Continuation-based Transformations 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 several operational semantics for a coordination language that supports first-class (shared) distributed data repositories. The computation sub-language considered is an untyped call-by-value functional language similar to pure Scheme. The first semantics describes a conventional synchronous message-passing implementation; the second is a formulation of a CmPS implementation; and, the third refines this implementation to support computation migration, a technique to lazily migrate control state. Using computation migration, an implementation "distributes" a continuation among multiple machines reducing bandwidth requirements needed to support thread mobility. We prove the equivalence of all three systems and describe optimizations and implementation issues that arise from using a CmPS-driven coordination language.

Keywords: continuations, distributed programming, operational semantics, task migration, tuple-spaces.

1 Introduction

Implementations of functional languages often first transform the source program into *continuation-passing style* (CPS) [App92, KH89]. 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

* This paper extends work previously published in the proceedings of the 1997 International Conference on Coordination Models and Languages.

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.

Ideally, a communication-passing style transform restricts the responsibility of the programmer to devising a 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.

The motivation for considering CmPS-based implementations for distributed coordination languages stems from the observation that networks of commodity workstations (NOWs) or clusters of PCs are likely soon to become the preferred platform for building long-lived parallel and distributed programs [ACPT95]. Under existing implementations, applications appear ill-suited to execute well on such platforms. Implementations of distributed (or parallel) languages, regardless of whether they assume a coordination model or not, rarely provide tasks any power in determining *where* their computation is performed. 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. Implementations often make decisions on where a computation executes without due consideration to the computation's dataflow properties. We believe that exploiting these properties will be crucial in constructing efficient implementations of distributed programs in a loosely-coupled, potentially time-shared environment, such as a NOW.

Besides efficiency considerations, correctness and consistency becomes important concerns in pure message-passing implementations. Indeed, typical implementations in this environment either require data shared among tasks running on different machines to be copied back-and-forth to ensure global consistency, or require administrative messages to be broadcast that invalidate old cached copies of shared data on different machines. 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 [CG89] 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 defines *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 expresses both message-passing and CmPS-style coordination.

The paper is organized as follows. The next section presents a direct-style formal operational

semantics for our language using a CEK-machine [FF86, FF87] formulation. Threads in the reference semantics are stationary and access global state using message-passing primitives. Section 3 presents a refinement of this semantics that describes a CmPS evaluation strategy. We proceed to prove a correspondence theorem that relates the two semantics 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. These optimizations are defined as a further refinement of the CmPS semantics; as before, correctness is shown in terms of a correspondence theorem that relates the refinements to behavior in the original reference semantics. Sections 6 and 7 provide comparison to related work and conclusions.

2 The Language

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_M machine, a parallel extension of the CEK-machine [FF86]. 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 machine implements coordination in terms of synchronous message-passing.

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. Unlike Scheme, the computation language does not include primitives such as pointer equality (`eq?`) or assignment (`set!`). Assignment can be simulated using operations provided by the coordination language which, like Linda, allows shared data to be mutated. Since these shared data structures are first-class, they can be used by the computation language to represent mutable cells. Pointer equality tests are problematic in a distributed environment in which objects may migrate among machines.

To simplify the presentation, we also assume programs are written in A-normal form [FSDF93, SF94]. Thus, the results of all expressions are named, and arguments in calls to functions and primitives are either constants, variables, or abstractions.

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` $AS\ e$), in environment r creates a new thread on address space AS to evaluate e in r.

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, and does not change the locus of control from the current address space. 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 with bindings found in shared environments cannot be renamed in the contexts in which they are referenced.

$$
\begin{array}{lcl}
x, y, z, \ldots & \in & \mathit{Var} \\
e \in E & ::= & \mathit{Comp} \mid \mathit{Coord} \\
e_\lambda \in \mathit{Comp} & ::= & b \\
 & \mid & x \\
 & \mid & \texttt{let}\, x = b\, \texttt{in}\, e\, \texttt{end} \\
 & \mid & \texttt{let}\, x = y\,(z)\, \texttt{in}\, e\, \texttt{end} \\
 & \mid & \texttt{let}\, x = (\texttt{if}\, y\, e_2\, e_3)\, \texttt{in}\, e\, \texttt{end} \\
 & \mid & (p\, x_1\, \ldots\, x_n) \\
e_\pi \in \mathit{Coord} & ::= & (\texttt{put}\, x\, [y\, z]\, e) \\
 & \mid & (\texttt{rd}\, x\, [y]\, e) \\
 & \mid & (\texttt{get}\, x\, [y]\, e_b) \\
 & \mid & (\texttt{spawn}\, x\, e_b) \\
b \in \mathit{BaseExp} & ::= & c \mid \lambda\, x.\, e \mid\ \mu\, f.\lambda\, x.\, e \\
p \in \mathit{Primop} & ::= & + \mid\ - \mid \ldots \mid \texttt{makeAS}
\end{array}
$$

Fig. 1. The language.

Readers familiar with Linda [CG89] 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 Examples

To illustrate our language, we consider two simple examples. To make the presentation more readable, we use pure Scheme as the computation sub-language, and italicize operators belonging to the coordination sublanguage.

As a first example, consider the specification of a simple RPC mechanism. The expression, `(remote <`*AS*`> <`*thunk*`>)` applies *thunk* on address space *AS* and returns the result of the application to its caller. We can define `remote` thus:

```
(define (remote AS thunk)
  (begin
    (spawn AS
        (let ((z  (thunk)))
          (put AS [x z] x)))
    (get AS [x] x)))
```

A new thread is spawned on the target address space to compute the application of `thunk`; the result is stored in a shared variable named `x` (assumed fresh) that is removed from the shared environment by the `get` operation on the sender[2]. By way of comparison, note that the expression,

[2] In practice, to avoid unwanted name capture, the address space would be created locally within

```
(spawn AS' (remote AS (lambda () (put AS [x' z'] e)))),
```

is not equivalent to

```
(spawn AS' (put AS [x' z'] e))
```

since `e` is evaluated on `AS` in the first case, and on `AS'` in the second.

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

Fig. 2. A simple tree-structured parallel program.

As a second example, consider the code fragment shown in Figure 2. 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 (using the procedure `merge`), and deposits the result as its binding-value for `val` on its address space.

In 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.

2.2 A Message-Passing Semantics

To specify a CEK_M-machine, we define a set of states, $State_M$, and a binary transition relation, $\longrightarrow_M$, on states:

$$CEK_M = \langle State_M, \longrightarrow_M \rangle$$

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 3 and 4.

`remote`, with the physical location on which the address space should be created supplied as the argument.

$$
\begin{array}{rl}
s_M \in \mathit{State}_M & = \mathit{ThreadMap}_M \times \mathit{SharedEnv} \\
T_M \in \mathit{ThreadMap}_M & = \mathit{AspaceUid} \times \mathit{ThreadUid} \to \mathit{ThreadState}_M \\
\rho \in \mathit{SharedEnv} & = \mathit{AspaceUid} \to \mathit{Env} \\
\alpha \in \mathit{AspaceUid} & \\
\tau \in \mathit{ThreadUid} &
\end{array}
$$

Fig. 3. CEK-machine Thread and Address Space States

$$
\begin{array}{rl}
t_M \in \mathit{ThreadState} & = \mathit{ControlState} + \mathit{ReturnState} + \\
& \quad \mathit{WaitState} + \mathit{HaltState} \\
\sigma \in \mathit{ControlState} & = \mathit{Exp} \times \mathit{Env} \times \mathit{Cont} \\
\mathit{ret} \in \mathit{ReturnState} & = \mathit{Cont} \times \mathit{Value} \\
w \in \mathit{WaitState} & = \mathit{WaitFrame} + \mathit{ReadFrame}_M + \mathit{GetFrame}_M + \\
& \quad \mathit{StoreFrame}_M + \mathit{SynchFrame} \\
\mathbf{halt}(v) \in \mathit{HaltState} & = \mathit{Value} \\
r \in \mathit{Env} & = \mathit{Var} \to \mathit{Value} \\
k \in \mathit{Cont} & = \mathit{Frame}^* \\
\mathbf{ret}\langle x, e, r\rangle \in \mathit{Frame} & = \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\
\mathbf{wait}\langle e, x, r, \alpha, k\rangle \in \mathit{WaitFrame} & = \mathit{Exp} \times \mathit{Var} \times \mathit{Env} \times \mathit{AspaceUid} \times \mathit{Cont} \\
\mathbf{read}_M\langle x\rangle \in \mathit{ReadFrame}_M & = \mathit{Var} \\
\mathbf{remove}_M\langle x\rangle \in \mathit{GetFrame}_M & = \mathit{Var} \\
\mathbf{store}_M\langle x, v\rangle \in \mathit{StoreFrame}_M & = \mathit{Var} \times \mathit{Value} \\
\mathbf{synch}\langle v\rangle \in \mathit{SynchFrame} & = \mathit{Value} \\
v \in \mathit{Value} & = \mathit{Constant} + \mathit{Closure} + \mathit{AspaceUid} + \\
& \quad \mathit{ThreadUid} \\
\langle \lambda\, x.\, e, r\rangle, \langle \mu\, f . \lambda\, e., r\rangle \in \mathit{Closure} & = (\mathit{LambdaExp} \times \mathit{Env}) + (\mathit{RecExp} \times \mathit{Env})
\end{array}
$$

Fig. 4. CEK_M-machine Thread States

In the following, we write $X + Y$ and $X \times Y$ to mean the cartesian sum and product of sequences (ordered sets) X and Y, resp. We write $X \to Y$ 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 and $X : Y$ to denote the append of sequence X onto sequence Y.

We also write $\langle T, \rho\rangle \xrightarrow{\langle \alpha, \tau\rangle}_M \langle T', \rho'\rangle$ to indicate a transition from state $\langle T, \rho\rangle$ to $\langle T', \rho'\rangle$ evaluating thread τ in address space α. Since we will be defining several semantics, we sometimes omit the suffix on $\longrightarrow$ when the transition rule is common to all the semantics we consider.

Machine States A *state* in our semantics is a pair consisting of a *ThreadMap*, T_M, and a global shared environment, ρ. T_M 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.[3]

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

A given thread τ may be in one of four states. If τ is in *control state*, σ, 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. Threads involved in reading or extending shared environments enter into a *wait state*, and exit when the operation completes. Wait states are thus used to model synchronization requirements. 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. Since the language is in A-normal form, continuations are used only to handle non tail-recursive function calls. In the following, we write k_0 to mean the initial continuation.

The language supports four kinds of values: constants, closures, and 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 $\longrightarrow_M$, we assume a partial function to apply primitive operations to constants:

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

Transition Relation We can think of $\longrightarrow_M$ 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 guarantees 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_M, we define $\longrightarrow_M$, 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 $\stackrel{*}{\longrightarrow}_M$ be the reflexive transitive closure of $\longrightarrow_M$. The semantics of a program P under a CEK_M machine is defined by a relation $\mathit{eval}_M(P)$ where

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

$$\langle T_0[\langle \alpha_0, \tau_0\rangle \mapsto \langle P, r_0, k_0\rangle], \rho_0\rangle \stackrel{*}{\longrightarrow}_M \quad \langle T_0[\langle \alpha_0, \tau_0\rangle \mapsto \mathbf{halt}(v_0) \ldots \langle \alpha_n, \tau_n\rangle \mapsto \mathbf{halt}(v_n)],\ \rho\rangle$$

The transition relation consists of three kinds of rules. The first group describes single-threaded behavior and is shown in Figure 5. The second group of rules defines the creation of new address spaces and threads; it is given in Figure 6. Finally, the last group of rules defines the semantics of the coordination language and is shown in Figure 7.

The Computation Sub-language. There are two kinds of rules for the computation sub-language. The first kind describes transitions on control states; the second describes transitions

on return states. The machine immediately evaluates constants, variables, and abstractions, transferring the result to the current continuation. For non-tail recursive applications, a **ret** frame is pushed on the stack. This frame holds the name to be bound to the result. Another frame is pushed to record the argument while the binding-value of the variable found in the function position is extracted from the environment.

To describe transitions dealing only with L's computation sub-language, it is often convenient to write $t \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} t'$ to mean:

$$\langle T_M[\langle \alpha\,,\tau\rangle \mapsto t], \rho\rangle \longrightarrow_M \langle T_M[\langle \alpha\,,\tau\rangle \mapsto t'], \rho\rangle$$

where $\langle \alpha\,,\tau\rangle \in Dom(T_M)$.

$\langle c, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle k, c\rangle$

$\langle x, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle k, r\,(x)\rangle$

$\langle \lambda\, x.\, e, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle k, \langle \lambda\, x.\, e, r\rangle\rangle$

$\langle \mu\, f.\lambda\, x.\, e, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle k, \langle \mu\, f.\lambda\, x.\, e, r\rangle\rangle$

$\langle \texttt{let}\ x\ = c\ \texttt{in}\ e\ \texttt{end}, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e, r[x \mapsto c], k\rangle$

$\langle \texttt{let}\ x\ = \lambda\, x.\, e\ \texttt{in}\ e_b\ \texttt{end}, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e_b, r[x \mapsto \langle \lambda\, x.\, e, r\rangle], k\rangle$

$\langle \texttt{let}\ x\ = \mu\, f.\lambda\, x.\, e\ \texttt{in}\ e_b\ \texttt{end}, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e_b, r[x \mapsto \langle \mu\, f.\lambda\, x.\, e, r\rangle], k\rangle$

$\langle \texttt{let}\ x\ = y\,(z)\ \texttt{in}\ z\ \texttt{end}, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle y\,(z), r, \mathbf{ret}\langle x, e, r\rangle : k\rangle$

$\langle \texttt{let}\ x\ = p\,(x_1, x_2, \ldots, x_n)\ \texttt{in}\ e\ \texttt{end}, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e, r[x \mapsto \mathcal{P}(p\,,\ r(x_1) \ldots r(x_n))], k\rangle$

$\langle y\,(z), r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e, r'[x \mapsto v], k\rangle$
provided $r\,(y) = \langle \lambda\, x.\, e, r'\rangle$ and $r\,(z) = v$

$\langle y\,(z), r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e, r'[x \mapsto v,\ f \mapsto \langle \mu\, f.\lambda\, x.\, e, r'\rangle], k\rangle$
provided $r\,(y) = \langle \mu\, f.\lambda\, x.\, e, r'\rangle$ and $r\,(z) = v$

$\langle \texttt{let}\ z\ = (\texttt{if}\ x\ e_t\ e_f)\ \texttt{in}\ e\ \texttt{end}, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e_t, r, \mathbf{ret}\langle z, e, r\rangle : k\rangle$
provided $r\,(x) = \mathbf{true}$

$\langle \texttt{let}\ z\ = (\texttt{if}\ x\ e_t\ e_f)\ \texttt{in}\ e\ \texttt{end}, r, k\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e_f, r, \mathbf{ret}\langle z, e, r\rangle : k\rangle$
provided $r\,(x) = \mathbf{false}$

$\langle \langle\rangle, v\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \mathbf{halt}(v)$

$\langle \mathbf{ret}\langle x, e, r\rangle : k, v\rangle \stackrel{\langle \alpha\,,\tau\rangle}{\longrightarrow} \langle e, r[x \mapsto v], k\rangle$

Fig. 5. Transition rules for the computation sub-language.

Threads and Address Spaces. There are two "top-level" rules that specify how new threads and address spaces are added to the global state. To create a new thread, a new thread uid is generated and the thread is instantiated on the specified address space 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. When a new thread is spawned, a unique thread id is created. This identifier may be retrieved from a central server, or may be constructed using globally unique information such as the machine name of the address space in which the thread executes. 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.

$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{spawn}\ AS\ e), r, k\rangle], \rho\rangle \longrightarrow_M$
$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle k, \tau'\rangle,\ \langle r(AS), \tau'\rangle \mapsto \langle e, r, k_0\rangle], \rho\rangle$
$\quad$ provided $r(AS) \in Dom(\rho)$ and τ' unique

$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{makeAS}), r, k\rangle], \rho\rangle \longrightarrow_M$
$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle k, \alpha\,'\rangle], \rho[\alpha\,' \mapsto \lambda\, x.\ undef]\rangle$
$\quad$ provided $\alpha\,' \notin Dom(\rho)$

Fig. 6. Transition rules for threads and address spaces.

The Coordination Sub-language. The rules defining the semantics of the coordination sub-language describe how shared environments and thread maps are manipulated by coordination operators. Consider a `put` operation executing as part of thread τ 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 creates an extension of τ on AS_t to update the shared environment resident on AS_t. The current thread of control on AS_s must block until the update has occurred. Note that our semantics thus allows τ to exist on two address spaces simultaneously. However, there is at most one transition available to a thread regardless of how many fragments exist. This is because a thread that extends itself on a different address space immediately enters a blocked (or **wait**) state on the source. There are no transitions from this state available until the subordinate thread completes and enters a **synch** state. At this point, the subordinate can induce no further transitions, and the parent can resume execution. The behavior described by this semantics thus conforms closely to a synchronous message-passing implementation of distributed data structures.

Transitions on `rd` and `get` are defined similarly. For example, a `rd` operation extends the current thread of control to the address space supplied as the operator's first argument. On this address space, the thread initiates a *local* read operation, then enters a **synch** state, and supplies the value read when complete. The portion of the thread containing `rd`'s continuation remains resident on the source and is blocked until this condition is achieved; blocking is modelled using the **wait** primitive as before.

3 Communication-Passing Style

Our message-passing semantics requires a thread to initiate a remote computation whenever shared data must be accessed or manipulated. Computation remains stationary, while shared data is copied to the threads requiring them. An alternative implementation of distributed data structures is based on a shift in perspective: instead of moving data to the computation that needs it, computation can move dynamically to where the required data resides. In this section, we present an implementation of this strategy.

$$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{put}\ AS\,[x\ y]\ e_b), r, k\rangle], \rho\rangle \longrightarrow_M$$
$$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{wait}\langle e_b, x, r, r\,(AS), k\rangle,\ \langle r\,(AS), \tau\rangle \mapsto \mathbf{store}_M\langle x, r\,(y)\rangle], \rho\rangle$$

$$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{rd}\ AS\,[x]\ e_b), r, k\rangle], \rho\rangle \longrightarrow_M$$
$$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{wait}\langle e_b, x, r, r\,(AS), k\rangle,\ \langle r\,(AS), \tau\rangle \mapsto \mathbf{read}_M\langle x\rangle], \rho\rangle$$

$$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{get}\ AS\,[x]\ e_b), r, k\rangle], \rho\rangle \longrightarrow_M$$
$$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{wait}\langle e_b, x, r, r\,(AS), k\rangle,\ \langle r\,(AS), \tau\rangle \mapsto \mathbf{remove}_M\langle x\rangle], \rho\rangle$$

$$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{store}_M\langle x, v\rangle], \rho\rangle \longrightarrow_M$$
$$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{synch}(v)], \rho[\alpha \mapsto \rho\,(\alpha\,)[x \mapsto v]]\rangle$$

$$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{read}_M\langle x\rangle], \rho\rangle \longrightarrow_M$$
$$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{synch}(v)], \rho\rangle$$
$$\quad \text{provided } x \in \mathit{Dom}(\rho\,(\alpha\,)) \text{ and } \rho\,(\alpha\,)\,(x) = v$$

$$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{remove}_M\langle x\rangle], \rho\rangle \longrightarrow_M$$
$$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{synch}(v)], \rho[\alpha \mapsto \rho\,(\alpha\,)/x]\rangle$$
$$\quad \text{provided } x \in \mathit{Dom}(\rho\,(\alpha\,)) \text{ and } \rho\,(\alpha\,)\,(x) = v$$

$$\langle T_M[\langle \alpha\,, \tau\rangle \mapsto \mathbf{wait}\langle e, x, r, \alpha\,', k\rangle,\ \langle \alpha\,', \tau\rangle \mapsto \mathbf{synch}(v)], \rho\rangle \longrightarrow_M$$
$$\quad \langle T_M[\langle \alpha\,, \tau\rangle \mapsto \langle e, r[x \mapsto v], k\rangle], \rho\rangle$$

Fig. 7. Transition rules for the coordination sub-language implemented using message-passing.

$$CEK_C = \langle \mathit{State}_C, \longrightarrow_C \rangle$$

$$\begin{aligned}
s_C \in \mathit{State}_C &= \langle\, \mathit{ThreadMap}_C, \mathit{SharedEnv}\rangle \\
T_{CM} \in \mathit{ThreadMap}_C &= \mathit{AspaceUid} \times \mathit{ThreadUid} \to \mathit{ThreadState}_C \\
t_C \in \mathit{ThreadState}_C &= \mathit{ControlState} + \mathit{ReturnState} + \\
&\quad \mathit{MigrateState} + \mathit{HaltState} \\
m \in \mathit{MigrateState} &= \mathit{StoreFrame}_C + \mathit{ReadFrame}_C + \\
&\quad \mathit{RemoveFrame}_C \\
\mathbf{store_C}\langle x, v, \sigma\rangle \in \mathit{StoreFrame}_C &= \mathit{Var} \times \mathit{Value} \times \mathit{ControlState} \\
\mathbf{read_C}\langle x, \sigma\rangle \in \mathit{ReadFrame}_C &= \mathit{Var} \times \mathit{ControlState} \\
\mathbf{remove_C}\langle x, \sigma\rangle \in \mathit{RemoveFrame}_C &= \mathit{Var} \times \mathit{ControlState}
\end{aligned}$$

Fig. 8. CEK_C machine states.

The specification uses a slightly different machine from the message-passing semantics, but no synchronization primitives are required. The modified domain equations for this semantics is shown in Figure 8. In particular, the continuation of a thread is never separated from the current frame, and thus there is never any need to enter into a **wait-** or **synch-**state. Instead, threads enter into a **migrate-**state that reflects the movement of the thread from one address space to another initiated when the thread accesses or manipulates (global) shared bindings. The computation core as well as operations to create address spaces and threads remain the

same with the caveat that the machine operates over $\langle T_{CM}, \rho\rangle$ states instead of $\langle T_M, \rho\rangle$ ones on `spawn` and `(makeAS)` expressions to conform to the signature of the transition relation. We define the meaning of a program under $\longrightarrow_C$ via a relation $eval_C$ defined similarly to $eval_M$.

We must now define new rules for reading, removing, and depositing bindings into a shared environment. These rules are given in Figure 9.

$$\langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{put}\ AS\,[x\ y]\ e_b), r, k\rangle], \rho\rangle \longrightarrow_C$$
$$\quad \langle T_{CM}/\langle \alpha\,, \tau\rangle[\langle r\,(AS), \tau\rangle \mapsto \mathbf{store_C}\langle x, r\,(y), \langle e_b, r, k\rangle\rangle], \rho\rangle$$

$$\langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{rd}\ AS\,[x]\ e_b), r, k\rangle], \rho\rangle \longrightarrow_C$$
$$\quad \langle T_{CM}/\langle \alpha\,, \tau\rangle[\langle r\,(AS), \tau\rangle \mapsto \mathbf{read_C}\langle x, \langle e_b, r, k\rangle\rangle], \rho\rangle$$

$$\langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \langle(\texttt{get}\ AS\,[x]\ e_b), r, k\rangle], \rho\rangle \longrightarrow_C$$
$$\quad \langle T_{CM}/\langle \alpha\,, \tau\rangle[\langle r\,(AS), \tau\rangle \mapsto \mathbf{remove_C}\langle x, \langle e_b, r, k\rangle\rangle], \rho'\rangle$$

$$\langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \mathbf{store_C}\langle x, v, \langle e, r, k\rangle\rangle], \rho\rangle \longrightarrow_C$$
$$\quad \langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \langle e, r[x \mapsto v], k\rangle], \rho[\alpha \mapsto \rho\,(\alpha\,)[x \mapsto v]]\rangle$$

$$\langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \mathbf{read_C}\langle x, \langle e, r, k\rangle\rangle], \rho\rangle \longrightarrow_C$$
$$\quad \langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \langle e, r[x \mapsto v], k\rangle], \rho\rangle$$
$$\quad \text{provided } x \in Dom(\rho\,(\alpha\,)) \text{ and } \rho\,(\alpha\,)\,(x) = v$$

$$\langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \mathbf{remove_C}\langle x, \langle e, r, k\rangle\rangle], \rho\rangle \longrightarrow_C$$
$$\quad \langle T_{CM}[\langle \alpha\,, \tau\rangle \mapsto \langle e, r[x \mapsto v], k\rangle], \rho'\rangle$$
$$\quad \text{provided } x \in Dom(\rho\,(\alpha\,)) \text{ and } \rho\,(\alpha\,)\,(x) = v \text{ where } \rho' = \rho[\alpha \mapsto \rho\,(\alpha\,)/x]$$

Fig. 9. Transition rule for coordination sublanguage using a CmPS evaluation strategy.

When a thread τ executing on address space AS_s wishes to deposit a new shared binding on an address space AS_t, a new thread is created on AS with the same thread uid as τ. Moreover, $\langle AS_s, \tau\rangle$ is removed from the thread map. This effectively models the act of migrating the thread from AS_s to AS_t. A $\mathbf{store_C}$ operation is subsequently performed; this operation records the new binding in the shared environment associated with AS_t, and resumes the execution of the `put` operation on this address space. It does so because `put`'s continuation is recorded as part of $\mathbf{store_C}$'s state, and is extracted once the binding has been deposited. A similar series of transitions are executed to read or remove a shared binding. First, the thread performing the operation is removed from the thread map and reinstantiated on the address space where the relevant binding is to be retrieved. A $\mathbf{remove_C}$ (or $\mathbf{read_C}$) operation is performed to extract the binding-value. In addition, a new control state that represents the continuation of the `get` (or `rd`) expression is installed.

3.1 Example Revisited

Consider a CmPS implementation of the example shown in Figure 2. 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 a message-passing implementation, three messages (rather than four) are initiated by each non-leaf task. Moreover, using a safe-for-space implementation of continuations [App92, SA94], the bandwidth requirement needed to support this migration is neglible since the only data referenced in the migrating continuation are to `node` and `right` both of which are accessed via global identifiers. As a result, we may expect significant performance improvement.

In this simple program, consistency issues were not critical since every binding in a shared-environment is read and written exactly once. Figure 10 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 (and removes) `val`'s binding-value on the left-child; the second performs the same computation except it reads (and removes) `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 (node) perform repeated calculation on
                           (get node [val] val)))
         (h (lambda (node)
              perform repeated calculation on
              (put node [val (merge (get node [left-val] left-val)
                                    (get node [right-val] right-val))]
                   val)))
         (tree (lambda (node v)
                 (cond ((leaf? node)
                        perform repeated calculation on
                        (put node [val (leaf v)] val))
                       (else
                         (let ((left (makeAS))
                               (right (makeAS)))
                           (spawn left (tree left (left-part v)))
                           (spawn right (tree right (right-part v)))
                           (spawn node (put node [left-val (g left)]
                                            left-val))
                           (spawn node (put node [right-val (g right)]
                                            right-val))
                           (spawn node (put node [val (h node)]
                                            val))))))))
   (tree (makeAS) initial-val))
```

Fig. 10. A simple tree-structured parallel program.

Consistency of shared data becomes an important issue here. In order to be faithful to the semantics, a message-passing implementation which chooses to cache the value yielded by `get` 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 `get` 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.

4 Equivalence

The message-passing specification of our language uses primitives for synchronization and copying to express coordination among threads. Its definition of threads is a conventional one: threads remain resident on the address space in which they are created regardless of their communication requirements. On the other hand, the CmPS specification uses no special primitives to express remote communication or synchronization. All coordination events occur locally. To achieve this, the thread model used is quite unconventional: coordination operations always require its executing thread to migrate to the address space containing the shared data of interest.

In this section, we show these two specifications are equivalent. Informally, a computation will produce the same answer regardless of which interpreter is used. In other words, the primary difference between the two systems stems from *where* computation is performed. Since the coordination language provides no reflective operations to allow threads to infer the address space on which they execute, the behavior of a computation cannot be influenced by the address space on which it resides.

We formalize this intuition by defining a third (reference) semantics that ignores address spaces entirely for the purposes of coordination and synchronization. We then proceed to construct mappings to this semantics from the message-passing and CmPS definitions and show that these mappings preserve meaning.

4.1 A Reference Semantics

Our reference semantics is defined via another interpreter, CEK_B that operates over domains which ignore address spaces:

$$CEK_B = \langle \overline{State}, \longrightarrow_B \rangle$$

In this definition, address spaces serve as indices into shared environments, and have no influence over where threads execute. The domain definition for shared environments is as before, but address spaces are used only to construct partitions of multiple bindings of the same variable in the shared environment, and unlike the earlier definitions have no physical connotation. Their definition is thus similar to the definition of first-class tuple-spaces used to construct a finer partition of a global tuple-space [Gel89] in a Linda-based coordination language.

The domain definition for states thus changes slightly:

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

$\langle \overline{T}[\tau \mapsto \langle(\texttt{spawn}\ AS\ e), r, k\rangle], \overline{\rho}\rangle \longrightarrow_B$
$\quad \langle \overline{T}[\tau \mapsto \langle k, \tau'\rangle,\ \tau' \mapsto \langle e, r, k_0\rangle], \overline{\rho}\rangle$
$\quad$ provided τ' unique.

$\langle \overline{T}[\tau \mapsto \langle(\texttt{makeAS}), r, k\rangle], \overline{\rho}\rangle \longrightarrow_B$
$\quad \langle \overline{T}[\tau \mapsto \langle k, \alpha\rangle], \overline{\rho}\rangle$
$\quad$ provided α unique.

$\langle \overline{T}[\tau \mapsto \langle(\texttt{put}\ AS\ [x\ y]\ e), r, k\rangle], \overline{\rho}\rangle \longrightarrow_B$
$\quad \langle \overline{T}[\tau \mapsto \langle e, r[x \mapsto r(y)], k\rangle], \overline{\rho}[\alpha \mapsto \overline{\rho}(\alpha)[x \mapsto r(y)]]\rangle$
$\quad$ provided $r(AS) = \alpha$

$\langle \overline{T}[\tau \mapsto \langle(\texttt{rd}\ AS\ [x]\ e), r, k\rangle], \overline{\rho}\rangle \longrightarrow_B$
$\quad \langle \overline{T}[\tau \mapsto \langle e, r[x \mapsto v], k\rangle], \overline{\rho}\rangle$
$\quad$ provided $x \in Dom(\overline{\rho}(r(AS)))$ and $\overline{\rho}(r(AS))(x) = v$

$\langle \overline{T}[\tau \mapsto \langle(\texttt{get}\ AS\ [x]\ e), r, k\rangle], \overline{\rho}\rangle \longrightarrow_B$
$\quad \langle \overline{T}[\tau \mapsto \langle e, r[x \mapsto v], k\rangle], \overline{\rho}[\alpha \mapsto \overline{\rho}(\alpha)/x]\rangle$
$\quad$ provided $r(r(AS)) = \alpha$, $x \in Dom(\overline{\rho}(r(AS)))$ and $\overline{\rho}(\alpha)(x) = v$

Fig. 11. A reference semantics for the coordination sublanguage.

The transition relation defining $\longrightarrow_B$ is shown in Figure 11. Synchronization among coordination operations in this semantics is expressed via side-conditions that enforce structure on shared environments. However, there are no explicit synchronization operations (as in the message-passing specification), nor are there any operations to move continuations or restrict the elements of the thread-map (as in the CmPS specification). This semantics closely mirrors the operational behavior of a concurrent implementation of distributed data structures. As before, we define the meaning of a program under $eval_B$ similarly to the definition of $eval_M$.

$CEK_M = CEK_B$. We now proceed to show a correspondence between states produced under $\longrightarrow_B$ and those produced under $\longrightarrow_M$. This correspondence is given via a translation function

$$F_B^M : State_M \rightarrow \overline{State}$$

shown in Figure 12 that removes address spaces and synchronization frames used to define the behavior of message-passing, but which are irrelevant for concurrency.

The translation preserves control, return and **halt** states between the two machines. However, it "abstracts" **wait** states by eliminating the distinction the CEK_M machine makes between

$$F_B^M\,(\langle T_M, \rho\rangle) = \langle F_B^M\,(T_M), \rho\rangle$$

$$F_B^M\,(T_M)\,(\tau) = \begin{cases} \langle e, r, k\rangle & \text{if } t_M = \langle e, r, k\rangle \\ \mathbf{halt}(v) & \text{if } t_M = \mathbf{halt}(v) \\ ret & \text{if } t_M = ret \\ \langle(\texttt{put}\ \alpha\ [x\ v]\ e), r, k\rangle & \text{if } t_M = \mathbf{store}_M\langle x, v\rangle \text{ and } T_M\,(\langle\alpha\,', \tau\rangle) = \\ & \mathbf{wait}\langle e, x, r, \alpha\,, k\rangle \\ \langle(\texttt{rd}\,\alpha\ [x]\ v)e, r, k\rangle & \text{if } t_M = \mathbf{read}_M\langle x\rangle \text{ and } T_M\,(\langle\alpha\,', \tau\rangle) = \\ & \mathbf{wait}\langle e, x, r, \alpha\,, k\rangle \\ \langle(\texttt{get}\ \alpha\ [x]\ v)e, r, k\rangle & \text{if } t_M = \mathbf{remove}_M\langle x\rangle \text{ and } T_M\,(\langle\alpha\,', \tau\rangle) = \\ & \mathbf{wait}\langle e, x, r, \alpha\,, k\rangle \\ \langle e, r[x \mapsto v], k\rangle & \text{if } t_M = \mathbf{synch}\langle v\rangle \text{ and } T_M\,(\langle\alpha\,', \tau\rangle) = \\ & \mathbf{wait}\langle e, x, r, \alpha\,, k\rangle \end{cases}$$

Fig. 12. Translation from machine states in a message-passing semantics to machine states in the reference semantics. We abbreviate $T_M\,(\langle\alpha\,, \tau\rangle)$ as simply t_M.

generating a communication event to effect an operation on the shared environment, and the operation itself. For example, a `put` expression in the CEK_M machine is implemented as three separate operations: (1) a store operation on the target address space, (2) a wait operation to suspend the current thread, and (3) a synchronization operation to resume execution of the blocked thread. In the CEK_B machine, on the other hand, all three operations are folded into a single transition.[4] The translation shown above reflects this abstraction. A similar transformation applies for $\mathbf{read}_M$ and $\mathbf{remove}_M$.

To show equivalence, we first prove that every state transition in a CEK_B machine can be modelled as a sequence of transitions in a CEK_M machine with the resulting states related under F_B^M as depicted in the following commutativity diagram:

$$\begin{array}{ccc} \langle\overline{T}, \overline{\rho}\rangle & \xrightarrow[B]{\tau} & \langle\overline{T}', \overline{\rho}'\rangle \\ \uparrow F_B^M & & \uparrow F_B^M \\ \langle T_M, \rho\rangle & \xrightarrow[M]{\exists\ +} & \langle T_M', \rho'\rangle \end{array}$$

We formalize this diagram in the following lemma:

Lemma 1. *If* $F_B^M\,(\langle T_M, \rho\rangle) = \langle\overline{T}, \overline{\rho}\rangle$ *and* $\langle\overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle\overline{T}', \overline{\rho}'\rangle$ *then there exists* $\langle T_M', \rho'\rangle$ *such that* $\langle T_M, \rho\rangle \xrightarrow{*}_M \langle T_M', \rho'\rangle$ *and* $F_B^M\,(\langle T_M', \rho'\rangle) = \langle\overline{T}', \overline{\rho}'\rangle$.

Proof. By cases on $\langle\overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle\overline{T}', \overline{\rho}'\rangle$.

[4] We abuse the language grammar slightly in presenting the translation and thus for example consider a control-state of the form: $\langle(\texttt{put}\ \alpha\ [x\ v]\ e), r, k\rangle$ to be equivalent to $\langle(\texttt{put}\ z\,[x\ v]\ e), r[z \mapsto \alpha\,], k\rangle$ for fresh z.

Case 1. Let $T_M(\langle\alpha,\tau\rangle) = \langle e_\lambda, r, k\rangle$. By definition of F_B^M and $\longrightarrow_B$, $\overline{T}(\tau) = \langle e_\lambda, r, k\rangle$. Since $\longrightarrow\!\!\!\!\!\rightarrow_M$ and $\longrightarrow\!\!\!\!\!\rightarrow_B$ share the same transitions on control states, simply choose T'_M from $\langle T_M, \rho\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T'_M, \rho\rangle$. By definition of $\longrightarrow$ $F_B^M(\langle T'_M, \rho\rangle) = \langle\overline{T}', \overline{\rho}\rangle$.

Case 2. Let $T_M(\langle\alpha,\tau\rangle) = \langle(\texttt{put}\, AS\, [x\, y]\, e), r, k\rangle$ where $r(AS) = \alpha'$. By definition of F_B^M and $\longrightarrow_B$, $\langle\overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle\overline{T}', \overline{\rho}'\rangle$ where $\overline{T}' = \overline{T}[\tau \mapsto \langle e, r[x \mapsto ry(]), k\rangle]$ and $\overline{\rho}' = \overline{\rho}[\alpha' \mapsto \overline{\rho}(\alpha')[x \mapsto r(y)]]$. Choose transition $\langle T_M, \rho\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T_{M1}, \rho\rangle$. By definition of $\longrightarrow_M$, $T_{M1}(\langle\alpha,\tau\rangle) = \mathbf{wait}\langle e, x, r, \alpha', k\rangle$ and $T_{M1}(\langle\alpha',\tau\rangle) = \mathbf{store}_M\langle x, r(y)\rangle$. Now, choose transition $\langle T_{M1}, \rho\rangle \xrightarrow{\langle\alpha',\tau\rangle}_M \langle T_{M2}, \rho'\rangle$. We know that $T_{M2}(\langle\alpha',\tau\rangle) = \mathbf{synch}\langle v\rangle$ where $r(y) = \rho'(\alpha')(x) = v$. Finally, consider transition $\langle T_{M2}, \rho'\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T'_M, \rho'\rangle$. By definition of F_B^M and $\longrightarrow_M$, $T'_M(\langle\alpha,\tau\rangle) = \langle e, r[x \mapsto v], k\rangle$, $\langle\alpha',\tau\rangle \notin Dom(T'_M)$ and since $F_B^M(\langle T_M, \rho\rangle) = \langle\overline{T}, \overline{\rho}\rangle$ by assumption $F_B^M(\langle T'_M, \rho'\rangle) = \langle\overline{T}', \overline{\rho}'\rangle$.

Case 3. Let $T_M(\langle\alpha,\tau\rangle) = \langle(\texttt{rd}\, AS\, [x]\, e), r, k\rangle$ where $r(AS) = \alpha'$. By definition of F_B^M and $\longrightarrow_B$, $\langle\overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle\overline{T}' = \overline{T}[\tau \mapsto \langle e, r, k\rangle], \overline{\rho}\rangle$ assuming $\overline{\rho}(\alpha')(x) = v$. (Suppose this were not true. Then, no transition from $\langle\overline{T}, \overline{\rho}\rangle$ via τ is allowed and the lemma assumption would fail.) Choose transition $\langle T_M, \rho\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T_{M1}, \rho\rangle$. By definition of $\longrightarrow_M$, $T_{M1}(\langle\alpha,\tau\rangle) = \mathbf{wait}\langle e, x, r, \alpha', k\rangle$ and $T_{M1}(\langle\alpha',\tau\rangle) = \mathbf{read}_M\langle x\rangle$. Now, choose transition $\langle T_{M1}, \rho\rangle \xrightarrow{\langle\alpha',\tau\rangle}_M \langle T_{M2}, \rho'\rangle$. We can choose this transition because $\overline{\rho}(\alpha')(x) = v$. Hence, we know that $T_{M2}(\langle\alpha',\tau\rangle) = \mathbf{synch}\langle v\rangle$ where $\rho(\alpha')(x) = v$. Finally, consider transition $\langle T_{M2}, \rho'\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T'_M, \rho'\rangle$. By definition of $\longrightarrow_M$ and F_B^M, $T'_M(\langle\alpha,\tau\rangle) = \langle e, r[x \mapsto v], k\rangle$, $\langle\alpha',\tau\rangle \notin Dom(T'_M)$ and since $F_B^M(\langle T_M, \rho\rangle) = \langle\overline{T}, \overline{\rho}\rangle$ by assumption $F_B^M(\langle T'_M, \rho'\rangle) = \langle\overline{T}', \overline{\rho}'\rangle$.

Case 4. Let $T_M(\langle\alpha,\tau\rangle) = \langle(\texttt{get}\, AS\, [x]\, e), r, k\rangle$ where $r(AS) = \alpha'$. Pick construction similar to (3).

Case 5. Let $T_M(\langle\alpha,\tau\rangle) = \langle(\texttt{spawn}\, AS\, e), r, k\rangle$ and consider transition $\langle T_M, \rho\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T'_M, \rho\rangle$. Choose fresh $\tau' \notin Dom(T')$ such that $\overline{T}'(\tau') = T'_M(\langle\alpha',\tau'\rangle) = \langle e, r, k\rangle$ where $r(AS) = \alpha'$. Then $F_B^M(T'_M) = \overline{T}'$.

Case 6. Let $T_M(\langle\alpha,\tau\rangle) = \langle(\texttt{makeAS}), r, k\rangle$ and $\langle T_M, \rho\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T'_M, \rho\rangle$. Choose fresh α' such that $T'_M(\langle\alpha,\tau\rangle) = \overline{T}'(\tau) = \langle k, \alpha'\rangle$. Then, $F_B^M(T'_M) = \overline{T}'$.

□

To prove equivalence of the two machines, however, we must also show that every transition in a CEK_M machine can be handled via a transition (possibly empty) in a CEK_B machine. In other words, we must show that every transition undertaken by the CEK_M machine corresponds to a legal transition in the CEK_B machine. Equivalence in this direction is depicted in the following diagram:

$$\begin{array}{ccc}
\langle T_M, \rho\rangle & \xrightarrow[M]{\langle\alpha,\tau\rangle} & \langle T'_M, \rho'\rangle \\
\Big\downarrow F_B^M & & \Big\downarrow F_B^M \\
\langle\overline{T}, \overline{\rho}\rangle & \underset{B}{\overset{*}{\xrightarrow{\qquad}}} & \langle\overline{T}', \overline{\rho}'\rangle \\
\exists & &
\end{array}$$

We formalize this picture in the following lemma:

Lemma 2. *If $F_B^M(\langle T_M, \rho\rangle) = \langle\overline{T}, \overline{\rho}\rangle$ and $\langle T_M, \rho\rangle \xrightarrow{\langle\alpha,\tau\rangle}_M \langle T_M, \rho'\rangle$ then there exists a transition (possibly empty) such that $\langle\overline{T}, \overline{\rho}\rangle \longrightarrow_B \langle\overline{T}', \overline{\rho}'\rangle$ and $F_B^M(\langle T'_M, \rho'\rangle) = \langle\overline{T}', \overline{\rho}'\rangle$.*

Proof. By cases on $\langle T_M, \rho\rangle \xrightarrow{\langle\alpha\,,\tau\rangle}_M \langle T'_M, \rho'\rangle$. We need only consider cases dealing with transitions unique to CEK_M; for transitions that are common to both machines, the lemma trivially holds.

Case 1. Let $T_M(\langle\alpha\,,\tau\rangle) = \mathbf{store}_M\langle x, v\rangle$. By definition of $\longrightarrow_M$, there exits $\langle\alpha\,',\tau\rangle$ such that $T_M(\langle\alpha\,',\tau\rangle) = \mathbf{wait}\langle e, x, r, \alpha\,, k\rangle$. This is because only evaluating a `put` expression pushes a **store** frame on the target address space, and a **wait** frame on the source. By definition of $F_B^M, \overline{T}(\tau) = \langle(\texttt{put}\ \alpha\ [x\ v]\ e), r, k\rangle$. Pick transition $\langle\overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle\overline{T}', \overline{\rho}'\rangle$. Now, $\overline{\rho}' = \rho'$ and $F_B^M(T'_M) = \overline{T}'$ by definition of $\longrightarrow_M$ and F_B^M.

Case 2. Let $T_M(\langle\alpha\,,\tau\rangle) = \mathbf{read}_M\langle x\rangle$. Follow construction used in (1).

Case 2. Let $T_M(\langle\alpha\,,\tau\rangle) = \mathbf{remove}_M\langle x\rangle$. Follow construction used in (1).

Case 3. Let $T_M(\langle\alpha\,,\tau\rangle) = \mathbf{synch}\langle v\rangle$. By definition of $\longrightarrow_M$, there exists $\langle\alpha\,',\tau\rangle$ such that $T_M(\langle\alpha\,',\tau\rangle) = \mathbf{wait}\langle e, x, r, \alpha\,, k\rangle$. By definition of F_B^M, $\overline{T}(\langle\alpha\,,\tau\rangle) = \langle e, r[x \mapsto v], k\rangle$ and $\overline{\rho} = \rho$. Let $\overline{T}' = \overline{T}$ and $\overline{\rho}' = \overline{\rho}$.

□

Theorem 3. *$eval_M = eval_B$.*

Proof. By Lemma 1 and Lemma 2 and induction on the length of transition sequences induced by $\longrightarrow_M$ and $\longrightarrow_B$. □

This theorem states that a program executed using the message-passing semantics will produce the same value under an interpreter that only exposes concurrency. The latter interpreter ignores issues of data and code placement in a distributed environment.

***CEK*$_C$ = *CEK*$_B$.** We now define a similar theorem for communication-passing. As before, the correspondence between communication-passing and the reference semantics is given a translation function

$$F_B^C : State_C \to \overline{State}$$

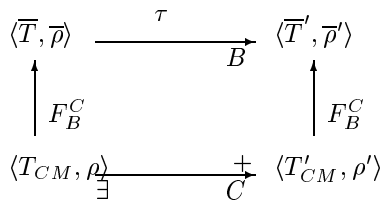
that ignores address spaces and states used to define the behavior of communication-passing style, but which are irrelevant for concurrency. In the following let $t_C = T_{CM}(\langle\alpha\,,\tau\rangle)$. Then,

$$F_B^C(\langle T_{CM}, \rho\rangle) = \langle F_B^C(T_{CM}), \rho\rangle$$

$$F_B^C(T_{CM})(\tau) = \begin{cases} \langle e, r, k\rangle & \text{if } t_C = \langle e, r, k\rangle \\ \mathbf{halt}(v) & \text{if } t_C = \mathbf{halt}(v) \\ ret & \text{if } t_C = ret \\ \langle(\texttt{put}\ \alpha\ [x\ v]\ e), r, k\rangle & \text{if } t_C = \mathbf{store_C}\langle x, v, \langle e, r, k\rangle\rangle \\ \langle(\texttt{rd}\ \alpha\ [x]\ e), r, k\rangle & \text{if } t_C = \mathbf{read_C}\langle x, \langle e, r, k\rangle\rangle. \\ \langle(\texttt{get}\ \alpha\ [x]\ e), r, k\rangle & \text{if } t_C = \mathbf{remove_C}\langle x, \langle e, r, k\rangle\rangle. \end{cases}$$

As before, the translation preserves control, return and **halt** states between the two machines. Operations to store, read, or get bindings from the shared environment are translated into `put`, `rd`, or `get` expressions. Because the CEK_C machine performs operations on shared environments locally (by migrating continuations), the translated expression takes as its address space argument, the address space recorded by the respective frame.

To show that CEK_C preserves meaning with respect to the reference semantics, we first show that every transition undertaken by $\longrightarrow_B$ corresponds to a sequence of transitions under $\longrightarrow_C$ with the resulting states related via F_B^C.



We formalize this diagram in the following lemma:

Lemma 4. *If* $F_B^M(\langle T_{CM}, \rho\rangle) = \langle \overline{T}, \overline{\rho}\rangle$ *and* $\langle \overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle \overline{T}', \overline{\rho}'\rangle$ *then there exists* $\langle T'_{CM}, \rho'\rangle$ *such that* $\langle T_{CM}, \rho\rangle \xrightarrow{*}_C \langle T'_{CM}, \rho'\rangle$ *and* $F_B^M(\langle T'_{CM}, \rho'\rangle) = \langle \overline{T}', \overline{\rho}'\rangle$.

Proof.

By construction on the structure of CEK_C:

Case 1. Let $T_{CM}(\langle \alpha, \tau\rangle) = \langle e, r, k\rangle$. The construction follows case (1) in the proof of lemma 1.

Case 2. Let $T_{CM}(\langle \alpha, \tau\rangle) = \langle(\texttt{put}\ AS\ [x\ y]\ e), r, k\rangle$ where $r(AS) = \alpha'$ and $r(y) = v$. By definition of F_B^C and $\longrightarrow_C$, $\langle \overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle \overline{T}', \overline{\rho}'\rangle$ where $\overline{T}' = \overline{T}[\tau \mapsto \langle e, r, k\rangle]$ and $\overline{\rho}' = \overline{\rho}[\alpha' \mapsto \overline{\rho}(\alpha)[x \mapsto v]]$. Choose transition $\langle T_{CM}, \rho\rangle \xrightarrow{\langle \alpha, \tau\rangle}_C \langle T_{CM1}, \rho\rangle$. By definition of $\longrightarrow_C$, $\langle \alpha, \tau\rangle \notin Dom(T_{CM1})$ and $T_{CM}(\langle \alpha', \tau\rangle) = \mathbf{store_C}\langle x, v, \langle e, r, k\rangle\rangle$. Now, choose transition $\langle T_{CM1}, \rho\rangle \xrightarrow{\langle \alpha', \tau\rangle}_C \langle T'_{CM}, \rho'\rangle$. By definition of $\longrightarrow_C$, $T'_{CM}(\langle \alpha', \tau\rangle) = \langle e, r[x \mapsto v], k\rangle$ and $\rho'(\alpha')(x) = v$. Since $F_B^C(\langle T_{CM}, \rho\rangle) = \langle \overline{T}, \overline{\rho}\rangle$, $F_B^C(\langle T'_{CM}, \rho'\rangle) = \langle \overline{T}', \overline{\rho}'\rangle$.

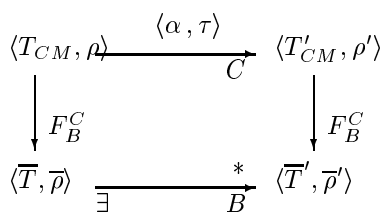
Case 3. Let $T_{CM}(\langle \alpha, \tau\rangle) = \langle(\texttt{rd}\ AS\ [x]\ e), r, k\rangle$ where $r(AS) = \alpha'$. By definition of F_B^C and $\longrightarrow_B$, $\langle \overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle \overline{T}' = \overline{T}[\tau \mapsto \langle e, r, k\rangle], \overline{\rho}\rangle$ assuming $\overline{\rho}(\alpha')(x) = v$. (Suppose this were not true. Then, no transition from $\langle \overline{T}, \overline{\rho}\rangle$ via τ is allowed and the lemma assumption would fail.) Choose transition $\langle T_{CM}, \rho\rangle \xrightarrow{\langle \alpha, \tau\rangle}_C \langle T_{CM1}, \rho\rangle$. By definition of $\longrightarrow_C$, $\langle \alpha, \tau\rangle \notin Dom(T_{CM1})$ and $T_{CM1}(\langle \alpha', \tau\rangle) = \mathbf{read_C}\langle x, \langle e, r, k\rangle\rangle$. Now, choose transition $\langle T_{CM1}, \rho\rangle \xrightarrow{\langle \alpha', \tau\rangle}_C \langle T'_{CM}, \rho'\rangle$. We can choose this transition because $\overline{\rho}(\alpha')(x) = v$. By definition of $\longrightarrow_C$ and F_B^C, $T'_{CM}(\langle \alpha', \tau\rangle) = \langle e, r[x \mapsto v], k\rangle$, $F_B^C(\langle T'_{CM}, \rho\rangle) = \langle \overline{T}', \overline{\rho}\rangle$.

Case 4. Let $T_{CM}(\langle \alpha, \tau\rangle) = \langle(\texttt{get}\ AS\ [x]e), r, k\rangle$ where $r(AS) = \alpha'$. Pick construction similar to (3).

Case 5. Let $T_{CM}(\langle \alpha, \tau\rangle) = \langle(\texttt{spawn}\ AS\ e), r, k\rangle$. Follow construction in proof of Lemma 1.

Case 6. Let $T_{CM}(\langle \alpha, \tau\rangle) = \langle(\texttt{makeAS}), r, k\rangle$. Follow construction in proof of Lemma 1. □

As before, we now show the correspondence in the other direction, *i.e.,* every transition taken by a CEK_C machine corresponds to zero or one transitions in a CEK_B machine:



We formalize this picture in the following lemma:

Lemma 5. *If* $F_B^C(\langle T_{CM}, \rho\rangle) = \langle \overline{T}, \overline{\rho}\rangle$ *and* $\langle T_{CM}, \rho\rangle \xrightarrow{\langle \alpha, \tau\rangle}_C \langle T'_{CM}, \rho'\rangle$ *then there exists a transition (possibly empty) such that* $\langle \overline{T}, \overline{\rho}\rangle \longrightarrow_B \langle \overline{T}', \overline{\rho}'\rangle$ *and* $F_B^C(\langle T'_{CM}, \rho'\rangle) = \langle \overline{T}', \overline{\rho}'\rangle$.

Proof.

By cases on $\langle T_{CM}, \rho\rangle \xrightarrow{\langle \alpha, \tau\rangle}_B \langle T'_{CM}, \rho'\rangle$. We need only consider cases dealing with transitions unique to CEK_C;

Case 1. Let $T_{CM}(\langle \alpha, \tau\rangle) = \mathbf{store_C}\langle x, v, \langle e, r, k\rangle\rangle$. By definition of $\longrightarrow_C, T'_{CM}(\langle \alpha, \tau\rangle) = \langle e, r[x \mapsto v], k\rangle$. By definition of F_B^C, $F_B^C(T_{CM})(\tau) = \langle(\texttt{put}\ \alpha\ [x\ v]\ ,)r, k\rangle$. Choose transition $\langle \overline{T}, \overline{\rho}\rangle \xrightarrow{\tau}_B \langle \overline{T}', \overline{\rho}'\rangle$. By definition of $\longrightarrow_C$ and F_B^C, $F_B^C(\overline{T}') = T'_{CM}$ and $\rho' = \overline{\rho}'$.

Case 2. Let $T_{CM}(\langle \alpha, \tau\rangle) = \mathbf{read_C}\langle x, v\rangle\langle e, r, k\rangle$. Follow construction used in (1).

Case 3. Let $T_{CM}(\langle \alpha, \tau\rangle) = \mathbf{remove_C}\langle x, v\rangle\langle e, r, k\rangle$. Follow construction used in (1).

□

Theorem 6. $eval_C = eval_B$.

Proof. By Lemma 4 and Lemma 5 and induction on the length of transition sequences induced by $\longrightarrow_C$ and $\longrightarrow_B$. □

Theorem 7. $eval_C = eval_M$

Proof. Follows trivially from Theorems 3 and 6. □

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. *Computation migration* [HWW93] refers to an implementation technique that lazily transmits portions of a thread's state when the thread is involved in a migration event. The technique reduces bandwidth requirements while potentially increasing the number of remote communication events. 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.

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" [Fel88], a mechanism to delimit the extent of continuation-capturing operations.

For example consider the expression:

```
(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 frame to hold the result of the second argument to `cons` since the call to `rd` is in a non-tail position. 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. Note that there is no series of intermediate returns executed: the base of the migrating continuation dictates that control return back to the address space initiating the `map` operation. This information is recorded as a result of the freeze ("!") annotation. In other words, for a list of length $O(n)$, only n (*not* $2n$) communication events are triggered.

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.

$$CEK_{CM} = \langle State_{CM}, \longrightarrow_{CM} \rangle$$

$$\begin{aligned}
s_{CM} \in State_{CM} &= \langle ThreadMap_{CM}, SharedEnv \rangle \\
T_{CM} \in ThreadMap_{CM} &= AspaceUid \times ThreadUid \rightarrow ThreadState_{CM} \\
t_{CM} \in ThreadState_{CM} &= ControlState + ReturnState + + \\
&\quad MigrateState + SuspendState + HaltState \\
\mathbf{suspend}\langle\, k \,\rangle \in SuspendState &= Cont \\
frame \in Frame_{CM} &= FunFrame + ReturnFrame + ArgFrame + \\
&\quad IfFrame + CMFrame \\
frame_{CM} \in CMFrame &= FreezeFrame + MigrateFrame \\
\mathbf{freeze}\langle\,\rangle \in FreezeFrame & \\
\mathbf{migrate}\langle\, \alpha, \tau \,\rangle \in MigrateFrame &= AspaceUid \times ThreadUid
\end{aligned}$$

Fig. 13. Domain definitions for a refinement of a CmPS machine that supports computation migration.

We define a new machine CEK_{CM} to express the behavior of computation migration in Figure 13. Only definitions different from the CEK_C specification are shown. Figure 14 shows the modifications to the state machine necessary to support computation migration. We add three new continuation frames: a **freeze**$\langle\,\rangle$ frame marks a continuation point below which continuation frames will not be migrated; the frame **suspend**$\langle\, k \,\rangle$ marks that portion of a continuation which is suspended pending a return of control-flow from a migrated sub-continuation; a frame **migrate**$\langle\, \alpha, \tau \,\rangle$ records the address space and thread containing a suspended continuation.

To evaluate the body e of a coordination expression, a new thread is created on the target address space. This thread begins evaluation of e 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*, $\mathbf{migrate}\langle\ \alpha\,,\tau\,\rangle$, 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 $\mathbf{freeze}\langle\,\rangle$, it simply pops the frame and proceeds with the continuation below it. If control reaches this point, it implies that the continuation of a $\mathbf{freeze}\langle\,\rangle$ expression did not initiate any coordination operations.

$$\langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle(!\ \ e), r, k\rangle], \rho\rangle \longrightarrow_{CM}$$
$$\quad \langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle e, r, \mathbf{freeze}\langle\,\rangle : k\rangle], \rho\rangle$$

$$\langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle\mathbf{freeze}\langle\,\rangle : k, v\rangle], \rho\rangle \longrightarrow_{CM}$$
$$\quad \langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle k, v\rangle], \rho\rangle$$

$$\langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle(\mathtt{put}\ AS\,[x\ y]\ e), r, k_u : \mathbf{freeze}\langle\,\rangle : k_d\rangle], \rho\rangle \longrightarrow_{CM}$$
$$\quad \langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \mathbf{suspend}\langle\ k_d\,\rangle,$$
$$\qquad \langle r\,(AS), \tau\rangle \mapsto \mathbf{store_C}\langle x, r\,(y), \langle e, r, k_u : \mathbf{migrate}\langle\ \alpha\,,\tau\,\rangle\rangle\rangle], \rho\rangle$$
$$\quad \text{provided } \mathbf{freeze}\langle\,\rangle \notin k_u.$$

$$\langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle(\mathtt{rd}\ AS\,[x]\ e), r, k_u : \mathbf{freeze}\langle\,\rangle : k_d\rangle], \rho\rangle \longrightarrow_{CM}$$
$$\quad \langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \mathbf{suspend}\langle\ k_d\,\rangle,$$
$$\qquad \langle r\,(AS), \tau\rangle \mapsto \mathbf{read_C}\langle x, \langle e, r, k_u : \mathbf{migrate}\langle\ \alpha\,,\tau\,\rangle\rangle\rangle], \rho\rangle$$
$$\quad \text{provided } \mathbf{freeze}\langle\,\rangle \notin k_u.$$

$$\langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle(\mathtt{get}\ AS\,[x]\ e), r, k_u : \mathbf{freeze}\langle\,\rangle : k_d\rangle], \rho\rangle \longrightarrow_{CM}$$
$$\quad \langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \mathbf{suspend}\langle\ k_d\,\rangle,$$
$$\qquad \langle r\,(AS), \tau\rangle \mapsto \mathbf{remove_C}\langle x, \langle e, r, k_u : \mathbf{migrate}\langle\ \alpha\,,\tau\,\rangle\rangle\rangle], \rho\rangle$$

$$\langle T_{CM}[\langle\alpha\,,\tau\rangle \mapsto \langle\mathbf{migrate}\langle\ \alpha\,',\tau\,\rangle, v\rangle,\ \langle\alpha\,',\tau\rangle \mapsto \mathbf{suspend}\langle\ k\,\rangle], \rho\rangle \longrightarrow_{CM}$$
$$\quad \langle T_{CM}/\langle\alpha\,,\tau\rangle[\langle\alpha\,',\tau\rangle \mapsto \langle k, v\rangle], \rho\rangle$$

Fig. 14. Transition rules to support computation migration.

The behavior of a CEK_{CM} machine is similar to its CEK_C counterpart except for the continuation frames it manipulates. Since *Migrate* and *Suspend* frames serve only to partition the locus of computation among address spaces, a CEK_{CM} machine preserves program meaning with respect to the other definitions presented earlier. As before, we can define a translation from a CEK_{CM} machine to our reference semantics, and prove equivalence of the two systems by cases on transitions they undertake. The translation function $F_B^{CM} : \mathit{State}_{CM} \rightarrow \overline{\mathit{State}}$ erases address spaces, states used to define the behavior of communication-passing style, and additionally frames used to split and join continuations involved in a computation migration. In the following, let $t_{CM} = T_{CM}\,(\langle\alpha\,,\tau\rangle)$. Portions of the translation elided in the definition are the same as in the definition of F_B^C.

$$F_B^{CM}\left(\langle T_{CM}, \rho\rangle\right) = \langle F_B^{CM}\left(T_{CM}\right), \rho\rangle$$

$$F_B^{CM}\left(T_{CM}\right)(\tau) = \begin{cases} \langle \mathbf{freeze}\langle\rangle : k, v\rangle & \langle k, v\rangle \\ \langle e, r, k_u : \mathbf{freeze}\langle\rangle : k_d\rangle & \langle e, r, k_u : k_d\rangle \\ \langle k, v\rangle & \text{if } t_{CM} = \mathbf{suspend}\langle\ k\ \rangle \text{ and } T_{CM}\left(\langle \alpha', \tau\rangle\right) = \\ & \langle \mathbf{migrate}\langle\ \alpha\ , \tau\ \rangle, v\rangle \\ \langle e, r, k_u : k\rangle & \text{if } t_{CM} = \mathbf{suspend}\langle\ k\ \rangle \text{ and } T_{CM}\left(\langle \alpha', \tau\rangle\right) = \\ & \langle e, r, k_u : \mathbf{migrate}\langle\ \alpha\ , \tau\ \rangle\rangle \\ \langle(\texttt{put}\ \alpha'\ [x\ v]\ e), r, k_u : k\rangle & \text{if } t_{CM} = \mathbf{suspend}\langle\ k\ \rangle \text{ and } T_{CM}\left(\langle \alpha', \tau\rangle\right) = \\ & \mathbf{store_C}\langle x, v, \langle e, r, k_u : \mathbf{migrate}\langle\ \alpha\ , \tau\ \rangle\rangle\rangle \\ \langle(\texttt{rd}\ \alpha'\ [x]\ e), r, k\rangle & \text{if } t_{CM} = \mathbf{suspend}\langle\ k\ \rangle \text{ and } T_{CM}\left(\langle \alpha', \tau\rangle\right) = \\ & \mathbf{read_C}\langle x, \langle e, r, k_u : \mathbf{migrate}\langle\ \alpha\ , \tau\ \rangle\rangle\rangle \\ \langle(\texttt{get}\ \alpha\ [x]\ e), r, k\rangle & \text{if } t_{CM} = \mathbf{suspend}\langle\ k\ \rangle \text{ and } T_{CM}\left(\langle \alpha', \tau\rangle\right) = \\ & \mathbf{remove_C}\langle x, \langle e, r, k_u : \mathbf{migrate}\langle\ \alpha\ , \tau\ \rangle\rangle\rangle \end{cases}$$

The translation erases **freeze** frames from continuations, and removes **suspend** states. Since such a state is always paired with a **migrate** frame associated with a thread having the same identifier, but located on a different address space, the translation combines the continuation captured by a suspend frame with the continuation in which the corresponding **migrate**-frame is recorded. For $\mathbf{store_C}$, $\mathbf{read_C}$, or $\mathbf{remove_C}$ operations, the translation produces `put`, `rd`, or `get` expressions whose address space argument is the address space on which the operation is to be performed. In any case, **migrate**-frames embedded in the control-state captured by these operations are also removed.

To show equivalence between the CEK_{CM} and CEK_B machines, we define a correspondence between programs with freeze annotations executed under CEK_{CM}, and the same program with these annotations removed executed under CEK_B. For program P, define $\overline{P}$ to be the same as P with all freeze annotations removed. Then, we state without proof the following theorem that relates the equivalence of a CEK_{CM} machine with a CEK_B machine:

Theorem 8. $eval_{CM}(P) = eval_B(\overline{P})$.

The proof of this theorem is similar in construction to the proof of theorem 6 and relies on defining a correspondence between states in the respective machines using the definition of F_B^{CM}.

6 Related Work

Most closely related to the coordination language developed here is Linda [CG89]. Piranha [GK92] 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 [RCRH95] 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 sub-languages, Olden specifies parallelism using continuation-capturing operations based on futures [Hal85] and lazy task creation [MKH90].

There has been much work in supporting efficient capture and restoration of continuations in language implementations such as Standard ML of New Jersey [AM87, App92] or Chez Scheme [HDB90, BWD96]. By using techniques found in these implementations, we believe the overhead of moving continuations in a distributed environment can also be made tractable. Indeed, experimental results from early work on computation migration [HWW93] provides evidence that the cost of lazy migration of continuation frames can be efficient in practice. We intend to validate these conjectures in a realistic implementation and believe the careful use of "freeze" annotations imposes a small burden on the programmer, but can greatly reduce the overhead incurred in migrating continuation frames.

Flanagan and Felleisen [FF95] and Moreau [Mor96b] present a CEK-based operational semantics for parallel languages supporting the *future* construct [Hal85]. Unlike the semantics described here, neither of the above papers consider implementation issues or language extensions to handle distributed computation, although Moreau does report on enhancing the semantics to support distribution and task migration in a separate report [Mor96a]. While the semantics is defined in a style similar to ours, Moreau's presentation does not consider programming issues related to coordination, communication-passing, or computation migration. In addition, correctness proofs in his system are defined with respect to a sequential semantics since *future* is considered as a semantics-preserving annotation whereas we define correctness with respect to a concurrent interpreter. Jagannathan and Weeks [JW94] also 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 [PM83] and computation migration [HWW93] 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 [Car95] and Kali [CJK95] 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. Queinnec [Que92] describes a distributed dialect of Scheme that supports task migration. Like continuation-passing style, migration in his system entails shifting the entire computation to the target machine. Unlike our approach, however, migration is not tied to data structure access, nor are primitives provided to distribute partial continuations, although we believe that his system can be extended to handle these issues. Nonetheless, none of these efforts have been conducted in the context of a coordination

language like Linda, nor have they explored the tradeoffs between data communication and task migration formally.

There has been much recent interest in developing expressive formal models of concurrency suitable to describe coordination languages which operate in a distributed environment [GM97]. More generally, systems such as the π-calculus [Mil89] are capable of describing the interaction of *agents* whose configuration and interaction may change dynamically. In the context of this model, researchers have developed encodings of other formal systems such as the λ-calculus [Mil93]. In particular, [Bou97, Mil92] have shown how continuation-based transformations in the λ-calculus correspond to transformations related to communication and naming events in the π-calculus. Indeed, "higher-order" extensions [Tho90] of the calculus can be used to model migration of processes among channels, thus presenting another means of understanding computation migration. Broadly related to these efforts are attempts to formalize systems of mobile agents [SY97, FG96] via a calculus that encodes migration, copying, and synchronization semantics. Cardelli and Gordon [CG98] describe a calculus for mobility that includes a notion of a protection domain called an *ambient*. Ambients are abstractions that serve to model access rights on agents which may migrate in a wide-area distributed network like the World-Wide Web. Insofar as ambients serve to model distributed computation, they are related to the notion of an address space as defined in our coordination language. However, the intended of role of ambients as an abstraction protected by capabilities is significantly different from the intended role of address spaces as structures encapsulating shared data and threads.

In summary, our primary interest is defining formal implementations of coordination languages that use various forms of thread migration to reduce communication overhead in a realistic distributed environment. Thus, our results can be viewed as providing an intersection between theoretical work on the expressive power of definitional interpreters for asynchronous concurrent systems and more practical work on implementations of high-level coordination languages.

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 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 sub-language 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

[ACPT95] Thomas Anderson, David Culler, David Patterson, and The Now Team. A Case For NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[AM87] Andrew W. Appel and David B. Macqueen. A Standard ML compiler. In *Conference on Functional Programming and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 301–324. Springer–Verlag, 1987.

[App92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Bou97] Gérard Boudol. The π-Calculus in Direct-Style. In 24^{th} *ACM Symposium on Principles of Programming Languages*, pages 228–241, 1997.

[BWD96] Carl Bruggeman, Oscar Waddell, and Kent Dybvig. Representing Control in the Presence of One-Shot Continuations. In *ACM Conference on Programming Language Design and Implementation*, June 1996.

[Car95] Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the 22^{nd} ACM Symposium on Principles of Programming Languages*, pages 286–298, New York, 1995. ACM.

[CG89] Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[CG98] Luca Cardelli and Andrew Gordon. Mobile Ambients. In *Foundations of Software Science and Computational Structures*, number 1378 in LNCS, pages 140–155. Springer-Verlag, 1998.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.

[Fel88] Matthias Felleisen. The Theory and Practice of First-Class Prompts. In 15^{th} *ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[FF86] Matthias Felleisen and Daniel Friedman. Control Operators, the SECD-machine, and the Lambda-calculus. In 3^{rd} *Working Conference on the Formal Description of Programming Concepts*, pages 193–219. North-Holland, 1986.

[FF87] Matthias Felleisen and Daniel Friedman. A Calculus for Assignments in Higher-Order Languages. In *Proceedings of the 14^{th} ACM Symposium on Principles of Programming Languages*, pages 314–325, 1987.

[FF95] Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the 12^{nd} ACM Symposium on Principles of Programming Languages*, pages 290–220, 1995.

[FG96] C. Fournet and G. Gonthier. A Calculus of Mobile Agents. In *CONCUR'96: Concurrency Theory*, number 1119 in LNCS, pages 406–421. Springer-Verlag, 1996.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *ACM Symposium on Programming Language Design and Implementation*, pages 237–247, 1993.

[Gel89] David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of Conference on Parallel Architectures and Languages, Europe*, number 365 in LNCS, pages 20–27. Springer-Verlag, 1989.

[GK92] David Gelernter and David Kaminsky. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. In *Proceedings of the 6^{th} ACM International Conference on Supercomputing*, pages 417–427, July 1992.

[GM97] David Garlan and Daniel Le Métayer, editors. *Coordination Languages and Models*, number 1282 in LNCS. Springer-Verlag, 1997.

[Hal85] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HDB90] Robert Hieb, Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. In *ACM Conference on Programming Language Design and Implementation*, June 1990.

[HWW93] Wilson Hsieh, Paul Wang, and William Weihl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. In *The 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 239–249, New York, May 1993. ACM.

[JW94] Suresh Jagannathan and Stephen Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proceedings of the 1994 ACM International Conference on Lisp and Functional Programming*, pages 294–305, 1994.

[KH89] Richard Kelsey and Paul Hudak. Realistic Compilation by Program Transformation. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mil92] Robin Milner. Functions as Processes. *Journal of Mathematical Structures in Computer Science*, pages 119–141, 1992.

[Mil93] Robin Milner. The Polyadic π-Calculus: A Tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[MKH90] Rick Mohr, David Kranz, and Robert 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.

[Mor96a] Luc Moreau. Correctness of a Distributed-Memory Model for Scheme. In *Second International Europar Conference (EURO-PAR'96)*, number 1123 in LNCS, pages 615–624. Springer-Verlag, 1996.

[Mor96b] Luc Moreau. The Semantics of Scheme with Future. In *Proceedings of the 1996 ACM International Conference on Functional Programming*, pages 146–156, 1996.

[PM83] M. Powell and B. Miller. Process Migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 110–119, New York, 1983. ACM.

[Que92] Christian Queinnec. A Concurrent and Distributed Extension of Scheme. In D. Etiemble and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE'92)*, number 605 in LNCS, pages 431–446. Springer-Verlag, June 1992.

[RCRH95] Anne Rogers, Martin Carlisle, John Reppy, and Laurie Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, 1995.

[SA94] Zhong Shao and Andrew Appel. Space-Efficient Closure Representations. In *Proceedings of the 1994 ACM Symposium on Lisp and Functional Programming*, pages 150–161, 1994.

[SF94] Amr Sabry and Matthias Felleisen. Is Continuation Passing Useful for Data Flow Analysis? In *ACM Symposium on Programming Language Design and Implementation*, pages 1–12, 1994.

[SY97] Tatsoru Sekiguchi and Akinori Yonezawa. A Calculus with Code Mobility. In *Formal Methods for Open Object-Oriented Distributed Systems.* Chapman-Hall, 1997.

[Tho90] Bent Thomsen. *Calculi For Higher-Order Communicating Processes.* PhD thesis, Imperial College, London University, 1990.

Acknowledgements

Thanks to the anonymous referees, Henry Cejtin, Richard Kelsey, and Stephen Weeks for many useful discussions and comments.

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