To appear in the *Proceedings of the 1995 Glasgow Workshop on Functional Programming*, Springer-Verlag, 1996

Integrating Multithreading into the Spineless Tagless G-machine

Manuel M.T. Chakravarty*
Software Engineering Research Group
Technical University Berlin

Abstract

To reduce the adverse effects of long latency communication operations in distributed implementations of the Spineless Tagless G-machine (STGM), a variant of the original abstract machine that contains explicit support for multithreading is introduced. In particular, source-to-source transformations can be used on the level of the abstract machine code to foster the tolerance to long latency communication.

The changes to the original STG-language include a separation of demand from case selection together with the introduction of a new construct that provides an abstract notion of thread boundaries and thread synchronization.

1 Introduction

A static mapping of the components of a parallel program to the physical processor elements of a parallel computer such that these components communicate in a regular fashion is only possible for restricted programs or programs that contain explicit layout information, e.g., [Kel89, DP93]. In contrast, distributed implementations of general lazy functional programs issue demands for remote data in a dynamic and unpredictable fashion; the resulting long latency communication operations have adverse effects on distributed implementations of abstract machines, such as the Spineless Tagless G-machine (STGM) [PS88, Pey92] with its implementations for GRIP [PCS89] and GUM [THJ+96].

The subject of this paper is a variant of the STGM that is designed to reduce the impact of long latency communication on the execution time; to this end, it exploits the inherent fine-grain parallelism contained in functional programs by employing multithreading and supporting stateless threads. A special feature of the new abstract machine, called STGM_{MT}, is that source-to-source transformations can be used on the level of the abstract machine code to foster the tolerance to long latency communication.

Section 2 introduces the basic ideas underlying the development of the STGM_{MT}. Section 3 describes the changes to the original machine language of the STGM, and Section 4 demonstrates the use of the new constructs. Then in Section 5, the operational semantics of the new machine language is presented. Related work is discussed in Section 6, and Section 7 contains the conclusions.

*E-mail: chak@cs.tu-berlin.de, URL: http://www.cs.tu-berlin.de/~chak/

2 The Use of Multithreading

We assume a distributed implementation of the STGM where on each of the many processor elements (PEs) a number of *tasks* are running interleaved. When the active task of one of these processors attempts the evaluation of the expression $(a + b) * (c + d)$, it may start by accessing the value of the variable a. If the value of a is stored in the memory of another PE, a remote access is triggered (by entering a FETCHME closure; cf. [Pey92]). When such a long latency communication operation stalls the progress of a task, it is common practice to suspend that task and, instead, to execute some other, runnable task. Such a task switch includes an expensive context switch where the current values of some processor registers have to be saved and other values have to be loaded; furthermore, the data in the processor cache is probably useless for the computation performed by the newly selected task.

Crucial for the techniques developed in this paper is, returning to the example, the observation that the evaluation of the subexpression $c + d$ is (i) independent from the result of the remote access induced by a and (ii) can be achieved with less context switching costs than switching to an arbitrary runable task. The context needed to evaluate $c + d$ is similar to the context needed to evaluate $a + b$, in particular, both computations are part of the same function activation.

Across function boundaries, a similar observation can be made. When the result of a function activation depends on a long latency operation and there are no more computations in this function activation that are independent from the long latency operation, then it is usually less costly to switch to the execution of some independent computation in the *calling* function than to activate a completely unrelated task—consider the processor cache.

Overall, when the delay of a long latency operation has to be covered by some independent computation, take a computation whose context is as close to the current context as possible. Such a computation can be chosen by the heuristic that computations that would have been executed shortly after the current computation are closely related in their context—an assumption justified by the locality that computations usually exhibit.

The fine-grain parallelism inherent in functional programs can be used to mechanically partition a program into *threads* realizing independent computations. Within a function activation, long latency operations can then be covered by switching to any of the ready threads. The abstract machine language described below provides a construct that explicitly displays independent computations on the level of the abstract machine language. These independent computations can be implemented by *stateless* threads [CGSE93, EAL93]. The crucial feature of stateless threads is that they never suspend, i.e., they start only when all resource requirements are satisfied and execute to completion without any further need for synchronization—in essence, they represent the smallest unit of non-synchronizing computation. There is evidence that the use of stateless threads minimizes the thread switching time while simultaneously allowing to exploit the properties of the memory hierarchy [CGSE93, H$^+$95].

3 The Abstract Machine Language

Starting from the STG-language as described in [Pey92], two principal changes are required to integrate support for multithreading: demand must be separated from case selection, and an abstract notion of thread boundaries and thread synchronization is needed. In the following, a third modification will also be applied: an abstract notion of distribution is added in order to be able to observe the effects of multithreading on the abstract level of the STG-language. The variant of the STG-language that is defined in this paper is called the STG_{MT}-language.

To focus the following presentation on issues relevant to multithreading, support for built-in data types, such as integers and floating point numbers, is omitted—they can be handled in a similar way as in the original STGM, namely by an explicit treatment of unboxed values. Furthermore, the explicit top-level, which contains the global definitions, is omitted—here also, the mechanisms of the STGM can still be applied in the STGM_{MT}.

3.1 The Grammar

A definition of the grammar of the STG_{MT}-language can be found in Figure 1.

In comparison to the original STG-language, note the addition of `letrem` and `letpar`, and the fact that in a case expression the keywords `case` and `of` enclose an identifier and not an arbitrary expression. An intuition of the behaviour of the added or changed constructs is provided in the following subsections.

3.2 An Abstract Notion of Threads

In the original STGM, there is only a single kind of bindings. It associates variable names with lambda forms. In the STGM_{MT} these bindings are called *function bindings* and are produced by the nonterminal *fbind*. In addition, *value bindings*—nonterminal *vbind*—are introduced in the STGM_{MT}.

Value bindings occur only in the `letpar` construct, which has the following general form:

$$\texttt{letpar}\ v_1\texttt{\#} = e_1\ \texttt{;}\ \ldots\ \texttt{;}\ v_n\texttt{\#} = e_n\ \texttt{in}\ e$$

In contrast to the `letrec` construct (cf. [Pey92]), no closures are created, but the expressions e_1 to e_n are evaluated, and only after all results are assigned to the v_i`#`, evaluation proceeds with e. Furthermore, the v_i`#` may not occur free in e_1 to e_n.

The last restriction guarantees the independence that we required in Section 2 for computations that may be used to cover long latency operations. More precisely, it allows to evaluate e_1 to e_n in an arbitrary order without any need to synchronize on the v_i`#`. Should the evaluation of any e_i suspend due to a remote access, then it is still possible to continue the computation locally with any e_j where $j \neq i$. In short, `letpar` allows to express the independence

exp	→	`letrec` *fbinds* `in` *exp*	(mutually recursive closures)
	|	`letrem` *fbind* `in` *exp*	(remote closure)
	|	`letpar` *vbinds* `in` *exp*	(parallel demands)
	|	`case` *uvid* `of` *alts*	(selection)
	|	*vid args*	(closure evaluation)
	|	*cid env*	(return constructor)
	|	*uvid*	(return unboxed value)
fbinds	→	$fbind_1$; ... ; $fbind_n$	($n \geq 1$)
fbind	→	*vid* `=` *env* `\`π *args* `->` *exp*	(function binding)
π	→	`u`	(updatable)
	|	`n`	(not updatable)
vbinds	→	$vbind_1$; ... ; $vbind_n$	($n \geq 1$)
vbind	→	*uvid* `=` *exp*	(value binding)
alts	→	alt_1 ; ... ; alt_n ; *dft*	($n \geq 1$)
alt	→	*cid args* `->` *exp*	(case alternative)
dft	→	`default ->` *exp*	(case default)
args	→	`{`arg_1`,` ... `,` arg_n`}`	(argument variables, $n \geq 0$)
arg	→	*vid*	(boxed value)
	|	*uvid*	(unboxed value)
env	→	`{`vid_1`,` ... `,` vid_n`}`	(environment variables, $n \geq 0$)
uvid	→	*vid*`#`	(variable for unboxed values)
vid	→	**lowercase identifier**	(variable)
cid	→	**uppercase identifier**	(data constructor)

Figure 1: The grammar of the STG_{MT}-language.

of local computations on the level of the abstract machine language. Furthermore, the fact that the evaluation of the body expression e must wait for the delivery of all v_i`#` can be seen as an abstract form of synchronization barrier.

The hash marks (`#`) behind the v_i indicate that the v_i`#` store unboxed values. The treatment of unboxed values in the STGM_{MT} is related to, but not identical to the use of unboxed values in the original STGM. In particular in the original STGM, which follows [JL91], all types of boxed and unboxed values have to be explicitly introduced, but in the STGM_{MT}, there is implicitly a corresponding unboxed value for each boxed value. The coercion from boxed to unboxed and from unboxed to boxed types is made explicitly by value bindings and function bindings, respectively. For example, the expression

```
letpar v# = w in ...
```

binds to `v#` the unboxed value associated with the boxed value stored in `w`.

Conversely,

```
letrec w = {} \n {} -> v# in ...
```

boxes the unboxed value contained in `v#`. The deviation from the technique used in the STGM becomes necessary due to the fact that demand for evaluation, in the STGM_{MT}, is issued when an expression occurs in the right-hand side of a value bindings while, in the STGM, the value of an expression is only demanded when it is scrutinized by a `case` expression.

Note that unboxed variables are not allowed to be in the list of free variables of a function binding, i.e., only boxed values can be stored in the environment of closures, and that it is forbidden to use them as arguments to constructors. These restrictions can be relaxed, but here they are enforced here to make the presentation simpler.

Furthermore, the expressions appearing as right-hand sides of value bindings must not be of functional type, i.e., must not be of type $\alpha \to \beta$. This restriction corresponds to the restriction of the original STGM that says that `case` expressions must not inspect values of functional type.

3.3 Selection Without Demand

In the original STGM, `case` expressions play two roles: first, they demand the evaluation of the scrutinized expression, i.e., the expression between the keywords `case` and `of`; second, they select one of several alternatives by matching the value of the scrutinized expression against the patterns of the alternatives.

It was already mentioned that value bindings issue demands for evaluation in the STGM_{MT}, and hence, the single purpose of `case` expressions is pattern matching. Overall, we have the following correspondence:

$$\begin{array}{cc} \text{Original STGM} & \text{STGM}_{\text{MT}} \\ \texttt{case}\ e\ \texttt{of}\ alt_1\,;\ \ldots\ ;\,alt_n\,;\,dft \quad \approx & \begin{array}{l}\texttt{letpar}\ v\texttt{\#} = e \\ \texttt{in case}\ v\texttt{\#}\ \texttt{of}\ alt_1\,;\ \ldots\ ;\,alt_n\,;\,dft\end{array} \end{array} \tag{1}$$

3.4 An Abstract Notion of Distribution

In the original presentation of the STGM [Pey92], the potential distribution of the heap of the abstract machine over multiple processing elements is left implicit. To make the need for long latency operations explicit, we expose the potential for distribution in the STGM_{MT}.

To this end, the concept of a *machine instance* is introduced. Each machine instance has a *local* heap and is able to evaluate closures in its local heap independent from the other instances. When the local evaluation depends on a closure stored within another machine instance—we call this a *remote* closure—a long latency operation is triggered.[1] On the level of the abstract

[1] Note that already in the original STGM, closures are the only kind of structure in the heap.

machine code, no assumptions are made about the number of machine instances available.

The `letrem` construct specifies those closures that *may* be allocated remotely. In contrast, the closures associated with the function bindings of a `letrec` are bound to be allocated locally. To simplify matters, there may only be one binding in a `letrem` and it must not be recursive—recursion can be introduced by using a `letrec` in the right-hand side of the single function binding of the `letrem`. Furthermore, the binding of a `letrem` must not have any arguments, i.e., it has to represent a nullary function.

Overall, the STG_{MT}-language allows, even requires, to explicitly specify the *partitioning* of a parallel program, but it abstracts over its *mapping* (cf. [Fos95] for a definition of these notions).

4 Using the New Constructs

In summary, the STG_{MT}-language modifies the original STG-language in three ways: it introduces an explicit, but abstract, notion of (i) local, independent computations (`letpar`) and (ii) closures that may be allocated on a remote instance (`letrem`); and (iii) demand and selection are separated. These modifications lead to a number of interesting properties that are discussed in the following.

4.1 Generating code for the STGM_{MT}

The translation of a functional program into the STG_{MT}-language corresponds closely to the generation of code for the original STGM. The main difference is that we have to observe the correspondence stated in Equation (1) and `case` expressions that are just used for unboxing require no `case` in the STGM_{MT}, but only a `letpar`. In contrast to `letpar` expressions, `letrem` constructs are not expected to be generated automatically; instead they are generated from explicit annotations, i.e., the programmer decides which computations are coarse-grained enough to be worth the shipping to another processor element.

4.2 Covering Long Latency Operations

Following the stated scheme for code generation, only `letpar` constructs containing a single value binding are generated. Such code does not exhibit any tolerance to long latency operations. An important characteristic of the STG_{MT}-language is that simple source-to-source transformations can be used to increase this tolerance. In particular, we can apply the following transformation rule:

$$\begin{array}{l}\texttt{letpar}\ v_1\texttt{\#} = e_1 \\ \texttt{in letpar}\ v_2\texttt{\#} = e_2\ \texttt{in}\ e_3\end{array} \quad \Rightarrow \quad \begin{array}{l}\texttt{letpar} \\ \quad v_1\texttt{\#} = e_1 \\ \quad v_2\texttt{\#} = e_2 \\ \texttt{in}\ e_3\end{array} \tag{2}$$

when v_1 is not free in e_2.

In case of the example from Section 2, $(a + b) * (c + d)$, the transformation rule (2) has a dramatic effect on the corresponding STGM_{MT}-code:

```
letpar av# = a []
in letpar bv# = b []
in letpar ab# = add# [av#, bv#]
in letpar cv# = c []
in letpar dv# = d []
in letpar cd# = add# [cv#, dv#]
in mul# [ab#, cd#]
```

$\Rightarrow$

```
letpar
  ab# = letpar
          av# = a [];
          bv# = b []
        in
          add# [av#, bv#];
  cd# = letpar
          cv# = c [];
          dv# = d []
        in
          add# [cv#, dv#];
in mul# [ab#, cd#]
```

The code to the left is similar to the `case` cascade used to represent this computation in the original STGM. The transformed code, to the right, explicitly represents the independence of those subcomputations that can be used to cover long latency operations. In particularly, when demanding `a` triggers a remote access, the demand of `b` as well as the demands to `c` and `d` together with the evaluation of `add# [cv#, dv#]` can be done while waiting for the value `av#`. In the worst case, when all data is remote, at least the access to `a`, `b`, `c`, and `d` is overlapped. In essence, the code to the right is a textual representation of the partial ordering induced on the code by the data dependencies.

Overall the separation of demand and selection allows to move demand, i.e., value bindings, outwards in order to collect multiple value bindings in a single `letpar` as in the following code fragment (the y_i are not free in e_2):

```
letpar x# = e1
in case x# of
  C {y1, ..., yn} ->
    letpar z# = e2
    in case z# of ...
```

$\Rightarrow$

```
letpar
  x# = e1
  z# = e2
in case x# of
  C {y1, ..., yn} -> case z# of ...
```

Apart from data dependencies, the outwards movement of value bindings is stopped by `case` expressions with more than one alternative; moving a value binding over a `case` with multiple alternatives can change the termination behaviour of a program.

In principle, function boundaries also stop the outwards movement, but this should, for the following reason, not become a problem in practice: either the function is very simple, then it can be inlined; or it is complex and, then, it usually contains a `case` with multiple alternatives, which hinders the outward movement anyway.

4.3 Distribution

The following program fragement displays the essentials of a parallel program exploiting pipelined parallelism:

```
consume        :: [Value] -> Result
consume []      = ⟨trivial result⟩
consume (x:xs)  = let r = consume xs
                  in
                    ⟨make result from x and r⟩

main = let stream = ⟨producer of some list of Values⟩
       in
         stream `par` consume stream     — consume in parallel
```

The `par` meta-function indicates that its two arguments may be evaluated (in parallel) on different machine instances; it is compiled into a `letrem` construct in the following STGM_{MT}-code:

```
consume = {} \n {l} ->
  letpar lv# = l {} in case lv# of
    Nil  {}      -> ⟨trivial result⟩
    Cons {x, xs} -> letpar
                      xv# = x {}              — independent
                      rv# = consume {xs}      — computations
                    in
                      ⟨make result from xv# and rv#⟩

main = {} \u {} ->
  letrem                        — potential remote allocation
    stream = {...} \u {} -> ⟨producer of some list of Values⟩
  in
    consume {stream}
```

The above code for `consume` is already transformed; the original code would place the expression `consume {xs}` into the function binding of a separate `letrec`. But the immediate following occurence of the bound variable in the right-hand side of a value binding allows the transformation into the shown code.

If there are machine instances that need additional work, then closures created with `letrem` can be shipped to those instances; otherwise they can also be allocated and evaluated locally—the latter case corresponds to the idea of the absorption of previously sparked children [HMP94]. In the above example, let us assume that `stream` is allocated remotely. Then, the value of `stream` and `consume {stream}` are evaluated on different instances, in parallel—any closures created with `letrec` while evaluating `stream` are also allocated and, thus, evaluated on the remote instance. This implies that the access to `x` in the body of `consume` triggers a remote access, which is, at least partially, covered by the recursive call to `consume` (in the same `letpar`).

5 The Meaning

To formalize the operational semantics of the STG_{MT}-language, a transition system is presented in this section; it is derived from the system in [Pey92]. It makes the effects of multithreading on the abstract level of the STG_{MT}-language explicit. The notation used in this section is similar to that used in [Pey92]; details are provided in Appendix A.

5.1 Machine Configurations

A *machine configuration* is represented by a mapping $\mathcal{I}$ from names to machine instances. Each instance consists of several components, including a code component, a task pool, an argument stack, a return stack, and a heap. A detailed description of these data structures is provided in Appendix B.

The machine instances in a configuration share a global name space, but the computations within one instance i may only access the components of i. When i needs to access a closure, named o, that is located in the heap of another instance j, it has to request j to evaluate the closure o and to return the WHNF of o back to i. This operation is the single form of long latency operation in the STGM_{MT}.

In the transition rules, we assume an unbound number of instances, and each closure allocated by a `letrem` is created on a not-yet-used instance. This exposes the maximal parallelism of the program. In a concrete implementation, the scheme outlined Section 4.3 is used, i.e., closures are only distributed upon request from processing elements with an insufficient amount of work.

5.2 The Transition Rules

The following transition rules affect either one or two instances at a time. To get a parallel and not only an interleaving semantics, we define a *parallel transition step* to be a set of applications of transition rules such that this set contains at least a single element and no instance is affected by more than one transition rule. A transition rule is said to affect an instance if this instance occurs in the rules pre- or postcondition.

5.2.1 The Initial State

The initial machine state used to evaluate an expression e is the following:

inst.	current task	task pool	frames	arg. stack	ret. stack	heap	dissem. map
$[\underline{i} \mapsto$	$Eval\ e\ []$	$\emptyset$	$[]$	$[]$	$[\langle \underline{d}, Next, [] \rangle]$	$[]$	$[]]$

The configuration consists of a single instance named i. Its current task is to evaluate e within an empty environment. The task pool is empty (i.e., there is no further work), just as the frame map and argument stack. The single continuation on the return stack indicates that the result of e has to be delivered via the (non-existing) slot d of the dissemination map.

The machine terminates when it attempts to distribute some value over the dissemination slot d; this is the value computed for e.

Intuitively, the roles of the components of an instance are as follows. The task pool contains the waiting (for the completion of a long latency operation) and the ready-to-run tasks that have to be executed on this instance. At this point it is important to clearly distinguish between tasks and threads. Tasks are unrelated, coarse-grain computations that are distributed over the machine instances to gain speedup by parallel evaluation; they are indirectly introduced by the `letrem` construct. Threads are clustered into closely related groups represented by the `letpar` construct and are fine-grain computations that are used to efficiently cover long latency operations. Only when a task contains no more ready-to-run threads and it is still waiting for a long latency operation, it is suspended and placed in the task pool. Every distributed implementation of the STGM uses tasks, but threads are the uncommon feature of the STGM_{MT}.

For every `letpar` construct that is executed, a frame is created; it contains a counter storing the number of value bindings that have not been completed yet and the local environment used to store the value bindings. The argument stack has the same function as in the original STGM, but the return stack assumes the functionality of both the return and the update stack of the STGM—this is necessary to correctly deal with updates of closures whose evaluation triggered a long latency operation. The heap is used to store closures—just as in the original STGM. Finally, the dissemination map supports the dissemination of the results of long latency operations to multiple receivers.

5.2.2 Applications

Execution of the application of a function to some arguments, pushes the arguments on the stack and enters the closure that represents the function. In contrast, the application of a data constructor, initiates a return operation.

$$\begin{array}{lllllllll} & \mathcal{I}[i \mapsto & \mathit{Eval}\ (f\ \{\overline{x_N}\})\ \rho & \Gamma & \mathit{fs} & & \mathit{as} & \mathit{rs} & \mathit{h} & \mathit{ds}] \\ \overset{(1)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Enter}\ (\rho\ f) & \Gamma & \mathit{fs} & [\overline{\rho\ x_N}] +\!\!+ & \mathit{as} & \mathit{rs} & \mathit{h} & \mathit{ds}] \end{array}$$

$$\begin{array}{lllllllll} & \mathcal{I}[i \mapsto & \mathit{Eval}\ (c\ \{\overline{x_N}\})\ \rho & \Gamma & \mathit{fs} & \mathit{as} & \mathit{rs} & \mathit{h} & \mathit{ds}] \\ \overset{(2)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{RetTerm}\ \langle c, [\overline{\rho\ x_N}]\rangle & \Gamma & \mathit{fs} & \mathit{as} & \mathit{rs} & \mathit{h} & \mathit{ds}] \end{array}$$

We use $\overline{\cdot}$ to indicate repetition, e.g., $[\overline{\rho\ x_N}]$ stands for $[\rho\ x_1, \ldots, \rho\ x_N]$

Evaluating an unboxed variable returns the unboxed value represented by this variable in the local environment ρ.

$$\begin{array}{lllllllll} & \mathcal{I}[i \mapsto & \mathit{Eval}\ x\#\ \rho & \Gamma & \mathit{fs} & \mathit{as} & \mathit{rs} & \mathit{h} & \mathit{ds}] \\ \overset{(3)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{RetTerm}\ (\rho\ x) & \Gamma & \mathit{fs} & \mathit{as} & \mathit{rs} & \mathit{h} & \mathit{ds}] \end{array}$$

5.2.3 Entering a Closure

A not updatable closure is entered by evaluating its code under an environment built from the closures arguments and the appropriate number of parameters from the argument stack. The body γ of the closure is a function that applied

to the environment ρ yields the code form that has to be executed. The environment ρ is constructed by taking *length xs* arguments from the stack and associating the free variables *vs* with the environment *eos* of the closure.

$$\begin{array}{lllllllll} & \mathcal{I}[i \mapsto & \mathit{Enter}\ o & \Gamma & \mathit{fs} & \mathit{as} & \mathit{rs} & h[o \mapsto \langle(\mathit{vs}\ \backslash n\ \mathit{xs} \to \gamma), \mathit{eos}\rangle] & \mathit{ds}] \\ \stackrel{(4)}{\Longrightarrow} & \mathcal{I}[i \mapsto & (\gamma\ \rho) & \Gamma & \mathit{fs} & \mathit{as'} & \mathit{rs} & h & \mathit{ds}] \end{array}$$

where

$$\begin{array}{lcl} \mathit{aos} +\!\!+ \mathit{as'} & = & \mathit{as}, \text{ such that } \mathit{length}\ \mathit{aos} = \mathit{length}\ \mathit{xs} \\ \rho & = & [\mathit{xs} \mapsto \mathit{aos}] +\!\!+ [\mathit{vs} \mapsto \mathit{eos}] \end{array}$$

Updatable closures are always nullary (cf. [Pey92]). In the original STGM, such closures push an update frame; in the STGM_{MT}, they create an *Upd* dissemination entry—as soon as a value is passed to this entry, the closure is updated with this value. Depending on the type of the value that is computed by the closure, we distinguish two cases: first, if the type is non-functional it is sufficient to extend the dissemination entry referenced by the topmost return continuation; second, if the type is functional, the closure has to be reentered after the update, i.e., a return continuation initiating the reentering is pushed and a new slot d is created in the dissemination map. Note that in the first case, the argument stack is guaranteed to be empty in type correct programs.

$$\mathcal{I}[i \mapsto \ \mathit{Enter}\ o\ \ \Gamma\ \ \mathit{fs}\ \ []\ \ \langle d, \mathit{cont}, \mathit{as}_p\rangle{:}\mathit{rs}\ \ h\ \ \mathit{ds}[d \mapsto \mathit{ms}]]$$

when $h\ o = \langle(\mathit{vs}\ \backslash u\ [] \to \gamma), \mathit{eos}\rangle$ and has non-functional type

$$\stackrel{(5)}{\Longrightarrow}\ \mathcal{I}[i \mapsto\ (\gamma\ \rho)\ \ \Gamma\ \ \mathit{fs}\ \ []\ \ \langle d, \mathit{cont}, \mathit{as}_p\rangle{:}\mathit{rs}\ \ h\ \ \mathit{ds}[d \mapsto (\mathit{Upd}\ o){:}\mathit{ms}]]$$

where

$$\rho\ \ =\ \ [\mathit{vs} \mapsto \mathit{eos}]$$

$$\mathcal{I}[i \mapsto\ \mathit{Enter}\ o\ \ \Gamma\ \ \mathit{fs}\ \ \mathit{as}\ \ \mathit{rs}\ \ h\ \ \mathit{ds}]$$

when $h\ o = \langle(\mathit{vs}\ \backslash u\ [] \to \gamma), \mathit{eos}\rangle$ and has functional type

$$\stackrel{(6)}{\Longrightarrow}\ \mathcal{I}[i \mapsto\ (\gamma\ \rho)\ \ \Gamma\ \ \mathit{fs}\ \ []\ \ \langle d, (\mathit{Enter}\ o), \mathit{as}\rangle{:}\mathit{rs}\ \ h\ \ \mathit{ds}[\underline{d} \mapsto [\mathit{Upd}\ o]]]$$

where

$$\rho\ \ =\ \ [\mathit{vs} \mapsto \mathit{eos}]$$

Finally, entering a (not updatable) closure needing more than the available arguments indicates that a partial application has to be passed to the topmost return continuation, i.e., the partial application must be distributed using the dissemination slot d referenced by the return continuation. This case occurs when either a thunk (cf. [Pey92]) has to be updated with a partial application or a partial application has to be communicated to a remote instance.

$$\mathcal{I}[i \mapsto\ \mathit{Enter}\ o\ \ \Gamma\ \ \mathit{fs}\ \ \mathit{as}\ \ \langle d, \mathit{cont}, \mathit{as}_p\rangle{:}\mathit{rs}\ \ h\ \ \mathit{ds}]$$

when $h\ o = \langle(\mathit{vs}\ \backslash n\ \mathit{xs} \to \gamma), \mathit{eos}\rangle$ and $\mathit{length}\ \mathit{as} < \mathit{length}\ \mathit{xs}$

$$\stackrel{(7)}{\Longrightarrow}\ \mathcal{I}[i \mapsto\ (\mathit{MsgPAPP}\ d\ o_p\ \mathit{cont})\ \ \Gamma\ \ \mathit{fs}\ \ \mathit{as}_p\ \ \mathit{rs}\ \ h'\ \ \mathit{ds}]$$

where

$$\begin{array}{lcl} \mathit{xs}_1 +\!\!+ \mathit{xs}_2 & = & \mathit{xs}, \text{ such that } \mathit{length}\ \mathit{xs}_1 = \mathit{length}\ \mathit{as} \\ h' & = & h[\underline{o_p} \mapsto \langle((\underline{f}{:}\mathit{xs}_1)\ \backslash n\ [] \to \mathit{Eval}\ (f\ \mathit{xs}_1)), o{:}\mathit{as}\rangle] \end{array}$$

A new closure, named o_p, is created, which implements the partial application; its structure corresponds to the representation of partial applications in the original STGM.

5.2.4 Local Bindings

`letrec` expressions behave as in the original STGM, but note the partial application of the code form *Eval* in the body of the new closures.

$$\begin{array}{lllllllll}
 & \mathcal{I}[i \mapsto & \mathit{Eval} \left(\begin{array}{l} \texttt{letrec} \\ \quad \overline{x_N \texttt{ = } vs_N \setminus \pi_N \; xs_N \texttt{ -> } e_N} \\ \texttt{in } e \end{array} \right) & \rho & \Gamma & fs & as & rs & h \quad ds] \\
\stackrel{(8)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Eval}\ e\ \rho' & & \Gamma & fs & as & rs & h' \quad ds]
\end{array}$$

where

$$\begin{array}{lcl}
h' & = & h[\overline{o_N \mapsto \langle (vs_N \setminus \pi_N \; xs_N \rightarrow (\mathit{Eval}\ e_N)), \rho'\ vs_N \rangle}] \\
\rho' & = & \rho[\overline{x_N \mapsto o_N}]
\end{array}$$

Evaluating a `letrem` expression allocates a closure on a new instance k. Additional forwarding closures that contain *EnterOn* code forms are used for two purposes: first, to reference the new closure o_k that is allocated on the new instance k from the current instance i and, second, to reference the closures (ρv_N) that are contained in the environment of the new closure but are located on the current instance i.

$$\begin{array}{lllllllll}
 & \mathcal{I}[i \mapsto & \mathit{Eval} \left(\begin{array}{l} \texttt{letrem} \\ \quad x \texttt{ = \{} \overline{v_N} \texttt{\} } \setminus \pi \texttt{ \{\} -> } e_1 \\ \texttt{in } e_2 \end{array} \right) & \rho & \Gamma_i & fs_i & as_i & rs_i & h_i \quad ds_i] \\
\stackrel{(9)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Eval}\ e_2\ \rho[x \mapsto o] & & \Gamma_i & fs_i & as_i & rs_i & h'_i \quad ds_i, \\
 & \underline{k} \mapsto & \mathit{Next} & & [] & [] & [] & [] & h_k \quad []]
\end{array}$$

where

$$\begin{array}{lcl}
h'_i & = & h_i[o \mapsto \langle ([\underline{v}] \setminus u\ [] \rightarrow \mathit{EnterOn}\ k\ v), o_k \rangle] \\
h_k & = & \left[\begin{array}{l} o_k \mapsto \langle ([\overline{v_N}] \setminus \pi\ [] \rightarrow (\mathit{Eval}\ e_1)), [\overline{o_N}] \rangle \\ \hline o_N \mapsto \langle ([\underline{v'}] \setminus u\ [] \rightarrow \mathit{EnterOn}\ i\ v'), \rho\ v_N \rangle \end{array} \right]
\end{array}$$

In Section 4.3, `letrem` was used to implement the meta-function `par`. Following the definition of `par` in [HMP94], the closure allocated on the remote instance k must be evaluated immediately. To achieve this behaviour, the initial code form of k must be $(\mathit{Enter}\ o)$ instead of *Next*.

`letpar` constructs specify related, but independent work; furthermore, the the evaluation of the body expression has to be synchronized with the delivery of the values demanded in the value bindings. To this end, the code form *Sync* together with a new frame f are employed. The first argument of *Sync* contains the still to be evaluated value bindings. The frame maintains a counter of the number of value bindings whose value was not yet added to the environment that is also held in the frame. Note that the number of still awaited values will be greater than the number of bindings in the *Sync* form when the computation of some values is hindered by long latency operations.

$$\begin{array}{lllllllll}
 & \mathcal{I}[i \mapsto & \mathit{Eval}\ (\texttt{letpar}\ b_1\texttt{;}\ \ldots\texttt{;}b_n\ \texttt{in}\ e)\ \rho & \Gamma & \mathit{fs} & \mathit{as} & \mathit{rs} & h & \mathit{ds}] \\
\stackrel{(10)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Sync}\ [b_1,\ldots,b_n]\ e\ f & \Gamma & \mathit{fs}' & \mathit{as} & \mathit{rs} & h & \mathit{ds}]
\end{array}$$

where

$$\mathit{fs}' \;=\; \mathit{fs}[\underline{f} \mapsto \langle n, \rho\rangle]$$

In a concrete implementation, the guaranteed independence of the value bindings within one `letpar` can be used to partition the code generated from an $\mathrm{STG}_{\mathrm{MT}}$-program into non-synchronizing threads, i.e., stateless threads.

If there are unprocessed value bindings in a *Sync*, one is selected and its right hand side e_1 is evaluated. A new return continuation is pushed on the stack; it contains the remaining part of the *Sync* code form and the values currently on the argument stack. The new slot d in the dissemination map is used to eventually distribute the result of e_1. The dissemination entry $(\mathit{Store}\ f\ x)$ indicates that the result has to be stored in the environment of frame f with the local name x.

$$\begin{array}{lllllllll}
 & \mathcal{I}[i \mapsto & \mathit{Sync}\ ((x\texttt{\#} = e_1)\texttt{:}bs)\ e_2\ f & \Gamma & \mathit{fs}[f \mapsto \langle n, \rho\rangle] & \mathit{as} & \mathit{rs} & h & \mathit{ds}] \\
\stackrel{(11)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Eval}\ e_1\ \rho & \Gamma & \mathit{fs} & [] & \mathit{rs}' & h & \mathit{ds}']
\end{array}$$

where

$$\begin{array}{lcl}
\mathit{rs}' & = & \langle d, (\mathit{Sync}\ bs\ e_2\ f), \mathit{as}\rangle\texttt{:}\mathit{rs} \\
\mathit{ds}' & = & \mathit{ds}[\underline{d} \mapsto [\mathit{Store}\ f\ x]]
\end{array}$$

If there are no more unprocessed value bindings in a *Sync* form, the behaviour depends on the value of the synchronization counter in the associated frame f. If it is zero, all values are available, and the body expression can be evaluated; otherwise, the evaluation of the body form has to wait for the delivery of the remaining values, but there is no more independent work in the `letpar` that created this *Sync* form. Nevertheless, it is usually not necessary to suspend the current task; there may be further work in textually enclosing `letpar`s or in the calling function. In order to utilize such work the code form *RetDelay* is used. The evaluation of the expression e is deferred to a new task *cont* that is activated only after the long latency operation is completed.

$$\begin{array}{lllllllll}
 & \mathcal{I}[i \mapsto & \mathit{Sync}\ []\ e\ f & \Gamma & \mathit{fs}[f \mapsto \langle 0, \rho\rangle] & \mathit{as} & \mathit{rs} & h & \mathit{ds}] \\
\stackrel{(12)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Eval}\ e\ \rho & \Gamma & \mathit{fs} & \mathit{as} & \mathit{rs} & h & \mathit{ds}]
\end{array}$$

$$\mathcal{I}[i \mapsto \quad \mathit{Sync}\ []\ e\ f \qquad \Gamma \qquad \mathit{fs}\ \ \mathit{as}\ \ \mathit{rs}\ \ h \qquad \mathit{ds}]$$

when $\mathit{fs}\ f = \langle n, \rho\rangle$ and $n > 0$

$$\stackrel{(13)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \quad \mathit{RetDelay}\ \underline{x}\ [x \mapsto d] \quad \Gamma.\mathit{cont} \quad \mathit{fs}\ \ \mathit{as}\ \ \mathit{rs}\ \ h \quad \mathit{ds}[\underline{d} \mapsto []]]$$

where

$$\mathit{cont} \;=\; \langle \mathit{Sync}\ []\ e\ f, \langle d, \mathit{Next}, []\rangle\rangle$$

The form *RetDelay* informs the enclosing computation about the fact that a long latency operation delayed the delivery of the requested value and that no more local work is available. This information is propagated through all the return continuations until a *Sync* form is found that has some work. When the delayed value eventually becomes available it is distributed using the new slot d of the dissemination map.

5.2.5 *Remote Method Invocation*

In the STGM_{MT} as in the original STGM, accessing a data structure means to evaluate a closure. Hence, the code form *EnterOn* represents a remote data access; it initiates the evaluation of the closure on the remote instance k by placing a new task *enter* into the task pool Γ_k. This task eventually distributes the result of the remote computation using the dissemination slot d_k, which forwards it to the slot d_i on the instance i that initiated the whole process.

Note that the time between the execution of an *EnterOn* and the delivery of its result is, in general, unbound. Hence, it is important that the instance i can do some useful work while waiting for the delivery of the result. The delay induced by the remote access is signaled with *RetDelay* to the enclosing computation. If the return continuation is a *Sync* form that still has some unevaluated value bindings left, it can continue by evaluating one of these bindings as they do not depend on the delayed value.

$$
\begin{array}{rlllllllll}
 & \mathcal{I}[i \mapsto & \mathit{EnterOn}\ k\ v\ \rho & \Gamma_i & fs_i & as_i & rs_i & h_i & ds_i, \\
 & k \mapsto & \mathit{task} & \Gamma_k & fs_k & as_k & rs_k & h_k & ds_k] \\
\overset{(14)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{RetDelay}\ \underline{x}\ [x \mapsto d_i] & \Gamma_i & fs_i & as_i & rs_i & h_i & ds_i', \\
 & k \mapsto & \mathit{task} & \Gamma_k.\mathit{enter} & fs_k & as_k & rs_k & h_k' & ds_k']
\end{array}
$$

where

$$
\begin{array}{lcl}
\mathit{enter} & = & \langle \mathit{Eval}\ v\ [v \mapsto \rho\ v], \langle d_k, \mathit{Next}, [] \rangle \rangle \\
ds_i' & = & ds_i[\underline{d_i} \mapsto []] \\
ds_k' & = & ds_k[\underline{d_k} \mapsto [\mathit{RetTo}\ i\ d_i]]
\end{array}
$$

5.2.6 *Selection*

A `case` selects the appropriate alternative on the basis of the scrutinized value.

$$
\begin{array}{rlllllllll}
 & \mathcal{I}[i \mapsto & \mathit{Eval} \left(\begin{array}{l} \texttt{case}\ x\texttt{\#}\ \texttt{of} \\ \quad \ldots\ ; c\ \texttt{\{}\overline{v_N}\texttt{\}}\ \texttt{->}\ e\ ;\ \ldots \end{array} \right) \rho & \Gamma & fs & as & rs & h & ds] \\
\text{when} & \rho\ x = \langle c, [\overline{w_N}] \rangle & & & & & & & \\
\overset{(15)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Eval}\ e\ \rho[\overline{v_N \mapsto w_N}] & \Gamma & fs & as & rs & h & ds]
\end{array}
$$

$$
\begin{array}{rlllllllll}
 & \mathcal{I}[i \mapsto & \mathit{Eval} \left(\begin{array}{l} \texttt{case}\ x\texttt{\#}\ \texttt{of} \\ \quad \overline{c_N\ vs_N\ \texttt{->}\ e_N}\ ;\ \texttt{default}\ \texttt{->}\ e \end{array} \right) \rho & \Gamma & fs & as & rs & h & ds] \\
\text{when} & \rho\ x = \langle c, ws \rangle \text{ and } \overline{c \neq c_N} & & & & & & & \\
\overset{(16)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Eval}\ e\ \rho & \Gamma & fs & as & rs & h & ds]
\end{array}
$$

If no alternative matches, a fatal failure occurs.

5.2.7 *Returning a Proper Value*

Returning the unboxed form of a data term implies to distribute it using the dissemination slot d referenced by the topmost return continuation; afterwards, the continuation is executed. Returning a data term, the argument stack always is empty for type correct programs.

$$\begin{array}{lllllllll}
\mathcal{I}[i \mapsto & \mathit{RetTerm}\ \langle c, ws\rangle & \Gamma & fs & [] & \langle d, cont, as_p\rangle{:}rs & h & ds] \\
\stackrel{(17)}{\Longrightarrow} \mathcal{I}[i \mapsto & (\mathit{MsgTerm}\ d\ \langle c, ws\rangle\ cont) & \Gamma & fs & as_p & rs & h & ds]
\end{array}$$

5.2.8 Returning a Delayed Value

When the code form *RetDelay* is executed, a long latency operation delayed the delivery of some intermediate result and local, but independent, computations should be employed to cover the delay, i.e., not yet evaluated value bindings in surrounding `letpar`s should be executed. For this mechanism to work properly, two jobs have to be carried out: first, some independent work has to be found and, second, when the delayed value finally arrives, it has to be introduced into the ongoing computation.

Imagine a *Sync* code form with multiple value bindings. When the first binding is evaluated, according to Rule (11), a return continuation is pushed that contains the *Sync* form with the remaining bindings. To utilize the independent work constituted by these bindings, we just have to return to this *Sync*. This is what *RetDelay* does while simultaneously taking care of the second issue, namely, preparing the asynchronous delivery of the delayed value. The latter is done by placing *Fwd* entries in the dissemination slot $(\rho\ v)$ that will eventually be used to deliver the remote value; as a result the asynchronous delivery updates closures that have to be updated with the remote value and it stores the value in frames whose associated *Sync* forms wait for that value.

When the topmost return continuation belongs to a closure that has to be updated (this is handled by the "if" in the rule below), this closure has to be overwritten with a new closure that contains a *RetDelay* code form. This ensures that repeatedly entering the closure does not cause multiple remote accesses; instead, remote accesses are shared.

When there are no arguments on the stack, the delayed value can be forwarded to the entry specified by the return continuation by using a *Fwd* entry, which sends any value delivered via this entry on to the entry given in its argument (in the following rule, to d).

$$\begin{array}{lllllllll}
\mathcal{I}[i \mapsto & \mathit{RetDelay}\ v\ \rho & \Gamma & fs & [] & rs & h & & ds[\rho\ v \mapsto ms]] \\
\multicolumn{9}{l}{\text{when} \quad rs = \langle d, cont, as_p\rangle{:}rs'} \\
\stackrel{(18)}{\Longrightarrow} & \mathcal{I}[i \mapsto & cont & \Gamma & fs & as_p & rs' & h' & ds[\rho\ v \mapsto (\mathit{Fwd}\ d){:}ms]] \\
\multicolumn{9}{l}{\text{where}} \\
\multicolumn{9}{l}{h' \quad = \quad \text{if } (\mathit{Enter}\ o_u) = cont} \\
\multicolumn{9}{l}{\qquad\quad \text{then } h[o_u \mapsto \langle([\underline{v'}] \backslash u\ [] \rightarrow \mathit{RetDelay}\ v'), \rho\ v\rangle]} \\
\multicolumn{9}{l}{\qquad\quad \text{else } h}
\end{array}$$

The above rule together with Rule (13) implies that the continuations on the return stack are popped in the process of exploiting work that is independent from the delayed value. Only when the stack is completely empty, it is necessary to switch to a completely unrelated task. An interesting consequence

of this property is that only one stack per machine instance is needed in the STGM_{MT}—instead of one stack per task.

If there are arguments remaining on the stack, the delayed value is a partial application, which has to be applied to the arguments on the stack when it it eventually delivered. To realize the synchronization between the delivery of the partial application via the dissemination entry $(\rho\, v)$ and the code performing the application, a new frame f and a task *enter* are created. The form *Store* in the dissemination entry is used to put the partial application into the frame f when it arrives; this, in turn, enables the task *enter*.

$$\begin{array}{lllllllll}
& \mathcal{I}[i \mapsto & \mathit{RetDelay}\ v\ \rho & \Gamma & fs & as & rs & h & ds[\rho\, v \mapsto ms]] \\
\text{when} & rs = \langle d, \mathit{cont}, as_p \rangle{:}rs' & & & & & & & \\
\overset{(19)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{cont} & \Gamma.\mathit{enter} & fs' & as_p & rs' & h' & ds']
\end{array}$$

where

$$\begin{array}{lcl}
h' & = & \text{if } (\mathit{Enter}\ o_u) = \mathit{cont} \\
 & & \text{then } h[o_u \mapsto \langle ([\underline{v}] \setminus u\ [] \to \mathit{RetDelay}\ v), \rho\, v \rangle] \\
 & & \text{else } h \\
fs' & = & fs[\underline{f} \mapsto (1, [\underline{vs} \mapsto as])] \\
\mathit{enter} & = & (\mathit{Sync}\ []\ (x\ vs)\ f, \langle d, \mathit{Next}, [] \rangle) \\
ds' & = & ds[\rho\, v \mapsto (\mathit{Store}\ f\ \underline{x}){:}ms]
\end{array}$$

As mentioned above, an empty return stack indicates that the current task has no more work to offer.

$$\begin{array}{lllllllll}
& \mathcal{I}[i \mapsto & \mathit{RetDelay}\ v\ \rho & \Gamma & fs & [] & [] & h & ds] \\
\overset{(20)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{Next} & \Gamma & fs & [] & [] & h & ds]
\end{array}$$

5.2.9 Task Management

Next passes control to some arbitrary task from the task pool. The return continuation of the new task is placed on the stack, then the task is executed.

$$\begin{array}{lllllllll}
& \mathcal{I}[i \mapsto & \mathit{Next} & \Gamma.(\mathit{task}, r) & fs & [] & [] & h & ds] \\
\overset{(21)}{\Longrightarrow} & \mathcal{I}[i \mapsto & \mathit{task} & \Gamma & fs & [] & [r] & h & ds]
\end{array}$$

A concrete implementation would, of course, ensure that the selected task is ready to run, e.g., by maintaining a list of such tasks; otherwise, the selected task may just suspended again immediately.

5.2.10 Dissemination of Messages

The distribution of unboxed data terms is performed by considering one entry of the dissemination slot after the other and use the value accordingly to store it into the environment of a frame (*Store*), update a closure (*Upd*), forward it to another dissemination slot (*Fwd*), or send it to another instance (*RetTo*).

$$\begin{array}{llllllllll}
\mathcal{I}[i \mapsto & \mathit{MsgTerm}\ d\ \langle c, ws \rangle\ \mathit{task} & \Gamma & fs & as & rs & h & ds[d \mapsto []]] \\
\mathcal{I}[i \mapsto & \mathit{task} & \Gamma & fs & as & rs & h & ds]
\end{array}$$

$$\mathcal{I}[i \mapsto \quad MsgTerm\ d\ \langle c, ws\rangle\ task \quad \Gamma \quad fs \quad as \quad rs \quad h \qquad\qquad ds]$$

when $\quad fs\ f = \langle m, \rho\rangle$ and $ds\ d = (Store\ f\ x){:}ms$

$\overset{(23)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \quad MsgTerm\ d\ \langle c, ws\rangle\ task \quad \Gamma \quad fs' \quad as \quad rs \quad h \quad ds[d \mapsto ms]]$

where

$fs' \quad = \quad fs[f \mapsto \langle m-1, \rho[x \mapsto \langle c, ws\rangle]\rangle]$

$$\mathcal{I}[i \mapsto \quad MsgTerm\ d\ \langle c, [\overline{w_N}]\rangle\ task \quad \Gamma \quad fs \quad as \quad rs \quad h \qquad\qquad ds]$$

when $\quad ds\ d = (Upd\ o_u){:}ms$

$\overset{(24)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \quad MsgTerm\ d\ \langle c, [\overline{w_N}]\rangle\ task \quad \Gamma \quad fs \quad as \quad rs \quad h_u \quad ds[d \mapsto ms]]$

where

$h_u \quad = \quad h[o_u \mapsto \langle([\overline{v_N}]\ \backslash n\ [] \to Eval\ (c\ \{\overline{v_N}\})), [\overline{w_N}]\rangle]$

$$\mathcal{I}[i \mapsto \quad MsgTerm\ d\ \langle c, ws\rangle\ task \quad \Gamma \quad fs \quad as \quad rs \quad h \qquad\qquad ds]$$

when $\quad ds\ d = (Fwd\ d'){:}ms$

$\overset{(25)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \quad MsgTerm\ d'\ \langle c, ws\rangle\ cont \quad \Gamma \quad fs \quad as \quad rs \quad h \quad ds[d \mapsto ms]]$

where

$cont \quad = \quad MsgTerm\ d\ \langle c, ws\rangle\ task$

$$\begin{array}{lllllllll} \mathcal{I}[i \mapsto & MsgTerm\ d_i\ \langle c, [\overline{w_N}]\rangle\ task_i & \Gamma_i & fs_i & as_i & rs_i & h_i & ds_i, \\ \quad k \mapsto & task_k & \Gamma_k & fs_k & as_k & rs_k & h_k & ds_k] \end{array}$$

when $\quad ds\ d_i = (RetTo\ k\ d_k){:}ms$

$$\begin{array}{lllllllll} \overset{(26)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto & MsgTerm\ d_i\ \langle c, [\overline{w_N}]\rangle\ task_i & \Gamma_i & fs_i & as_i & rs_i & h_i & ds_i', \\ \quad k \mapsto & MsgTerm\ d_k\ \langle c, [\overline{w'_N}]\rangle\ task_i & \Gamma_k & fs_k & as_k & rs_k & h_k' & ds_k] \end{array}$$

where

$ds_i' \quad = \quad ds_i[d_i \mapsto ms]$

$h_k' \quad = \quad h_k[\overline{w'_N \mapsto \langle([\underline{v}]\ \backslash u\ [] \to EnterOn\ i\ v), [w_N]\rangle}]$

In the last rule, where the value is transmitted to another instance, the arguments must be replaced by forwarding closures using the *EnterOn* code form (compare this to Rule (9)).

The distribution of partial applications is similar to that of data terms.

$$\begin{array}{lllllllll} \mathcal{I}[i \mapsto & MsgPAPP\ d\ o\ task & \Gamma & fs & as & rs & h & ds[d \mapsto []]] \\ \mathcal{I}[i \mapsto & task & \Gamma & fs & as & rs & h & ds] \end{array}$$

$$\mathcal{I}[i \mapsto \quad MsgPAPP\ d\ o\ task \quad \Gamma \quad fs \quad as \quad rs \quad h \qquad ds]$$

when $\quad fs\ f = \langle m, \rho\rangle$ and $ds\ d = (Store\ f\ x){:}ms]$

$\overset{(28)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \quad MsgPAPP\ d\ o\ task \quad \Gamma \quad fs' \quad as \quad rs \quad h \quad ds[d \mapsto ms]]$

where

$fs' \quad = \quad fs[f \mapsto \langle m-1, \rho[x \mapsto o]\rangle]$

$$\mathcal{I}[i \mapsto \quad MsgPAPP\ d\ o\ task \quad \Gamma \quad fs \quad as \quad rs \quad h \quad ds[d \mapsto (Upd\ o_u){:}ms]]$$

$\overset{(29)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \quad MsgPAPP\ d\ o\ task \quad \Gamma \quad fs \quad as \quad rs \quad h_u \qquad ds[d \mapsto ms]]$

where

$h_u \quad = \quad h[o_u \mapsto \langle([\underline{f}]\ \backslash n\ [] \to Eval\ (f\ \{\})), o\rangle]$

$$\mathcal{I}[i \mapsto \ \mathit{MsgPAPP}\ d\ o\ \mathit{task} \quad \Gamma \quad \mathit{fs} \quad \mathit{as} \quad \mathit{rs} \quad h \quad \mathit{ds}[d \mapsto (\mathit{Fwd}\ d'){:}\mathit{ms}]]$$
$$\overset{(30)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \ \mathit{MsgPAPP}\ d'\ o\ \mathit{cont} \quad \Gamma \quad \mathit{fs} \quad \mathit{as} \quad \mathit{rs} \quad h \quad \mathit{ds}[d \mapsto \mathit{ms}]]$$

where

$$\mathit{cont} \ = \ \mathit{MsgPAPP}\ d\ o\ \mathit{task}$$

$$\mathcal{I}[i \mapsto \ \mathit{MsgPAPP}\ d_i\ o_i\ \mathit{task}_i \quad \Gamma_i \quad \mathit{fs}_i \quad \mathit{as}_i \quad \mathit{rs}_i \quad h_i \quad \mathit{ds}_i,$$
$$k \mapsto \ \mathit{task}_k \quad \Gamma_k \quad \mathit{fs}_k \quad \mathit{as}_k \quad \mathit{rs}_k \quad h_k \quad \mathit{ds}_k]$$

when $\left(\begin{array}{l} \mathit{ds}\ d_i = (\mathit{RetTo}\ k\ d_k){:}\mathit{ms}, \\ h_i\ o_i = \langle (\mathit{vs}\ \backslash n\ [] \rightarrow d), o_f{:}[\overline{w_N}] \rangle, \text{ and} \\ h_i\ o_f = \langle (\mathit{vs}_f\ \backslash \pi\ \mathit{xs}_f \rightarrow d_f),\ [\overline{w_{f_N}}]\ \rangle \end{array} \right)$

$$\overset{(31)}{\Longrightarrow} \quad \mathcal{I}[i \mapsto \ \mathit{MsgPAPP}\ d_i\ o_i\ \mathit{task}_i \quad \Gamma_i \quad \mathit{fs}_i \quad \mathit{as}_i \quad \mathit{rs}_i \quad h_i \quad \mathit{ds}_i[d_i \mapsto \mathit{ms}],$$
$$k \mapsto \ \mathit{MsgPAPP}\ d_k\ o_k\ \mathit{task}_k \quad \Gamma_k \quad \mathit{fs}_k \quad \mathit{as}_k \quad \mathit{rs}_k \quad h'_k \quad \mathit{ds}_k]$$

where

$$h'_k \ = \ h_k \left[\begin{array}{l} o_k \mapsto \langle (\mathit{vs}\ \backslash n\ [] \rightarrow d), o'_f{:}[\overline{w'_N}] \rangle \\ o'_f \mapsto \langle (\mathit{vs}_f\ \backslash \pi\ \mathit{xs}_f \rightarrow d_f),\ [\overline{w'_{f_N}}]\ \rangle \\ \overline{w'_N \mapsto \langle ([\underline{v}]\ \backslash u\ [] \rightarrow \mathit{EnterOn}\ i\ v), [w_N] \rangle} \\ \overline{w'_{f_N} \mapsto \langle ([\underline{v}]\ \backslash u\ [] \rightarrow \mathit{EnterOn}\ i\ v),\ [w_{f_N}] \rangle} \end{array} \right]$$

The complexity of the last rule is due to the fact that forwarding *EnterOn* closures have to be created for all the objects referenced in the environments of the transmitted closures.

It is not sufficient to transmit only the partial application $(h_i\ o_i)$, but the closure $(h_i\ o_f)$ referenced in the partial application's first environment argument has to be transmitted, too. This closure represents the function that was (partially) applied to the w_i. To execute the partial application on the instance k, the closure $(h_i\ o_f)$ has obviously to be on instance k also.

6 Related Work

The Threaded Abstract Machine (TAM) [CGSE93] is designed to implement the dataflow language Id [Nik90]. It applies multithreading based on stateless threads to tolerate long latency operations. In this respect it is close to the work presented in this paper, but the realization of this basic idea differs considerably. Instead of source-to-source transformations, the Id compiler based on the TAM builds a structured dataflow graph as an intermediate representation and has to apply sophisticated thread partitioning schemes [SCG95]. These partitioning algorithms require graphs without cyclic dependencies, which can not be guaranteed for a lazy language. In contrast to the STGM_{MT}, which employs asynchronous operations only when a long latency operation is actually encountered, the TAM uses asynchronous operations by default.

In comparison to the parallel implementation of the STGM, it is interesting to note that the STGM_{MT}, only needs a single argument and return stack per machine instance, instead of one stack per task. Furthermore, on entry, closures need not be overwritten with a "queue-me" code pointer. The updating performed in the rules (18) and (19) is sufficient and happens only when a remote

access occurred. A proper comparison with a distributed implementation of the original STGM, e.g., GUM [THJ+96], has to wait until a first implementation of the STGM_{MT} is working.

7 Conclusion

To decrease the impact of long latency operations on the execution time of distributed implementations of the Spineless Tagless G-machine, the STGM_{MT} extends the abstract machine language with abstract notions of independent local computations (thread boundaries and thread synchronization) and with an abstract notion of distribution. This enables the use of source-to-source transformations to increase the tolerance of the code to long latency operations.

While the behaviour of the new abstract machine can be studied using the operational semantics provided in this paper, it remains to be shown that the proposed techniques decrease the impact of the communication overhead in an actual implementation of a lazy functional language on parallel computers with distributed memory.

Acknowledgments. I am grateful to Paul H. J. Kelly, Hans-Wolfgang Loidl, and Simon L. Peyton Jones for their excellent comments on a previous version of this paper.

References

[CGSE93] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM—a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347—370, 1993.

[DP93] Marco Danelutto and Susanna Pelagatti. Structuring parallelism in a functional framework. Technical Report TR-29/93, Dipartimento di Informatica, Universitá di Pisa, 1993.

[EAL93] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. TR 93-13a, University of Arizona, 1993.

[Fos95] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.
⟨URL: `http://www.mcs.anl.gov/dbpp/text/book.html`⟩.

[H+95] Herbert H.J. Hum et al. A design study of the EARTH multiprocessor. In *Proceedings of Parallel Architectures and Compilation Techniques*, 1995.

[HMP94] Kevin Hammond, Jim S. Mattson, and Simon L. Peyton Jones. Automatic spark strategies and granularity for a parellel functional language reducer. In *CONPAR '94*, 1994.

[JL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *FPCA '91*, 1991.

[Kel89] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman, 1989.

[Nik90] R.S. Nikhil. Id (version 90.0) reference manual. Technical report, MIT Laboratory for Computer Science, July 1990. CSG Memo 284-1.

[PCS89] Simon L. Peyton Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Proceedings of PARLE (Volume 1).* Springer Verlag, 1989. LNCS 365.

[Pey92] Simon L. Peyton-Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), 1992.

[PS88] Simon L. Peyton-Jones and J. Salkild. The spineless tagless G-machine. In *Workshop on Implementations of Lazy Functional Languages,* Aspensas, Schweden, 1988.

[SCG95] Klaus E. Schauser, David E. Culler, and Seth C. Goldstein. Separation constraint partitioning - a new algorithm for partitioning non-strict programs into sequential threads. In *Proceedings of the Symposium on Principles of Programming Languages*, 1995.

[THJ+96] P. W. Trinder, K. Hammond, J. S. Mattson Jr, A. S. Partridge, and S. L. Peyton Jones. Gum: a portable implementation of Haskell. Submitted for publication, 1996.

A Notation

Tuples with n components are written as $\langle x_1, \ldots, x_n \rangle$ and have type $\langle \alpha_1, \ldots, \alpha_n \rangle$ when the x_i have type α_i.

Sequences have the form $[x_1, \ldots, x_n]$ and type $[\alpha]$ when α is the type of all elements x_i. The functions $(+\!\!+)$, $(:)$, and *length* represent concatenation of two sequences, prepending an element at a sequence, and obtaining the length of a sequence, respectively—they have the obvious types.

Maps are sequences of associations, denoted by $[o_1 \mapsto x_1, \ldots, o_n \mapsto x_n]$, with type $[Name \mapsto \alpha]$ when the x_i are of type α. The notation $m[o \mapsto x]$ is, in left-hand sides of transition rules, used to indicate that the name o is mapped

to x in the map m and, in right-hand sides, to denote the map that is obtained by replacing the value associated with o in m with x. We write for $[o_1 \mapsto x_1, \ldots, o_n \mapsto x_n]$ also $[os \mapsto xs]$ with $os = [o_1, \ldots, o_n]$ and $xs = [x_1, \ldots, x_n]$. Sometimes maps of type $[Name \mapsto \alpha]$ are used as representatives of functions from $Name$s to values of type α. Hence, $(m[o \mapsto x])\ o = x$.

Multisets with elements of type α are of type $\{\alpha\}$; we use the notation $\Gamma.e$ as a shorthand for $\Gamma \cup \{e\}$.

To support conciseness, we use $\overline{form}$ to represent the repetition of $form$. Using N as an index for a subform in $form$ denotes a family of the subform, e.g., $[\overline{x_N \mapsto a_N}]$ denotes $[x_1 \mapsto a_1, \ldots, x_N \mapsto a_N]$.

When new, unique names are required, they are marked by underlining them. Underlying a variable representing a sequence means a sequence of new names.

B The Structure of Machine Configurations

Machine configurations are maps from instance names to machine instances:

$$Config \quad = \quad [IName \mapsto Inst]$$

The type $IName$ is a synonym for $Name$, we use it for names of machine instances. In the transition rules from Section 5.2, we have $\mathcal{I} :: Config$.

A machine instance is a tuple containing seven components. It is of the following type:

$$\begin{array}{rcll} Inst & = & \langle Code, & \text{— currently executed code} \\ & & \{Task\}, & \text{— remaining tasks (ready \& waiting)} \\ & & [FName \mapsto Frame], & \text{— sync. frames for } \texttt{letpar}\text{s} \\ & & [Value], & \text{— stack of pending arguments} \\ & & [Cont], & \text{— return continuations} \\ & & [HName \mapsto Cls], & \text{— heap: named closures} \\ & & [DName \mapsto [DEntry]]\rangle & \text{— target locations for results} \end{array}$$

In the transition rules, the $\langle$, $\rangle$, and comma are omitted, and the seven components are just placed side-by-side. The types $FName$, $HName$, and $DName$ are synonyms of the type $Name$ and are used for names of frames, heap-allocated closures, and dissemination entries, respectively.

The types of tasks and frames are

$$\begin{array}{rcll} Task & = & \langle Code, Cont\rangle & \text{— return to } Cont \text{ with result of } Code \\ Frame & = & \langle Int, [Env]\rangle & \text{— sync. counter \& completed bindings} \\ Env & = & [LName \mapsto Value] & \text{— local environment} \end{array}$$

The type $LName$ is a synonym for $Name$, it is used for names of local variables.

Values appear either boxed, i.e., they are the name of a closure in the heap, or they are unboxed, i.e., a data constructor with its arguments. In addition a value can be the name of a dissemination entry. In the following definition of

the data type *Value*, data constructors are omitted—this avoids some clutter in the transition rules and there is no danger that ambiguities arise.

$$
\begin{array}{rcll}
\mathit{Value} & = & \mathit{HName} & \text{— boxed: name of a closure} \\
 & | & \mathit{UValue} & \text{— unboxed: constructor \& arguments} \\
 & | & \mathit{DName} & \text{— name of a dissemination entry} \\
\mathit{UValue} & = & \langle \mathit{CName}, [\mathit{HName}] \rangle & \text{— constructor \& arguments}
\end{array}
$$

The type *CName* is a synonym for *Name*, it is used for names of data constructors.

The type of closures is defined as

$$\mathit{Cls} \quad = \quad \langle (\underbrace{[\mathit{LName}]}_{\text{free}} \backslash \pi \underbrace{[\mathit{LName}]}_{\text{args}} \rightarrow \underbrace{(\mathit{Env} \rightarrow \mathit{Code})}_{\text{body}}, \underbrace{[\mathit{HName}]}_{\text{env}} \rangle$$

The sequences of free and argument variables correspond to the list of environment variables and the list of argument variables, respectively, in the function bindings of the $\mathrm{STG_{MT}}$-language. The body of a closure is, in contrast to the original STGM, not an expression of the $\mathrm{STG_{MT}}$-language, but a function from environments to code forms—the reason for this change is that we sometimes need to place code forms other than *Eval* (e.g., *EnterOn*) into the body of closures.

Return continuations, the elements of the return stack, are triples made of the name of a dissemination entry, a code form, and a sequence of values. The meaning is that the currently executed task distributes its result value using the dissemination entry, and then continues with the code after placing the values on the argument stack.

$$\mathit{Cont} \quad = \quad \langle \mathit{DName}, \mathit{Code}, [\mathit{Value}] \rangle$$

The various code forms are defined in the following data type (*Eval* and *Enter* are equal to the corresponding forms of the original STGM):

$$
\begin{array}{rcll}
\mathit{Code} & = & \mathit{Eval\ Expr\ Env} & \text{— evaluate expression} \\
 & | & \mathit{Enter\ HName} & \text{— enter the given closure} \\
 & | & \mathit{EnterOn\ IName\ LName\ Env} & \text{— enter on other instance} \\
 & | & \mathit{RetTerm\ UValue} & \text{— return unboxed value} \\
 & | & \mathit{RetDelay\ LName\ Env} & \text{— cover long latency} \\
 & | & \mathit{Sync\ [VBind]\ Expr\ FName} & \text{— wait for local bindings} \\
 & | & \mathit{Next} & \text{— switch to other task} \\
 & | & \mathit{MsgTerm\ DName\ UValue\ Code} & \text{— dissem. unboxed value} \\
 & | & \mathit{MsgPAPP\ DName\ HName\ Code} & \text{— dissem. partial appl.}
\end{array}
$$

Details on the meaning of the variants are provided in Section 5.2.

Finally, elements of the lists of the dissemination map describe the locations where the code forms *MsgTerm* and *MsgPAPP* have to place disseminated values:

$$
\begin{array}{rcll}
\mathit{DEntry} & = & \mathit{Store\ FName\ LName} & \text{— store in the env. of a frame} \\
 & | & \mathit{Upd\ HName} & \text{— update closure} \\
 & | & \mathit{Fwd\ DName} & \text{— forward to other dissem. entry} \\
 & | & \mathit{RetTo\ IName\ DName} & \text{— communicate to other instance}
\end{array}
$$