Communication-Passing Style for Coordination Languages

Suresh Jagannathan

Computer Science Research, NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, *suresh@research.nj.nec.com*

Abstract. Coordination languages for parallel and distributed systems specify mechanisms for creating tasks and communicating data among them. These languages typically assume that (a) once a task begins execution on some processor, it will remain resident on that processor throughout its lifetime, and (b) communicating shared data among tasks is through some form of message-passing and data migration. In this paper, we investigate an alternative approach to understanding coordination. *Communication-passing style* (CmPS) refers to a coordination semantics in which data communication is always undertaken by migrating the continuation of the task requiring the data to the processor where the data resides.

Communication-passing style is closely related to continuation-passing style (CPS), a useful transformation for compiling functional languages. Just as CPS eliminates implicit call-return sequences, CmPS eliminates implicit inter-processor data communication and synchronization requests. In a CmPS-transformed program, only continuations (i.e., control contexts) are transmitted across machines; all synchronization and data communication occurs locally. Besides providing significant optimization opportunities, CmPS is a natural representation for implementations on networks of workstations.

This paper presents an operational semantics for a coordination language that supports first-class distributed data repositories. The computation sublanguage considered is an untyped call-by-value functional language similar to pure Scheme. Optimizations and implementation issues that arise from using a CmPS-driven coordination language are also described.

1 Introduction

Implementations of functional languages often first transform the source program into *continuation-passing style* (CPS) [1, 16]. In a CPS'ed program, every procedure is supplied an extra argument, its *continuation*, that represents the "rest of the computation" following a call to this procedure. When a procedure is applied, the supplied continuation conceptually represents the return point to which control should be transferred upon completion of the procedure body. The most important advantage of the CPS transform is that it makes all control-flow in a program explicit, thus simplifying and enabling various source-level optimizations.

Communication-passing style (CmPS) is a natural extension of CPS in a distributed context. Assume a collection of tasks executing on some collection of processors, and assume some distribution of shared data on these processors. In a CmPS-transformed program, inter-processor communication is always undertaken by *migrating* the continuation of a task to the processor where the shared data it requires resides. Just as CPS eliminates implicit call-return sequences, CmPS eliminates implicit inter-processor data communication and synchronization requests. In a CmPS-transformed program, only continuations (i.e., control contexts) are transmitted across machines; all synchronization and data communication occurs locally. Just as CPS can be thought as transforming procedure calls to gotos, CmPS can be thought as transforming data communication to task migration.

Using a communication-passing style transform, the responsibility of the programmer is restricted to devising reasonable data distribution among the computing elements available; issues of global consistency and synchronization are no longer relevant. Shared data remains resident on the node where it was created, and tasks migrate as needed among those nodes containing the shared data they require.

Networks of commodity workstations (NOWs) are likely soon to become the preferred platform for building long-lived parallel and distributed programs. Programs written under existing languages, however, appear ill-suited to execute on such platforms. In traditional languages, tasks usually have a great deal of control over *what* they compute, but rarely have any power in determining *where* or *how* their computation is exercised. Thus, while a program creating many tasks may be able to specify where these tasks should be initially allocated, the executing process usually does not have the capability of efficiently altering or refining this mapping as computation progresses. Yet, it is precisely this feature that is crucial when executing parallel and distributed programs in a loosely-coupled, potentially time-shared environment, such as a NOW.

Furthermore, typical implementations in this environment require data shared among tasks running on different machines to be copied back-and-forth to ensure global consistency. Locality often suffers if the primary accessor of the data executes on a machine different from where the data actually resides. In addition to ensuring global consistency of multiple copies of a shared object, implementations must also overcome bandwidth limitations if shared data is accessed by many tasks on many different machines.

In this paper, we consider the integration of a coordination language loosely based on Linda [4] into a simple untyped call-by-value higher-order computation language. Rather than tuple-spaces, our coordination language supports first-class *address spaces*. An address space defines a separate locus of execution that may contain globally visible shared-data. Instead of distributed data structures, our coordination language supports *naming environments* implicitly defined with each address space. Tasks may deposit, retrieve, or remove bindings on any address space, blocking if a desired binding is not found. Within this context, we consider a formal language specification and abstract implementation that supports CmPS-style coordination.

The paper is organized as follows. The next section presents a direct-style formal operational semantics for the kernel language using a CEK-machine [7, 8] formulation. Section 3 provides further motivation for communication-passing style through a series of simple examples. Because the semantics is presented in direct-style, continuations are implicitly managed by the machine, and not exposed in the source program. In the context of a higher-order computation language, a CmPS strategy, like CPS, can be expressed entirely as a source-to-source transformation; this transform is described in Section 4. Section 5 discusses some optimizations in the context of the semantics that may significantly reduce the bandwidth requirements needed to support a CmPS-based implementation. Sections 6 and 7 provide comparison to related work and conclusions.

2 The Language

In this section, we define a simple parallel and distributed language L. We proceed to give a small-step operational semantics for L in terms of the CEK^T machine, a parallel extension of the CEK-machine [7]. We regard the specification of this machine as a state transition system whose objects of interest include expressions, environments, continuations, threads, and shared environments.

The source language for L, whose grammar is shown in Figure 1, consists of two parts. L's computation component is a simple untyped call-by-value lambda-calculus with constants, variables, conditionals, single-argument functions, recursion, applications, and (primitive) applications. Like Scheme, the computation language is latently typed—no static typing discipline is imposed on programs.

L's coordination language supports the creation of multiple threads of control and multiple address spaces. An address space defines a new logical locus of execution; when a new address space is created, resources on some machine in a network ensemble maybe allocated for it. We leave unspecified the manner in which the mapping between address spaces and physical machines is chosen. New threads are always created on a specific address space. Evaluating (`makeAS`) creates a new address space, and evaluating the expression, (`spawn` $AS\, e$) creates a new thread on address space AS to evaluate e. Associated with each address space is a shared environment accessible to expressions evaluating on other address spaces. We provide three operations on this environment: (`put` $AS\, [x\; v]\; e$) deposits a binding $\langle x, v\rangle$ that binds variable x to value v in the environment associated with address space AS and evaluates e once the binding has been deposited; this binding shadows any previous binding for x in AS. The expression, (`rd` $AS\, [x]\, e$) reads a binding for x from AS, and binds x in e. If no binding for x is available, evaluation of the expression blocks until a `put` operation to deposit a binding for x is executed by some other thread. Similar to `rd` is `get` which reads and then *removes* the appropriate binding from the target environment. Note that because synchronization is tied to name lookup, variables associated

$$
\begin{array}{lcl}
e \in E & ::= & Comp \mid Coord \\
\\
c_\lambda \in Comp & ::= & c \mid x \mid \lambda x.e \mid \mu f.\lambda x.e \\
 & \mid & (e_1\, e_2) \\
 & \mid & (p\, e_1 \ldots e_n) \\
 & \mid & (\texttt{if}\, e_1\; e_2\; e_3) \\
\\
c_\pi \in Coord & ::= & (\texttt{put}\, e\, [x\; e_a]\, e_b) \\
 & \mid & (\texttt{rd}\, e\, [x]\, e_b) \\
 & \mid & (\texttt{get}\, e\, [x]\, e_b) \\
 & \mid & (\texttt{spawn}\, e_a\, e_b) \\
\\
p \in Primop & ::= & + \mid - \mid \ldots \mid \texttt{makeAS} \mid \texttt{spawn}
\end{array}
$$

Fig. 1. The language.

with bindings found in shared environments cannot be renamed in the contexts in which they are referenced.

Readers familiar with Linda [4] will see obvious similarities between these operations and Linda's support for distributed data structures via tuple-spaces. We believe the coordination operators defined here can be easily generalized to support associative matching in the style supported by Linda, but such extensions are orthogonal to our motivation for investigating the semantics of communication-passing style transformations, and thus not included here.

2.1 Semantics

To specify a CEK^T-machine, we define a set of states, *State*, and a binary transition relation, $\longrightarrow$, on states. A particular state includes the state of all threads in all address spaces, and the contents of the shared environment in each address space. The domain definitions for states are shown in Figures 2 and 3.

$$
\begin{array}{ll}
s \in State & = ThreadMap \times SharedEnv \\
T \in ThreadMap & = AspaceUid \times ThreadUid \rightarrow ThreadState \\
\rho \in SharedEnv & = AspaceUid \rightarrow Env \\
\alpha \in AspaceUid & \\
\tau \in ThreadUid &
\end{array}
$$

Fig. 2. CEK^T-machine Thread and Address Space States

$$
\begin{array}{rll}
t \in \mathit{ThreadState} & = \mathit{ControlState} + \mathit{ReturnState} + \mathit{HaltState} \\
\langle e, r, k\rangle \in \mathit{ControlState} & = \mathit{Exp} \times \mathit{Env} \times \mathit{Cont} \\
\langle k, v\rangle \in \mathit{ReturnState} & = \mathit{Cont} \times \mathit{Value} \\
\mathbf{halt}(v) \in \mathit{HaltState} & = \mathit{Value} \\
r \in \mathit{Env} & = \mathit{Var} \rightarrow \mathit{Value} \\
k \in \mathit{Cont} & = \mathit{Frame}^* \\
\mathit{frame} \in \mathit{Frame} & = \mathit{CompFrame} + \mathit{CoordFrame} \\
\mathit{frame}_\lambda \in \mathit{CompFrame} & = \mathit{ArgFrame} + \mathit{FunFrame} + \\
 & \quad \mathit{PrimFrame} + \mathit{IfFrame} + \mathit{CoordFrame} \\
\mathit{frame}_\pi \in \mathit{CoordFrame} & = \mathit{SpawnFrame} + \mathit{PutArgFrame} + \\
 & \quad \mathit{RdArgFrame} + \mathit{GetArgFrame} + \mathit{PutBodyFrame} \\
\mathbf{arg}\langle e, r\rangle \in \mathit{ArgFrame} & = \mathit{Exp} \times \mathit{Env} \\
\mathbf{fun}\langle v\rangle \in \mathit{FunFrame} & = \mathit{Value} \\
\mathbf{cond}\langle e_t, e_f, r\rangle \in \mathit{IfFrame} & = \mathit{Exp} \times \mathit{Exp} \times \mathit{Env} \\
\mathbf{spawn}\langle e, r\rangle \in \mathit{SpawnFrame} & = \mathit{Exp} \times \mathit{Env} \\
\mathbf{prim}\langle p, r, \vec{v}, \vec{e}\rangle \in \mathit{PrimFrame} & = \mathit{Primitive} \times \mathit{Env} \times \mathit{Value}^* \times \mathit{Exp}^* \\
v \in \mathit{Value} & = \mathit{Constant} + \mathit{Closure} + \mathit{AspaceUid} + \mathit{ThreadUid} \\
\lambda x.\, e, \mu f.\lambda e. \in \mathit{Closure} & = (\mathit{LambdaExp} \times \mathit{Env}) + (\mathit{RecExp} \times \mathit{Env})
\end{array}
$$

Fig. 3. CEK^T-machine Thread States

In the following, we write $X + Y$ and $X \times Y$ to mean the cartesian sum and product of sets X and Y, resp. We write $X \rightarrow Y$ to denote the set of partial functions from X to Y. The notation $f[x \mapsto v]$ denotes the function that is identical to f on all elements except x for which it returns y; f/x denotes the function identical to f on all elements except x for which it is undefined. We write X^* to denote the finite sequences of elements of X. If x ranges over X, then $\vec{x}$ denotes a list of elements drawn from X. We write "$v_1 @ \vec{v}$" to indicate the list whose head is v_1 and whose tail is $\vec{v}$.

Machine States A *state* in our semantics is a pair consisting of a *ThreadMap*, T, and a global shared environment, ρ. T is a function that maps pairs of address spaces and threads to thread states. The shared environment is a map from address spaces to environments. We assume all address spaces and threads are associated with a unique identifier that names them.[1]

A given thread τ may be in one of three states. If τ is in *control state*, $\langle e, r, k\rangle$, it means τ is currently evaluating expression e in environment r. If τ is in a *return state*, it has computed a value for a subexpression that must now be passed to continuation. When τ completes, it enters a *halt state*.

The continuation k of e is the "remainder" of the computation to be performed by this thread after evaluation of e. Continuations are represented as a stack of frames. There are two kinds of continuation frames, those that deal

[1] We freely denote address spaces and threads by their uids when obvious from context.

with the computation sublanguage and those that deal with the coordination sublanguage. For the computation sublanguage, an *ArgFrame* is used to hold the argument value of a function call while the expression in the function position of the call is evaluated; a *FunFrame* is used to combine the function found at a call with its arguments; a *PrimFrame* is used to collect arguments to primitives; and, an *IfFrame* is used to hold the true and false branches of a conditional while the test is being evaluated. For the coordination sublanguage, a *SpawnFrame* is used to hold the expression to be spawned while the value of the address space in which this thread should evaluate is being computed; *PutArg* frames and *PutBody* frames are used to hold intermediate results of subexpressions in a `put` operation; and, *RdArg* and *GetArg* frames serve similar purposes for `rd` and `get` operations, respectively.

The language supports four kinds of values: constants, closures, and tagged unique identifiers for address spaces and threads. Note that there is mutual reference in the definition of *Env*, *Closure*, and *Value*. Rather than appealing to a domain-theoretic interpretation of these structures, we simply choose the smallest sets satisfying the desired equations; the empty environment serves as the base case.

In order to define $\longrightarrow$, we assume a partial function to apply primitive operations to constants:

$$\mathcal{P} : \mathit{Primitive} \times \mathit{Constant}^* \rightarrow \mathit{Constant}$$

Transition Relation We can think of $\longrightarrow$ as an implementation of an interpreter for this language. The interpreter chooses some thread to evaluate, matches the current state with a left-hand side of a rule, and "transforms" the state to the right-hand side. At any point during evaluation, there may be many thread states that satisfy the left-hand side of a rule; any one of these threads may be chosen for evaluation. The only dependencies among threads are those induced by lookups and updates of shared environments. Thus, any two threads not involved in a communication event via a shared environment may be evaluated concurrently; the semantics imposes no conditions on the *order* in which threads are evaluated. The semantics makes no assumptions about fairness or liveness.

However, the semantics must guarantee that for any given thread, there is at most one rule that satisfies the thread's current state. Moreover, a thread in a state whose evaluation is associated with a rule that has side-conditions, must satisfy these conditions before it can be evaluated to the right-hand side. If no thread in the machine can be further reduced, and not all threads are in a halt state, the program is deadlocked.

Given CEK^T, we define *eval*, a partial function that maps a program to a pair $\langle \rho', \{v_0, v_1, v_2, \ldots, v_n\} \rangle$ where ρ' is a shared environment, and v_i represents the value yielded by evaluating thread with uid τ_i. Let $\stackrel{*}{\longrightarrow}$ be the reflexive transitive closure of $\longrightarrow$. Then, the semantics of a program P is given by:

$$\mathit{eval}(P) = \langle \rho', \{v_1, v_2, \ldots, v_n\} \rangle \Longleftrightarrow$$

$$\langle T_\perp[\langle \alpha_0, \tau_0\rangle \mapsto \langle P, r_\perp, k_\perp\rangle], \rho_\perp\rangle \xrightarrow{*}$$
$$\langle T_\perp[\langle \alpha_0, \tau_0\rangle \mapsto \mathbf{halt}(v_0), \ldots, \langle \alpha_n, \tau_n\rangle \mapsto \mathbf{halt}(v_n)], \rho'\rangle$$

The transition relation consists of four kinds of rules. The first group describes actions on the global state; these rules define the creation of new address spaces and threads. The second group of rules defines the semantics of the coordination language, and describes how shared environments are manipulated. The third and fourth group of rules defines the action of the machine on control and return states, and describe single-threaded behavior. The first two groups of rules are given in Figures 4 and 5. The latter two are shown in the appendix.

There are three "top-level" rules that specify how new threads and address spaces are added to the global state. To create a new thread, we first determine the address space on which the thread will be evaluated, pushing the expression to be spawned along with the current environment on the continuation stack. Provided that the address space exists, a new uid is associated with this thread, and the thread is instantiated with an initial state consisting of the thread body, the environment in which the `spawn` expression is being evaluated, and the empty continuation. Thus, threads do not inherit the dynamic control context of their parent. Creating a new address space involves generating a new address space uid α', augmenting the thread map to include a reference to the new address space, and augmenting the shared environment to map α' to an initial empty environment.

$$\langle T[\langle \alpha, \tau\rangle \mapsto \langle (\texttt{spawn}\; e_1\; e_2), r, k\rangle], \rho\rangle \longrightarrow \langle T[\langle \alpha, \tau\rangle \mapsto \langle e_1, r, \mathbf{spawn}\langle e_2, r\rangle : k\rangle], \rho\rangle$$

$$\langle T[\langle \alpha, \tau\rangle \mapsto \langle \alpha', \mathbf{spawn}\langle e_2, r\rangle : k\rangle], \rho\rangle \longrightarrow$$
$$\langle T[\langle \alpha, \tau\rangle \mapsto \langle k, \tau'\rangle,\ \langle \alpha', \tau'\rangle \mapsto \langle e_2, r, k_\perp\rangle], \rho\rangle$$
provided $\alpha' \in Dom(\rho)$ and τ' fresh

$$\langle T[\langle \alpha, \tau\rangle \mapsto \langle \texttt{makeAS}, r, k\rangle], \rho\rangle \longrightarrow$$
$$\langle T[\langle \alpha', \tau_0\rangle \mapsto \mathbf{halt}(v_0),\ \langle \alpha, \tau\rangle \mapsto \langle k, \alpha'\rangle], \rho[\alpha' \mapsto \lambda\, x.\, undef]\rangle$$
provided $\alpha' \notin Dom(\rho)$ and α' fresh

Fig. 4. Transition rules for threads and address spaces.

The rules defining the semantics of the coordination sublanguage are also divided into two groups. The first group defines the order of evaluation of subexpressions on the coordination operators; the second defines how the shared environment and thread map are manipulated. Consider a `put` operation executing as part of thread σ on address space AS_s. Let AS_t be the target address space where the generated binding $\langle x, v\rangle$ is to be deposited. To perform this operation, the machine extends the shared environment on AS_t with $\langle x, v\rangle$, and "migrates" σ to AS_t. Migration is expressed by executing a transition that yields a new thread map which removes the current thread from AS_s and installs a new thread (with the same thread uid) on AS_t.

A similar transition is performed for `rd` and `get` operations. The relevant transition rules for these operators also have side-conditions that guarantee that the transition is satisfiable only when the relevant binding is available on the appropriate shared environment. Note that the introduction of a new computation sublanguage will not entail any modification to the semantics of the coordination sublanguage.

The CmPS-transform is expressed explicitly in the semantics (and *implicitly* in the coordination language) by manipulating threads and their continuations. Since continuations are manifest in the semantics, and since, by definition, there is a unique continuation associated with every executing thread, it becomes straightforward to specify the act of migrating a continuation from one address space to another in terms of simple restrictions and extensions on the global thread map. Moreover, because each transition is atomic, and because thread uids are globally unique, the machine will never enter a state in which the same thread is concurrently executing on multiple address spaces.

$$\langle (\mathtt{put}\; e_a\,[x\; e_x]\; e_b), r, k\rangle \longrightarrow \langle e_a, r, \mathbf{putArg}\langle\, x, e_x, e_b, r\,\rangle : k\rangle$$
$$\langle (\mathtt{rd}\; e\,[x]\; e_b), r, k\rangle \longrightarrow \langle e, r, \mathbf{rdArg}\langle\, x, e_b, r\,\rangle : k\rangle$$
$$\langle (\mathtt{rd}\; e\,[x]\; e_b), r, k\rangle \longrightarrow \langle e, r, \mathbf{getArg}\langle\, x, e_b, r\,\rangle : k\rangle$$

$$\langle \mathbf{putArg}\langle\, x, e_x, e_b, r\,\rangle : k, v\rangle \longrightarrow \langle e_x, r, \mathbf{putBody}\langle\, x, e_b, v, r\,\rangle : k\rangle$$
$$\langle T[\langle \alpha,\tau\rangle \mapsto \langle \mathbf{putBody}\langle\, x, e_b, v, r\,\rangle : k, v_a\rangle], \rho\rangle \longrightarrow$$
$$\quad \langle T/\langle \alpha,\tau\rangle[\langle v,\tau\rangle \mapsto \langle e_b, r, k\rangle], (\rho\, v)[x \mapsto v_a]\rangle$$

$$\langle T[\langle \alpha,\tau\rangle \mapsto \langle \mathbf{rdArg}\langle\, x, e_b, r\,\rangle : k, v\rangle], \rho\rangle \longrightarrow \langle T/\langle \alpha,\tau\rangle[\langle v,\tau\rangle \mapsto \langle e_b, r[x\mapsto v_x], k\rangle], \rho\rangle$$
$$\quad \text{provided } x \in \mathit{Dom}((\rho\, v)) \text{ and } ((\rho\, v)\, x) = v_x$$

$$\langle T[\langle \alpha,\tau\rangle \mapsto \langle \mathbf{getArg}\langle\, x, e_b, r\,\rangle : k, v\rangle], \rho\rangle \longrightarrow \langle T/\langle \alpha,\tau\rangle[\langle v,\tau\rangle \mapsto \langle e_b, r[x\mapsto v_x], k\rangle], \rho'\rangle$$
$$\quad \text{where } \rho' = (\rho\, v)/x,\; x \in \mathit{Dom}((\rho\, v)) \text{ and } ((\rho\, v)\, x) = v_x$$

Fig. 5. Transition rule for coordination sublanguage

3 An Example

To motivate communication-passing style, consider the simple example shown in Figure 6. To make the presentation more readable, we use pure Scheme as the computation sublanguage, and italicize operators belonging to the coordination sublanguage.

```
(letrec ((tree
          (lambda (AS v)
            (cond ((leaf?) (put AS [val (leaf v)] v))
                  (else (let ((left (makeAS))
                              (right (makeAS)))
                          (spawn left (tree left (left-part v)))
                          (spawn right (tree right (right-part v)))
                          (put AS [val (C (rd left [val] val)
                                          (rd right [val] val))]
                          val)))))))
  (tree (makeAS) initial-val))
```

Fig. 6. A simple tree-structured parallel program.

This program creates a tree of address spaces, with a single task running on each address space. All tasks communicate via shared variable `val` that has a potentially different binding value on each address space. A leaf task simply deposits a binding for `val` in its associated address space. A non-leaf task creates two child address spaces, and spawns a thread on each of them. These threads simply perform a recursive call, using some portion of the initial value supplied to the root of the tree as their argument. A non-leaf task waits for its children to supply a binding-value for `val` on their corresponding address spaces, combines these values, and deposits the result as its binding-value for `val` on its address space.

Consider a straightforward implementation of this program. Each address space would be mapped to some physical node in a network ensemble. Every non-leaf task would initiate two messages for each of `C`'s `rd` arguments. The first message is a request for `val`'s binding-value on the target address space; the request may be enqueued if the value is not present. The second is the reply containing the data. For any non-leaf task T, the bandwidth requirements imposed by T is the sum of the sizes of the binding-values for `val` in its descendents.

Now, consider a CmPS implementation. For a non-leaf task, the `rd` operation on the left child will cause the continuation of the parent to migrate to the address space on which the left child resides. Once `val` has been read on this address space, a continuation migrates to the right child, and finally returns back to the parent. Unlike the implementation sketched above, three messages (rather than four) are initiated by each non-leaf task. Moreover, using a safe-for-space implementation of continuations [1, 21], the bandwidth requirement needed to support this migration is neglible since the only data referenced in the migrating continuation is an empty shared environment. As a result, we may expect significant performance improvement over the non-CmPS implementation.

In this simple program, consistency issues were not critical since every binding in a shared-environment is read and written exactly once. Figure 7 is a slightly more sophisticated version of the same program. Instead of shared data being read exactly once, each non-leaf task spawns three new threads which read a given piece of shared data multiple times. The first thread performs some computation that repeatedly reads `val`'s binding-value on the left-child; the second performs the same computation except it reads `val`'s binding value multiple times on the right-child; and, the third repeatedly refines `val`'s binding-value on the current address space. Thus, instead of a single communication event between parent and child tasks, there are multiple events, and instead of a single unique value computed for each task, there is effectively a stream of values produced.

```
(letrec ((g (lambda (AS) perform repeated calculation on (rd AS [val] val)))
         (h (lambda (AS)
               perform repeated calculation on
               (put AS [val (C (rd AS [left-val] left-val)
                               (rd AS [right-val] right-val))]
                        val)))
         (tree (lambda (AS v)
                  (cond ((leaf?) (put  [val (leaf v)] v))
                        (else
                          (let ((left (makeAS))
                                (right (makeAS)))
                            (spawn left (tree left (left-part v)))
                            (spawn right (tree right (right-part v)))
                            (spawn AS (put AS [left-val (g left)]
                                               left-val))
                            (spawn AS (put AS [right-val (g right)]
                                               right-val))
                            (spawn AS (put AS [val (h AS)] val)))))))
   (tree (makeAS) initial-val))
```

Fig. 7. A simple tree-structured parallel program.

In this example, consistency of shared data becomes an important issue. In order to be faithful to the semantics, a non-CmPS implementation which chooses to cache the value yielded by `rd` on its local address space must ensure that the address space is notified whenever the read value is updated. Rather than implementing a sophisticated consistency protocol, an implementation may choose to have every `rd` operation on a remote address space entail a remote communication event.

In a CmPS-implementation, on the other hand, the computation of a call to `g` will always occur on either the left or right child. The size of the continuation is again neglible because it includes only a reference to the current address space. This is because spawned threads begin execution in an empty continuation. Unlike the earlier example, the continuation corresponding to the two threads responsible for computing `left-val` and `right-val` are sent to two different address spaces, namely the address spaces corresponding to the left and right child. When the call to `g` is finished, these continuations migrate back to their parent, updating the shared environment appropriately with a new binding for `left-val` and `right-val`. Thus, exactly four communication events occur between a non-leaf node and its two children. In contrast, a non-CmPS implementation will incur a number of communication events proportional to the number of iterations performed by g.

For fine-grained programs such as these, a CmPS-style communication model would appear to be significantly more effective at reducing bandwidth requirements, than implementations based on more traditional data-migration [15] or RPC-style [2] semantics. Nonetheless, it is clear that the effectiveness of this communication strategy depends upon the size of the continuations being migrated. If these continuations are large, bandwidth requirements may in fact exceed what would have been necessary in a more traditional implementation. In Section 5, we examine some simple optimizations and alterations to our semantics that address this concern.

4 A Source-Level Transformation

The direct-style semantics given in Section 2 manipulated continuations as part of the machine specification dealing with the creation and coordination of threads. For example, whenever a coordination operator accesses an address space AS, its continuation is migrated to AS. These actions are not manifest in the source program, but defined explicitly in the transitions undertaken by the machine. Given that the computation sublanguage we are considering is higher-order, however, it is straightforward to express this behavior as a source-level transformation. Demonstrating this transform is illustrative because it provides further motivation for why we regard the specification of task migration in this language to be closely related to a continuation-passing transformation.

Figure 8 shows the CmPS source-level transformation for the coordination sublanguage. We omit presentation of the transform for the computation sublanguage since it would be identical to an ordinary CPS [1, 10] translation.

Assume a thread τ executing on address space AS has continuation k and is to evaluate a `put` expression. The CmPS-transform for this expression evaluates the operator's first argument to an address space, v_{AS}, evaluates the binding-expression to get value v_{arg}, and creates a new thread on v_{AS} that deposits a binding on the address space associated with v_{AS}. Finally, k is applied to the CmPS-transformed value of `put`'s body. The continuation of the call to k is the continuation of the `spawn` operator, which we have defined in our CEK^T specification to be the empty continuation. Thus, no further evaluation of τ takes place on AS; the rest of τ's computation is undertaken by its proxy on v_{AS}.

$$
\begin{aligned}
&\mathcal{C}[\![(\texttt{put}\ e_{AS}\ [x\ e_a]\ e_b)]\!]\ k = (\texttt{let}\ ((k_1\ (\texttt{lambda}\ (v_{AS})\\
&\qquad\qquad (\texttt{let}\ ((k_2\ (\texttt{lambda}\ (v_{arg})\\
&\qquad\qquad\qquad (\texttt{let}\ ((k_3\ (\texttt{lambda}\ (z)\ (k\ z))))\\
&\qquad\qquad\qquad\quad (\texttt{spawn}\ v_{AS}\ (\texttt{put}\ v_{AS}\ [x\ \ v_{arg}]\\
&\qquad\qquad\qquad\qquad \mathcal{C}[\![e_b]\!]\ k_3))))))\\
&\qquad\qquad\quad \mathcal{C}[\![e_a]\!]\ k_2))))\\
&\qquad\quad \mathcal{C}[\![e_{AS}]\!]\ k_1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}[\![(\texttt{rd}\ e_{AS}\ [x]\ e_b)]\!]\ k = (\texttt{let}\ ((k_1\ (\texttt{lambda}\ (v_{AS})\\
&\qquad\qquad (\texttt{let}\ ((k_2\ (\texttt{lambda}\ (z)\ (k\ z))))\\
&\qquad\qquad\quad (\texttt{spawn}\ v_{AS}\ (\texttt{rd}\ v_{AS}\ [x]\ \mathcal{C}[\![e_b]\!]\ k_2))))))\\
&\qquad\quad \mathcal{C}[\![e_{AS}]\!]\ k_1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}[\![(\texttt{get}\ e_{AS}\ [x]\ e_b)]\!]\ k = (\texttt{let}\ ((k_1\ (\texttt{lambda}\ (v_{AS})\\
&\qquad\qquad (\texttt{let}\ ((k_2\ (\texttt{lambda}\ (z)\ (k\ z))))\\
&\qquad\qquad\quad (\texttt{spawn}\ v_{AS}\ (\texttt{get}\ v_{AS}\ [x]\ \mathcal{C}[\![e_b]\!]\ k_2))))))\\
&\qquad\quad \mathcal{C}[\![e_{AS}]\!]\ k_1)
\end{aligned}
$$

Fig. 8. A source-level CmPS transformation for the coordination sublanguage. Italicized variables are fresh.

This transformation produces the desired behavior: the body of the `put` expression as well as `put`'s continuation evaluate on v_{AS}. Moreover, there is no further evaluation of τ on AS. The transformation for `rd` and `get` are similarly defined. In these cases as well, the body of the operator along with the operator's continuation are evaluated in a new thread instantiated on the address space where the relevant binding exists. Like CPS, this transform makes all control-flow explicit. It now becomes syntactically apparent that coordination operations introduced by the transformation all occur as arguments to `spawn`. Moreover, the target address spaces for these operators (i.e., the address space on which the desired binding is to be deposited or retrieved) is the same as the address space on which the thread instantiated by the enclosing `spawn` begins execution.

Note that the body of the `spawn` expression for all three operators is a *non-transformed* variant of the input to $\mathcal{C}$. The meaning of these expressions is as shown in Figure 5. Since the evaluation of these expressions occur on the same address space as the bindings they manipulate, there is no observable effect on the thread map when they are evaluated. Indeed, we can simplify the transition rules for the coordination sublanguage by assuming the existence of this transformation. These simplified rules are shown in Figure 9.

$\langle (\texttt{put}\ e_a\ [x\ e_x]\ e_b), r, k \rangle \longrightarrow \langle e_a, r, \textbf{putArg}\langle\, x, e_x, e_b, r\,\rangle : k \rangle$
$\langle (\texttt{rd}\ e\ [x]\ e_b), r, k \rangle \longrightarrow \langle e, r, \textbf{rdArg}\langle\, x, e_b, r\,\rangle : k \rangle$
$\langle (\texttt{get}\ e\ [x]\ e_b), r, k \rangle \longrightarrow \langle e, r, \textbf{getArg}\langle\, x, e_b, r\,\rangle : k \rangle$

$\langle \textbf{putArg}\langle\, x, e_x, e_b, r\,\rangle : k, v_{AS} \rangle \longrightarrow \langle e_x, r, \textbf{putBody}\langle\, x, e_b, v_{AS}, r\,\rangle : k \rangle$
$\langle T[\langle \alpha, \tau \rangle \mapsto \langle \textbf{putBody}\langle\, x, e_b, v, r\,\rangle : k, v_a \rangle], \rho \rangle \longrightarrow$
$\quad \langle T[\langle \alpha, \tau \rangle \mapsto \langle e_b, r, k \rangle], (\rho\, \alpha)[x \mapsto v_a] \rangle$

$\langle T[\langle \alpha, \tau \rangle \mapsto \langle \textbf{rdArg}\langle\, x, e_b, r\,\rangle : k, \alpha \rangle], \rho \rangle \longrightarrow \langle T[\langle \alpha, \tau \rangle \mapsto \langle e_b, r[x \mapsto v_x], k \rangle], \rho \rangle$
provided $x \in Dom((\rho\, \alpha))$ and $((\rho\, \alpha)\, x) = v_x$

$\langle T[\langle \alpha, \tau \rangle \mapsto \langle \textbf{getArg}\langle\, x, e_b, r\,\rangle : k, \alpha \rangle], \rho \rangle \longrightarrow \langle T[\langle \alpha, \tau \rangle \mapsto \langle e_b, r[x \mapsto v_x], k \rangle], \rho' \rangle$
where $\rho' = (\rho\, \alpha)/x$, $x \in Dom((\rho\, \alpha))$ and $((\rho\, \alpha)\, x) = v_x$

Fig. 9. Transition rule for coordination sublanguage assuming a CmPS source-level transformation.

5 Computation Migration

The effectiveness of communication-passing style depends greatly on the size of continuations of coordination operations. The larger these continuations, the greater the bandwidth requirements imposed. Moreover, if a continuation is closed over a large structure that never happens to be accessed on the address spaces to which the continuation migrates, efficiency is likely to be significantly impacted. As an example, consider the following program fragment:

```
(let ((z big object))
  (lambda (AS-list)
    (map (lambda (AS) (cons z (rd AS [x] x)))
         AS-list)))
```

The procedure returned by this expression maps over a list of address spaces `AS-list`, pairing a large locally-defined structure `z` with the value of shared variable `x` on each address space in `AS-list`. The continuation of the `rd` expression includes `z`. Using a CmPS-based implementation defined earlier, this continuation would migrate to each address space in the list, performing the pairing on the remote site.

Note that in this example, no interesting portion of `rd`'s continuation, other than the topmost frame, is necessary to evaluate the operation's body. Suppose only this piece of the continuation were sent as the context within which the `rd` operation evaluates on `AS`, with the rest of the continuation resident on the sender S. In this case, the value of `x` found on the different address spaces in `AS-list` would be sent back to S whereupon the pairing within `z` would be performed. Communication costs would be significantly reduced since `z` would not be unnecessarily copied as the part of the migrated continuation.

The tradeoff here is added communication: whereas the CmPS scheme described earlier would migrate the task's continuation to each address space in `AS-list` in turn, this reduced copying scheme would cause the task's continuation to always return back to the original address space before migrating to the next address space in the list because of the need to perform the pairing on the sender. Conceptually, this scheme suggests that a continuation be distributed among a collection of address spaces. If control ever passes through the base of a migrated continuation, the continuation below the base is resumed. Thus, falling off the end of a migrated continuation is tantamount to shifting control back to the address space where the suspended portion resides. This approach, sometimes referred to as *computation migration* [13] reduces bandwidth requirements while potentially increasing the number of remote communication events.

To support computation migration, we extend the coordination language with one new operation. The expression, (! e) marks the top frame of the current continuation stack, and returns the result of evaluating e. The specification of the abstract machine is now changed to ensure that no continuation frames below the most recently marked "frozen" frame are migrated. This annotation is closely related to "prompts" [6], a mechanism to delimit the extent of continuation-capturing operations.

To achieve the desired degree of migration, the above expression could be rewritten thus:

```
(let ((z big object))
  (lambda (AS-list)
    (map (lambda (AS) (cons z (! (rd AS [x] x))))
         AS-list)))
```

The `freeze` operation ensures that the continuation within which the `rd` expression evaluates on `AS` is a single *ArgFrame* to hold the result of the second argument to `cons`. When control passes back to the suspended continuation on the caller address-space, the result is supplied as the continuation's argument. Further evaluation of this expression now proceeds on the sender.

As another example, consider a slightly different implementation of the above program:

```
(let ((z big object))
  (lambda (AS-list)
    (map (lambda (elt) (cons z (list elt)))
         (! (map (lambda (AS) (rd AS [x] x)) AS-list)))))
```

In this program, the values of `x` are first explicitly collected before being paired with `z`. The expression surrounded by "`!`" performs a series of `rd` operations on the address spaces found in `AS-list`. Each of these operations involve migrating a continuation to a new address space. The program ensures that the continuation sent to perform the `rd` does not include `z`. This implementation is quite unlike the previous one, which caused every remote `rd` operation to return its result back to the sender. In this version, `rd`'s continuation appends to the list built by `map`, and executes a recursive call on the next address space in the list. Consequently, control moves among the address spaces found in `AS-list`, and returns to the sender only after all distributed members have been collected.

As this example illustrates, judicious use of the "freeze" annotation can make a computation migration strategy an effective optimization in a CmPS-based implementation, and may significantly reduce the bandwidth requirements of communication-intensive applications.

$\langle (!\ e), r, k \rangle \longrightarrow \langle e, r, \mathbf{freeze}\langle\,\rangle : k \rangle$
$\langle \mathbf{freeze}\langle\,\rangle : k, v \rangle \longrightarrow \langle k, v \rangle$
$\langle (\mathtt{put}\ e_a\ [x\ e_x]\ e_b), r, k \rangle \longrightarrow \langle e_a, r, \mathbf{putArg}\langle\, x, e_x, e_b, r\,\rangle : k \rangle$
$\langle (\mathtt{rd}\ e\ [x]\ e_b), r, k \rangle \longrightarrow \langle e, r, \mathbf{rdArg}\langle\, x, e_b, r\,\rangle : k \rangle$
$\langle (\mathtt{get}\ e\ [x]\ e_b), r, k \rangle \longrightarrow \langle e, r, \mathbf{getArg}\langle\, x, e_b, r\,\rangle : k \rangle$

$\langle \mathbf{putArg}\langle\, x, e_x, e_b, r\,\rangle : k, v \rangle \longrightarrow \langle e_x, r, \mathbf{putBody}\langle\, x, e_b, v, r\,\rangle : k \rangle$
$\langle T[\langle \alpha, \tau \rangle \mapsto \langle \mathbf{putBody}\langle\, x, e_b, v, r\,\rangle : k^* : \mathbf{freeze}\langle\,\rangle : k, v_a \rangle], \rho \rangle \longrightarrow$
$\quad \langle T[\langle \alpha, \tau \rangle \mapsto \mathbf{suspend}\langle\, k\,\rangle,\ \langle v, \tau' \rangle \mapsto \langle e_b, r, k^* : \mathbf{migrate}\langle\, \alpha, \tau\,\rangle \rangle], (\rho\, v)[x \mapsto v_a] \rangle$
$\quad$ where τ' is fresh, $k^* = k_1 : \ldots : k_n$ and $k_i \neq \mathbf{freeze}\langle\,\rangle$, $1 \leq i \leq n$

$\langle T[\langle \alpha, \tau \rangle \mapsto \langle \mathbf{rdArg}\langle\, x, e_b, r\,\rangle : k^* : \mathbf{freeze}\langle\,\rangle : k, v \rangle], \rho \rangle \longrightarrow$
$\quad \langle T[\langle \alpha, \tau \rangle \mapsto \mathbf{suspend}\langle\, k\,\rangle,\ \langle v, \tau' \rangle \mapsto \langle e_b, r[x \mapsto v_x], k^* : \mathbf{migrate}\langle\, \alpha, \tau\,\rangle \rangle], \rho \rangle$
$\quad$ where τ' fresh, $x \in Dom((\rho\, v))$, $((\rho\, v)\, x) = v_x$,
$\quad k^* = k_1 : \ldots : k_n$, and $k_i \neq \mathbf{freeze}\langle\,\rangle$, $1 \leq i \leq n$

$\langle T[\langle \alpha, \tau \rangle \mapsto \langle \mathbf{rdArg}\langle\, x, e_b, r\,\rangle : k^* : \mathbf{freeze}\langle\,\rangle : k, v \rangle], \rho \rangle \longrightarrow$
$\quad \langle T[\langle \alpha, \tau \rangle \mapsto \mathbf{suspend}\langle\, k\,\rangle,\ \langle v, \tau' \rangle \mapsto \langle e_b, r[x \mapsto v_x], k^* : \mathbf{migrate}\langle\, \alpha, \tau\,\rangle \rangle], \rho \rangle$
$\quad$ where τ' fresh, $\rho' = (\rho\, v)/x$, $x \in Dom((\rho\, v))$, $((\rho\, v)\, x) = v_x$,
$\quad k^* = k_1 : \ldots : k_n$, and $k_i \neq \mathbf{freeze}\langle\,\rangle$, $1 \leq i \leq n$

$\langle T[\langle \alpha, \tau \rangle \mapsto \langle \mathbf{migrate}\langle\, \alpha', \tau'\,\rangle, v \rangle,\ \langle \alpha', \tau' \rangle \mapsto \mathbf{suspend}\langle\, k\,\rangle], \rho \rangle \longrightarrow$
$\quad \langle T/\langle \alpha, \tau \rangle[\langle \alpha', \tau' \rangle \mapsto \langle k, v \rangle], \rho \rangle$

Fig. 10. Transition rules to support computation migration.

Figure 10 shows the modifications to the state machine necessary to support computation migration. We add three new continuation frames: a *Freeze Frame* marks a continuation point below which continuation frames will not be migrated; a *Suspend Frame* marks that portion of a continuation which is suspended pending a return of control-flow from a migrated sub-continuation; a *Migrate Frame* records the address space and thread containing a suspended continuation. To evaluate the body e_b of a coordination expression, a new thread is created on the target address space. This thread begins evaluation of e_b using a continuation containing a series of frames whose bottom-most frame is a *Migrate Frame*. The size of this continuation depends upon freeze points currently on the continuation stack. When the machine encounters a *Migrate Frame*, **migrate**$\langle\ \alpha, \tau\ \rangle$, it enters a new state in which the suspended continuation on α is resumed using thread τ. The thread executing the *Migrate Frame* is also removed since control has now shifted back to α. When the machine encounters a *Freeze Frame*, it simply pops the frame and proceeds with the continuation below it.

It should be obvious that the behavior of any terminating program implemented using computation migration will be observationally equivalent to one using just CmPS. This is because there are no transitions from a suspended continuation, and a continuation embedded inside a *Suspend Frame* can only be extracted by a thread evaluating a *Migrate Frame* whose arguments uniquely identify the suspended computation.

6 Related Work

Most closely related to the coordination language developed here is Linda [4]. Piranha [11] is a Linda implementation of master-worker style parallelism targeted for networks of workstations. If any node becomes unavailable due to system load, Piranha causes the Piranha processes executing on that node to *retreat*. Retreating involves executing a user-supplied procedure that cleans up intermediate state generated by these processes. Because Piranha does not consider migration of full-fledged control-contexts, programmers are required to structure their programs so that intermediate computation can be easily moved by the retreat procedure.

Olden [20] supports thread migration for a distributed dialect of C. To make the implementation tractable, Olden restricts pointers into the runtime stack; thus migrated tasks are guaranteed not to have remote references into their local data stack. Prohibiting pointers into the stack, however, is likely to greatly impact expressivity in C. Like CmPS, task migration in Olden is initiated whenever a task attempts to access data resident on another processor. Olden also uses a migration scheme similar to computation migration that faults only the top-most frame of the runtime stack. It does not provide linguistic extensions to delimit the effect of a migration operation. Rather than devising separate coordination and computation sublanguages, Olden specifies parallelism using continuation-capturing operations based on futures [12] and lazy task creation [17].

Flanagan and Felleisen [9] and Moreau [18] present a CEK-based operational semantics for parallel languages supporting the *future* construct [12]. Unlike the semantics described here, neither of the above reports consider implementation issues or language extensions to handle distributed computation. Jagannathan and Weeks [14] present an abstract interpretation of a core language similar to the one described here. The language analyzed did not include continuations, however. The formal exact semantics was given in direct-style, and thus is not well-suited to specifying issues related to task migration as done here.

Process migration [19] and computation migration [13] are two approaches to moving threads in distributed environments. We have presented the semantics of computation migration using a well-specified operational semantics. A pleasant property of our formal description of computation migration is that only minor modifications to the machine state domain used by the original semantics, which migrates full continuations, is necessary to specify computation migration. We believe ours is the first attempt to present a precise operational characterization of computation migration for a coordination language.

Obliq [3] and Kali [5] are two implementations of higher-order distributed languages that would form a natural basis on which to implement the coordination language described here. Indeed, we have incorporated the ideas presented here into Kali and expect to report on its practical utility in the near future.

7 Conclusions

As networks of workstations (NOWs) become increasingly the platform of choice for parallel and distributed systems, language abstractions tailored to the particular characteristics of a NOW are required. In this paper, we have developed a semantics for a coordination language with this idea in mind. The language, based loosely on Linda, supports first-class address spaces. Address spaces can serve as virtual processors closed over a shared naming environment in a NOW ensemble.

We have presented an operational semantics in which all communication events are treated by migrating the continuation of the task initiating the event to the address space where the desired data resides. We have argued that this strategy, called CmPS, can significantly reduce communication costs in a loosely-coupled, high-latency environment such as a NOW. Furthermore, we have shown that when embedded within a computation sublanguage supporting first-class procedures, CmPS can be expressed entirely as a source-level transformation on top of a more traditional implementation. Finally, we have also sketched an optimization that may significantly reduce communication costs by allowing programmers to delimit the dynamic context of migrating continuations.

References

1. A. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.
2. A. D. Birrell and B. Nelson, *Implementing remote procedure call*, ACM Transactions on Computer Systems, 2 (1984), pp. 39–59.
3. L. Cardelli, *A Language with Distributed Scope*, in Proceedings of the 22^{nd} ACM Symposium on Principles of Programming Languages, New York, 1995, ACM, pp. 286–298.
4. N. Carriero and D. Gelernter, *Linda in Context*, Communications of the ACM, 32 (1989), pp. 444 – 458.
5. H. Cejtin, S. Jagannathan, and R. Kelsey, *Higher-Order Distributed Objects*, ACM Transactions on Programming Languages and Systems, 17 (1995), pp. 704–739.

6. M. Felleisen, *The Theory and Practice of First-Class Prompts*, in 15^{th} ACM Symposium on Principles of Programming Languages, 1988, pp. 180–190.
7. M. Felleisen and D. Friedman, *Control Operators, the SECD-machine, and the Lambda-calculus*, in 3^{rd} Working Conference on the Formal Description of Programming Concepts, North-Holland, 1986, pp. 193–219.
8. ———, *A Calculus for Assignments in Higher-Order Languages*, in Proceedings of the 14^{th} ACM Symposium on Principles of Programming Languages, 1987, pp. 314–325.
9. C. Flanagan and M. Felleisen, *The Semantics of Future and Its Use in Program Optimization*, in Proceedings of the 12^{nd} ACM Symposium on Principles of Programming Languages, 1995, pp. 290–220.
10. C. Flanagan, A. Sabry, B. Duba, and M. Felleisen, *The Essence of Compiling with Continuations*, in 1993 ACM Symposium on Programming Language Design and Implementation, 1993, pp. 237–247.
11. D. Gelernter and D. Kaminsky, *Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha*, in Proceedings of the 6^{th} ACM International Conference on Supercomputing, July 1992, pp. 417–427.
12. R. Halstead, *Multilisp: A Language for Concurrent Symbolic Computation*, Transactions on Programming Languages and Systems, 7 (1985), pp. 501–538.
13. W. Hsieh, P. Wang, and W. Weihl, *Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems*, in The 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, May 1993, ACM, pp. 239–249.
14. S. Jagannathan and S. Weeks, *Analyzing Stores and References in a Parallel Symbolic Language*, in Proceedings of the 1994 ACM International Conference on Lisp and Functional Programming, 1994, pp. 294–305.
15. E. Jul, H. Levy, N. Hutchison, and A. Black, *Fine-Grained Mobility in the Emerald System*, ACM Transactions on Computer Systems, 6 (1988), pp. 109–133.
16. R. Kelsey and P. Hudak, *Realistic Compilation by Program Transformation*, in Proceedings of the 16^{th} ACM Symposium on Principles of Programming Languages, 1989, pp. 281–292.
17. R. Mohr, D. Kranz, and R. Halstead, *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*, in Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, June 1990.
18. L. Moreau, *The Semantics of Scheme with Future*, in Proceedings of the 1996 ACM International Conference on Functional Programming, 1996, pp. 146–156.
19. M. Powell and B. Miller, *Process Migration in DEMOS/MP*, in Proceedings of the 9th ACM Symposium on Operating Systems Principles, New York, 1983, ACM, pp. 110–119.
20. A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, *Supporting Dynamic Data Structures on Distributed-Memory Machines*, ACM Transactions on Programming Languages and Systems, 17 (1995), pp. 233–263.
21. Z. Shao and A. Appel, *Space-Efficient Closure Representations*, in Proceedings of the 1994 ACM Symposium on Lisp and Functional Programming, 1994, pp. 150–161.

A Transition Rules for the Computation Language

To describe transitions dealing only with L's computation sublanguage, it is often convenient to write $t \longrightarrow t'$
to mean $\langle T[\langle \alpha, \tau \rangle \mapsto t], \rho \rangle \longrightarrow \langle T[\langle \alpha, \tau \rangle \mapsto t'], \rho \rangle$ where $\langle \alpha, \tau \rangle \in \mathit{Dom}(T)$.

There are two groups of rules for the computation sublanguage. The first group describes transitions on control states; the second group describes transitions on return states. The machine immediately evaluates constants, variables, and abstractions, transferring the result to the current continuation. For applications, a frame holding the value of the argument to the call is pushed on the stack while the function position is evaluated. Primitives and conditionals are treated similarly.

Return states manipulate the continuation stack. The machine enters such a state when a continuation must be applied. This event occurs whenever the machine reduces an expression to a value and must transfer control to the current continuation. To illustrate, consider the actions that the machine must perform to handle a function call. Assuming that the function position yields a closure, the top frame of the current continuation will hold the argument to the call. To evaluate this argument, the machine executes a transition to a new control state in which the argument frame has been popped, and the argument expression is now the next expression to be evaluated. A new frame holding the value of the closure is also pushed. At some later point, when the argument is also reduced to a value, the body of the closure becomes the current expression, the environment in which the expression is evaluated is augmented with a binding for the formal to the value yielded by evaluation of the argument, and the function frame is popped from the continuation stack.

$$
\begin{array}{l}
\langle c, r, k \rangle \longrightarrow \langle k, c \rangle \\
\langle x, r, k \rangle \longrightarrow \langle k, (r\, x) \rangle \\
\langle \lambda\, x.\, e, r, k \rangle \longrightarrow \langle k, \langle \lambda\, x.\, e, r \rangle \rangle \\
\langle \mu\, f.\lambda\, x.\, e, r, k \rangle \longrightarrow \langle k, \langle \mu\, f.\lambda\, x.\, e, r \rangle \rangle \\
\langle (e_1\, e_2), r, k \rangle \longrightarrow \langle e_1, r, \mathbf{arg}\langle e_2, r \rangle : k \rangle \\
\langle (p\ e_1\ \ \vec{e}), r, k \rangle \longrightarrow \langle e_1, r, \mathbf{prim}\langle p, \langle \rangle, \vec{e}, r \rangle : k \rangle \\
\langle (\mathtt{if}\, e_1\ e_2\ e_3), r, k \rangle \longrightarrow \langle e_1, r, \mathbf{cond}\langle e_2, e_3, r \rangle : k \rangle
\end{array}
$$

$$
\begin{array}{l}
\langle \langle \rangle, v \rangle \longrightarrow \mathbf{halt}(v) \\
\langle \mathbf{arg}\langle e', r \rangle : k, v \rangle \longrightarrow \langle e', r, \mathbf{fun}\langle v \rangle : k \rangle \\
\langle \mathbf{fun}\langle \langle \lambda\, x.\, e, r \rangle \rangle : k, v \rangle \longrightarrow \langle e, r[x \mapsto v], k \rangle \\
\langle \mathbf{fun}\langle \langle \mu\, f.\lambda\, x.\, e, r \rangle \rangle : k, v \rangle \longrightarrow \langle e, r[x \mapsto v,\ f \mapsto \langle \mu\, f.\lambda\, x.\, e, r \rangle], k \rangle \\
\langle \mathbf{prim}\langle p, \vec{v}, \langle \rangle, : \rangle k, v \rangle \longrightarrow \langle k, \mathcal{P}(p\, ,\, v @\ \vec{v}) \rangle \\
\langle \mathbf{prim}\langle p, \vec{v}, \vec{e} : e, : \rangle k, v \rangle \longrightarrow \langle e, r, \mathbf{prim}\langle p, v : \vec{v}, \vec{e}, : \rangle k \rangle \\
\langle \mathbf{cond}\langle e_t, e_f, r \rangle : k, \mathbf{true} \rangle \longrightarrow \langle e_t, r, k \rangle \\
\langle \mathbf{cond}\langle e_t, e_f, r \rangle : k, \mathbf{false} \rangle \longrightarrow \langle e_f, r, k \rangle
\end{array}
$$

This article was processed using the LaTeX macro package with LLNCS style