Type-safe Higher-order Channels in ML-like Languages *

Sungwoo Park

Pohang University of Science and Technology
Republic of Korea
gla@postech.ac.kr

Abstract

As a means of transmitting not only data but also code encapsulated within functions, higher-order channels provide an advanced form of task parallelism in parallel computations. In the presence of mutable references, however, they pose a safety problem because references may be transmitted to remote threads where they are no longer valid.

This paper presents an ML-like parallel language with *type-safe* higher-order channels. By type safety, we mean that no value written to a channel contains references, or equivalently, that no reference escapes via a channel from the thread where it is created. The type system uses a typing judgment that is capable of deciding whether the value to which a term evaluates contains references or not. The use of such a typing judgment also makes it easy to achieve another desirable feature of channels, *channel locality*, that associates every channel with a unique thread for serving all values addressed to it.

Our type system permits mutable references in sequential computations and also ensures that mutable references never interfere with parallel computations. Thus it provides both flexibility in sequential programming and ease of implementing parallel computations.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics, Syntax

General Terms Languages

Keywords Higher-order channels, Channel locality, Parallel languages, Distributed languages

1. Introduction

The advent of multicore processors and the impending demise of free lunch (Sutter 2005) have changed the conventional wisdom on computer hardware (Asanovic et al. 2006). There is little room for increasing clock frequency while increasing hardware parallelism is now the only viable way of improving processor performance. Such a radical change in the trend of computer hardware has revitalized research on language support for parallel programming, as evidenced by recent parallel languages such as X10 (Charles et al. 2005), Fortress (Allan et al. 2007), Chapel (Cray Inc. 2005), Data Parallel Haskell (Chakravarty et al. 2007), and Manticore (Fluet et al. 2007).

Depending on the granularity of parallel computations, parallel programming models are divided into *data parallelism* or *task parallelism*. Data parallelism applies an independent operation to each element of a collection of homogeneous data and often results in massive parallel computations. A simple case of data parallelism is *flat data parallelism* in which the independent operation is sequential. A more sophisticated model called *nested data parallelism* (Blelloch 1996) allows the independent operation itself to be a parallel computation. Task parallelism executes cooperative threads in parallel which communicate via shared memory or channels in order to perform synchronization. The use of shared memory simplifies programming tasks, especially if a high-level abstraction such as software transactional memory (Shavit and Touitou 1995) is provided, but it entails the memory consistency problem. The use of channels empirically requires more effort than programming with shared memory (Hochstein et al. 2005), but it eliminates the memory consistency problem.

This paper is primarily concerned with an extension of an ML-like language, *i.e.*, a call-by-value functional language with mutable references, that offers task parallelism with *higher-order channels* (but without shared memory). Higher-order channels transmit not only data, such as integers and channel names, but also pieces of code encapsulated within functions. The capability to transmit code between threads opens a new range of communication constructs such as futures, remote evaluation, code on demand, and hot code replacement.

Because of the similarity between task parallelism in parallel computations and process parallelism in distributed computations, our target language can also be thought of as a distributed language in which processes share no global memory and communicate only via higher-order channels. Thus, although our work primarily aims at designing higher-order channels for parallel languages, it can be equally applied to distributed languages.

1.1 Type-safe higher-order channels

The main problem with higher-order channels is that in the presence of mutable references, code containing references may travel between threads, but in the absence of shared memory, a reference created by a thread cannot be dereferenced at another thread. Previous parallel or distributed languages with a similar programming model avoid this problem by dispensing with mutable references altogether, as in the parallel language Manticore (Fluet et al. 2007), or by designing the runtime system so as to create copies of heap cells whenever their references are transmitted, as in the distributed languages Facile (Knabe 1995) and JoCAML (Fournet et al. 2003).

This paper takes a different approach by developing *type-safe* higher-order channels. The type system ensures that no reference

* This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No. R01-2007-000-11087-0) and POSTECH BSRI research fund-2007.

escapes from the thread where it is created so that all values written to channels remain valid even after being transmitted to remote threads. To be specific, suppose that a thread is executing a channel write $a\,!\,M$ where a is a channel for a certain type A and M is a term of the same type A. Evaluating M yields a value V which is then transmitted to another thread executing a channel read $a\,?$. Type safety of channels means that V contains no references and that V is valid at both threads regardless of its type A.

Conventional type systems, however, are inadequate to guarantee type safety of channels. The crux of the problem is that a typical typing judgment $M : A$ does not indicate whether the result of evaluating M contains references or not. For example, a term

$$(\lambda v\!:\!\mathsf{int}.\,\lambda x\!:\!\mathsf{int}.\,x + v)\ 0$$

of type $\mathsf{int} \rightarrow \mathsf{int}$ evaluates to a value containing no reference, but another term

$$(\lambda r\!:\!\mathsf{ref\ int}.\,\lambda x\!:\!\mathsf{int}.\,x + !r)\ (\mathsf{ref}\ 0)$$

of the same type evaluates to a value containing a reference. In order to guarantee type safety of channels, therefore, we need at least two kinds of typing judgments: an ordinary typing judgment asserting that a given term evaluates to a value that may contain references, or a *local value*, and another stronger typing judgment asserting that a given term evaluates to a value containing no references, or a *global value*.

Our work is based on the type system in our previous work (Park 2006) which uses two typing judgments to distinguish between local values and global values in the context of distributed computations. We simplify the type system by combining the two typing judgments into a single typing judgment $M : A_{@L}$ where L is a *locality* indicating whether the value to which M evaluates is local ($L = \mathsf{L}$) or global ($L = \mathsf{G}$). The connection between two localities G and L is established by a new construct $\mathsf{box}\ M$ and a modal type $\boxdot A$ such that $M : A_{@\mathsf{G}}$ implies $\mathsf{box}\ M : \boxdot A_{@\mathsf{L}}$ (*i.e.*, if M evaluates to a global value of type A, then $\mathsf{box}\ M$ has type $\boxdot A$). Now a channel write $a\,!\,M$ typechecks only if $M : A_{@\mathsf{G}}$ holds. As a result, a channel read $a\,?$ always returns a global value and thus $a\,? : A_{@\mathsf{G}}$ automatically holds.

1.2 Channel locality

The type system developed for higher-order channels makes it easy to achieve an important feature of task parallelism, *channel locality*, which states that every channel is associated with a unique thread for reading off all values sent to it. Channel locality obviates the need for a sophisticated mechanism in the runtime system for dynamically determining the destination of each value written to a channel.

Most of the previous approaches to enforcing channel locality (Fournet et al. 1996; Amadio 1997; Yoshida and Hennessy 1999; Schmitt and Stefani 2003) have been developed for calculi for concurrent processes based on the pi-calculus. We find that these approaches are difficult to apply to our setting which uses as a base language the simply-typed lambda calculus with mutable references instead of the pi-calculus or its variant.

The main idea for achieving channel locality in our work is to split channels into two kinds, *read channels* and *write channels,* and treat read channels as local values but write channels as global values. $\mathsf{new}_{\langle A \rangle}$, a construct for creating channels for type A, evaluates to a pair of read channel a^r and write channel a^w such that a^r accepts only those values written to a^w. Since a^r is a local value, channel reads from a^r are allowed only at the thread where a^r is created. Channel writes to a^w are, however, allowed at any thread because a^w is a global value. Thus a pair of a^r and a^w can open unidirectional communications from any thread to the thread to which a^r belongs.

It is easy to classify read channels as local values and write channels as global values. First we assign different types, namely *read channel types* and *write channel types,* to read channels and write channels, respectively. Then we treat read channel types like reference types so that read channels cannot be part of a global value, but define write channel types as primitive types whose values are all inherently global (like integers). For this reason, our decision to distinguish between read channels and write channels has a different motivation from previous work in which read channels and write channels are also distinguished, but channel locality is not enforced (Odersky 1995) or enforced only syntactically (Zhang and Potter 2002).

1.3 Contributions

We develop an ML-like parallel language $\lambda_{\boxdot}^{\mathsf{PC}}$ which features type-safe higher-order channels with channel locality. Its type system inherits from the type system of (Park 2006) the ability to distinguish between local values and global values. Its operational semantics models parallel computations where multiple threads communicate via higher-order channels. Channel locality is a direct consequence of assigning different types to read channels and write channels. To the best of our knowledge, type-safe higher-order channels in the presence of mutable references have not been investigated.

Although $\lambda_{\boxdot}^{\mathsf{PC}}$ deals primarily with type safety for task parallelism, providing type safety for data parallelism is also straightforward by virtue of its ability to distinguish between local values and global values. As an example, consider a parallel map construct mapP which applies a function f to each element of an array in parallel. By requiring that f be a global value so that child threads share no references, we can prevent mapP from running into the memory consistency problem. In this regard, $\lambda_{\boxdot}^{\mathsf{PC}}$ serves as a unified framework for achieving type safety for both data parallelism and task parallelism when mutable references are allowed.

Removing mutable references simplifies the implementation of a parallel language because lack of mutable references implies automatic data separation in parallel computations. For example, Manticore (Fluet et al. 2007) excludes mutable references in its base language (a subset of Standard ML) in order to simplify its implementation. Parallel dialects of Haskell, such as pH (Nikhil and Arvind 2001) and Data Parallel Haskell (Chakravarty et al. 2007), also benefit from lack of mutable references. In comparison, $\lambda_{\boxdot}^{\mathsf{PC}}$ permits mutable references in sequential computations, but its type system ensures that mutable references never interfere with parallel computations. Thus the type system of $\lambda_{\boxdot}^{\mathsf{PC}}$ wins us both flexibility in sequential programming and ease of implementing parallel computations.

1.4 Organization of the paper

Section 2 presents the base language $\lambda_{\boxdot}$ for our work. Although the type system differentiates global values from local values, $\lambda_{\boxdot}$ is just a sequential language to which global values are of no use. Hence we develop a parallel operational semantics for modeling parallel computations. Section 3 presents the resultant language $\lambda_{\boxdot}^{\mathsf{P}}$ which comes with a new construct $\mathsf{spawn}\ M$ for creating new threads. Section 4 extends $\lambda_{\boxdot}^{\mathsf{P}}$ with new constructs for higher-order channels to obtain $\lambda_{\boxdot}^{\mathsf{PC}}$. Section 5 illustrates how to implement various communication constructs in $\lambda_{\boxdot}^{\mathsf{PC}}$ such as futures, bidirectional communications, shared references, remote evaluation, code on demand, and hot code replacement. Section 6 discusses two extensions of $\lambda_{\boxdot}^{\mathsf{PC}}$ (including type safety for data parallelism). Section 7 discusses related work and Section 8 concludes.

type	A, B, C	$::=$	$\mathsf{unit} \mid A \to A \mid A \times A \mid A{+}A \mid \mathsf{ref}\ A \mid\ \boxdot\, A$
primitive type	P		
term	M, N	$::=$	$x \mid \lambda x\!:\!A.\,M \mid M\ M \mid$
			$(M, M) \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid$
			$\mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow M \mid \mathsf{inr}\ x \Rightarrow M \mid$
			$() \mid \mathsf{fix}\, x\!:\!A.\,M \mid$
			$\mathsf{ref}\ M \mid\ !M \mid M := M \mid l \mid$
			$\mathsf{box}\ M \mid \mathsf{letbox}\ x = M\ \mathsf{in}\ M$
value	V	$::=$	$() \mid \lambda x\!:\!A.\,M \mid (V, V) \mid \mathsf{inl}\ V \mid \mathsf{inr}\ V \mid l \mid \mathsf{box}\ M$
locality	L	$::=$	$\mathsf{G} \mid \mathsf{L}$
typing context	Γ	$::=$	$\cdot \mid \Gamma, x : A_{@L}$
store	ψ	$::=$	$\cdot \mid \psi, l \mapsto V$
store typing	Ψ	$::=$	$\cdot \mid \Psi, l \mapsto A$

Figure 1. Abstract syntax for $\lambda_{\boxdot}$

2. Base language $\lambda_{\boxdot}$

This section presents the base language $\lambda_{\boxdot}$ which is a reformulation of the call-by-value language with modal types $\boxdot A$ in our previous work (Park 2006). In the previous work, we use two separate typing judgments to distinguish between local values (which may contain references) and global values (which contains no references): an ordinary typing judgment $M : A$ to mean that M evaluates to a local value, and a stronger typing judgment $M \sim A$ to mean that M evaluates to global value. We combine the two typing judgments into a single typing judgment $M : A_{@L}$ where a locality L, either L or G, indicates whether M evaluates to a local value or a global value. The simplification of typing judgments is essential to maintaining the complexity of the type system at a manageable level. For example, the introduction of sum types requires eight new typing rules in the type system of (Park 2006) whereas the base language $\lambda_{\boxdot}$ here uses only three typing rules.

2.1 Definition of $\lambda_{\boxdot}$

Figure 1 shows the abstract syntax for $\lambda_{\boxdot}$ which is based on the simply-typed λ-calculus with product types $A \times A$, sum types $A{+}A$, reference types $\mathsf{ref}\ A$, and the fixed point construct $\mathsf{fix}\, x\!:\!A.\,M$. (We will define the set of primitive types later.) $\mathsf{ref}\ M$ allocates a fresh reference, $!M$ dereferences an existing reference, and $M := N$ assigns a new value to a reference. A location l, of type $\mathsf{ref}\ A$, is a value for a reference. A new construct $\mathsf{box}\ M$ has a modal type $\boxdot A$, and another new construct $\mathsf{letbox}\ x = M\ \mathsf{in}\ N$ expects M to be of type $\boxdot A$.

We use a typing judgment $\Gamma \mid \Psi \vdash M : A_{@L}$ to mean that under typing context Γ and store typing Ψ, term M evaluates to a value of type A with locality L. The resultant value is local if $L = \mathsf{L}$ and global if $L = \mathsf{G}$. (That is, L means "here only" and G "everywhere.") A binding $x : A_{@L}$ in a typing context Γ means that x holds a local value of type A if $L = \mathsf{L}$, or a global value of type A if $L = \mathsf{G}$. A store ψ maps locations to values, and a store typing Ψ maps locations to types of these values. We assume a relation $\mathsf{G} < \mathsf{L}$ to reflect the fact that a global value may be used as a local value, but not vice versa.

Figure 2 shows the type system of $\lambda_{\boxdot}$. The rule Var uses $L \leq L'$ to mean either $L = L'$ or $L < L'$. The rules $\to\!\mathsf{I}$ through Loc are all derived from typing rules in the simply-typed λ-calculus by annotating each typing judgment with a locality L. Each of the rules $\times\mathsf{I}$, $\times\mathsf{E_L}$, and $\times\mathsf{E_R}$ for product types uses the same unspecified locality L in its premise and conclusion. Note that these rules make sense only under the call-by-value semantics where (M, N) is a value only when both M and N are also values. If (M, N) is considered as a value for any two terms M and N, for example, all these rules become incorrect when $L = \mathsf{G}$. Similarly the rules $+\mathsf{I_L}$, $+\mathsf{I_R}$, and $+\mathsf{E}$ for sum types make sense only under the call-by-value semantics.

The modality $\boxdot$ has an introduction rule $\boxdot\mathsf{I}$ which states that M in $\mathsf{box}\ M$ always evaluates to a global value. The conclusion of the rule $\boxdot\mathsf{I}$ uses a locality L because $\mathsf{box}\ M$ itself may not be a global value if M contains references. For example, if l is a location of type $\mathsf{ref\ int}$, $\mathsf{box}\ !l$ is not a global value although it has a modal type $\boxdot\mathsf{int}$. The elimination rule $\boxdot\mathsf{E}$ uses an unspecified locality L in the conclusion because M in $\mathsf{letbox}\ x = M\ \mathsf{in}\ N$ does not specify whether N evaluates to a local value or a global value. That is, regardless of the type of M, the value to which N evaluates can still be either local or global.

The rule GVal is the main rule for connecting the two localities L and G. Its premise uses the following definition of Γ_{G} which extracts bindings for variables holding global values from Γ:

$$\Gamma_{\mathsf{G}} \quad = \quad \{x : A_{@\mathsf{G}} \mid x : A_{@\mathsf{G}} \in \Gamma\}$$

Then the premise $\Gamma_{\mathsf{G}} \mid \cdot \vdash V : A_{@\mathsf{L}}$ states that V is a value that uses no local values (because of Γ_{G}) and no references (because of an empty store typing); hence V is a mobile value. Since V is already a value and thus requires no further evaluation, we may say that V evaluates to a global value, which is expressed in the conclusion $\Gamma \mid \Psi \vdash V : A_{@\mathsf{G}}$.

The rule Prim uses the notion of primitive type to provide another way to connect the two localities L and G. A type is primitive if all its values are inherently mobile. For example, unit is a primitive type because its only value, $()$, typechecks under any typing context and store typing, as shown in the rule Unit. (Other examples of primitive types would be int for integers and bool for boolean values.) Formally we define primitive types as follows:

Definition 2.1. *P is a primitive type if $\Gamma \mid \Psi \vdash V : P_{@\mathsf{L}}$ implies $\Gamma_{\mathsf{G}} \mid \cdot \vdash V : P_{@\mathsf{L}}$.*

Then terms of primitive types always evaluate to mobile values, and $\Gamma \mid \Psi \vdash M : P_{@\mathsf{L}}$ automatically implies $\Gamma \mid \Psi \vdash M : P_{@\mathsf{G}}$ as shown in the rule Prim.

Under the type system in Figure 2, the rule Unit justifies the use of unit as a primitive type. In addition, $P_1 \times P_2$ and $P_1{+}P_2$ are primitive types if both P_1 and P_2 are primitive types, since values of product type $P_1 \times P_2$ have the form (V_1, V_2), and values of sum type $P_1{+}P_2$ have the form $\mathsf{inl}\ V$ or $\mathsf{inr}\ V$. Thus we use the following set of primitive types for $\lambda_{\boxdot}$:

$$\text{primitive type} \quad P \quad ::= \quad \mathsf{unit} \mid P \times P \mid P{+}P$$

Proposition 2.2 justifies the relation $\mathsf{G} < \mathsf{L}$.

Proposition 2.2.

The rule $\dfrac{\Gamma \mid \Psi \vdash M : A_{@\mathsf{G}}}{\Gamma \mid \Psi \vdash M : A_{@\mathsf{L}}}\ \mathsf{Global}$ *is admissible.*

$$\frac{x : A_{@L} \in \Gamma \quad L \leq L'}{\Gamma \mid \Psi \vdash x : A_{@L'}}\ \mathsf{Var} \qquad \frac{\Gamma, x : A_{@\mathsf{L}} \mid \Psi \vdash M : B_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash \lambda x{:}A.\, M : A \to B_{@\mathsf{L}}}\ {\to}\mathsf{I} \qquad \frac{\Gamma \mid \Psi \vdash M : A \to B_{@\mathsf{L}} \quad \Gamma \mid \Psi \vdash N : A_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash M\, N : B_{@\mathsf{L}}}\ {\to}\mathsf{E}$$

$$\frac{}{\Gamma \mid \Psi \vdash () : \mathsf{unit}_{@\mathsf{L}}}\ \mathsf{Unit} \qquad \frac{\Gamma, x : A_{@\mathsf{L}} \mid \Psi \vdash M : A_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash \mathsf{fix}\, x{:}A.\, M : A_{@\mathsf{L}}}\ \mathsf{Fix} \qquad \frac{\Gamma \mid \Psi \vdash M : A_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash \mathsf{ref}\ M : \mathsf{ref}\ A_{@\mathsf{L}}}\ \mathsf{Ref}$$

$$\frac{\Gamma \mid \Psi \vdash M : \mathsf{ref}\ A_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash {!M} : A_{@\mathsf{L}}}\ \mathsf{Drf} \qquad \frac{\Gamma \mid \Psi \vdash M : \mathsf{ref}\ A_{@\mathsf{L}} \quad \Gamma \mid \Psi \vdash N : A_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash M := N : \mathsf{unit}_{@\mathsf{L}}}\ \mathsf{Asn} \qquad \frac{\Psi(l) = A}{\Gamma \mid \Psi \vdash l : \mathsf{ref}\ A_{@\mathsf{L}}}\ \mathsf{Loc}$$

$$\frac{\Gamma \mid \Psi \vdash M : A_{@L} \quad \Gamma \mid \Psi \vdash M : B_{@L}}{\Gamma \mid \Psi \vdash (M, N) : A \times B_{@L}}\ {\times}\mathsf{I} \qquad \frac{\Gamma \mid \Psi \vdash M : A \times B_{@L}}{\Gamma \mid \Psi \vdash \mathsf{fst}\ M : A_{@L}}\ {\times}\mathsf{E_L} \qquad \frac{\Gamma \mid \Psi \vdash M : A \times B_{@L}}{\Gamma \mid \Psi \vdash \mathsf{snd}\ M : B_{@L}}\ {\times}\mathsf{E_R}$$

$$\frac{\Gamma \mid \Psi \vdash M : A_{@L}}{\Gamma \mid \Psi \vdash \mathsf{inl}\ M : A{+}B_{@L}}\ {+}\mathsf{I_L} \qquad \frac{\Gamma \mid \Psi \vdash M : B_{@L}}{\Gamma \mid \Psi \vdash \mathsf{inr}\ M : A{+}B_{@L}}\ {+}\mathsf{I_R} \qquad \frac{\Gamma \mid \Psi \vdash M : A{+}B_{@L'} \quad \begin{array}{c}\Gamma, x : A_{@L'} \mid \Psi \vdash N : C_{@L} \\ \Gamma, x' : B_{@L'} \mid \Psi \vdash N' : C_{@L}\end{array}}{\Gamma \mid \Psi \vdash \mathsf{case}\ M\ \mathsf{of\ inl}\ x \Rightarrow N \mid \mathsf{inr}\ x' \Rightarrow N' : C_{@L}}\ {+}\mathsf{E}$$

$$\frac{\Gamma \mid \Psi \vdash M : A_{@\mathsf{G}}}{\Gamma \mid \Psi \vdash \mathsf{box}\ M : \boxdot A_{@\mathsf{L}}}\ \boxdot\mathsf{I} \qquad \frac{\Gamma \mid \Psi \vdash M : \boxdot A_{@L} \quad \Gamma, x : A_{@\mathsf{G}} \mid \Psi \vdash N : C_{@L}}{\Gamma \mid \Psi \vdash \mathsf{letbox}\ x = M\ \mathsf{in}\ N : C_{@L}}\ \boxdot\mathsf{E}$$

$$\frac{\Gamma_{\mathsf{G}} \mid \cdot \vdash V : A_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash V : A_{@\mathsf{G}}}\ \mathsf{GVal} \qquad \frac{\Gamma \mid \Psi \vdash M : P_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash M : P_{@\mathsf{G}}}\ \mathsf{Prim}$$

Figure 2. Type system of $\lambda_{\boxdot}$

$$\begin{array}{lll} \text{evaluation context} & \phi & ::= & [] \mid \phi\ M \mid V\ \phi \mid (\phi, M) \mid (V, \phi) \mid \mathsf{fst}\ \phi \mid \mathsf{snd}\ \phi \mid \\ & & & \mathsf{inl}\ \phi \mid \mathsf{inr}\ \phi \mid \mathsf{case}\ \phi\ \mathsf{of\ inl}\ x \Rightarrow M \mid \mathsf{inr}\ x \Rightarrow M\ \mathsf{ref}\ \phi \mid {!\phi} \mid \phi := M \mid V := \phi \mid \\ & & & \mathsf{letbox}\ x = \phi\ \mathsf{in}\ M \mid \mathsf{letbox}\ x = \mathsf{box}\ \phi\ \mathsf{in}\ M \end{array}$$

$$\begin{array}{rcl} (\lambda x{:}A.\, M)\ V & \to_\beta & [V/x]M \\ \mathsf{fix}\, x{:}A.\, M & \to_\beta & [\mathsf{fix}\, x{:}A.\, M/x]M \\ \mathsf{letbox}\ x = \mathsf{box}\ V\ \mathsf{in}\ M & \to_\beta & [V/x]M \end{array} \qquad \begin{array}{rcl} \mathsf{fst}\ (V, V') & \to_\beta & V \\ \mathsf{snd}\ (V, V') & \to_\beta & V' \\ \mathsf{case\ inl}\ V\ \mathsf{of\ inl}\ x \Rightarrow M \mid \mathsf{inr}\ x' \Rightarrow M' & \to_\beta & [V/x]M \\ \mathsf{case\ inr}\ V\ \mathsf{of\ inl}\ x \Rightarrow M \mid \mathsf{inr}\ x' \Rightarrow M' & \to_\beta & [V/x']M' \end{array}$$

$$\frac{M \to_\beta M'}{\phi[\![M]\!] \mid \psi \to \phi[\![M']\!] \mid \psi}\ R_\beta$$

$$\frac{\mathit{fresh}\ l \notin \mathit{dom}(\psi)}{\phi[\![\mathsf{ref}\ V]\!] \mid \psi \to \phi[\![l]\!] \mid \psi, l \mapsto V}\ \mathit{Ref} \qquad \frac{\psi(l) = V}{\phi[\![!l]\!] \mid \psi \to \phi[\![V]\!] \mid \psi}\ \mathit{Drf} \qquad \frac{}{\phi[\![l := V]\!] \mid \psi \to \phi[\![()]\!] \mid [l \mapsto V]\psi}\ \mathit{Asn}$$

Figure 3. Operational semantics for $\lambda_{\boxdot}$

Figure 3 shows the operational semantics for $\lambda_{\boxdot}$. It uses a reduction judgment $M \mid \psi \to M' \mid \psi'$ which means that term M with store ψ reduces to term M' with store ψ'. A β-reduction $M \to_\beta M'$ uses a capture-avoiding substitution $[N/x]M$ defined in a standard way. $\phi[\![M]\!]$ fills the hole $[]$ in evaluation context ϕ with term M. $[l \mapsto V]\psi$ replaces $l \mapsto V'$ in store ψ by $l \mapsto V$. $\psi(l)$ denotes the value to which l is mapped under ψ; $\mathit{dom}(\psi)$ denotes the set of locations mapped under ψ.

Since $\mathsf{letbox}\ x = \mathsf{box}\ \phi\ \mathsf{in}\ N$ is an evaluation context, $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$ further evaluates M so as to substitute the resultant value for x in N, even though $\mathsf{box}\ M$ is already a value. Thus $\mathsf{letbox}\ x = M'\ \mathsf{in}\ N$ first evaluates M' to identify how to obtain a value to be substituted for x, and then obtains such a value by evaluating M, if M' evaluates to $\mathsf{box}\ M$. Since M evaluates to a global value, all occurrences of x in N become bound to a global value, as required by the rules $\boxdot\mathsf{E}$.

2.2 Type safety of $\lambda_{\boxdot}$

The proof of type safety of $\lambda_{\boxdot}$ needs a store typing judgment $\Psi \vdash \psi\ \mathsf{okay}$ which means that store ψ conforms to store typing Ψ. $\Psi(l)$ denotes the type to which l is mapped under Ψ; $\mathit{dom}(\Psi)$ denotes the set of locations mapped under Ψ.

$$\frac{\mathit{dom}(\Psi) = \mathit{dom}(\psi) \qquad \begin{array}{c}\text{for every } l \in \mathit{dom}(\psi) \\ \cdot \mid \Psi \vdash \psi(l) : \Psi(l)_{@\mathsf{L}}\end{array}}{\Psi \vdash \psi\ \mathsf{okay}}\ \mathsf{Store}$$

The proof of progress (Theorem 2.3) uses a canonical forms lemma, as usual. The proof of type preservation (Theorem 2.6) uses a substitution theorem (Theorem 2.4). It also uses Lemma 2.5 whose proof uses the definition of primitive types (Definition 2.1).

Theorem 2.3 (Progress)**.** *Suppose* $\cdot \mid \Psi \vdash M : A_{@L}$. *Then either:*
(1) M is a value, or
(2) for any store ψ such that $\Psi \vdash \psi$ okay, there exist some term M' and store ψ' such that $M \mid \psi \to M' \mid \psi'$.

Theorem 2.4 (Substitution)**.**
If $\Gamma \mid \Psi \vdash N : A_{@\mathsf{L}}$, *then*
$\Gamma, x : A_{@\mathsf{L}} \mid \Psi \vdash M : C_{@L}$ *implies* $\Gamma \mid \Psi \vdash [N/x]M : C_{@L}$.
If $\Gamma_{\mathsf{G}} \mid \cdot \vdash V : A_{@\mathsf{L}}$, *then*
$\Gamma, x : A_{@\mathsf{G}} \mid \Psi \vdash M : C_{@L}$ *implies* $\Gamma \mid \Psi \vdash [V/x]M : C_{@L}$.

$$
\begin{array}{llll}
\text{term} & M & ::= & \cdots \mid \mathsf{spawn}\ M \\
\text{evaluation context} & \phi & ::= & \cdots \mid \mathsf{spawn}\ \phi \mid \mathsf{spawn\ box}\ \phi \\
\text{thread} & \gamma & & \\
\text{configuration} & \pi & ::= & \cdot \mid \pi, \{M \mid \psi\ @\ \gamma\} \\
\text{configuration typing} & \Pi & ::= & \cdot \mid \Pi, \gamma : A
\end{array}
$$

$$\frac{\Gamma \mid \Psi \vdash M : \boxdot(\mathsf{unit} \to A)_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash \mathsf{spawn}\ M : \mathsf{unit}_{@\mathsf{L}}}\ \mathsf{Spawn}$$

$$\frac{dom(\Pi) = dom(\pi) \quad \begin{array}{l} \text{for each } \{M \mid \psi\ @\ \gamma\} \in \pi, \\ \quad \text{there exist } A_\gamma, \Psi_\gamma \text{ such that} \\ \quad\quad \gamma : A_\gamma \in \Pi \\ \quad\quad \Psi_\gamma \vdash \psi\ \mathsf{okay} \\ \quad\quad \cdot \mid \Psi_\gamma \vdash M : A_{\gamma @\mathsf{L}} \end{array}}{\Pi \vdash \pi\ \mathsf{okay}}\ \mathsf{Conf}$$

$$\frac{M \mid \psi \to M' \mid \psi'}{\pi, \{M \mid \psi\ @\ \gamma\} \Rightarrow \pi, \{M' \mid \psi'\ @\ \gamma\}}\ \mathit{Red}$$

$$\frac{\mathit{fresh}\ \gamma'}{\begin{array}{l}\pi, \{\phi[\![\mathsf{spawn\ box}\ V]\!] \mid \psi\ @\ \gamma\} \Rightarrow \\ \qquad \pi, \{\phi[\![()]\!] \mid \psi\ @\ \gamma\}, \{V\ () \mid \cdot\ @\ \gamma'\}\end{array}}\ \mathit{Spawn}$$

Figure 4. Definition of $\lambda^{\mathsf{P}}_{\boxdot}$

Lemma 2.5. *If* $\cdot \mid \Psi \vdash V : A_{@\mathsf{G}}$, *then* $\cdot \mid \cdot \vdash V : A_{@\mathsf{L}}$.

Proof. By induction on the structure of the proof of $\cdot \mid \Psi \vdash V : A_{@\mathsf{G}}$. We need to consider the rules Prim, GVal, $\times\mathsf{I}$, $+\mathsf{I_L}$, and $+\mathsf{I_R}$. In the case of the rule Prim, $\cdot \mid \Psi \vdash V : A_{@\mathsf{L}}$ holds where A is a primitive type, and $\cdot \mid \cdot \vdash V : A_{@\mathsf{L}}$ follows by Definition 2.1. □

Theorem 2.6 (Type preservation)**.**
Suppose $\cdot \mid \Psi \vdash M : A_{@L}$, $\Psi \vdash \psi\ \mathsf{okay}$, *and* $M \mid \psi \to M' \mid \psi'$. *Then there exists a store typing* Ψ' *such that* $\cdot \mid \Psi' \vdash M' : A_{@L}$, $\Psi \subset \Psi'$, *and* $\Psi' \vdash \psi'\ \mathsf{okay}$.

3. $\lambda^{\mathsf{P}}_{\boxdot}$ with a parallel operational semantics

Although its type system uses localities and modal types to differentiate global values from local values, $\lambda_{\boxdot}$ is just a base language for sequential computations to which global values are of no use. That is, its operational semantics focuses only on the sequential computation at a hypothetical thread and there is no way to exploit global values for communications between threads. This section extends $\lambda_{\boxdot}$ with a parallel operational semantics which models multiple threads running concurrently (but not communicating with each other yet). The resultant language is called $\lambda^{\mathsf{P}}_{\boxdot}$.

Figure 4 shows the definition for $\lambda^{\mathsf{P}}_{\boxdot}$. At the syntax level, $\lambda^{\mathsf{P}}_{\boxdot}$ augments $\lambda_{\boxdot}$ with a new construct $\mathsf{spawn}\ M$ whose typing rule Spawn requires M to be of type $\boxdot(\mathsf{unit} \to A)$. $\mathsf{spawn}\ M$ starts a fresh thread with an empty store by evaluating a term obtained as follows. First it evaluates M to obtain $\mathsf{box}\ M'$. Then it evaluates the inner term M' (as indicated by the evaluation context $\mathsf{spawn\ box}\ \phi$) to obtain $\lambda\_ \!: \mathsf{unit}.\ M''$ (or a similar form of thunk). Since $\mathsf{box}\ M'$ has type $\boxdot(\mathsf{unit} \to A)$, M' evaluates to a global value, which means that $\lambda\_ : \mathsf{unit}.\ M''$ contains no references. Therefore it is safe to evaluate $(\lambda\_ \!: \mathsf{unit}.\ M'')\ ()$ at a fresh thread with an empty store, which effectively starts to evaluate M'' at the fresh thread. In a realistic language, we could use syntactic sugar $\mathsf{spawnT}\ \{M\}$ defined as $\mathsf{spawn\ box}\ \lambda\_ \!: \mathsf{unit}.\ M$ in order to start a fresh thread by evaluating M without such intervening evaluations:

$$\frac{\Gamma_{\mathsf{G}} \mid \cdot \vdash M : A_{@\mathsf{L}}}{\Gamma \mid \Psi \vdash \mathsf{spawnT}\ \{M\} : \mathsf{unit}_{@\mathsf{L}}}\ \mathsf{SpawnTerm}$$

Note that the premise $\Gamma_{\mathsf{G}} \mid \cdot \vdash M : A_{@\mathsf{L}}$ implies $\Gamma \mid \Psi \vdash \mathsf{box}\ \lambda\_ \!: \mathsf{unit}.\ M : \boxdot(\mathsf{unit} \to A)_{@\mathsf{L}}$.

A *configuration* π, as an unordered set, represents the state of a parallel computation by associating each thread γ with a term M being evaluated at γ and a store ψ allocated at γ, written as $\{M \mid \psi\ @\ \gamma\}$. A *configuration typing* Π records the type of the term being evaluated at each thread. The type system of $\lambda^{\mathsf{P}}_{\boxdot}$ uses the same typing judgment as $\lambda_{\boxdot}$, except that it also uses a *configuration typing judgment* $\Pi \vdash \pi\ \mathsf{okay}$ to mean that configuration π has configuration typing Π. The rule Conf may be regarded as the definition of the configuration typing judgment, where $dom(\Pi)$ and $dom(\pi)$ denote the set of threads in Π and π, respectively. Note that for each thread γ such that $\{M \mid \psi\ @\ \gamma\} \in \pi$, it infers a store typing Ψ_γ by typechecking all locations present in M and ψ.

The parallel operational semantics uses a *configuration transition judgment* $\pi \Rightarrow \pi'$ to mean that configuration π reduces (by the rule *Red*) or evolves (by the rule *Spawn*) to configuration π'. The rule *Red* says that a parallel computation consists primarily of sequential computations performed at individual threads. The rule *Spawn* starts a new thread γ' by evaluating $V\ ()$. Since $\mathsf{box}\ V$ guarantees that V is a global value, it is safe to evaluate $V\ ()$ with an empty store at γ'. Note that a configuration transition is non-deterministic because a configuration may choose an arbitrary thread to which either rule is applied.

Type safety of $\lambda^{\mathsf{P}}_{\boxdot}$ consists of *configuration progress* (Theorem 3.1) and *configuration typing preservation* (Theorem 3.2). It is a corollary of type safety of $\lambda^{\mathsf{PC}}_{\boxdot}$ to be presented in the next section.

Theorem 3.1 (Configuration progress)**.** *Suppose* $\Pi \vdash \pi\ \mathsf{okay}$. *Then either:*
(1) π *consists only of* $\{V \mid \psi\ @\ \gamma\}$, *or*
(2) there exists π' *such that* $\pi \Rightarrow \pi'$.

Theorem 3.2 (Configuration typing preservation)**.**
Suppose $\Pi \vdash \pi\ \mathsf{okay}$ *and* $\pi \Rightarrow \pi'$. *Then there exists a configuration typing* Π' *such that* $\Pi \subset \Pi'$ *and* $\Pi' \vdash \pi'\ \mathsf{okay}$.

4. $\lambda^{\mathsf{PC}}_{\boxdot}$ with higher-order channels

This section extends $\lambda^{\mathsf{P}}_{\boxdot}$ with higher-order channels to obtain $\lambda^{\mathsf{PC}}_{\boxdot}$. In order to achieve channel locality, $\lambda^{\mathsf{PC}}_{\boxdot}$ splits channels into two kinds: read channels and write channels. A read channel a^r is assigned a read channel type $\langle A\rangle^r$, and is treated as a local value so that it cannot escape the thread at which it is created. A write channel a^w is assigned a write channel type $\langle A\rangle^w$, and is treated as a global value so that it can be transmitted to another thread. Since values written to write channels travel between threads, they cannot be local values. That is, only global values are allowed to be written to write channels, and consequently all values read from read channels are also global.

Figure 5 shows the definition for $\lambda^{\mathsf{PC}}_{\boxdot}$. A read channel a^r of type $\langle A\rangle^r$ receives a global value of type A from its corresponding write channel a^w via a channel read $a^r\,?$. A write channel a^w of type $\langle A\rangle^w$ transmits a global value V of type A to its corresponding read channel a^r via a channel write $a^w\,!\,V$. A new construct $\mathsf{new}_{\langle A\rangle}$ dynamically creates a pair of read channel a^r and write channel a^w to open unidirectional communications from a^w to a^r. $a^r \mapsto A$ in a *read channel typing* Φ^r means that a^r has type $\langle A\rangle^r$; similarly $a^w \mapsto A$ in a *write channel typing* Φ^w means that a^w has type $\langle A\rangle^w$.

The type system of $\lambda^{\mathsf{PC}}_{\boxdot}$ uses a read channel typing Φ^r and a write channel typing Φ^w in each typing judgment:

$$\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash M : A_{@L}$$

$$
\begin{array}{llll}
\text{type} & A & ::= & \cdots \mid \langle A\rangle^r \mid \langle A\rangle^w \\
\text{term} & M & ::= & \cdots \mid \mathsf{new}_{\langle A\rangle} \mid a^r \mid a^w \mid M\,? \mid M\,!\,M \\
\text{value} & V & ::= & \cdots \mid a^r \mid a^w \\
\text{evaluation context} & \phi & ::= & \cdots \mid \phi\,? \mid \phi\,!\,M \mid V\,!\,\phi \\
\text{read channel typing} & \Phi^r & ::= & \cdot \mid \Phi^r, a^r \mapsto A \\
\text{write channel typing} & \Phi^w & ::= & \cdot \mid \Phi^w, a^w \mapsto A
\end{array}
$$

$$
\frac{\Gamma_{\mathsf{G}} \mid \Phi^w; \cdot; \cdot \vdash V : A_{@\mathsf{L}}}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash V : A_{@\mathsf{G}}}\ \mathsf{GVal}
\qquad
\frac{}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash \mathsf{new}_{\langle A\rangle} : \langle A\rangle^r \times \langle A\rangle^w{}_{@\mathsf{L}}}\ \mathsf{New}
$$

$$
\frac{a^r \mapsto A \in \Phi^r}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash a^r : \langle A\rangle^r{}_{@\mathsf{L}}}\ \mathsf{ChanR}
\qquad
\frac{a^w \mapsto A \in \Phi^w}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash a^w : \langle A\rangle^w{}_{@\mathsf{L}}}\ \mathsf{ChanW}
$$

$$
\frac{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash M : \langle A\rangle^r{}_{@\mathsf{L}}}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash M\,? : A_{@L}}\ \mathsf{R?}\ (L = \mathsf{G} \text{ or } L = \mathsf{L})
\qquad
\frac{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash M : \langle A\rangle^w{}_{@\mathsf{L}} \quad \Gamma \mid \Phi^w; \Phi^r; \Psi \vdash N : A_{@\mathsf{G}}}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash M\,!\,N : \mathsf{unit}_{@\mathsf{L}}}\ \mathsf{W!}
$$

$$
\frac{dom(\Psi) = dom(\psi) \qquad \begin{array}{c}\text{for every } l \in dom(\psi) \\ \cdot \mid \Phi^w; \Phi^r; \Psi \vdash \psi(l) : \Psi(l)_{@\mathsf{L}}\end{array}}{\Phi^w; \Phi^r; \Psi \vdash \psi\ \mathsf{okay}}\ \mathsf{Store}
$$

$$
\frac{dom(\Pi) = dom(\pi) \qquad \begin{array}{l}\text{for each } \{M \mid \psi \mathbin{@} \gamma\} \in \pi, \\ \quad \text{there exist } A_\gamma, \Phi^r_\gamma, \Psi_\gamma \text{ such that} \\ \quad\quad \gamma : A_\gamma \in \Pi \\ \quad\quad \Phi^w; \Phi^r_\gamma; \Psi_\gamma \vdash \psi\ \mathsf{okay} \\ \quad\quad \cdot \mid \Phi^w; \Phi^r_\gamma; \Psi_\gamma \vdash M : A_{\gamma @\mathsf{L}} \end{array} \qquad \begin{array}{l}\text{if } \gamma \neq \gamma', \\ \quad \Phi^r_\gamma \cap \Phi^r_{\gamma'} = \varnothing\end{array}}{\Phi^w; \Pi \vdash \pi\ \mathsf{okay}}\ \mathsf{Conf}
$$

$$
\frac{\mathit{fresh}\ (a^r, a^w)}{\begin{array}{l}\pi, \{\phi[\![\mathsf{new}_{\langle A\rangle}]\!] \mid \psi \mathbin{@} \gamma\} \Rightarrow \\ \qquad \pi, \{\phi[\![(a^r, a^w)]\!] \mid \psi \mathbin{@} \gamma\}\end{array}}\ \mathit{New}
\qquad
\frac{}{\begin{array}{l}\pi, \{\phi[\![a^r\,?]\!] \mid \psi \mathbin{@} \gamma\}, \{\phi'[\![a^w\,!\,V]\!] \mid \psi' \mathbin{@} \gamma'\} \Rightarrow \\ \qquad \pi, \{\phi[\![V]\!] \mid \psi \mathbin{@} \gamma\}, \{\phi'[\![()]\!] \mid \psi' \mathbin{@} \gamma'\}\end{array}}\ \mathit{Sync}
$$

Figure 5. Definition of $\lambda^{\mathsf{PC}}_{\boxdot}$

All the previous typing rules are extended in a straightforward way by adding Φ^r and Φ^w to each typing judgment. The only exception is the rule GVal whose premise ($\Gamma_{\mathsf{G}} \mid \Phi^w; \cdot; \cdot \vdash V : A_{@\mathsf{L}}$) uses an empty read channel typing because read channels are local values. Accordingly we redefine primitive types as follows:

Definition 4.1.
P is a primitive type if and only if $\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash V : P_{@\mathsf{L}}$ *implies* $\Gamma_{\mathsf{G}} \mid \Phi^w; \cdot; \cdot \vdash V : P_{@\mathsf{L}}$.

Under the new definition of primitive types, a write channel type $\langle A\rangle^w$ becomes a primitive type (see the rule ChanW):

$$\text{primitive type} \quad P \quad ::= \quad \cdots \mid \langle A\rangle^w$$

The rule R? uses an unspecified locality L in the conclusion so that Proposition 2.2 continues to hold. The rule W! says that channel writes accept only global values. The rule Store uses an extended store typing judgment $\Phi^w; \Phi^r; \Psi \vdash \psi\ \mathsf{okay}$ to check that store ψ conforms to store typing Ψ under Φ^r and Φ^w.

The rule Conf is of particular importance in $\lambda^{\mathsf{PC}}_{\boxdot}$ because it verifies channel locality in a given configuration. Note that an extended configuration typing judgment $\Phi^w; \Pi \vdash \pi\ \mathsf{okay}$ shares a write channel typing Φ^w for all threads (because write channels are global values), but does not assume a specific read channel typing. Instead, for each thread γ such that $\{M \mid \psi \mathbin{@} \gamma\} \in \pi$, it infers a *unique* read channel typing Φ^r_γ, in addition to a store typing Ψ_γ, by typechecking all read channels present in M and ψ. The uniqueness of Φ^r_γ for each thread γ, as stated in the third premise, implies that no two threads share common read channels, and thus amounts to channel locality in π.

The parallel operational semantics uses the same configuration transition judgment $\pi \Rightarrow \pi'$ as in $\lambda^{\mathsf{P}}_{\boxdot}$. The premise of the rule *New* means that read channel a^r and write channel a^w are unique across the entire set of threads. The rule *Sync* says that a channel read $a^r\,?$ and a channel write $a^w\,!\,V$ occur synchronously. (An asynchronous version of $\lambda^{\mathsf{PC}}_{\boxdot}$ would require a different parallel operational semantics, but the same type system would continue to work.)

As with $\lambda^{\mathsf{P}}_{\boxdot}$, type safety of $\lambda^{\mathsf{PC}}_{\boxdot}$ consists of configuration progress (Theorem 4.3) and configuration typing preservation (Theorem 4.5). Proofs of Theorems 4.3 and 4.5 use type safety for sequential computations in $\lambda^{\mathsf{PC}}_{\boxdot}$ (Propositions 4.2 and 4.4). Note that in Theorem 4.5, a configuration transition $\pi \Rightarrow \pi'$ preserves channel locality because of the use of extended configuration typing judgments.

Proposition 4.2 (Progress)**.**
Suppose $\cdot \mid \Phi^w; \Phi^r; \Psi \vdash M : A_{@L}$*. Then either:*
(1) M *is a value,*
(2) $M = \phi[\![\mathsf{new}_{\langle A\rangle}]\!]$*,*
(3) $M = \phi[\![a^r\,?]\!]$*,*
(4) $M = \phi[\![a^w\,!\,V]\!]$*,*
(5) $M = \phi[\![\mathsf{spawn\ box}\ V]\!]$*, or*
(6) for any store ψ *such that* $\Phi^w; \Phi^r; \Psi \vdash \psi\ \mathsf{okay}$*, there exist some term* M' *and store* ψ' *such that* $M \mid \psi \longrightarrow M' \mid \psi'$*.*

Theorem 4.3 (Configuration progress)**.**
Suppose $\Phi^w; \Pi \vdash \pi\ \mathsf{okay}$*. Then either:*
(1) π *consists only of*
$\{V \mid \psi \mathbin{@} \gamma\}$,
$\{\phi[\![a^r\,?]\!] \mid \psi \mathbin{@} \gamma\}$,
$\{\phi[\![a^w\,!\,V]\!] \mid \psi \mathbin{@} \gamma\}$*, or*
(2) there exists π' *such that* $\pi \Rightarrow \pi'$*.*

Proposition 4.4 (Type preservation)**.**
Suppose $\cdot \mid \Phi^w; \Phi^r; \Psi \vdash M : A_{@\mathsf{L}}$*,* $\Phi^w; \Phi^r; \Psi \vdash \psi$ okay*, and* $M \mid \psi \longrightarrow M' \mid \psi'$*. Then there exists a store typing* Ψ' *such that* $\cdot \mid \Phi^w; \Phi^r; \Psi' \vdash M' : A_{@\mathsf{L}}$*,* $\Psi \subset \Psi'$*, and* $\Phi^w; \Phi^r; \Psi' \vdash \psi'$ okay*.*

Theorem 4.5 (Configuration typing preservation)**.**
Suppose $\Phi^w; \Pi \vdash \pi$ okay *and* $\pi \Rightarrow \pi'$*. Then there exist a write channel typing* ${\Phi'}^w$ *and a configuration typing* Π' *such that* $\Phi^w \subset {\Phi'}^w$*,* $\Pi \subset \Pi'$*, and* ${\Phi'}^w; \Pi' \vdash \pi'$ okay*.*

5. Examples

This section presents examples of implementing various communication constructs in $\lambda_{\boxdot}^{\mathsf{PC}}$: futures, bidirectional communications, shared references, remote evaluation, code on demand, and hot code replacement. Throughout the examples, we use the following syntactic sugar:

$$\begin{array}{rcl}
\mathsf{let}\ x = M\ \mathsf{in}\ N & \equiv & (\lambda x{:}\_.\, N)\ M \\
\mathsf{let}\ (x,y) = M\ \mathsf{in}\ N & \equiv & \mathsf{let}\ z = M\ \mathsf{in} \\
& & \mathsf{let}\ x = \mathsf{fst}\ z\ \mathsf{in}\ \mathsf{let}\ y = \mathsf{snd}\ z\ \mathsf{in}\ N \\
\mathsf{let\ rec}\ x = M\ \mathsf{in}\ N & \equiv & \mathsf{let}\ x = \mathsf{fix}\ x{:}\_.\, M\ \mathsf{in}\ N
\end{array}$$

We also use syntactic sugar spawnT $\{M\}$ defined in Section 3.

5.1 Futures

A future (Halstead, Jr. 1985) is a communication construct which can be thought of as a pointer to a thread. After the thread finishes its computation, the pointer may be dereferenced to obtain the result. Below we simulate futures using read channels.

In order to create a future, we first need to evaluate a term M at a fresh thread. Since M is to be evaluated at a remote thread, it must contain no references. We implement the future with a read channel a^r such that the result of evaluating M is written to the corresponding write channel a^w. Since a channel write entails a communication between threads, M must evaluate to a global value (in accordance with the rule W!). Hence future, our construct for futures, expects a value of type $\boxdot\boxdot A$, say box box M, and returns a read channel of type $\langle A \rangle^r$ to which the result of evaluating M is sent:

```
future : $\boxdot\boxdot A \to \langle A \rangle^r{}_{@\mathsf{G}}$
future =
  $\lambda x{:}\boxdot\boxdot A.$
    letbox $x' = x$ in
    let $(y_r, y_w) = \mathsf{new}_{\langle A \rangle}$ in
    letbox $y'_w = \mathsf{box}\ y_w$ in
    let $\_ = \mathsf{spawnT}\ \{\mathsf{letbox}\ z = x'\ \mathsf{in}\ y'_w\,!\,z\}$ in
    $y_r$
```

5.2 Bidirectional communications

Communications from a child thread back to its parent thread are easy to implement because write channels are global values — we start the child thread by evaluating a term containing write channels whose corresponding read channels belong to the parent thread. For communications in the other direction, however, the child thread needs to somehow return a write channel to the parent thread, which can be done only via another write channel originating from the parent thread. Thus, in order to receive a write channel of type $\langle A \rangle^w$ from a child thread, the parent thread creates a pair of read channel a^r of type $\langle \langle A \rangle^w \rangle^r$ and write channel a^w of type $\langle \langle A \rangle^w \rangle^w$. Then the child thread is passed a^w, through which a write channel of type $\langle A \rangle^w$ is sent to the parent thread.

We design spawn$^{\mathsf{BI}}$, our construct for bidirectional communications (or for unidirectional communications from a parent thread to a child thread), in such a way that given a value of type $\boxdot(\langle A \rangle^r \to C)$, say box $\lambda x_r{:}\langle A \rangle^r.\, M$, it creates a child thread evaluating M with x_r bound to a read channel a^r of type $\langle A \rangle^r$ and returns a corresponding write channel a^w to the parent thread:

```
spawn$^{\mathsf{BI}}$ : $\boxdot(\langle A \rangle^r \to C) \to \langle A \rangle^w{}_{@\mathsf{G}}$
spawn$^{\mathsf{BI}}$ =
  $\lambda z{:}\boxdot(\langle A \rangle^r \to C).$
    let $(x_r, x_w) = \mathsf{new}_{\langle \langle A \rangle^w \rangle}$ in
    letbox $x'_w = \mathsf{box}\ x_w$ in
    letbox $z' = z$ in
    let $\_$ =
      spawnT { let $(y_r, y_w) = \mathsf{new}_{\langle A \rangle}$ in
               let $\_ = x'_w\,!\,y_w$ in
               $z'\ y_r$ }
    in
    $x_r\,?$
```

Note that by the rule *Sync*, the channel write $x'_w\,!\,y_w$ blocks the child thread until it synchronizes with the corresponding channel read $x_r\,?$. Thus the evaluation of $z'\ y_r$, the core of the child thread, may not start immediately after $\mathsf{new}_{\langle A \rangle}$ returns a pair of read and write channels. To start the evaluation of $z'\ y_r$ immediately, the child thread could delegate the channel write to another new thread. In general, a channel write $M\,!\,N$ can be replaced by the following term which spawns a new thread for performing the channel write and returns immediately (unless an evaluation of M or N gets stuck with a channel read or a channel write):

```
letbox $x_M = \mathsf{box}\ M$ in
letbox $x_N = \mathsf{box}\ N$ in
spawnT $\{x_M\,!\,x_N\}$
```

The same idea can be applied to all instances of channel writes in the examples below.

5.3 Shared references

We implement a shared reference for type A as a thread with two components: a read channel of type $\langle \langle A \rangle^w + A \rangle^r$, as an interface, and a global value of type A, as its current content. Any thread may request the current content of the reference by sending a value inl a^w (of type $\langle A \rangle^w + A$) to the read channel, in which case the current content is written back to a^w. To update the content of the reference, it sends a value inr V (also of type $\langle A \rangle^w + A$) to the read channel, in which case the current content is updated with a new value V.

As an illustration, we use spawn$^{\mathsf{BI}}$ to create a shared reference for type int initialized with a value of 0:

```
let $cell_w$ =
  spawn$^{\mathsf{BI}}$
    box $\lambda cell_r{:}\langle \langle \mathsf{int} \rangle^w + \mathsf{int} \rangle^r.$
      let rec $f = \lambda n{:}\mathsf{int}.$
        case $cell_r\,?$ of inl $ch_w \Rightarrow$ let $\_ = ch_w\,!\,n$ in $f\ n$
                          | inr $v \Rightarrow f\ v$
        in
      $f\ 0$
```

Then we use a write channel in $cell_w$ to implement get, of type unit $\to$ int, for requesting the current content of the reference and set, of type int $\to$ unit, for updating the content of the reference; note that each call to get creates a fresh pair of read channel and write channel:

```
letbox $cell'_w = \mathsf{box}\ cell_w$ in
letbox $get$ = box $\lambda\_{:}\mathsf{unit}.$ let $(x_r, x_w) = \mathsf{new}_{\langle \mathsf{int} \rangle}$ in
                             let $\_ = cell'_w\,!\,(\mathsf{inl}\ x_w)$ in
                             $x_r\,?$
  in
letbox $set$ = box $\lambda n{:}\mathsf{int}.\ cell'_w\,!\,(\mathsf{inr}\ n)$ in
$(get, set)$
```

Since both get and set are global values, they may be present at any thread, which means that the reference can be shared by all threads.

5.4 Remote evaluation

Remote evaluation (Stamos and Gifford 1990) is a mechanism for exploiting a set of services exported by a server in a flexible way. A client sends to the server a program whose execution *by the server* may invoke these services. That is, it sends not arguments for these services but a program utilizing these services. We demonstrate how to implement remote evaluation in $\lambda^{\mathsf{PC}}_{\boxdot}$ with a server providing a service for calculating the successor of an integer.

We use a write channel of type $\langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^w$ as an interface to the service. The idea is that the service responds to (n, a^w) written to the write channel by writing $n+1$ back to a^w. A call to $\mathsf{spawn}^{\mathsf{BI}}$ with the following global value $V_{service}$ as an argument starts the service and returns its interface:

```
$V_{service}$ : $\boxdot(\langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^r \to \mathsf{unit})_{@\mathsf{G}}$
$V_{service}$ =  box $\lambda s_r : \langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^r$.
                   let rec $f = \lambda\_ : \mathsf{unit}$.  let $(n, ch) = s_r\,?$ in
                                                let $\_ = ch\,!\,(n+1)$ in
                                                $f\ ()$
                     in
                   $f\ ()$
```

The following term starts the server and returns its interface which is a write channel req_w of type $\langle \langle \langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^w \to \mathsf{unit} \rangle^w$. Since req_w is a global value, any client can use the service by sending a global value $\lambda s : \langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^w .\, M$, upon receiving which the server binds s to the interface to the service and evaluates M:

```
let $req_w$ =
  spawn$^{\mathsf{BI}}$
    box $\lambda req_r : \langle \langle \langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^w \to \mathsf{unit} \rangle^r$.
      let $s_w$ = spawn$^{\mathsf{BI}}$ $V_{service}$ in
      let rec $f = \lambda\_ : \mathsf{unit}$. let $\_ = (req_r\,?)\ s_w$ in $f\ ()$ in
      $f\ ()$
```

Here is an example of a client which calculates the successor of the successor of 0. Note that it sends a global value f to the server only once while the server invokes the service twice. The client waits for the result by performing a channel read $c_r\,?$.

```
let $(c_r, c_w)$ = new$_{\langle \mathsf{int} \rangle}$ in
letbox $c'_w$ = box $c_w$ in
letbox $f$ =  box $\lambda s : \langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^w$.
                 let $(ch_r, ch_w)$ = new$_{\langle \mathsf{int} \rangle}$ in
                 let $\_ = s\,!\,(0, ch_w)$ in
                 let $\_ = s\,!\,(ch_r\,?, ch_w)$ in
                 let $\_ = c'_w\,!\,(ch_r\,?)$ in
                 $()$
   in
let $\_ = req_w\,!\,f$ in
$c_r\,?$
```

5.5 Code on demand

Code on demand is the opposite mechanism of remote evaluation in that a client can download code from a code server instead of sending code to a remote server. As an example, we implement a code server which, upon request, returns the global value $V_{service}$ of type $\boxdot(\langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^r \to \mathsf{unit})$ given in Section 5.4. It expects from a client a write channel of type $\langle \boxdot(\langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^r \to \mathsf{unit}) \rangle^w$, to which $V_{service}$ is sent back:

```
let $req_w$ =
  spawn$^{\mathsf{BI}}$
     box $\lambda req_r : \langle \langle \boxdot(\langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^r \to \mathsf{unit}) \rangle^w \rangle^r$.
        let rec $f = \lambda\_ : \mathsf{unit}$. let $\_ = (req_r\,?)\,!\,V_{service}$ in $f\ ()$ in
        $f\ ()$
```

The following client downloads $V_{service}$ and starts the service for its own use:

```
let $(c_r, c_w)$ = new$_{\langle \boxdot(\langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^r \to \mathsf{unit}) \rangle}$ in
let $\_ = req_w\,!\,c_w$ in
let $s_w$ = spawn$^{\mathsf{BI}}$ $(c_r\,?)$ in
let $(ch_r, ch_w)$ = new$_{\langle \mathsf{int} \rangle}$ in
let $\_ = s_w\,!\,(0, ch_w)$ in
$ch_r\,?$
```

5.6 Hot code replacement

The capability to transmit code between threads enables us to implement a server whose code can be replaced at runtime without stopping it. The following term starts a server which accepts a pair of integer n and write channel ch and writes to ch the result of applying a certain function f to n. Initially f is given as an identity function, but we can change the behavior of the server at runtime by performing a channel write $s_w\,!\,(\mathsf{inr}\ f_{new})$ for a certain global value f_{new} of type $\mathsf{int} \to \mathsf{int}$.

```
let $s_w$ =
  spawn$^{\mathsf{BI}}$
    box $\lambda s_r : \langle (\mathsf{int} \times \langle \mathsf{int} \rangle^w) + (\mathsf{int} \to \mathsf{int}) \rangle^r$.
       let rec $loop = \lambda f : \mathsf{int} \to \mathsf{int}$.
          case $s_r\,?$ of  inl $(n, ch) \Rightarrow$  let $\_ = ch\,!\,(f\ n)$ in
                                         $loop\ f$
                        | inr $f_{new} \Rightarrow loop\ f_{new}$
          in
       $loop\ (\lambda x : \mathsf{int}.\, x)$
```

6. Extensions to $\lambda^{\mathsf{PC}}_{\boxdot}$

This section discusses two extensions to $\lambda^{\mathsf{PC}}_{\boxdot}$. As it is routine to incorporate these extensions into the definition of $\lambda^{\mathsf{PC}}_{\boxdot}$, we only sketch the main ideas and omit the details.

6.1 Local threads

Consider two read channels a_1^r and a_2^r (of the same type $\langle A \rangle^r$) belonging to different threads γ_1 and γ_2, respectively. a_1^r wishes to forward every incoming message V to a_2^r with a channel write $a_2^w\,!\,V$ where a_2^w is a write channel corresponding to a_2^r. (As write channels are global values, we assume that a_2^w has already been transmitted to thread γ_1.) An easy solution is to call the following construct forward with arguments a_1^r and a_2^w at thread γ_1:

```
forward : $\langle A \rangle^r \to (\langle A \rangle^w \to \mathsf{unit})_{@\mathsf{G}}$
forward = $\lambda x : \langle A \rangle^r.\, \lambda y : \langle A \rangle^w$.
   let rec $f = \lambda\_ : \mathsf{unit}$. let $\_ = y\,!\,(x\,?)$ in $f\ ()$ in
   $f\ ()$
```

The problem with such a call to forward is that thread γ_1 goes into an infinite loop and degenerates to a trivial thread that only repeats a channel read $x\,?$ followed by a channel write $y\,!\,(x\,?)$. Hence a better solution would be to spawn a separate thread with a call to the following construct forward$'$ with arguments a_1^r and a_2^w:

```
forward' : $\langle A\rangle^r \rightarrow (\langle A\rangle^w \rightarrow \mathsf{unit})_{@\mathsf{G}}$
forward' = $\lambda x : \langle A\rangle^r.\ \lambda y : \langle A\rangle^w.$
    letbox $x'$ = box $x$ in        (* does not typecheck *)
    letbox $y'$ = box $y$ in
    spawnT {  let rec $f$ = $\lambda$_: unit. let _ = $y'$!($x'$?) in $f$ () in
              $f$ ()}
```

Unfortunately forward$'$ fails to typecheck because a read channel of type $\langle A\rangle^r$ cannot be a global value.

Instead of spawning an ordinary thread, therefore, we spawn a *local thread* that starts by evaluating forward a_1^r a_2^w. The underlying assumption is that a thread consists of one or more local threads running concurrently which share *both the store and read channels.* (There arises the memory consistency problem, but it is a separate issue.) Hence both references and read channels can be part of a term for creating a new local thread. In other words, as far as creating local threads is concerned, both references and read channels can be regarded as global values. We do not, however, allow communications of read channels even between local threads, since the syntax for a channel write $a^w\,!\,V$ does not indicate whether the corresponding read channel a^r resides at the same thread (in which case V may be another read channel) or at a remote thread (in which case V may not be another read channel). Thus channel writes still expect global values which do not include read channels. Fournet et al. (1996) make a similar assumption in their study of the distributed join-calculus: every channel has a unique reflexive solution, corresponding to a thread in $\lambda_{\boxdot}^{\mathsf{PC}}$, which may run multiple processes, corresponding to local threads in $\lambda_{\boxdot}^{\mathsf{PC}}$; all these processes can interact with any message addressed to the channel.

We introduce a new construct lthread $\{M\}$ for creating a local thread that starts by evaluating M:

$$\text{term} \quad M \quad ::= \quad \cdots \mid \mathsf{lthread}\ \{M\}$$

Since local threads running at the same thread share references and read channels, lthread $\{M\}$ typechecks whenever M typechecks:

$$\frac{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash M : A_{@\mathsf{L}}}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash \mathsf{lthread}\ \{M\} : \mathsf{unit}_{@\mathsf{L}}}\ \mathsf{LThread}$$

Now that a thread may run multiple local threads, the state of a thread γ is represented by $\{M_1, \cdots, M_n \mid \psi\ @\ \gamma\}$ where each term M_i, $1 \leq i \leq n$, is being evaluated by a local thread. All the previous rules for configuration transitions are extended in a straightforward way by rewriting $\{M \mid \psi\ @\ \gamma\}$ as $\{M, M_1, \cdots, M_n \mid \psi\ @\ \gamma\}$. An exception is the rule *Spawn* which creates a fresh thread consisting only of a single local thread, as shown in Figure 6. A new rule *Sync$'$* accounts for a channel read and a channel write occurring synchronously within the same thread γ. Another new rule *LThread* evaluates lthread $\{M\}$ to creates a fresh local thread.

Now we rewrite forward by exploiting lthread:

```
forward = $\lambda x : \langle A\rangle^r.\ \lambda y : \langle A\rangle^w.$
  lthread {  let rec $f$ = $\lambda$_: unit. let _ = $y$!($x$?) in $f$ () in
             $f$ () }
```

Another application of lthread is to simulate channels not subject to channel locality, *i.e.*, channels that can be read from and written to at any thread. The idea is the same as in implementing shared references in Section 5.3 except that each channel read from $cell_r$ starts a new local thread. Here is an example of creating such a channel for type int. We call *read* for channel reads and *write* for channel writes, both of which are global values and thus can be shared by all threads.

```
let $cell_w$ =
  spawn$^{\mathsf{BI}}$
    box $\lambda cell_r : \langle\langle \mathsf{int}\rangle^w + \mathsf{int}\rangle^r$.
      let $(c_r, c_w)$ = new$_{\langle \mathsf{int}\rangle}$ in
      let rec $f$ = $\lambda$_: unit.
        case $cell_r$? of  inl $ch_w$ ⇒ lthread {$ch_w$!($c_r$?)}
                         | inr $v$ ⇒ lthread {$c_w$!$v$}
        in
      $f$ ()

  letbox $cell'_w$ = box $cell_w$ in
  letbox $read$ = box $\lambda$_: unit.  let $(x_r, x_w)$ = new$_{\langle \mathsf{int}\rangle}$ in
                                  let _ = $cell'_w$!(inl $x_w$) in
                                  $x_r$?
    in
  letbox $write$ = box $\lambda x$: int. $cell'_w$!(inr $x$) in
  ($read$, $write$)
```

6.2 Type-safe data parallelism

The characteristic feature of the type system of $\lambda_{\boxdot}^{\mathsf{PC}}$ is to be able to decide whether a given term evaluates to a global value or a local value. We can exploit this feature to provide type safety for data parallelism in $\lambda_{\boxdot}^{\mathsf{PC}}$, as illustrated below.

Consider a parallel map construct mapP which applies a function f to each element of an array e in parallel:

$$\mathsf{mapP}\ f\ e$$

The parent thread evaluating mapP f e spawns multiple child threads each of which applies f to an element of e. In order to avoid the memory consistency problem, we require that f do not inherit references from the parent thread (because f may attempt to update such references). On the other hand, in order to provide more flexibility in programming, we allow f to allocate new references, which cannot be transmitted between child threads and thus are safe to use.

Checking if f satisfies the above two conditions is simple: we just test whether f is a global value or not. Note that although child threads do not share writable data, they can still share read-only data because variables bound to global values can serve as shared read-only data. That is, if a binding $x : A_{@\mathsf{G}}$ is available at the parent thread, all child threads may read variable x to obtain a global value.

A global value f, however, does not protect against write channels being shared by child threads. An easy solution is to introduce another locality $\mathsf{G}^\star$ with a relation $\mathsf{G}^\star < \mathsf{G}$ and the following typing rule

$$\frac{\Gamma_{\mathsf{G}^\star} \mid \cdot; \cdot; \cdot \vdash V : A_{@\mathsf{L}}}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash V : A_{@\mathsf{G}^\star}}\ \mathsf{GVal}'$$

where

$$\Gamma_{\mathsf{G}^\star} \quad = \quad \{x : A_{@\mathsf{G}^\star} \mid x : A_{@\mathsf{G}^\star} \in \Gamma\}.$$

New constructs and types for $\mathsf{G}^\star$ can be designed in an analogous way to those for G. Then a typing judgment $f : A_{@\mathsf{G}^\star}$ ensures that f contains neither references nor write channels.

7. Related work

7.1 Mutable references in parallel or distributed languages

As there is no definitive standard for shared memory model in parallel or distributed languages, different policies for mutable references or similar constructs have been proposed. X10 (Charles et al. 2005) permits remote mutable variables belonging to remote threads, but accessing the contents of a remote mutable variable results in a runtime exception. Fortress (Allan et al. 2007) permits

$$\frac{\mathit{fresh}\ \gamma'}{\pi, \{\phi[\![\mathsf{spawn\ box}\ V]\!], M_1, \cdots, M_n \mid \psi \mathbin{@} \gamma\} \Rightarrow \pi, \{\phi[\![()]\!], M_1, \cdots, M_n \mid \psi \mathbin{@} \gamma\}, \{V\ () \mid \cdot \mathbin{@} \gamma'\}}\ \mathit{Spawn}$$

$$\frac{}{\pi, \{\phi[\![a^r\,?]\!], \phi'[\![a^w\,!\,V]\!], M_1, \cdots, M_n \mid \psi \mathbin{@} \gamma\} \Rightarrow \pi, \{\phi[\![V]\!], \phi'[\![()]\!], M_1, \cdots, M_n \mid \psi \mathbin{@} \gamma\}}\ \mathit{Sync}'$$

$$\frac{}{\pi, \{\phi[\![\mathsf{lthread}\ \{M\}]\!], M_1, \cdots, M_n \mid \psi \mathbin{@} \gamma\} \Rightarrow \pi, \{\phi[\![()]\!], M, M_1, \cdots, M_n \mid \psi \mathbin{@} \gamma\}}\ \mathit{LThread}$$

Figure 6. Configuration transition rules for local threads

shared objects accessible to every thread, but at the cost of implicitly maintaining the sharedness (either *local* or *shared*) of every object. Titanium (Hilfinger et al. 2005), which extends Java, takes a different approach by allowing shared memory, but also distinguishing between local references and global references at the type level. UPC (El-Ghazawi et al. 2003), which extends C, takes a similar approach by using two kinds of pointers: private pointers and global pointers. Facile (Knabe 1995) and JoCAML (Fournet et al. 2003) dispense with remote references by sending copies of heap cells whenever their references are transmitted to remote threads. Erlang (Armstrong 1997) and Manticore (Fluet et al. 2007) do not use mutable references at all.

$\lambda^{\mathsf{PC}}_{\boxdot}$ is similar to Facile and JoCAML in that it permits mutable references but assumes no shared memory (and also in that it is a dialect of ML). The main difference is that Facile and JoCAML rely on the runtime system to avoid remote references whereas $\lambda^{\mathsf{PC}}_{\boxdot}$ relies on the type system to forestall remote references.

7.2 Channel locality

The issue of channel locality has been studied mainly for the pi-calculus and its relatives. Amadio (1997) develops a type system for a fragment of the pi-calculus in which typing judgments use channels linearly to ensure channel locality. The distributed join-calculus (Fournet et al. 1996) assumes a syntactic restriction to enforce channel locality. Specifically it considers only syntactically well-formed DRCHAMs (distributed reflexive chemical machines) in which every channel is defined in at most one RCHAM (reflexive chemical machine), where a DRCHAM corresponds to a configuration and a RCHAM to a thread in $\lambda^{\mathsf{PC}}_{\boxdot}$. Schmitt and Stefani (2003) present a higher-order version of the distributed join calculus which uses a polymorphic type system to guarantee that the destination for each message is uniquely determined. It achieves channel locality in a slightly different sense than in $\lambda^{\mathsf{PC}}_{\boxdot}$ because, for example, a message may be intercepted before reaching its destination. The dynamic join-calculus (Schmitt 2002) achieves channel locality in a similar vein, in which the destination for a message is determined according to its current position.

Yoshida and Hennessy (1999) present a calculus similar to $\lambda^{\mathsf{PC}}_{\boxdot}$ in that it uses a *type system* to enforce channel locality. It combines the call-by-value lambda calculus and a higher-order extension of the pi-calculus, and allows processes themselves to be transmitted via channels. The type system introduces *sendable types* whose values can be transmitted between processes without destroying channel locality, and exploits a subtyping relation to enforce channel locality. The type system, however, is not a general solution applicable to our setting for two reasons. First there is a semantic restrictions on function types which severely limits the generality of the calculus: for a function type $\tau \rightarrow \sigma$, if σ is sendable, τ must also be sendable, in which case $\tau \rightarrow \sigma$ is also automatically regarded as sendable. In our setting, such a restriction means, for example, that no term of type $\mathsf{ref\ int} \rightarrow \mathsf{int}$ or $\langle\mathsf{int}\rangle^r \rightarrow \mathsf{int}$ is even allowed and that every function of type $\mathsf{int} \rightarrow \mathsf{int}$ must be global, neither of which is the case in $\lambda^{\mathsf{PC}}_{\boxdot}$. Second the type system ignores the call-by-value semantics of the underlying lambda calculus. Specifically it includes a typing rule (TERM$_l$) assigning a sendable type to a term whenever it typechecks under a typing context using sendable types only. In our setting, the typing rule would correspond to the following rule which fails to comply with the call-by-value semantics:

$$\frac{\Gamma_{\mathsf{G}} \mid \Phi^w; \cdot; \cdot \vdash M : A_{@\mathsf{L}}}{\Gamma \mid \Phi^w; \Phi^r; \Psi \vdash M : A_{@\mathsf{G}}}\ (\mathit{wrong})$$

In comparison, $\lambda^{\mathsf{PC}}_{\boxdot}$ imposes no syntactic or semantic restriction on function types and properly accounts for the call-by-value semantics in the presence of mutable references.

7.3 Splitting channels

The idea of using two kinds of channels, namely read channels and write channels (or input channels and output channels), is not new. For example, the type system of (Pierce and Sangiorgi 1993) can effectively distinguish between read channels and write channels because every channel is assigned a tag indicating its usage (read, write, or both). The polarized pi-calculus of (Odersky 1995) syntactically distinguishes between read channels and write channels. In the polarized pi-calculus, using read channels for channel writes or write channels for channel reads results in no reaction with other processes. The polar pi-calculus of (Zhang and Potter 2002) also syntactically distinguishes between read channels and write channels with an additional requirement that only write channels can be transmitted via channels.

$\lambda^{\mathsf{PC}}_{\boxdot}$ is, however, different from previous work in that it assigns different types to different kinds of channels and *exploits these types,* instead of the syntax, to enforce channel locality. In fact, such notions as local values, global values, and primitive types, all established in the base language $\lambda_{\boxdot}$, have naturally led to the main idea for achieving channel locality in $\lambda^{\mathsf{PC}}_{\boxdot}$. Thus our decision to distinguish between read channels and write channels has a different motivation from previous work.

7.4 Modal types for parallel and distributed computations

There are a few distributed languages (Murphy et al. 2004; Jia and Walker 2004) whose type systems are based on the spatial interpretation of modal logic. Murphy et al. (2004) use the necessity modality $\Box$ in modal types for global terms which can be evaluated at any node in the network, and the possibility modality $\Diamond$ in modal types for references to local resources belonging to certain nodes. Jia and Walker (2004) use $\Box$ for the same purpose, but $\Diamond$ in modal types for terms that can be evaluated at certain nodes.

Our earlier work (Park 2006) is the first departure from the typical spatial interpretation of modal logic in that it uses a new modality $\boxdot$ to focus on global values rather than global terms. (See (Park 2006) for a discussion on logical properties of the modality $\boxdot$. For example, $\boxdot A \rightarrow A$ and $\boxdot A \rightarrow \boxdot\boxdot A$ are both inhabited, but $\boxdot(A \rightarrow B) \rightarrow \boxdot A \rightarrow \boxdot B$ is uninhabited if we ignore non-terminating terms.) Our present work adds communication constructs for higher-order channels, thereby providing improved flexibility in programming for parallel and distributed computations.

8. Conclusion and future work

We present an ML-like parallel language $\lambda_{\boxdot}^{\mathsf{PC}}$ which features type-safe higher-order channels with channel locality. Sequential programming in $\lambda_{\boxdot}^{\mathsf{PC}}$ may exploit mutable references as usual, since the type system ensures that mutable references never interfere with parallel computations. Thus implementing the parallel operational semantics of $\lambda_{\boxdot}^{\mathsf{PC}}$ is no more complicated than implementing a similar parallel language without mutable references.

Our long-term goal is to build a programming system that supports three levels of parallelism within a unified framework. At the highest level, it implements distributed computations taking place in a network of nodes. Each node performs a stand-alone computation and also communicates with other nodes via higher-order channels. (Hence there is no distributed shared memory.) The next level implements task parallelism with higher-order channels as in $\lambda_{\boxdot}^{\mathsf{PC}}$. Thus a stand-alone computation at a node actually consists of multiple threads running in parallel. At the lowest level, multiple local threads with shared memory run within each thread. Task parallelism with shared memory as well as data parallelism can be implemented at this level. Then the type system of $\lambda_{\boxdot}^{\mathsf{PC}}$ can be extended to provide type safety at all the three levels.

Acknowledgments

The author is grateful to Hyeonseung Im and anonymous reviewers for their helpful comments.

References

Eric Allan, David Chase, Joe Hallet, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0beta. Technical report, Sun Microsystems Inc., March 2007.

Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings of the 2nd International Conference on Coordination Languages and Models, LNCS 1282*, pages 374–391. Springer-Verlag, 1997.

Joe Armstrong. The development of Erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 196–203. ACM Press, 1997.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

Guy Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

Manuel Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, pages 11–18, 2007.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

Cray Inc. The Chapel language specification version 0.4. Technical report, February 2005.

Tarek El-Ghazawi, William Carlson, and Jesse Draper. UPC language specifications, v1.1.1, October 2003.

Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, pages 25–32, 2007.

Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR: 7th International Conference on Concurrency Theory, LNCS 1119*, pages 406–421. Springer, 1996.

Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: a language for concurrent distributed and mobile programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School, 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer-Verlag, 2003.

Robert Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

Paul Hilfinger, Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Benjamin Liblit, Geoffrey Pike, Jimmy Su, and Katherine Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, EECS Department, University of California, Berkeley, November 2005.

Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey Hollingsworth, and Marvin Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 35. IEEE Computer Society, 2005.

Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). In *Proceedings of the European Symposium on Programming, LNCS 2986*, pages 219–233. Springer, 2004.

Frederick Knabe. *Language Support for Mobile Agents*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1995.

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science*. IEEE Press, 2004.

R. Nikhil and Arvind. *Implicit parallel programming in pH*. Morgan Kaufmann, 2001.

Martin Odersky. Polarized name passing. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS 1026*, pages 324–337. Springer-Verlag, 1995.

Sungwoo Park. A modal language for the safety of mobile values. In Naoki Kobayashi, editor, *Proceedings of the 4th Asian Symposium on Programming Languages and Systems, LNCS 4279*, pages 217–233. Springer, 2006.

Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 376–385, 1993.

Alan Schmitt. Safe dynamic binding in the join calculus. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science*, pages 563–575. Kluwer, B.V., 2002.

Alan Schmitt and Jean-Bernard Stefani. The M-calculus: a higher-order distributed process calculus. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 50–61. ACM Press, 2003.

Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth ACM symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

James Stamos and David Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4): 537–564, 1990.

Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

Nobuko Yoshida and Matthew Hennessy. Subtyping and locality in distributed higher order processes (extended abstract). In *CONCUR: 10th International Conference on Concurrency Theory, LNCS 1664*, pages 557–572. Springer-Verlag, 1999.

Xiaogang Zhang and John Potter. Responsive bisimulation. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science*, pages 601–612. Kluwer, B.V., 2002.