parallelize the STG machine, we define a few more closures and entry points for each closure. A closure may be entered via one of these entry points according to a context in the same way and at the same cost. In addition to the existing lambda form, we define the *Locked* node to synchronize multiple attempts to evaluate a closure and the *OutInd* node to point to a remote closure. Six more entry points are added to the *standard entry point*, to break a stack into small pieces and to spark, export, or print out a closure.

The following sections describe ν–STG machine, a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture. This paper contains all rules for the ν–STG machine, but explains them briefly. We are therefore assuming that the reader is already familiar with the STG machine. We would also like to point out that this represents work in progress, so real performance figures are not yet available.

2 The ν–STG language

The abstract machine code for the ν–STG machine, the ν–STG *language*, is very similar to the STG language[Peyton Jones]. The main differences are as follows:

2.1 Algebraic typed case

The STG machine allows arbitrary expressions as *case switches*. Whilst the ν–STG machine retain this for the primitive typed expressions, if the type of switch is algebraic, the expression must be a variable applied to no arguments. This results in all variables the expression needs being explicitly *free* in the closure in the heap, and so it is not necessary to transfer items between argument stacks during *EnterAlloc* states. This aspect will be discussed below in Section 5.

An arbitrary *algebraic case expression*,

```
case e of alts
```

can be easily translated into

```
let t = vs \r {} -> e in case t of alts
```

where the *vs* are the free variables for the expression, *e*. Although this translation requires an extra heap allocation, it may not be necessary in many cases, when the value scrutinized by a *case* expression is a simple variable.

2.2 casep expression

The ν–*STG* machine supports a *lazy task creation mechanism*, which allows *unboxed* expressions to be evaluated in *parallel*, if machine requires work. To represent this potential parallelism, the `casep` expression is used. This consists of two or more primitive typed expressions that may be evaluated in parallel and a list of variables which are eventually bound with the results of these expressions. It is left to the programmer, or the analytical passes of the compiler, to ensure that each of the *parallel* expressions represents a suitable amount of work that would be worth sparking a parallel task. The following example shows how a number of nested primitive *case* expressions may be transformed into a *casep* expression.

```
case e1# of
 x1# -> case e2# of
          x2# -> case e3# of
                   x3# -> e

   where e1#, e2# and e3# are of primitive type.
         e2# is not dependent on x1#, and
         e3# is not dependent on x1# or x2#

 ==>

casep {e1#, ! {vs2} e2#, ! {vs3} e3#} of
        {x1#, x2#, x3#} -> e

   where vs2 and vs3 are free variables
   for e2# and e3# respectively.
```

All expressions but the first one are annotated to indicate that which type of sparking mechanism any new task would be sparked by. If work is required, the continuation can be changed, and one of the expressions boxed up into a new task. This will be discussed later in Section 9.

2.3 Three flags for lambda form

In the STG machine, closures are marked as updatable (`\u`) and non–updatable (`\n`) explicitly. For parallel evaluation, it is convenient to have three classes for closures:

\n, *Nonreducible*	closure is in Weak Head Normal Form(WHNF) and so is not reducible and will never change.
\r, *Reducible*	closure is not in WHNF, but is known not to be shared. In this case an update may avoided if evaluation is carried out in-line.
\u, *Updatable*	closure should be updated when brought to WHNF, as it may be shared.

2.4 Annotation of parallelism

The variables in the argument list of the lambda form and the arguments of functions, built–in operators and constructors may be annotated by relevant sparking mechanism. For basic sparks the annotation "!!" is used, whilst ignorable spark are denoted with a single "!". The annotated variable must be a boxed type.

3 Operational semantics of the ν–*STG language*

The ν–*STG language* is the abstract machine code for the ν–*STG* machine. The operational semantics for the ν–*STG* language can be described by a state transition system. In distributed memory model architecture, each processing element has its own transition system with a local state. The processing elements communicate by sending and receiving messages via a global state which has only one component, the *network*. Some of the rules of the local transition system access the global state by sending messages. If a message arrives, the current evaluation is interrupted at the relevant time and a message handler is launched to process the arrived message. The semantics for the message handler can also be described by a state transition system.

The local state has seven components:

1. the *code, c*, which takes one of several forms, given below;
2. the *context heap, cs*, which contains *context frames*;
3. the *heap, h*, which contains *closures*;
4. the *task pool, tp*, which contains details of the status of current *tasks*;
5. the *work pool, wp*, which contains *jobs* that can be performed by an idle processor;
6. the *printing mechanism state, pr*, which contains a *print stack* and an *output stream*;
7. The *global environment,* σ, which gives the addresses of all closures defined at top level.

The ν–STG code state can take one of several forms. Firstly,there are forms that correspond directly to the four forms in the STG machine:

Eval $e\ \rho\ \nu$	Evaluate the expression e in environment ρ using the stacks of context ν. The expression e is an arbitrary complex ν–*STG language* expression.
Enter $a\ \nu$	Apply the closure at address a to the arguments on the arguments stack of context ν.
ReturnCon $c\ ws\ \nu$	Return the constructor c applied to values ws, to the continuation in context ν.
ReturnInt $k\ \nu$	Return the primitive value k to the continuation in context ν.

The next code state allows the creation of a new context frame for a sequence of tail–calls.

EnterAlloc $a\ \nu$	Enter the closure a, allocating a new context frame if necessary, in which case the old context will be pointed by the new context.

The third set of code states deal with parallel execution; creating *tasks* and finding *work* for idle processors.

EnterISpark $a\ ra_u$	Create a task to evaluate closure at a, if not in WHNF, and update the closures in ra_u(if not empty) when evaluation is complete.
EnterBSpark $a\ (t \vert ra_u)$	Create a task to evaluate closure at a, if not in WHNF, and notify task, t, or update the closures in ra_u when evaluation is complete.
GetTask	Attempt to obtain a task from the *work pool*. If none is available then request work from a neighbour and wait for a reply.
WaitTask	Similar to the *GetTask*, but wait for a reply to the work request which was made in the *GetTask* state.

Finally, there are the rules for the printing mechanism:

EnterPrint a	Cause the closure a to be evaluated if necessary, and then eventually written to the *output*.
PrintNext	Cause the next item on the *print stack* to be written to the *output*.
Stop	Halt the machine. (Evaluation is complete.)

A major change from the STG to ν–*STG* is in the form of the stacks. Whilst the STG has three stacks, the ν–STG has only two, distributed in several *context frames*. Some code states have a context field ν. This maps to a *context frame* in the *context heap* and has the following fields.

t	the task's name that owns this *context*.
ν_{old}	the name of the parent context frame.
as	the argument stack, which contains *values*.
rs	the combined continuation stack, which contains entries of several forms, given below.

A *value* is exactly the same as one in the STG machine, and takes one of the following forms:

Addr a	A heap address
Int k	A primitive value, such as integer, character, boolean, etc.

The *combined continuation stack* is a departure from the original STG machine. This takes the place of the STG's continuation, *rs* and update, *us* stacks. In the physical implementation of the STG machine, these stacks are combined with *update frames* being treated as special continuations. The ν–*STG* combined continuation stack may contain a few entries of the following forms:

$R(alts,\rho)$	*return–to continuation*, which contains case alternatives, *alts*, and an environment, ρ, and is similar to the *rs* stack in the STG.
$U(as,a)$	*update continuation*, which contains old argument stack, *as*, and address of the closure to update, *a*, and is similar to the *us* stack in the STG.
$U'(pid,a)$	*remote update continuation*, which contains a global address of the remote closure to update. The global address consists of a processor's name, *pid*, and address of closure, *a*, which is placed in the heap of *pid*.
$R'(es,xs,\rho',e,\rho)$	*parallel return–to continuation*, which is built when evaluating a *casep* expression and contains a list of expressions still requiring evaluation, *es*, a list of variables, *xs*, to which they will be bound, an environment ρ' containing things already evaluated in this *casep* expression, the body expression, *e*, that will be evaluated finally, and the environment, ρ, in which the list of expressions, *es*, are to be evaluated.

The last entry in the continuation stack is always one of the following forms. It is an error to have an empty continuation stack.

D	deallocate this context frame and continue in the parent context.
P	deallocate this context frame and print the value being returned.
E	deallocate this context frame, end this thread of computation and get a new job.

The heap, *h*, is a mapping from addresses to closures. Every closure is one of the forms:

$(vs \setminus\pi\ xs \rightarrow e)\ ws$	This is the same form as the closure in STG heap. This consist of a lambda form $(vs \setminus\pi\ xs \rightarrow e)$, which denotes the code of the closure and a sequence of values, *ws*, that gives the value of each of the free variables, *vs*.
$Locked\ t\ pd\ \overline{pd}$	This represents that the evaluation of the closure in progress. The *t* is the identity of the task performing the evaluation. Two pending lists *pd* and $\overline{pd}$ contain lists of tasks blocked awaiting the result. The $(pd +\!\!+ \overline{pd})$ corresponds to the pending list of the HDG machine[Burn, Kingdon etc, Lester etc]. The role of $\overline{pd}$ will be discussed later.
OutInd pid a	This points the closure which is allocated in a heap of another processing element. (pid,a) is a global address of the remote closure pointed to by this *OutInd* closure.

The ν–STG task pool, *tp* contains details of work either currently being done by a processor, or that has been started and will be continued in the future. Each task status record has three fields, $(exec,pc,\overline{pc})$. Two pending counts, *pc* and $\overline{pc}$ represent how many closures the task has declared as being needed, but have not yet been returned. $(pc+\overline{pc})$ corresponds to the pending count of the HDG machine. The role of $\overline{pc}$ will be discussed below. The *exec* field indicates the actual state of the task at the moment:

R	*Running*, The task is currently being run on some processor.
$B(a,\nu)$	*Blocked*, The task is blocked on the closure at address, a, in context, ν.
$C(a,\nu)$	*Continued*, Previously blocked task that can now be resumed, and should proceed by entering closure, a in context ν.
T	*Terminated*, The task has completed execution, but its record remains until the two pending counts are *zero*.

The work pool, *wp* contains either resumed tasks which will be continued or sparked jobs which can be evaluated by a new task in the future. The *wp* contains five pools:

ap	*active pool*, which is a set of tasks that have been blocked, but are now available for execution.
sap	*semi–active pool*, which is a set of tasks that have been blocked, but are now available for execution if no other work is available. They have a zero *pc* but non–zero $\overline{pc}$ and so may be resumed at the risk of almost immediate re–blocking. See section 7.1.
bs	*basic spark pool*, which is a list of jobs which may be used to create a new task, and cannot be ignored.
is	*ignorable spark pool*, which is a list of jobs that can be used to create a new task, but may be ignored.
ls	*lazy task pool*, which is a list of places where primitive expressions may be found that can be used to create new tasks, on demand.

The above five pools are grouped into two classes, *rp* and *sp*, to explain each transition rule conveniently. The active task pool, *rp* contains *ap* and *sap*, while the others belong to the spark pool, *sp*.

The above pools contain a minimal amount of information for each candidate task. The active pool, *ap* is simply a set of task identifiers that are currently in a $C(a,\nu)$ state, while the semi–active pool *sap* is a set of tasks that are currently in $B(a,\nu)$ with zero *pc* and non–zero $\overline{pc}$. The ignorable spark pool, *is* is a list of pairs of a closure addresses, a and a list of global addresses, ra_u. An *ISparked* job declared by a local task has an empty ra_u, while an imported ispark has a list of closure to be updated when the evaluation is complete. The basic spark pool is a list of two types of components: (a,t) and (a,ra_u). The a is an address of the closure containing the work. The t is a task to notify when the evaluation is complete. The ra_u is a list of global addresses of closures to update with the result when the evaluation is complete. As we will described below, the basic spark declared by local task needs the task name to notify when the evaluation is complete, while the basic spark declared in other processing element and imported needs global addresses to update with the result. One should note that remote updating is the way to notify to remote tasks when computation is complete. Finally, the lazy task pool contains addresses of $R'()$ *continuations* where valid work remains. The address of $R'()$ is defined as the context ν and the depth of the stack beneath the continuation.

The printing mechanism state, *pr*, contains two components, the print stack, *ps*, and the output stream, *op*. These both contain print data, which can be of the following forms:

Addr a	address of closure awaiting printing. (*print stack* only)
Int k	literal value.
c	constructor name. (*out put* only)
RBRACK	right bracket.
LBRACK	left bracket. (*out put* only)

The local environment ρ maps variable names to values. The *val* function takes an atom and delivers a value:

$$\begin{aligned} val\ \rho\ \sigma\ k &= Int\ k \\ val\ \rho\ \sigma\ v &= \rho\ \mathrm{v}\ if\ v \in dom\ (\rho) \\ &= \sigma\ \mathrm{v}\ otherwise \end{aligned}$$

The *mkUpdate* function takes a global address and make either update or remote update continuation. This notation also extends to sequences of global addresses like the local environment augmenting or the function *val*.

$$\begin{aligned} mkUpdate\ (pid,a) &= U(\{\},a) \quad \text{if } pid \text{ is the local processor's name.} \\ &= U'(pid,a) \text{ otherwise} \end{aligned}$$

3.1 Message Handler

To support parallel execution in distributed memory model, messages are sent or received via network. When a message arrives a processor, the processor is interrupted by the message handler to react to the arrived message. After the message is handled, the interrupted process is resumed with a new state which would be changed by the message. The message handler can interrupt the current process only between the codes which are described as above, ie. each transition rule is treated atomically. In a real implementation, the amount of work which need to be treated atomically would be much smaller than the ν–STG codes.

The semantics of the message handler can also be described by a state transition system with the state which consists of the message handler code and the ν–STG machine state which is described above.

The message handler code has one of the following forms:

$MsgISpark\ a\ ra_u$	When it is used as a code for a message handler, it places a job information being imported in the *ignorable spark pool*. If it is used as a message, it tells the message contains an ignorable sparked job information.
$MsgBSpark\ a\ ra_u$	When it is used as a code for a message handler, it places a job information being imported in the *basic spark pool*. If it is used as a message, it tells the message contains a basic sparked job information.
$MsgRequestTask\ pid$	When it is used as a code for a message handler, it looks for an useful job and exports it to the processor, *pid*. As a message, it tells that the processor, *pid*, is asking a new job to you.
$MsgReceiveTask\ closure\ ra_u$	It allocates a closure being imported to the local *heap*, creats a new task to evaluate it, and put the task in the *active task pool*. As a message, it tells that the message contains a job being exported.
$MsgUpdate\ a\ closure$	It updates a closure at address, a, with the closure, which arrives with the message. As a message, it tells that the message asks to update the closure at a with the closure in the message.
$MsgNoTask$	If there is no useful job to export, the MsgNoTask message is sent.

The above code can be used as messages, whilst the following codes are used internally in the message handler.

$MsgGetTask\ pid$	looks up a useful job. This is similar to the *GetTask* or *WaitTask*.
$Resume$	wakes up the interrupted task at the end of message handling

Finally, each closure has two more entry points to export the closure, which are entered by *MsgGetTask* code.

$EnterExportISpark$	If a closure is not evaluated yet, it is exported. If a closure is evaluated, being evaluated or an OutInd node, this code behaves like the *EnterISpark*.
$EnterExportBSpark$	If a closure is not evaluated yet, it is exported. If a closure is evaluated, being evaluated or an OutInd node, this code behaves like the *EnterBSpark*.

The message handler's behaviour is described by the transition rules which is numbered by Mxx and the above codes will be explained in the rules.

3.2 The initial state

The initial machine state is defined below. One processor is instructed to print the value of the closure defined by *main*. All other processors in the machine are placed into a *GetTask* mode. Each processor has its own global environment by copying the closure in WHNF or indirecting via *OutInd* closure to prevent re–evaluation. Only the processor which is instructed to print the result of *main* accesses the print state and has the original global environment in which some closures are indirected by other processors.

```
---------------------------------------------------------------------------------
code          context heap task_pool      work_pool        print_state globals
                                   active_task  sparked_work
EnterPrint a     {}    h     {}    (({},{}),    ({},{},{}))  ({},{})       s
            h = / a1 -> (vs1 \p1 xs1 -> e1) (s vs1) \
               |   ...                              |
                \ an -> (vsn \pn xsn -> en) (s vsn) /
            s = / g1 -> Addr a1 \
               |   ...          |
                \ gn -> Addr an /
            s 'main' -> Addr a
---------------------------------------------------------------------------------
```

In addition, it is worth noting here that the final machine state is when one processor reaches as *Stop* state. In that case, all other processors should be in a *GetTask* or *WaitTask* state, and the task pools should be empty, if the program was annotated correctly.

If all processors are in a *WaitTask* state with empty task pools and there is no message which could make a new task somewhere, it is recognized as a deadlock state. This should not occur in correct programs.

The first rule for the message handler is about resuming:

```
----------------------------------------
M1)  Resume   c  cs  h  tp  wp  pr  s
     ==>
     c           cs  h  tp  wp  pr  s
----------------------------------------
```

4 The conventional STG machine transition rules

This section describes the rules taken from the original STG machine, but expressed in the ν–*STG* term. Some of them are slightly changed to accommodate new features in the ν–*STG* like the distributed context frame, the parallelism, etc.

4.1 Application

Transition rules for application are almost the same as those in the STG machine. Annotated arguments must be closures and their addresses (and the current task name for the *BSpark*) are placed in the work pool to create a new task in the future.

```
-----------------------------------------------------------------------------------------
1)  Eval (f xs) p v  cs[v -> t v' as  rs]  h tp[t -> exec pc pc*]  (rp,(bs,is,ls)     pr s
            val p s f = Addr a
    ==>
    Enter a v        cs[v -> t v' as' rs]  h tp[t -> exec pc' pc*] (rp,(bs',is',ls))  pr s
            as' = (val p s xs) ++ as
            is' = {(a,{})| a<-(isparked xs)} ++ is
            bs' = {(a,t) | a<-(bsparked xs)} ++ bs
            pc' = pc + length (bsparked xs)
-----------------------------------------------------------------------------------------
```

Since the *non–updatable* closure in the STG machine is replaced with the *non–reducible*(`\n`) and the *reducible*(`\r`) closure, there are two rules for entering the closure which is not updatable.

A non-reducible closure with a non-empty argument list, a function value, may have some of its arguments annotated for parallel evaluation. *Work pool* should be updated with these annotations.

```
-------------------------------------------------------------------------------------
2a) Enter a v    cs[v -> t v' as  rs]    h   tp[t -> exec pc] (rp,(bs,is,ls))    pr   s
            h a = [a -> (vs \n xs -> e) wsf]
    ==>
    Eval e p v   cs[v -> t v' as' rs]    h   tp[t -> exec pc] (rp,(bs',is',ls))  pr   s
            wsa ++ as' = as
            length wsa = length xs
            p   = [vs -> wsf, xs -> wsa]
            is' = {(a,{})| a<-(isparked xs)} ++ is
            bs' = {(a,t) | a<-(bsparked xs)} ++ bs
            pc' = pc + length (bsparked xs)
-------------------------------------------------------------------------------------
```

Each time a *reducible closure*(`\r`) in entered it is locked, just in case it has been sparked.

```
---------------------------------------------------------------------
2b) Enter a v    cs    h[a -> (vs \r {} -> e) wsf]    tp    wp    pr    s
            cs v = t v' as rs
    ==>
    Eval e p v   cs    h[a -> Locked t {} {}]         tp    wp    pr    s
            p          = [vs -> wsf]
---------------------------------------------------------------------
```

4.2 let(rec) expressions

This rule is almost identical to the one in the STG machine. A *let(rec)* expression constructs one or more closures in the heap and continues to evaluate the body expression.

```
--------------------------------------------------------------------------------
3)  Eval / let(rec) x1 = vs1 \f1 xs1 -> e1 \ p v   cs   h         tp  wp  pr  s
        |           ...                    |
        |           xn = vsn \fn xsn -> en  |
         \ in e                           /
    ==>
    Eval e p' v   cs  h / a1 -> (vs1 \f1 xs1 -> e1) (prhs vs1) \  tp  wp  pr  s
                       |  ...                                  |
                        \ an -> (vsn \fn xsn -> en) (prhs vsn) /
            p'   = p[x1 -> Addr a1, ... , xn -> Addr an]
            prhs | isa let    = p
                 | isa letrec = p'
--------------------------------------------------------------------------------
```

4.3 Case expressions and data constructors

In the STG machine, given the expression

```
case e of alts
```

the operational interpretation is "push a continuation onto the return stack, and evaluate e". This interpretation is however retained for the primitive typed case expression in the ν–*STG* machine. A new context frame is allocated for each tail–call sequence to distribute a large context into several small chunks, as described later.

```
--------------------------------------------------------------------------------
4)  Eval (case e of alts) p v  cs[v -> t v' as rs]            h  tp  wp  pr  s
               The 'e' is primitive typed.
    ==>
    Eval e p v                 cs[v -> t v' as R(alts,p):rs]  h  tp  wp  pr  s
--------------------------------------------------------------------------------
```

Evaluating a constructor and returning to the appropriate continuation taken from the continuation stack is almost the same as in the STG machine.

Evaluating a constructor application simply moves into the *ReturnCon* state. Some arguments of a constructor application may also be annotated with parallelism. The addresses of the annotated arguments (and the current task name) are placed in the work pool to create new tasks based on them.

```
-------------------------------------------------------------------------------------
5) Eval (c xs) p v              cs    h   tp[t -> exec pc pc*]   (rp,(bs,is,ls))    pr  s
   ==>
   ReturnCon c (val p s xs) v   cs    h   tp[t -> exec pc' pc*]  (rp,(bs',is',ls))  pr  s
           is' = {(a,{})| a<-(isparked xs)} ++ is
           bs' = {(a,t) | a<-(bsparked xs)} ++ bs
           pc' = pc + length (bsparked xs)
-------------------------------------------------------------------------------------
```

The rules for *ReturnCon* return to the appropriate continuation taken from the context frame.

```
-----------------------------------------------------------------------------------------
6) ReturnCon c ws v      cs[v -> v' as R(...;c vs -> e;..., p):rs]   h   tp   wp   pr   s
   ==>
   Eval e p[vs -> ws] v  cs[v -> v' as rs]                           h   tp   wp   pr   s
-----------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
7) ReturnCon c ws v   cs[v -> v' as R/ c1 vs1  -> e1; , p \:rs]  h  tp  wp  pr  s
                                     |  ...                 |
                                     |  cn vsn  -> en;      |
                                      \ default -> ed      /
          c <> ci (1 <= i <= n)
   ==>
   Eval ed p v        cs[v -> v' as rs                     ]     h  tp  wp  pr  s
-------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
8) ReturnCon c ws v  cs[v -> v' as R/ c1 vs1 -> e1; , p \:rs]  h   tp  wp  pr  s
                                    |  ...                |
                                    |  cn vsn -> en;      |
                                     \ x      -> ed      /
          c <> ci (1 <= i <= n)
   ==>
   Eval ed p' v      cs[v -> v' as rs                    ]   h'  tp  wp  pr  s
          h'         = h[a -> (vs \n {} -> c vs) ws]
          p'         = p[x -> Addr a]
          length vs = length ws
          vs is a sequence of arbitrary distinct variables
------------------------------------------------------------------------------
```

4.4 Built in operations

The rules which handle primitive values are almost identical to those in the STG machine.

Evaluating of a primitive literal or a variable bound to an primitive value moves into *ReturnInt* state:

```
----------------------------------------------------------------------
9) Eval k p v      cs    h   tp   wp   pr   s
   ==>
   ReturnInt k v   cs    h   tp   wp   pr   s
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
10) Eval (f {}) p[f -> Int k] v   cs    h   tp   wp   pr   s
   ==>
   ReturnInt k v                  cs    h   tp   wp   pr   s
----------------------------------------------------------------------
```

ReturnInt takes a appropriate continuation from the combined continuation stack of the context frame and returns to the continuation:

```
---------------------------------------------------------------------------
11) ReturnInt k v  cs[v -> v' as R(...;k -> e;..., p):rs]  h  tp  wp  pr  s
   ==>
   Eval e p v     cs[v -> v' as rs                     ]  h  tp  wp  pr  s
---------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------
12) ReturnInt k v   cs[v -> v' as R/ k1 -> e1; , p \:rs]  h  tp  wp  pr  s
                                 |  ...             |
                                 |  kn -> en;       |
                                 \ x  -> ed        /
            k <> ki (1 <= i <= n)
    ==>
    Eval ed p' v    cs[v -> v' as rs                   ]  h  tp  wp  pr  s
            p' = p[x -> Int k]
---------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------------
13) ReturnInt k v   cs[v -> v' as R/ k1       -> e1; , p \:rs]   h  tp  wp  pr  s
                                 |  ...                  |
                                 |  kn       -> en;      |
                                 \ default  -> ed       /
            k <> ki (1 <= i <= n)
    ==>
    Eval ed p v     cs[v -> v' as rs                          ]   h  tp  wp  pr  s
----------------------------------------------------------------------------------
```

Evaluating of built–in arithmetic operation returns the result by *ReturnInt*:

```
----------------------------------------------------------------------
14) Eval (+ x1 x2) p v      cs   h   tp   wp   pr   s
            p x1 = Int i1
            p x2 = Int i2
    ==>
    ReturnInt (i1 + i2) v   cs   h   tp   wp   pr   s
----------------------------------------------------------------------
```

4.5 Updating

Updating rules are almost the same as those in the STG machine. When an updatable closure is entered, a new update continuation is created and placed in the continuation stack and the closure is locked, since it is shared. The update continuation contains the argument stack and closure being entered.

```
----------------------------------------------------------------------------------------------
15) Enter a v    cs[v -> t v' as rs]          h[a -> (vs \u {} -> e) wsf]   tp   wp   pr   s
    ==>
    Eval e p v   cs[v -> t v' {} U(as,a):rs]   h[a -> Locked t {} {}]        tp   wp   pr   s
            p  = [vs -> wsf]
----------------------------------------------------------------------------------------------
```

In the STG machine, the failure of popping from the return stack triggers an update. While completion of the evaluation with seeing an update continuation on the continuation stack starts updating in the ν*–STG* machine. The closure to be updated is just updated with a *standard constructor closure* which can be built for each constructor at compile time.

```
----------------------------------------------------------------------------------
16) ReturnCon c ws v   cs[v -> t v' {}  U(asu,au):rs]  h    tp    (rp,sp)    pr  s
            h au  = Locked t pd pd*
    ==>
    ReturnCon c ws v   cs[v -> t v' asu rs]            h'   tp'   (rp',sp)   pr  s
            h' au = (vs \n {} -> c vs) ws
            length vs = length ws
            (tp',rp') = decPC tp rp pd pd*
            vs is a sequence of arbitrary distinct variables
----------------------------------------------------------------------------------
```

Finally if there are not enough arguments on the stack, an update with a partial function is triggered as in the STG machine.

```
--------------------------------------------------------------------------------
17) Enter a v   cs[v -> t v' as U(asu,au):rs]   h    tp    (rp,(bs,is,ls))    pr   s
            h a  = (vs \n xs -> e) wsf
            h au = Locked t pd pd*
            length as < length xs
    ==>
    Enter a v   cs[v -> t v' (as++asu) rs]      h'   tp'   (rp',(bs,is,ls))   pr   s
            xs1 ++ xs2 = xs
            length xs1 = length as
            h' au      = (f:xs1 \n {} -> f xs1) (a:as)
            (tp',rp')  = decPC tp rp pd pd*
            f is a new variables
--------------------------------------------------------------------------------
```

4.6 Arithmetic optimization

Primitive arithmetic operation can be optimized by the following rules like the STG machine.

```
--------------------------------------------------------------------
18) Eval  / case (+ x1 x2) of \ p v       cs    h    tp    wp    pr    s
         |    k1 -> e1;        |
         |    ...              |
         |    kn -> en;        |
          \   x  -> ed;       /
            p x1 = Ink i1
            p x2 = Int i2
            ki  <> (i1 + i2)    (1 <= i <= n)
    ==>
    Eval ed p[x -> Int (i1 + i2)] v        cs    h    tp    wp    pr    s
--------------------------------------------------------------------
```

```
--------------------------------------------------------------------
18b) Eval / case (+ x1 x2) of \ p v       cs    h    tp    wp    pr    s
         |    k1 -> e1;        |
         |    ...              |
         |    kn -> en;        |
          \   default -> ed;  /
            p x1 = Ink i1
            p x2 = Int i2
            ki  <> (i1 + i2)    (1 <= i <= n)
     ==>
     Eval ed p v                           cs    h    tp    wp    pr    s
--------------------------------------------------------------------
```

```
--------------------------------------------------------------------
19) Eval  / case (+ x1 x2) of \ p v       cs    h    tp    wp    pr    s
         |  ...                |
         |  k -> e;            |
          \ ...               /
            p x1 = Ink i1
            p x2 = Int i2
            k    = i1 + i2
    ==>
    Eval e p v                             cs    h    tp    wp    pr    s
--------------------------------------------------------------------
```

5 Distributed stack

In the ν–*STG* machine, a new context frame is allocated for each tail–call sequence and it is abandoned after the last reduction in the sequence. The distribution critically depends on the ability of the compiler to differentiate between tail and non–tail calls. A moderate sized frame is allocated initially. If it is too small, larger frame is reallocated as needed. The low frequency of reallocation may be achieved by expanding frame exponentially and choosing a good initial size.

Each closure has the second entry point *EnterAlloc*. The *EnterAlloc* is entered as the first call in a tail–call sequence and subsequent reductions use normal entries *Enter*. *EnterAlloc* saves the old context frame pointer and sets up a new frame, while *Enter* uses the existing frame. The compiler can easily decide between two:

```
case x of alts
=>
build x; ...  CALL x_EnterAlloc; alts-code

let e1= ... en= .. in e
=>
build e1 ... en; JUMP e_Enter
```

The switch expression of algebraic type must be a variable applied to no argument as described above. In a well typed program, this closure will evaluate to a constructor application. A *return–to* is pushed, and the closure is entered, with a new context being allocated if necessary.

```
---------------------------------------------------------------------------------
21) Eval (case (x {}) of alts) p v   cs[v -> t v' as rs]           h  tp  wp  pr  s
            val p s x = Addr a
            (iff closure does not reduce to partial application)
    ==>
    EnterAlloc a v                   cs[v -> t v' as R(alts,p):rs]  h  tp  wp  pr  s
---------------------------------------------------------------------------------
```

EnterAlloc behaves in a similar way to *Enter*, except that a new context frame is allocated and used for the evaluation of the closure if necessary.

For non–reducible closures, the *EnterAlloc* is the same as the *Enter*. It doesn't have to allocate a new context frame. In this case the closure must be a constructor in WHNF if the program is well typed.

```
----------------------------------------------------------------------
22) EnterAlloc a v        cs   h   tp    wp    pr    s
            h a = [a -> (vs \n {} -> c xs) wsf]
    ==>
    Eval (c xs) p v       cs   h   tp    wp    pr    s
            p   = [vs -> wsf]
----------------------------------------------------------------------
```

For reducible or updatable closures, a new context frame is allocated. In the bottom of the new context frame, the *deallocation continuation* is placed to abandon this frame at the last reduction of the evaluation of the closure.

```
-----------------------------------------------------------------------------------------
23) EnterAlloc a v    cs                     h[a -> (vs \r {} -> e) wsf]   tp   wp   pr   s
            cs v = t v' as rs
    ==>
    Eval e p v_n      cs[v_n -> t v {} {D}]  h[a -> Locked t {} {}]        tp   wp   pr   s
            p = [vs -> wsf]
            v_n is not in domain cs.
-----------------------------------------------------------------------------------------
```

For an updatable closure, *update continuation* is placed in the new context frame to update the closure when the evaluation is complete.

```
--------------------------------------------------------------------------------------------
24) EnterAlloc a v    cs                             h[a -> (vs \u {} -> e) wsf]  tp  wp  pr  s
            cs v = t v' as rs
    ==>
    Eval e p v_n      cs[v_n -> t v {} U(as,a):{D}]  h[a -> Locked t {} {}]       tp  wp  pr  s
            p = [vs -> wsf]
    v_n is not in domain cs.
--------------------------------------------------------------------------------------------
```

The next rule deals with the case when a deallocate continuation is seen by *ReturnCon*. In this case, the current context frame is discarded and the *ReturnCon* repeated in the old context.

```
----------------------------------------------------------------------
25) ReturnCon c xs v      cs[v -> t v' {} {D}]   h   tp   wp   pr   s
    ==>
    ReturnCon c xs v'     cs                     h   tp   wp   pr   s
----------------------------------------------------------------------
```

6 Sparking Mechanism

To introduce parallelism, the sparking mechanism is used. In the *ν–STG language*, some variables may be annotated to spark at run time. Sparking happens when the expression in which the spark occurs is evaluated. When sparking occurs, neither context frame nor any detail record of the task is allocated, while minimal amounts of job informations are saved in the *work pool* as described above. When a job which a local task has sparked is looked up, it becomes a real task, allocating a detail record of the task status and a context frame. If the closure pointed to by a job in the *work pool* is already evaluated or being evaluated, no task is created from the job information.

The job information of the *work pool* may be exported to other remote processors on demand. If a closure evaluated or being evaluated is looked up by message handler, the closure is not exported and the next job is looked up. When a job information is received from a remote processor, a new task is created from the information to prevent a job cycling around the machine without performing useful work. Once a task is created from a job information, the task is performed in a local processor until it is terminated. Task's name is known to only the local processor, while other remote processors don't have to know any remote task's name.

The *ν–STG* machine support two mechanism of sparking, *ISpark* and *BSpark*, like the HDG machine. An *ISparked* task may be ignored, whilst an *BSparked* task should be synchronized with the parent task and cannot be ignored. How and by whom the sparking annotation is given will not be discussed in this paper.

Each closure has two more entry points to support the above two sparking mechanism, *EnterISpark* and *EnterBSpark*. When the *work pool* is looked up to find a useful work for an idle processor, the closure which has been sparked and placed in the work pool is entered via one of the above two entry points according to the information. The following sections will describe these mechanism.

6.1 EnterISpark

EnterISpark creates a process to evaluate a closure to WHNF.

In the case when the closure is not in WHNF (and so has a `\r` or `\u` flag) *EnterISpark* locks the closure, and creates a context frame to contain the stacks. Two continuations are stored in the continuation stack. The first of these (in order of popping) will result in the closure being updated. This has the effect of treating a `\r` closure as a `\u` closure when sparked which can be seen to be correct as sparking a closure effectively causes it to be shared, and the result must be stored in some place holder in order that the other task can find it. The second continuation is an *end–thread* continuation. This will result in the process being halted and destroyed once the evaluation and update have been completed.

```
-----------------------------------------------------------------------------------------------
26) EnterISpark a rau  cs                    h[a -> (vs \u\r {} -> e) wsf]  tp           wp pr s
    ==>
    Eval e p v         cs[v -> t _ {} cont]  h[a -> Locked t {} {}]         tp[t -> R 0] wp pr s
            p = [vs -> wsf]
            v is not in domain cs
            t is not in domain tp
            cont = mkUpdate rau ++ {U({},a),E}
-----------------------------------------------------------------------------------------------
```

If evaluation is already complete the spark is ignored and try to find another job from the work pool, *wp*.

```
----------------------------------------------------------------------
27) EnterISpark a rau         cs   h   tp   wp   pr   s
            h a = (vs \n xs -> e) wsf
    ==>
    GetTask                   cs   h   tp   wp   pr   s
----------------------------------------------------------------------
```

6.2 EnterBSpark

EnterBSpark, like *EnterISpark*, creates a process to evaluate a closure to WHNF if evaluation has not already been started. In addition, it carries an extra parameter: either the name of the task requesting the spark or a list of global addresses. The state of the machine is set such either that this task will be notified or that the closures in the address list are updated with a result once evaluation of closure is complete.

In the case when the closure is not in WHNF (and so has a `\r` or `\u` flag) *EnterBSpark* locks the closure and stores the task name in the pending list. It then proceeds as *EnterISpark*, creating a context frame, and putting the *update* and *end–thread* continuations in the continuation stack.

In this rule, you can see that the task name is placed the second pending list and the pending counts are adjusted by decreasing the first one and increasing the second one. By doing so, it can be recoreded that a task is launched for a *BSparked* closure. This information can release a deadlock situation which would occur as described below.

```
-----------------------------------------------------------------------------------------
28) EnterBSpark a t'  cs   h[a -> (vs \u\r {} -> e) wsf]  tp[t' -> exec pc pc*]      wp  pr  s
    ==>
    Eval e p v        cs'  h[a -> Locked t {} {t'}]       tp[t' -> exec pc-1 pc*+1]  wp  pr  s
                                                            [t  -> R 0            ]
            p   = [vs -> wsf]
            cs' = cs[v -> t _ {} U({},a):{E}]
            v is not in domain cs
            t is not in domain tp
-----------------------------------------------------------------------------------------
```

Presented with a *non–reducible* closure we once again have to look for more work. However before doing so we must notify the process that evaluation is complete to avoid deadlock.

```
-----------------------------------------------------------------------
29) EnterBSpark a t        cs    h    tp    (rp,sp)    pr    s
            h a = (vs \n xs -> e) wsf
    ==>
    GetTask                cs    h    tp'   (rp',sp)   pr    s
            (tp',rp') = decPC tp rp {t} {}
-----------------------------------------------------------------------
```

If the closure is not in WHNF and the basic sparked work is an imported one, it locks the closure and puts the *update* or *remote update continuation* in the continuation stack to update the relevant closures. In the case of an imported basic spark, *updating* is used to notify the completion of evaluation. So, the name of task created here is not placed in the pending lists.

```
------------------------------------------------------------------------------------
30) EnterBSpark a rau    cs   h[a -> (vs \u\r {} -> e) wsf]   tp            wp   pr   s
    ==>
    Eval e p v           cs'  h[a -> Locked t {} {}]          tp[t -> R 0]  wp   pr   s
            p   = [vs -> wsf]
            cs' = cs[v -> t _ {} cont++{U({},a),E}]
            cont = mkUpdate rau
            v is not in domain cs
            t is not in domain tp
------------------------------------------------------------------------------------
```

In the case of *non–reducible* closure, it notifies that evaluation is complete. As described before, *notifying* for an imported basic spark is *updating*. Therefor, messages are sent to the relevant closures to update them with the entered closure which is in WHNF already.

```
-----------------------------------------------------------------------
31) EnterBSpark a rau        cs    h    tp    wp    pr    s
            h a = (vs \n xs -> e) wsf
    ==>
    GetTask                  cs    h    tp    wp    pr    s
                send messages to update the closures in rau with
                the closure ((vs \n xs -> e) wsf)
-----------------------------------------------------------------------
```

6.3 Ending thread

In the case of an *end–thread* continuation being present in the *ReturnCon* state the machine simply discards the context and enters a *GetTask* phase. This may seem unusual in that the constructor and the arguments are discarded. However for this continuation to have been reached, these will have previously been saved in an update.

```
-----------------------------------------------------------------------------------
32) ReturnCon c xs v           cs[v -> t _ {} {E}]  h  tp[t -> R pc]  wp  pr  s
    ==>
    GetTask                    cs                   h  tp[t -> T pc]  wp  pr  s
-----------------------------------------------------------------------------------
```

If a sparked closure results in a partial application, the *end–thread* continuation can be seen at the time of *Enter*. In this case, discards the context and try to find new job. *Update continuation* which must exist previously has updated the closure with the *standard partial application form*.

```
----------------------------------------------------------------------
33) Enter a v   cs[v -> t _ as {E}]  h   tp[t -> R pc]   wp   pr   s
            h a       = (vs \n xs -> e) wsf
            length as < length xs
    ==>
    GetTask     cs                   h   tp[t -> T pc]   wp   pr   s
----------------------------------------------------------------------
```

7 Pending Mechanism

When a task attempt to evaluate a closure being evaluated, the task's name is placed the pending list of the closure, the pending count of the task is increased by one and the task is blocked to wait the result. When the evaluation of the closure is complete, it is notified to the tasks in the pending list by decrementing the pending counts of the tasks.

```
---------------------------------------------------------------------------------------------
34) Enter a v      cs   h[a -> Locked t' pd pd*]     h   tp[t -> R pc pc*]          wp   pr   s
            cs v = t v' as rs
    ==>
    GetTask        cs   h[a -> Locked t' t:pd pd*]   h   tp[t -> B(a,v) pc+1 pc*]   wp   pr   s
---------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------
35) EnterAlloc a v     cs  h[a -> Locked t' pd pd*]     tp[t -> R pc pc*]          wp  pr  s
            cs v = t v' as rs
    ==>
    GetTask            cs  h[a -> Locked t' t:pd pd*]   tp[t -> B(a,v) pc+1 pc*]   wp  pr  s
-------------------------------------------------------------------------------------------
```

When a *ISparked* work is looked up from the work pool and the closure pointed by the *ISparked* work is being evaluated, the spark is ignored without creating any task and the processor looks for work once again.

```
----------------------------------------------------------------------
36) EnterISpark a rau    cs   h   tp   wp   pr   s
            h a = Locked _ _ _
    ==>
    GetTask              cs   h   tp   wp   pr   s
----------------------------------------------------------------------
```

If a *BSparked* work is looked up and the closure is being evaluated, the name of the task which declare to need the result of the closure is placed in the pending list. The pending count of the task must be increased when spawning a *BSparked* work. No task has to be created for the *BSparked* work.

```
-------------------------------------------------------------------------
37) EnterBSpark a t   cs   h[a -> Locked t' pd pd*]     tp   wp   pr   s
            t <> t'
    ==>
    GetTask           cs   h[a -> Locked t' t:pd pd*]   tp   wp   pr   s
-------------------------------------------------------------------------
```

If the task to be notified is the task actually calculating the value of the closure, ie the evaluation has been *in–lined*, one simply notifies the process. This means the process will not be waiting for the evaluation of the result it is evaluating itself, which would result in deadlock.

```
----------------------------------------------------------------------
38) EnterBSpark a t   cs   h   tp   (rp,sp)    pr   s
        h a = Locked t _ _
    ==>
    GetTask           cs   h   tp'  (rp',sp)   pr   s
        (tp',rp') = decPC tp rp {t} {}
----------------------------------------------------------------------
```

If an imported *BSparked* work is seen, a new task is created to update the closures which are waiting the result of the closure being evaluated, the new task's name is placed in the pending list and the task is blocked.

To prevent a new task being created in blocked state, the *Locked* closure could be changed to have a field to save a list of global addresses of closures to be updated with the result when the evaluation is complete. We should then check whether or not there is any closure to be updated whenever a *Locked* closure is updated with a result. This is however the only case that needs to create a new task in blocked state. If this rule is applied very rarely, creating a blocked task wouldn't be a problem.

```
-----------------------------------------------------------------------------------------------
39) EnterBSpark a rau  cs                    h[a -> Locked t' pd pd*]   tp              wp pr s
    ==>
    GetTask           cs[v -> t _ {} cont]  h[a -> Locked t' t:pd pd*] tp[t -> B(a,v)] wp pr s
            cont = mkUpdate rau ++ {E}
            v is not in domain cs
            t is not in domain tp
-----------------------------------------------------------------------------------------------
```

7.1 Double Pending Mechanism

You can see now how the double pending mechanism works.

When a *BSparked* task shares a subexpression with its parent task, deadlock would occur as the following:

1. The parent task comes first to a shared subexpression and continues to evaluate it in–line.
2. When the shared subexpression is being evaluated in–line by the parent task, the *BSparked* task attempts to evaluate the subexpression and be blocked.
3. If we assume that the parent task is blocked by somebody else after the above situation, deadlock occurs because the parent task cannot be waken up again since the *BSparked* task is not complete, whilst the *BSparked* task cannot be waken up since the shared subexpression being evaluated by the parent task is not complete yet.

To release the above deadlock, the parent task should be waken up. However, to recognize the above deadlock, whenever the *BSparked* task attempts to evaluate an expression being evaluated, it should check whether or not the expression is being evaluated by one of its ancestor tasks. It is too expensive because the names of the ancestor tasks should be kept in every sparked task records.

Now we introduce the *double pending mechanism*, which can release the above deadlock, but not recognize it. In the double pending mechanism, when a new task is created for a *BSparked* work, the pending counts of the parent are adjusted by decreasing the first pending count and increasing the second one which tell how many *BSparked* tasks are spawned from this task. The parent task name is placed in the second pending list to be matched with the pending counts when the sparked closure is *Locked* by the *BSparked* task. If we add the two pending counts and combine the two pending list, it becomes equivalent to the pending mechanism which can be found in the HDG machine.

When an evaluation is complete, it is notified to the tasks in the pending lists by decrementing the pending counts of the tasks and updating the work pool according to the change of the task status.

If the first pending count is being decremented to *zero*, the second pending count is already *zero*, and the task is in the *blocked* state, it's state is changed to *continued*, and it's identifier is placed in the *active task pool*. If the second pending count is *not zero* at the previous case, the task is placed in the *semi–active task pool* where a task is tried to wake up at the risk of an almost immediate re–blocking to release a deadlock as described earlier. If the first pending count is already *zero* and the second pending count is being *zero* in a *blocked* task, the task's state is changed to *continued*, queued up in the *active task pool*, and removed from the *semi–active task pool* if necessary. The following rules detail this operation:

```
----------------------------------------------------------------------
decPC  tp  rp  pd  pd* = (decPC' pd . decPC'' pd*) (tp, rp)
----------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------
decPC'  [] (tp, rp) = (tp, rp)
decPC'  t:pd  (tp[t -> B(a,v) 1 0],  (ns1, ns2)) = (tp[t -> C(a,v) 0 0],  (ns1++[t], ns2))
decPC'  t:pd  (tp[t -> B(a,v) 1 pc'],(ns1, ns2)) = (tp[t -> B(a,v) 0 pc'],(ns1, ns2++[t]))
decPC'  t:pd  (tp[t -> exec pc pc'], rp)         = (tp[t -> exec pc-1 pc'],rp)
------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------
decPC'' [] (tp, rp) = (tp, rp)
decPC'' [t']  (tp[t' -> B(a,v) 0 1], (ns1, ns2)) = (tp[t' -> C(a,v) 0 0],(ns1++[t'], ns2--[t']))
decPC'' [t']  (tp[t' -> exec pc pc'],rp)         = (tp[t' -> exec pc pc'-1],rp)
------------------------------------------------------------------------------------------
```

If there is no job available, a task in *semi–active task pool* is tried to run at the risk of immediate reblocking. This attempt can release the above deadlock, if happens, which would occur by sharing an expression between a *BSparked* task and its parent task.

```
---------------------------------------------------------------------------------
40) WaitTask        cs   h   tp[t -> B pc pc*]   (({},tid:ns2),({},{},ls))  pr  s
    ==>
    Enter a v       cs   h   tp[t -> R pc pc*]   (({},ns2),({},{},ls))      pr  s
---------------------------------------------------------------------------------
```

The double pending mechanism can release the above deadlock if it happens as described above, but not recognize the deadlock. Even though it has a risk of immediate re–blocking when it wakes up a task in the *semi–active task pool*, it doesn't affect to the result of a well–typed program.

8 Get a new task

8.1 Get a task from the local work pool

When a task is blocked or terminated, the processor looks for a new job from the work pool.

We now examine the rules for *GetTask* and *WaitTask* which detail how an idle processor attempts to find work. The *GetTask* requests a job to other remote processor if necessary, while the *WaitTask* looks for a new job from only the local work pool with waiting a new job from a remote processor which was requested in the *GetTask* phase.

Firstly, if there are any active tasks one of these is taken. The resumption is made by entering the closure in the context stored in the active task pool.

```
--------------------------------------------------------------------------
41) GetTask     cs   h   tp[t -> C(a,v) 0]   wp@((t:ns1,ns2),sp)   pr   s
    ==>
    Enter a v   cs   h   tp[t -> R 0]        wp@((ns1,ns2),sp)     pr   s
                if size wp < k then request a new task.
                k is a thresh hold value.
--------------------------------------------------------------------------
```

```
------------------------------------------------------------------------
42) WaitTask     cs   h   tp[t -> C(a,v) 0]   ((t:ns1,ns2),sp)   pr   s
    ==>
    Enter a v    cs   h   tp[t -> R 0]        ((ns1,ns2),sp)     pr   s
------------------------------------------------------------------------
```

In the event of no active tasks being available, a sparked process is sought. Basic sparks take priority over ignorable sparks. The new task is started with an *EnterBSpark*, or an *EnterISpark*.

```
------------------------------------------------------------------------
43) GetTask          cs   h   tp   wp@(({},ns2),(b:bs,is,ls))   pr   s
    ==>
    EnterBSpark b     cs   h   tp   wp@(({},ns2),(bs,is,ls))     pr   s
                if size wp < k then request a new task.
                k is thresh hold value.
------------------------------------------------------------------------
```

```
---------------------------------------------------------------------
44) WaitTask          cs   h   tp   (({},ns2),(b:bs,is,ls))   pr   s
    ==>
    EnterBSpark b      cs   h   tp   (({},ns2),(bs,is,ls))     pr   s
---------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
45) GetTask            cs   h  tp wp@(({},ns2),({},((a,rau):is),ls))   pr   s
    ==>
    EnterISpark a rau   cs   h  tp wp@(({},ns2),({},is,ls))            pr   s
                if size wp < k then request a new task.
                k is thresh hold value.
-------------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------
46) WaitTask            cs   h  tp (({},ns2),({},((a,rau):is),ls))   pr   s
    ==>
    EnterISpark a rau    cs   h  tp (({},ns2),({},is,ls))            pr   s
-----------------------------------------------------------------------------
```

The final rule for *GetTask* deals with the case when no work is available at the present. In this case the unemployed processor sends a message to get a new task and go to the *WaitTask* state.

```
---------------------------------------------------------------------
47) GetTask    cs   h   tp   wp@(({},ns2),({},{},ls))   pr   s
    ==>
    WaitTask   cs   h   tp   wp@(({},ns2),({},{},ls))   pr   s
                if size wp < k then request a new task.
                k is thresh hold value.
---------------------------------------------------------------------
```

The final rule for *WaitTask* deals with the case when no work is available at present. In this case the unemployed processor tries to get a job next time.

```
---------------------------------------------------------------------
48) WaitTask   cs   h   tp  (({},{}),({},{},{}))   pr   s
    ==>
    WaitTask   cs   h   tp  (({},{}),({},{},{}))   pr   s
---------------------------------------------------------------------
```

8.2 Exporting a task

When a message which requests a new job has arrived, the message handler starts a *MsgGetTask* phase to find a useful job to export.

```
-----------------------------------------------------------------------
M2) MsgRequestTask pid'    c  cs  h  tp  wp  pr  s
               There are enough jobs to export in 'wp'
    ==>
    MsgGetTask pid'        c  cs  h  tp  wp  pr  s
-----------------------------------------------------------------------
```

If there are not enough works to export, *NoTask* message is sent back.

```
-----------------------------------------------------------------------
M3) MsgRequestTask pid'    c  cs  h  tp  wp  pr  s
               There are not enough jobs to export in 'wp'
    ==>
    Resume                 c  cs  h  tp  wp  pr  s
               sendMessage pid' MsgNoTask
-----------------------------------------------------------------------
```

MsgGetTask phase attempts to find a useful job to export from the *basic sparked pool*, the *ignorable sparked pool*, and the *lazy task pool*, in the order. This is very similar to the *GetTask* phase, but different entry points are used to export a useful job.

If a sparked job is found in the *BSparked* pool, the sparked closure is entered via the *EnterExportBSpark* which exports the closure and locks it.

```
------------------------------------------------------------------------
M4) MsgGetTask pid'            c  cs  h  tp  (rp,(bs++{b},is,ls)) pr  s
    ==>
    EnterExportBSpark b pid'   c  cs  h  tp  (rp,(bs,is,ls))      pr  s
------------------------------------------------------------------------
```

If the *basic spark pool* is empty, the *ignorable spark pool* is looked up to find an exportable job. If one is available, the related closure is entered via *EnterExportISpark* entry point.

```
------------------------------------------------------------------------------------
M5) MsgGetTask pid'                c  cs  h  tp  (rp,({},is++{(a,rau)},ls))  pr   s
    ==>
    EnterExportISpark a rau pid'   c  cs  h  tp  (rp,({},is,ls)              pr   s
------------------------------------------------------------------------------------
```

If the message handler cannot find a closure for export, it tries to build one from the *lazy task pool*, using the *lazy task creation mechanism*.

If a closure that is found from the *work pool* is not evaluated yet, it is worthwhile to export it. *EnterExportBSpark* locks the closure and export it by the *MsgReceiveTask* message. Adjusting the pending counts and placing the task name to be notified when the exported closure is evaluated are the same as the *EnterBSpark*.

```
---------------------------------------------------------------------------------------------
M6) EnterExportBSpark a t pid'  c  cs  h[(vs \u\r xs -> e) ws]  tp[t -> exec pc pc*]     wp pr s
    ==>
    Resume                      c  cs  h[a -> Locked rt {} {t}] tp[t -> exec pc-1 pc*+1] wp pr s
           sendMessage pid' (MsgReceiveTask clo {(pid,a)})
           pid is the current processor's name.
           clo = "(vs \u\r xs -> e) ws"
           rt is a distinguished name to represent remote task.
---------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
M7) EnterExportBSpark a rau pid'   c  cs  h[(vs \u\r xs -> e) ws]  tp  wp  pr  s
    ==>
    Resume                         c  cs  h[a -> Locked rt {} {}]  tp  wp  pr  s
           sendMessage pid' (MsgReceiveTask clo (pid,a):rau)
           clo = "(vs \u\r xs -> e) ws"
           rt is a distinguished name to represent remote task.
           pid is the current processor's name.
------------------------------------------------------------------------------
```

EnterExportISpark behaves like *EnterISpark*, but locks the closure and export it.

```
-----------------------------------------------------------------------------
M8) EnterExportISpark a rau pid'   cs  h[a -> (vs \u\r xs -> e) ws] tp wp pr s
    ==>
    Resume                         cs  h[a -> Locked rt {} {}]      tp wp pr s
       sendMessage pid' (MsgReceiveTask clo {})
       clo = "(vs \u\r xs -> e) ws"
       rt is a distinguished name to represent remote task.
       pid is the current processor's name.
-----------------------------------------------------------------------------
```

If a closure, which is already evaluated, being evaluated, or an *OutInd* node, is entered via *EnterExportBSpark* or *EnterExportISpark*, it is treated as entered via *EnterBSpark* or *EnterISpark* and the *MsgRequestTask* phase is tried once again to find a useful job to export.

```
---------------------------------------------------------------------
M9) EnterExportISpark a rau pid'     c   cs   h   tp   wp   pr   s
            h a = (vs \n xs -> e) wsf
    ==>
    MsgRequestTask pid'              c   cs   h   tp   wp   pr   s
---------------------------------------------------------------------


---------------------------------------------------------------------
M10) EnterExportBSpark a t pid'   c   cs   h   tp   (rp,sp)    pr   s
            h a = (vs \n xs -> e) wsf
    ==>
    MsgRequestTask pid'           c   cs   h   tp'  (rp',sp)   pr   s
            (tp',rp') = decPC tp rp {t} {}
---------------------------------------------------------------------


---------------------------------------------------------------------
M11) EnterExportBSpark a rau pid'   c   cs   h   tp   wp    pr   s
            h a = (vs \n xs -> e) wsf
     ==>
     MsgRequestTask pid'            c   cs   h   tp   wp    pr   s
                send messages to update the closures in rau with
                the closure ((vs \n xs -> e) wsf)
---------------------------------------------------------------------


---------------------------------------------------------------------
M12) EnterExportISpark a rau pid'   c   cs   h   tp   wp   pr   s
            h a = Locked _ _ _
     ==>
     MsgRequestTask pid'            c   cs   h   tp   wp   pr   s
---------------------------------------------------------------------


---------------------------------------------------------------------------------
M13) EnterExportBSpark a t pid'   c  cs  h[a -> Locked t' pd pd*]    tp  wp  pr  s
            t <> t'
     ==>
     MsgRequestTask pid'          c  cs  h[a -> Locked t' t:pd pd*]  tp  wp  pr  s
---------------------------------------------------------------------------------


---------------------------------------------------------------------
M14) EnterExportBSpark a t pid'   c   cs   h   tp   (rp,sp)    pr   s
            h a = Locked t _ _
     ==>
     MsgRequestTask pid'          c   cs   h   tp'  (rp',sp)   pr   s
            (tp',rp') = decPC tp rp {t} {}
---------------------------------------------------------------------


-----------------------------------------------------------------------------------------------
M15) EnterExportBSpark a rau pid'  c  cs  h[a -> Locked t' pd pd*]   tp               wp  pr  s
     ==>
     MsgRequestTask pid'           c  cs' h[a -> Locked t' t:pd pd*] tp[t -> B(a,v)]  wp  pr  s
                cs'  = cs[v -> t _ {} cont++{E}]
                cont = mkUpdate rau
                v is not in domain cs
                t is not in domain tp
-----------------------------------------------------------------------------------------------
```

If a new job arrives, a new task is created and placed in the *active task pool* with a new context frame which contents *update* or *remote update continuations* at the end.

```
------------------------------------------------------------------------------
M16) MsgReceiveTask clo rau     c  cs  h  tp  ((ns1,ns2),sp)      pr  s
     ==>
     Resume                     c  cs' h' tp' ((ns1++{t},ns2),sp)  pr  s
                         h'  = h[a -> clo]
                         cs' = cs[v -> t _ {} cont++{E}]
                         cont = map mkUpdate rau
                         tp' = tp[t -> C(a,v) 0]
                         a is not in domain h
                         v is not in domain cs
                         t is not in domain tp
------------------------------------------------------------------------------
```

If a message is replied with no jobs, the message handler tries again to request a new one in the case that a new one is still needed.

```
--------------------------------------------------------------------------
M17) MsgNoTask        c  cs  h  tp  wp  pr  s
     ==>
     Resume           c  cs  h  tp  wp  pr  s
             if size wp < k then request a new task.
             k is thresh hold value.
--------------------------------------------------------------------------
```

9 Lazy Task Creation Mechanism

A *lazy task creation mechanism* is designed to exploit parallelism in *unboxed arithmetic expressions* by *boxing up* expressions to create parallel tasks on demand.

Unboxed expressions suitable for parallel execution are explicitly exposed in the ν–*STG* code by the parallel default alternative of the case construct.

Evaluation of a *casep expression* causes a $R'()$ continuation to be placed on the continuation stack. This stores all the details of the *casep* and the machine begins evaluation of the first switch expression. In addition, the address of the $R'()$ continuation is placed in the *lazy task pool*.

```
----------------------------------------------------------------------------------------------------
49) Eval (casep (e:es) of (vs) -> exc) p v   cs[v -> t v' as rs]   h  tp  (rp,(bs,is,ls))  pr  s
    ==>
    Eval e p v                               cs'                   h  tp  (rp,(bs,is,ls')) pr  s
              cs' = cs[v -> t v' as R'(es,vs,{},exc,p):rs]
              ls' = (v,length rs):ls
----------------------------------------------------------------------------------------------------
```

If the machine is so busy and so no processing element asks a new task from the *lazy task pool*, the parallel expressions will be evaluated in–line by a single task from the front expression. After each expression has been evaluated, its result is stored in a mini environment, ρ' and evaluation of the next expression is started.

```
------------------------------------------------------------------------------------------
50a) ReturnInt k v   cs[v -> t v' as R'(e:es,x:xs,p',exc,p):rs]          tp   wp   pr   s
     ==>
     Eval e p v      cs[v -> t v' as R'(es,xs,p'[x -> Int k],exc,p):rs]  tp   wp   pr   s
------------------------------------------------------------------------------------------
```

If the $R'()$ continuation contains exactly one parallel expression, the address of the $R'()$ should be removed from the *lazy task pool* before evaluating the last expression. This is important because if we leave an useless item, which points a $R'()$ continuation with no parallel expression, in the *lazy task pool*, and the continuation is poped up, the useless item is dangled and there is no way to remove it from the *lazy task pool*.

```
----------------------------------------------------------------------------------------------------
50b) ReturnInt k v   cs[v -> t v' as R'({e},x:xs,p',exc,p):rs]         tp (rp,(bs,is,ls))  pr s
     ==>
     Eval e p v      cs[v -> t v' as R'({},xs,p'[x -> Int k],exc,p):rs] tp (rp,(bs,is,ls')) pr s
                ls' = ls -- {(v,length rs)}
----------------------------------------------------------------------------------------------------
```

In the case of the continuation not containing any more parallel expressions, the machine begins the evaluation of the body expression. A new environment is built consisting of the environment external to the original *casep* (ρ), the environment built up during the evaluation of previous parallel expressions (ρ') and the result of the last expression bound to the remaining variable in the *xs* list.

```
-----------------------------------------------------------------------------
51) ReturnInt k v   cs[v -> t v' as R'({},{x},p',exc,p):rs]  h  tp  wp  pr  s
    ==>
    Eval exc pc v   cs[v -> t v' as rs]                      h  tp  wp  pr  s
            pc      = (p' ++ p)[x -> Int k]
-----------------------------------------------------------------------------
```

If during the evaluation of any, but the last, of the parallel expressions an idle processor requests work, the last expression will be removed from the $R'()$ continuation. It is boxed up into a closure and exported to the idle processor. In addition, the body expression is replaced with an expression which will unbox the result and bind it to the matched variable, before continuing with the evaluation of the body expression.

If a $R'()$ continuation has at least one expression, the last expression in the list (e) is taken, along with it's corresponding variable (x), and the continuation changed to account for the fact that what was originally an unboxed expression is now a boxed one.

First the ρ' environment is augmented with a new variable that is bound to a new, locked heap location. Then the continuation expression is replaced with code exc' to unbox the value at that location, before continuing with evaluation.

In addition to the continuation of the other task being changed, a new closure to evaluate the expression is created and exported with leaving the locked closure, which is used to augment the ρ' environment, to be notified by updating when the evaluation is completed. The evaluation of this closure will produce a boxed typed value, and this boxed value will be unboxed by the code exc'.

```
-----------------------------------------------------------------------------------------------
M18) MsgGetTask pid'   c  cs[v -> t v' as (rs'++(r:rs))]   h
                             tp[t -> exec pc pc*]    (rp,({},{},(v,length rs):ls))   pr   s
           r     = R'(es++{e},xs++{x},p',exc,p)
     ==>
     Resume            c  cs[v -> t v' as (rs'++(r':rs))]  h[a -> Locked rt {} pd*]
                             tp[t -> exec pc pc*']   (rp,({},{},ls'))                pr   s

           r'    = R'(es,xs,p'[w -> Addr a],exc',p)
           exc'  = "case w of MAGIC x -> exc"

           ls'   | es == {}   = ls
                 | otherwise  = (v,length rs) : ls

           (pc*',pd*) | doingBSpark   = (pc*+1, {t})
                      | otherwise     = (pc*,   {})

           sendMessage pid' (MsgReceiveTask clo {(pid,a)})
           clo   = "(vs \r {} -> case e of x -> MAGIC x) (p vs)"
           vs is free variables of e
           pid is the current processor's name.

           rt is a distinguished name to represent remote task.
           a  is not in domain h
           w  is not in domain (p' ++ p)
---------------------------------------------------------------------------------------------
```

10 Remote Accessing

10.1 Remote updating

If a remote update continuation is observed in the *ReturnCon* state, a message is sent to update the relevant closure in a remote processor.

```
-------------------------------------------------------------------------
52) ReturnCon c ws v   cs[v -> t v' []  U'(pid',au):rs]  h  tp  wp  pr  s
    ==>
    ReturnCon c ws v   cs[v -> t v' [] rs]               h  tp  wp  pr  s
            sendMessage pid' (MsgUpdate au ((vs \n {} -> c vs) ws))
            length vs = length ws
            vs is a sequence of arbitrary distinct variables
-------------------------------------------------------------------------
```

A remote update continuation may be found when a closure is entered, if the closure is a partial application. In this case, the closure is entered again after sending a message to update a remote closure. Then update or remote update continuation may be seen and the *end–thread* continuation must be found at the end. This rule could be optimized to see a sequence of continuations together without entering to the closure again. The sequence of the continuation must be the form of $\{U', (U|U')*, E\}$, ie start with *remote update* and zero or more of *update* or *remote update continuations* and the *end–thread continuation* at the end.

```
------------------------------------------------------------------------------
53) Enter a v   cs[v -> t v' as  U'(pid,au):rs]    h  tp  wp  pr  s
                h a = (vs \n xs -> e) wsf
                length as < length xs
    ==>
    Enter a v   cs[v -> t v' as  rs]               h  tp  wp  pr  s

                xs1 ++ xs2 = xs
                length xs1 = length as
                sendMessage pid (MsgUpdate au ((f:xs1 \n {} -> f xs1) (a:as)))
                f is a new variables
------------------------------------------------------------------------------
```

If the message is for updating a closure which is placed locally and locked by a remote task, the closure is updated and it is notified to tasks in the pending list.

```
------------------------------------------------------------------------------
M19) MsgUpdate a clo      c  cs  h[a -> Locked _ pd pd*]  tp   (rp,sp)    pr  s
     ==>
     Resume               c  cs  h[a -> clo]              tp'  (rp',sp)   pr  s
               (tp',rp')  = decPC tp rp pd pd*
------------------------------------------------------------------------------
```

10.2 Remote evaluating

If an out indirection node is attempted to be evaluated by entering, the node is locked and a relevant message is sent to trigger an evaluation of the remote closure which is pointed by the out indirection node.

When an *OutInd* closure is entered via *Enter* or *EnterAlloc*, a message is sent as described above, the current task is blocked, its pending count is increased and its name is placed in the pending list of the *Locked* closure which was a *OutInd* node.

```
----------------------------------------------------------------------------------
54) Enter a v  cs   h[a -> OutInd ex pid' a']  tp[t -> R pc pc*]         wp  pr  s
                cs v = t v' as rs
    ==>
    GetTask    cs   h[a -> Locked rt {t} {}]  tp[t -> B(a,v) pc+1 pc*]   wp  pr  s
                sendMessage pid' (MsgBSpark a' {(pid,a)})
                rt is a distinguished name to represent remote task.
                pid is the current processor's name.
----------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------
55) EnterAlloc a v    cs   h[a -> OutInd ex pid' a']  tp[t -> R pc pc*]         wp  pr  s
            cs v = t v' as rs
    ==>
    GetTask         cs   h[a -> Locked rt {t} {}]   tp[t -> B(a,v) pc+1 pc*]   wp  pr  s
            sendMessage pid' (MsgBSpark a' {(pid,a)})
            rt is a distinguished name to represent remote task.
            pid is the current processor's name.
 ------------------------------------------------------------------------------------------
```

If an *OutInd* node is entered via *EnterISpark*, the node is locked and a message is sent to spark the closure pointed by the *OutInd* node.

```
-------------------------------------------------------------------------
56) EnterISpark a rau      cs   h[a -> OutInd pid' a']   tp   wp   pr   s
    ==>
    GetTask                cs   h[a -> Locked rt {} {}]  tp   wp   pr   s
            sendMessage pid' (MsgISpark a' (pid,a):rau)
            pid is the current processor's name.
-------------------------------------------------------------------------
-------------------------------------------------------------------------------
M20) EnterExportISpark a rau pid''  c  cs  h[a -> OutInd pid' a']  tp  wp  pr  s
     ==>
     MsgRequestTask pid''           c  cs  h[a -> Locked rt {} {}] tp  wp  pr  s
            sendMessage pid' (MsgISpark a' (pid,a):rau)
            pid is the current processor's name.
-------------------------------------------------------------------------------
```

MsgISpark places an imported *ISpark* information in the *local work pool*.

```
---------------------------------------------------------------------
M21) MsgISpark a rau      c  cs  h  tp  (rp,(bs,is,ls))         pr  s
     ==>
     Resume               c  cs  h  tp  (rp,(bs,(a,rau):is,ls))  pr  s
---------------------------------------------------------------------
```

EnterBSpark to an *OutInd* node makes the pending counts be adjusted, because a new task will be created somewhere eventually. Even a task for the *BSpark* information is not created immediately, the processor which exports the *BSparked* work regards the task as created because the *OutInd* node is locked already and waiting the result which will be evaluated by a task which will be created somewhere in the future.

```
---------------------------------------------------------------------
58) EnterBSpark a t  cs   h   tp[t -> exec pc pc*]       wp   pr   s
            h a = OutInd pid' a'
    ==>
    GetTask          cs   h'  tp[t -> exec pc-1 pc*+1]   wp   pr   s
            h' a = Locked rt {} {t}
            sendMessage pid' (MsgBSpark a' {(pid,a)})
            rt is a distinguished name to represent remote task.
            pid is the current processor's name.
---------------------------------------------------------------------
-------------------------------------------------------------------------------
M22) EnterExportBSpark a t pid'   c  cs  h  tp[t -> exec pc pc*]       wp  pr  s
            h a = OutInd pid' a'
     ==>
     MsgRequestTask pid'          c  cs  h' tp[t -> exec pc-1 pc*+1]   wp  pr  s
            h' a = Locked rt {} {t}
            sendMessage pid' (MsgBSpark a' {(pid,a)})
            rt is a distinguished name to represent remote task.
            pid is the current processor's name.
-------------------------------------------------------------------------------
```

If an imported basic spark see an out indirection node again, it is exported again with the global address of the indirection node to update with a final result. Nothing is added on pending list, because no task is created locally for the imported basic spark.

```
---------------------------------------------------------------------
59) EnterBSpark a rau  cs   h   tp   wp   pr   s
            h a = OutInd pid' a'
    ==>
    GetTask            cs   h'  tp   wp   pr   s
            h' a = Locked rt {} {}
            sendMessage pid' (MsgBSpark a' (pid,a):rau)
            rt is a distinguished name to represent remote task.
            pid is the current processor's name.
---------------------------------------------------------------------
```

```
---------------------------------------------------------------------
M23) EnterExportBSpark a rau pid'   c  cs   h   tp   wp   pr   s
            h a = OutInd pid' a'
     ==>
     MsgRequestTask pid'            c  cs   h'  tp   wp   pr   s
            h' a = Locked rt {} {}
            sendMessage pid' (MsgBSpark a' (pid,a):rau)
        rt is a distinguished name to represent remote task.
            pid is the current processor's name.
---------------------------------------------------------------------
```

MsgBSpark places an imported *BSparked* work in the *local work pool.*

```
---------------------------------------------------------------------
M24) MsgBSpark a rau    c  cs  h  tp  (rp,(bs,is,ls))         pr  s
     ==>
     Resume             c  cs  h  tp  (rp,((a,rau):bs,is,ls))  pr  s
---------------------------------------------------------------------
```

11 Printing mechanism

We now come to the rules for the *printing mechanism*. The first of these, *EnterPrint*, causes a closure (normally assumed to be a constructor) to be evaluated and printed on the *output*. In the case when the closure is already a constructor application in WHNF, an opening bracket and the constructor are sent to the output stream. In addition, the arguments are pushed onto the *print stack*, along with a closing bracket.

```
--------------------------------------------------------------------------
60) EnterPrint a    cs  h  tp  wp  (ps,op)                              s
        h a = (vs \n {} -> c xs) wsf
    ==>
    PrintNext       cs  h  tp  wp  (xs'++{RBRACK}++ps,op++{LBRACK}++c)  s
        xs' = val [vs -> wsf] s xs
--------------------------------------------------------------------------
```

In the case that the closure is not already a WHNF constructor, a new context frame is created with the *print–result* continuation, and the closure is entered in the new context.

```
---------------------------------------------------------------------
61) EnterPrint a    cs                  h  tp            wp  pr  s
    ==>
    Enter a v       cs[v -> t _ {} {P}]  h  tp[t -> R 0]  wp  pr  s
        v is not in domain cs
        t is not in domain tp
---------------------------------------------------------------------
```

PrintNext simply causes the next item on the print stack to be sent to the *output*. This is trivial in the case of a closing bracket or a basic value.

```
---------------------------------------------------------------------
62) PrintNext       cs   h   tp   wp   (RBRACK:ps,op)      s
    ==>
    PrintNext       cs   h   tp   wp   (ps,op++{RBRACK})   s
---------------------------------------------------------------------
```

```
----------------------------------------------------------------------
63) PrintNext      cs   h   tp   wp   (Ink k:ps,op)      s
    ==>
    PrintNext      cs   h   tp   wp   (ps,op++{Int k})   s
----------------------------------------------------------------------
```

If the value at the top of the *print stack* is a closure address, we simply enter the *EnterPrint* state with that address.

```
----------------------------------------------------------------------
64) PrintNext      cs   h   tp   wp   (Addr a:ps,op)   s
    ==>
    EnterPrint a   cs   h   tp   wp   (ps,op)          s
----------------------------------------------------------------------
```

The next rule for *PrintNext* is the case when there are no entries on the *print stack*. This is the termination case for the machine as if there is nothing left to print, there is no more useful work to do.

```
----------------------------------------------------------------------
65) PrintNext      cs   h   tp   wp   ({},op)   s
    ==>
    STOP           cs   h   tp   wp   ({},op)   s
----------------------------------------------------------------------
```

Finally, we shall consider the case when the continuation is a *print–result* continuation in the *ReturnCon* phase. The machine enters the *PrintNext* phase, after discarding the context frame and task. In this rule, the entry point to *PrintNext* doesn't seem to be found at run time. Fortunately, an entry table whose key is the constructor's name can be built at compile time and the relevant entry point of the *PrintNext* can be searched by the constructor's name.

```
-----------------------------------------------------------------------------------------------
66) ReturnCon c xs v  cs[v -> t _ {} {P}] h tp[t -> R pc]  wp (ps,             op)             s
    ==>
    PrintNext         cs                  h tp[t -> T pc]  wp (xs++{RBRACK}++ps,op++{LBRACK}++c) s
-----------------------------------------------------------------------------------------------
```

12 Conclusion

A complete set of transition rules are defined to explain the semantics of the *ν–STG language*. We now have an executable specification of the rules, which is implemented in Haskell(actually Gofer[1]) To investigate the *ν–STG* machine more realistically, we defined an intermediate level language which treats the memory in a word level. A primitive version of a simulator for the intermediate language is built and a few hand-translated serial programs are tested successfully.

Acknowledgements

We would like to thank Dr. David Lester and Julian Seward for many useful discussion. The first author wish to express thanks to the Korean Government for the financial support.

References

[Augustsson and Johnsson] Lennart Augustsson and Thomas Johnsson, "Parallel Graph Reduction with the $\langle \nu, G \rangle$–machine", *ACM, IFIP Conference on Functional Programming Languages and Computer Architecture*, September 1989.

[Burn] G L Burn, "A Shared Memory Parallel G–machine Based on the Evaluation Transformer Model of Computation", In *Workshop on Implementation of Lazy Functional Language, Aspenas*, 5–8 September 1988.

[Kingdon etc] Huge Kingdon, David Lester and Geoffrey Burn, "The HDG–machine: a highly distributed graph–reducer for a transputer network", The Computer Journal, Vol. 34, No. 4, 1991.

[1] Gofer is a non–strict functional language interpreter written by Mark P. Jones in Oxford University.

[Lester etc] David R. Lester and Geoffrey L. Burn, "An Executable Specification of the HDG–Machine", In *Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy,* 9–15 October 1989.

[Peyton Jones] Simon L Peyton Jones, "Implementing lazy functional language on stock hardware: the Spineless Tagless G-machine", *to appear in the Journal of Functional Programming*, 1992.