JFP **19** (1): 107–142, 2009.
doi:10.1017/S0956796808006989 First published online 9 September 2008 Printed in the United Kingdom

Type-safe higher-order channels with channel locality[1]

SUNGWOO PARK and HYEONSEUNG IM
Department of Computer Science and Engineering,
Pohang University of Science and Technology, Republic of Korea
(*e-mail:* {gla,genilhs}@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references, however, they pose a safety problem because references may be transmitted to remote threads where they are no longer valid.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.

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

[1] This is an extended version of the paper that appeared in The 12th ACM SIGPLAN International Conference on Functional Programming (Park, 2007).

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 & 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 logically 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 via higher-order channels only. 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 logically 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 Manticore (Fluet *et al*., 2007), by designing the runtime system so as to create copies of heap cells whenever their references are transmitted, as in Facile (Knabe, 1995) and JoCAML (Fournet *et al*., 2003), or by raising runtime exceptions when references are transmitted, as in Alice (Rossberg *et al*., 2005).

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

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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 $\mathsf{int} \to \mathsf{int}$ 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. Therefore, in order to guarantee type safety of channels, 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*.

The present work is based on the modal type system in our previous work (Park, 2006) which uses two typing judgments and a new modality $\boxdot$ 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 L and G 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 $\boxdot A$. In order to explicitly bind variables to global values, we use another construct $\mathsf{letbox}\ x = M\ \mathsf{in}\ N$ such that $\mathsf{letbox}\ x = \mathsf{box}\ M'\ \mathsf{in}\ N$ evaluates N after binding variable x to the global value to which M' evaluates to.

As our modal type system enables us to express that the result of evaluating a term is a global value and contains no references, it is straightforward to guarantee type safety of channels: we simply require that a channel write $a\,!\,M$ typecheck 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 & Hennessy, 1999; Schmitt & 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

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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 is allowed only at the thread where a^r is created. However, channel writes to a^w is 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 a 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 & Potter, 2002).

1.3 Contributions

We develop an ML-like parallel language $\lambda^{\mathsf{PC}}_{\Box}$ 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.

Although, $\lambda^{\mathsf{PC}}_{\Box}$ 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, $\lambda^{\mathsf{PC}}_{\Box}$ 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 from its base language (a subset of Standard ML) in order to simplify its implementation. Parallel dialects of Haskell, such as pH (Nikhil & Arvind, 2001) and Data Parallel Haskell (Chakravarty *et al.*, 2007), also benefit from lack of mutable references. In comparison, $\lambda^{\mathsf{PC}}_{\Box}$ permits mutable references in sequential computations, but its type system ensures that mutable references never interfere with parallel computations. Thus the type system of $\lambda^{\mathsf{PC}}_{\Box}$ wins us both flexibility in sequential programming and ease of implementing parallel computations.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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	::=	$\cdot \mid \Psi, l \mapsto A$

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

1.4 Organization of the paper

Section 2 presents the base language $\lambda_{\boxdot}$ for our work. Section 3 discusses the logical meaning of the modality $\boxdot$. 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 4 presents the resultant language $\lambda^{\mathsf{P}}_{\boxdot}$ which comes with a new construct $\mathsf{spawn}\ \{M\}$ for creating new threads. Section 5 extends $\lambda^{\mathsf{P}}_{\boxdot}$ with new constructs for higher-order channels to obtain $\lambda^{\mathsf{PC}}_{\boxdot}$. Section 6 illustrates how to implement various communication constructs in $\lambda^{\mathsf{PC}}_{\boxdot}$ such as futures, bidirectional communications, shared references, remote evaluation, code on demand, and hot code replacement. Section 7 discusses two extensions of $\lambda^{\mathsf{PC}}_{\boxdot}$ (including type safety for data parallelism). Section 8 discusses related work, and Section 9 concludes. Most proofs are given in Appendix.

2 Base language $\lambda_{\boxdot}$

This section presents the base language $\lambda_{\boxdot}$ which is essentially a reformulation of the call-by-value language with modal types $\boxdot A$ defined in our previous work (Park, 2006).

2.1 Definition of $\lambda_{\boxdot}$

Figure 1 shows the abstract syntax for $\lambda_{\boxdot}$ which is based on the simply typed lambda-calculus with product types $A \times A$, sum types $A+A$, reference types $\mathsf{ref}\ A$, and the fixed point construct $\mathsf{fix}\ x{:}A.M$. Primitive types P are a subset of ordinary types A whose values are all inherently global. (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"

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

$$\frac{x : A_{@L} \in \Gamma \quad L \leq L'}{\Gamma \mid \Psi \vdash x : A_{@L'}}\ \mathsf{Var}$$

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

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

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

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

$$\frac{\Gamma \mid \Psi \vdash M : A_{@L}}{\Gamma \mid \Psi \vdash \mathsf{inl}\, M : A + B_{@L}}\ {+}\mathsf{I_L} \qquad \frac{\Gamma \mid \Psi \vdash M : B_{@L}}{\Gamma \mid \Psi \vdash \mathsf{inr}\, M : A + B_{@L}}\ {+}\mathsf{I_R}$$

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

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

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

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

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 $L \leqslant L'$ to mean either $L = L'$ or $L < L'$. The rules $\to$I through Loc are all derived from typing rules in the simply typed lambda-calculus by annotating each typing judgment with "here only" locality L. In order to relate terms in these rules with "everywhere" locality G, we need a separate application of the rule GVal or Prim (to be explained later) which connects the two localities L and G. In contrast, the rules $\times$I, $\times\mathsf{E_L}$, and $\times\mathsf{E_R}$ for product types use the same unspecified locality L in their premise and conclusion, which makes sense only because we assume the call-by-value semantics (where (M, N) is a value only when both M and N are also values). For example, if both M and N evaluate to global values V and V', respectively, we may associate (M, N) with locality G because (M, N) indeed evaluates to a global value (V, V'). If we assume the call-by-name semantics and consider (M, N) as a value for any term M and N, however, all these rules become invalid when $L = \mathsf{G}$. For example, if both $!l_1$ and $!l_2$ evaluate to global values such as $()$, the rule $\times$I incorrectly deduces that $(!l_1, !l_2)$ (which contains two locations and thus is not a global value) is a global

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

value. Similarly the call-by-value semantics allows the rules $+\mathsf{I}_\mathsf{L}$, $+\mathsf{I}_\mathsf{R}$, and $+\mathsf{E}$ for sum types to use the same unspecified locality L in their premise and conclusion.

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 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:

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

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 global 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 global. 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 global 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 $\lambda_\boxdot$:

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

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.

Proof
By induction on the structure of $\Gamma \mid \Psi \vdash M : A_{@\mathsf{G}}$. □

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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

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

$$\frac{M \to_\beta M'}{\phi[\![M]\!] \mid \psi \to \phi[\![M']\!] \mid \psi}\ R_\beta \qquad \frac{\mathit{fresh}\ l \notin \mathit{dom}(\psi)}{\phi[\![\mathsf{ref}\ V]\!] \mid \psi \to \phi[\![l]\!] \mid \psi, l \mapsto V}\ \mathit{Ref}$$

$$\frac{\psi(l) = V}{\phi[\![!l]\!] \mid \psi \to \phi[\![V]\!] \mid \psi}\ \mathit{Drf} \qquad \frac{}{\phi[\![l := V]\!] \mid \psi \to \phi[\![()]\!] \mid [l \mapsto V]\psi}\ \mathit{Asn}$$

Fig. 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 $\boxdot$E.

2.2 Examples

We illustrate the use of the modality $\boxdot$ with the two examples given in Section 1.1. We assume a primitive type int, a typing rule Int for integers, and an infix operator $+$ for adding two integers.

The first term

$$M_1 = (\lambda v : \mathsf{int}.\ \lambda x : \mathsf{int}.\ x + v)\ 0$$

has type $\mathsf{int} \to \mathsf{int}$ and indeed evaluates to a global value $\lambda x : \mathsf{int}.\ x + 0$, but cannot be used to build a term of type $\boxdot(\mathsf{int} \to \mathsf{int})$:

$$\frac{\dfrac{(\mathit{no\ typing\ rule\ applicable})}{\Gamma \mid \Psi \vdash (\lambda v : \mathsf{int}.\ \lambda x : \mathsf{int}.\ x + v)\ 0 : \mathsf{int} \to \mathsf{int}_{@\mathsf{G}}}}{\Gamma \mid \Psi \vdash \mathsf{box}\ (\lambda v : \mathsf{int}.\ \lambda x : \mathsf{int}.\ x + v)\ 0 : \boxdot(\mathsf{int} \to \mathsf{int})_{@\mathsf{L}}}\ \boxdot\mathsf{I}$$

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

The reason why box M_1 fails to have type $\boxdot(\mathsf{int} \to \mathsf{int})$ is that the type of $\lambda v : \mathsf{int}.\, \lambda x : \mathsf{int}.\, x + v$, namely $\mathsf{int} \to (\mathsf{int} \to \mathsf{int})$, does not express that a global value of type $\mathsf{int} \to \mathsf{int}$ is returned. That is, the type system is unaware that the inner λ-abstraction $\lambda x : \mathsf{int}.\, x + v$ can be a global value, since a binding $v : \mathsf{int}_{@\mathsf{G}}$ is not added to the typing context. In order to build an operationally equivalent term of type $\boxdot(\mathsf{int} \to \mathsf{int})$, we need to explicitly specify that 0 is a global value as follows:

$$M_1' = \mathsf{letbox}\ v = \mathsf{box}\ 0\ \mathsf{in}\ \lambda x : \mathsf{int}.\, x + v$$

Note that typechecking box M_1' adds a binding $v : \mathsf{int}_{@\mathsf{G}}$ to the typing context:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Psi \vdash 0 : \mathsf{int}_{@\mathsf{L}}}\,\mathsf{Int}}{\Gamma \mid \Psi \vdash 0 : \mathsf{int}_{@\mathsf{G}}}\,\mathsf{Prim}}{\Gamma \mid \Psi \vdash \mathsf{box}\ 0 : \boxdot\mathsf{int}_{@\mathsf{L}}}\,\boxdot\mathsf{I} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma_{\mathsf{G}}, v : \mathsf{int}_{@\mathsf{G}}, x : \mathsf{int}_{@\mathsf{L}} \mid \cdot \vdash x + v : \mathsf{int}_{@\mathsf{L}}}}{\Gamma_{\mathsf{G}}, v : \mathsf{int}_{@\mathsf{G}} \mid \cdot \vdash \lambda x : \mathsf{int}.\, x + v : \mathsf{int} \to \mathsf{int}_{@\mathsf{L}}}\,{\to}\mathsf{I}}{\Gamma, v : \mathsf{int}_{@\mathsf{G}} \mid \Psi \vdash \lambda x : \mathsf{int}.\, x + v : \mathsf{int} \to \mathsf{int}_{@\mathsf{G}}}\,\mathsf{GVal}}{\Gamma \mid \Psi \vdash \mathsf{letbox}\ v = \mathsf{box}\ 0\ \mathsf{in}\ \lambda x : \mathsf{int}.\, x + v : \mathsf{int} \to \mathsf{int}_{@\mathsf{G}}}\,\boxdot\mathsf{E}$$
$$\dfrac{}{\Gamma \mid \Psi \vdash \mathsf{box}\ \mathsf{letbox}\ v = \mathsf{box}\ 0\ \mathsf{in}\ \lambda x : \mathsf{int}.\, x + v : \boxdot(\mathsf{int} \to \mathsf{int})_{@\mathsf{L}}}\,\boxdot\mathsf{I}$$

The second term

$$M_2 = (\lambda r : \mathsf{ref}\ \mathsf{int}.\, \lambda x : \mathsf{int}.\, x + !r)\ (\mathsf{ref}\ 0)$$

has type $\mathsf{int} \to \mathsf{int}$ and does not evaluate to a global value. The type system also prevents M_2 from being used to build a term of type $\boxdot(\mathsf{int} \to \mathsf{int})$ because there is no typing derivation of $\Gamma \mid \Psi \vdash M_2 : \mathsf{int} \to \mathsf{int}_{@\mathsf{G}}$:

$$\dfrac{\dfrac{(\textit{no typing rule applicable})}{\Gamma \mid \Psi \vdash (\lambda r : \mathsf{ref}\ \mathsf{int}.\, \lambda x : \mathsf{int}.\, x + !r)\ (\mathsf{ref}\ 0) : \mathsf{int} \to \mathsf{int}_{@\mathsf{G}}}}{\Gamma \mid \Psi \vdash \mathsf{box}\ (\lambda r : \mathsf{ref}\ \mathsf{int}.\, \lambda x : \mathsf{int}.\, x + !r)\ (\mathsf{ref}\ 0) : \boxdot(\mathsf{int} \to \mathsf{int})_{@\mathsf{L}}}\,\boxdot\mathsf{I}$$

Rewriting M_2 as $\mathsf{letbox}\ r = \mathsf{box}\ \mathsf{ref}\ 0\ \mathsf{in}\ \lambda x : \mathsf{int}.\, x + !r$ does not help because ref 0 does not return a global value.

Note that although M_2 does not evaluate to a global value, M_2 itself is closed and contains no free references. Hence, for example, $\lambda \_ : \mathsf{unit}.\, M_2$ is a global value (where $\_$ denotes a fresh variable) and box $\lambda \_ : \mathsf{unit}.\, M_2$ has type $\boxdot(\mathsf{unit} \to (\mathsf{int} \to \mathsf{int}))$:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_{\mathsf{G}} \mid \cdot \vdash \lambda \_ : \mathsf{unit}.\, (\lambda r : \mathsf{ref}\ \mathsf{int}.\, \lambda x : \mathsf{int}.\, x + !r)\ (\mathsf{ref}\ 0) : \mathsf{unit} \to (\mathsf{int} \to \mathsf{int})_{@\mathsf{L}}}}{\Gamma \mid \Psi \vdash \lambda \_ : \mathsf{unit}.\, (\lambda r : \mathsf{ref}\ \mathsf{int}.\, \lambda x : \mathsf{int}.\, x + !r)\ (\mathsf{ref}\ 0) : \mathsf{unit} \to (\mathsf{int} \to \mathsf{int})_{@\mathsf{G}}}\,\mathsf{GVal}}{\Gamma \mid \Psi \vdash \mathsf{box}\ \lambda \_ : \mathsf{unit}.\, (\lambda r : \mathsf{ref}\ \mathsf{int}.\, \lambda x : \mathsf{int}.\, x + !r)\ (\mathsf{ref}\ 0) : \boxdot(\mathsf{unit} \to (\mathsf{int} \to \mathsf{int}))_{@\mathsf{L}}}\,\boxdot\mathsf{I}$$

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

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

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

mapped under Ψ; $dom(\Psi)$ denotes the set of locations mapped under Ψ.

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

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 $M \mid \psi \rightarrow M' \mid \psi'$.

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 $\Gamma \mid \Psi \vdash [V/x]M : C_{@L}$.

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, $\times\mathsf{I}$, $+\mathsf{I}_{\mathsf{L}}$, and $+\mathsf{I}_{\mathsf{R}}$. 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 $\cdot \mid \Psi \vdash M : A_{@L}$, $\Psi \vdash \psi$ okay, and $M \mid \psi \rightarrow M' \mid \psi'$. Then there exists a store typing Ψ' such that $\cdot \mid \Psi' \vdash M' : A_{@L}$, $\Psi \subset \Psi'$, and $\Psi' \vdash \psi'$ okay.

3 Logic for $\lambda_{\Box}$

The type system for modal types $\Box A$ is unusual in that it differentiates values (i.e., terms in weak head normal form) from ordinary terms, as shown in the rule GVal. This differentiation implies that the logic corresponding to the modality $\Box$ via the Curry–Howard isomorphism requires a judgment that inspects not only hypotheses in a proof but also the proof structure itself (e.g., inference rules used in the proof). Thus, the modality $\Box$ sets itself apart from other modalities and is not found in any other logic. Later we will compare the modality $\Box$ with other modalities in modal logic.

In the pure fragment of $\lambda_{\Box}$ without primitive types and reference types, the modality $\Box$ shows similarities with modal possibility $\Diamond$ and lax modality $\bigcirc$ in Pfenning and Davies (2001). Specifically a proof-theoretic analysis of $\Box$ gives rise to a new form of substitution $\langle M/x\rangle N$ which is defined inductively on the structure

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

of the term being substituted (namely M) instead of the term being substituted into (namely N). Let us interpret a β-reduction as the reduction of a typing derivation in which an introduction rule is followed by a corresponding elimination rule. For example, the β-reduction for the connective $\rightarrow$ may be seen as the reduction of the following typing derivation in the pure fragment of $\lambda_{\boxdot}$ where we omit following store typings:

$$\dfrac{\dfrac{\Gamma, x : A_{@\mathsf{L}} \vdash M : B_{@\mathsf{L}}}{\Gamma \vdash \lambda x{:}A.\, M : A \rightarrow B_{@\mathsf{L}}}\ {\rightarrow}\mathsf{I} \qquad \Gamma \vdash N : A_{@\mathsf{L}}}{\Gamma \vdash (\lambda x{:}A.\, M)\ N : B_{@\mathsf{L}}}\ {\rightarrow}\mathsf{E} \qquad \rightarrow_\beta \qquad \Gamma \vdash [N/x]M : B_{@\mathsf{L}}$$

Likewise we obtain a β-reduction for $\boxdot$ from the reduction of a typing derivation in which the introduction rule $\boxdot\mathsf{I}$ is followed by the elimination rule $\boxdot\mathsf{E}$:

$$\dfrac{\dfrac{\Gamma \vdash M : A_{@\mathsf{G}}}{\Gamma \vdash \mathsf{box}\ M : \boxdot A_{@\mathsf{L}}}\ \boxdot\mathsf{I} \qquad \Gamma, x : A_{@\mathsf{G}} \vdash N : C_{@L}}{\Gamma \vdash \mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N : C_{@L}}\ \boxdot\mathsf{E} \qquad \rightarrow_\beta \qquad \Gamma \vdash \langle M/x\rangle N : C_{@L}$$

To see why $\langle M/x\rangle N$ should be defined inductively on the structure of M, observe that the reduction of $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$ requires an analysis of M instead of N. The reason is that only a value can be substituted for x, but M may not be a value yet; therefore we have to analyze M to decide how to transform the whole term so that x is eventually replaced by a value. Conceptually N should be replicated at those subterms within M that can be taken as the result of evaluating M, so that M and N are evaluated exactly once and in that order. If M is already a value V, we reduce the whole term to $[V/x]N$. Thus we are led to define $\langle M/x\rangle N$ as follows:

$$\langle V/x\rangle N = [V/x]N$$
$$\langle \mathsf{letbox}\ x' = M'\ \mathsf{in}\ M''/x\rangle N = \mathsf{letbox}\ x' = M'\ \mathsf{in}\ \langle M''/x\rangle N$$

Note that we cannot define $\langle M_1\ M_2/x\rangle N$ because in the absence of primitive types and the rule Prim, there is no typing derivation of $\Gamma \vdash M_1\ M_2 : A_{@\mathsf{G}}$ and thus $\mathsf{box}\ M_1\ M_2$ cannot be well typed.

In the presence of primitive types, the β-reduction

$$\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N \quad \rightarrow_\beta \quad \langle M/x\rangle N$$

is no longer valid because $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$ may typecheck while $\langle M/x\rangle N$ is undefined. For example, $M = M_1\ M_2$ of type unit satisfies $\Gamma \vdash M : \mathsf{unit}_{@\mathsf{G}}$ by the rule Prim, but $\langle M_1\ M_2/x\rangle N$ is undefined. Intuitively the rule Prim disguises an unanalyzable term of a primitive type as an analyzable term. Thus, in order to reduce $\mathsf{letbox}\ x = \mathsf{box}\ M\ \mathsf{in}\ N$, the operational semantics of $\lambda_{\boxdot}$ is forced to reduce M into a value V first, instead of analyzing M to transform the whole term. Then an ordinary substitution $[V/x]N$ suffices for the reduction of $\mathsf{letbox}\ x = \mathsf{box}\ V\ \mathsf{in}\ N$.

We close this section with a brief discussion of the properties of the modality $\boxdot$. For a given type in $\lambda_{\boxdot}$, we give a term of that type if it is inhabited. We ignore the fixed point construct, in the presence of which all types are inhabited.

- $A \not\rightarrow \boxdot A$
  An ordinary term does not necessarily evaluate to a global value. This implies

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

that $\boxdot$ is different from modal possibility $\diamond$ and lax modality $\bigcirc$ because both $A \to \diamond\, A$ and $A \to \bigcirc A$ hold.

- $\boxdot A \to A$
  $\lambda x{:}\boxdot A.\, \mathsf{letbox}\ y = x\ \mathsf{in}\ y$
  A global value is a special case of an ordinary term. A special case $\boxdot\boxdot A \to \boxdot A$ means that $\boxdot$ is an idempotent modality.
- $\boxdot A \to \boxdot\boxdot A$
  $\lambda x{:}\boxdot A.\, \mathsf{letbox}\ y = x\ \mathsf{in}\ \mathsf{box}\ \mathsf{box}\ y$
  A global value itself is global.
- $\boxdot(A \to B) \not\to (\boxdot A \to \boxdot B)$
  A global λ-abstraction does not necessarily return a global value. This implies that $\boxdot$ is different from modal necessity $\Box$ because $\Box(A \to B) \to (\Box A \to \Box B)$ holds (Pfenning & Davies, 2001).
- $(\boxdot A \to \boxdot B) \not\to \boxdot(A \to B)$
  A λ-abstraction that accepts and returns global values is not necessarily global.
- $\boxdot(A \times B) \to (\boxdot A \times \boxdot B)$
  $\lambda x{:}\boxdot(A \times B).\, \mathsf{letbox}\ y = x\ \mathsf{in}\ (\mathsf{box}\ \mathsf{fst}\ y, \mathsf{box}\ \mathsf{snd}\ y)$
  This type is inhabited because the typing rules $\times\mathsf{E}_\mathsf{L}$ and $\times\mathsf{E}_\mathsf{R}$ use an unspecified locality L instead of a specific locality L or G.
- $(\boxdot A \times \boxdot B) \to \boxdot(A \times B)$
  $\lambda x{:}\boxdot A \times \boxdot B.\, \mathsf{letbox}\ y_1 = \mathsf{fst}\ x\ \mathsf{in}\ \mathsf{letbox}\ y_2 = \mathsf{snd}\ x\ \mathsf{in}\ \mathsf{box}\ (y_1, y_2)$
  This type is inhabited because the typing rule $\times\mathsf{I}$ uses an unspecified locality L.
- $\boxdot(A+B) \to (\boxdot A+\boxdot B)$
  $\lambda x{:}\boxdot(A+B).\, \mathsf{letbox}\ y = x\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inl}\ z_1 \Rightarrow \mathsf{inl}\ \mathsf{box}\ z_1 \mid \mathsf{inr}\ z_2 \Rightarrow \mathsf{inr}\ \mathsf{box}\ z_2$
  This type is inhabited because the typing rule $+\mathsf{E}$ uses an unspecified locality L.
- $(\boxdot A+\boxdot B) \to \boxdot(A+B)$
  $\lambda x{:}\boxdot A+\boxdot B.$
  $\mathsf{box}\ (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ y_1 \Rightarrow \mathsf{letbox}\ z_1 = y_1\ \mathsf{in}\ \mathsf{inl}\ z_1 \mid \mathsf{inr}\ y_2 \Rightarrow \mathsf{letbox}\ z_2 = y_2\ \mathsf{in}\ \mathsf{inr}\ z_2)$ This type is inhabited because the typing rules $+\mathsf{I}_\mathsf{L}$ and $+\mathsf{I}_\mathsf{R}$ use an unspecified locality L.

4 $\lambda^{\mathsf{P}}_{\boxdot}$ 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 of $\lambda^{\mathsf{P}}_{\boxdot}$. At the syntax level, $\lambda^{\mathsf{P}}_{\boxdot}$ augments $\lambda_{\boxdot}$ with a new construct $\mathsf{spawn}\ \{M\}$. The typing rule Spawn requires M in $\mathsf{spawn}\ \{M\}$ to

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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

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

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

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

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

Fig. 4. Definition of $\lambda^{\mathsf{P}}_{\Box}$.

contain no dangling references so that it is safe to evaluate M at a fresh thread with an empty store, as shown in the rule *Spawn*. (It does not matter whether M evaluates to a global value or not.)

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 $\{M \mid \psi \mathbin{@} \gamma\}$. A *configuration typing* Π records the type of the term being evaluated at each thread. The type system of $\lambda^{\mathsf{P}}_{\Box}$ uses the same typing judgment as $\lambda_{\Box}$, 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. For each thread γ such that $\{M \mid \psi \mathbin{@} \gamma\} \in \pi$, it checks the existence of a store typing Ψ_γ whose domain includes 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 nondeterministic because a configuration may choose an arbitrary thread to which either rule is applied.

Type safety of $\lambda^{\mathsf{P}}_{\Box}$ consists of *configuration progress* (Theorem 4.1) and *configuration typing preservation* (Theorem 4.2).

Theorem 4.1 (*Configuration progress*)
Suppose $\Pi \vdash \pi\ \mathsf{okay}$. Then either
(1) π consists only of $\{V \mid \psi \mathbin{@} \gamma\}$, or
(2) there exists π' such that $\pi \Rightarrow \pi'$.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

Theorem 4.2 (*Configuration typing preservation*)
Suppose $\Pi \vdash \pi$ okay and $\pi \Rightarrow \pi'$. Then there exists a configuration typing Π' such that $\Pi \subset \Pi'$ and $\Pi' \vdash \pi'$ okay.

5 $\lambda^{\mathsf{PC}}_{\Box}$ with higher-order channels

This section extends $\lambda^{\mathsf{P}}_{\Box}$ with higher-order channels to obtain $\lambda^{\mathsf{PC}}_{\Box}$. In order to achieve channel locality, $\lambda^{\mathsf{PC}}_{\Box}$ 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 from 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 $\lambda^{\mathsf{PC}}_{\Box}$. 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 $\lambda^{\mathsf{PC}}_{\Box}$ uses a read channel typing Φ^r and a write channel typing Φ^w in each typing judgment:

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

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 5.1
P is a primitive type if and only if $\Gamma \mid \Phi^w ; \Phi^r ; \Psi \vdash V : P_{@\mathsf{L}}$ implies $\Gamma_{\mathsf{G}} | \Phi^w ; \cdot ; \cdot \vdash V : P_{@\mathsf{L}}$.

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 $\Phi^w ; \Phi^r ; \Psi \vdash \psi$ 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 $\lambda^{\mathsf{PC}}_{\Box}$ because it verifies channel locality in a given configuration. Note that an extended configuration typing judgment

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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

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

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

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

$$\frac{\Gamma \mid \Phi^w;\Phi^r;\Psi \vdash M : \langle A\rangle^w{}_{@\mathsf{L}} \quad \Gamma \mid \Phi^w;\Phi^r;\Psi \vdash N : A_{@\mathsf{G}}}{\Gamma \mid \Phi^w;\Phi^r;\Psi \vdash M\,!N : \mathsf{unit}_{@\mathsf{L}}}\ \mathsf{W!}$$

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

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

$$\frac{fresh\ (a^r, a^w)}{\pi, \{\phi[\![\mathsf{new}_{\langle A\rangle}]\!] \mid \psi\ @\ \gamma\} \Rightarrow \pi, \{\phi[\![(a^r, a^w)]\!] \mid \psi\ @\ \gamma\}}\ \mathit{New}$$

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

Fig. 5. Definition of $\lambda^{\mathsf{PC}}_{\square}$.

$\Phi^w;\Pi \vdash \pi\ \mathsf{okay}$ 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 $\{M \mid \psi\ @\ \gamma\} \in \pi$, 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 $\lambda^{\mathsf{P}}_{\square}$. 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 $\lambda^{\mathsf{PC}}_{\square}$ would require a different parallel operational semantics, but the same type system would continue to work.) The rule *Sync* is nondeterministic in that it does not specify how to choose a pair of channels read and write to be synchronized together.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

As with $\lambda^{\mathsf{P}}_{\square}$, type safety of $\lambda^{\mathsf{PC}}_{\square}$ consists of configuration progress (Theorem 5.3) and configuration typing preservation (Theorem 5.5). Proofs of Theorems 5.3 and 5.5 use type safety for sequential computations in $\lambda^{\mathsf{PC}}_{\square}$ (Propositions 5.2 and 5.4). Note that in Theorem 5.5, a configuration transition $\pi \Rightarrow \pi'$ preserves channel locality: $\Phi^w ; \Pi \vdash \pi$ okay and $\Phi'^w ; \Pi' \vdash \pi'$ okay imply that π and π' satisfy channel locality, respectively.

Proposition 5.2 (*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) $M = \phi[\![\mathsf{spawn}\ \{M\}]\!]$, or
(6) for any store ψ such that $\Phi^w ; \Phi^r ; \Psi \vdash \psi$ okay, there exist some term M' and store ψ' such that $M \mid \psi \rightarrow M' \mid \psi'$.

Theorem 5.3 (*Configuration progress*)
Suppose $\Phi^w ; \Pi \vdash \pi$ okay. Then either
(1) π consists only of $\{V \mid \psi\ @\ \gamma\}$, $\{\phi[\![a^r\ ?]\!] \mid \psi\ @\ \gamma\}$, and $\{\phi[\![a^w\ !V]\!] \mid \psi\ @\ \gamma\}$, or
(2) there exists π' such that $\pi \Rightarrow \pi'$.

Proposition 5.4 (*Type preservation*)
Suppose $\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash M : A_{@\mathsf{L}}$, $\Phi^w ; \Phi^r ; \Psi \vdash \psi$ okay, and $M \mid \psi \rightarrow M' \mid \psi'$. 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 5.5 (*Configuration typing preservation*)
Suppose $\Phi^w ; \Pi \vdash \pi$ okay and $\pi \Rightarrow \pi'$. Then there exist a write channel typing Φ'^w and a configuration typing Π' such that $\Phi^w \subset \Phi'^w$, $\Pi \subset \Pi'$, and $\Phi'^w ; \Pi' \vdash \pi'$ okay.

6 Examples

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

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

6.1 Futures

Future (Baker & Hewitt, 1977; Halstead 1985) is a communication construct which can be thought of as a pointer to a thread. After the thread finishes its computation,

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

the pointer is implicitly dereferenced whenever the result is requested. We use read channels to simulate a restricted variant of the future that should be explicitly dereferenced by the programmer.

In order to create a future, first we need to evaluate a term M at a fresh thread. 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 the result of evaluating M is written to a write channel, M must evaluate to a global value; hence box M must typecheck:

$$\mathsf{box}\ M : \boxdot A$$

Moreover, M is to be evaluated at a remote thread and must contain no references. Hence, box M itself must be a global value, otherwise M cannot be transmitted to a remote thread

$$\mathsf{box}\ \mathsf{box}\ M : \boxdot\boxdot A$$

Thus future, our construct for futures, expects a value of type $\boxdot\boxdot A$ and returns a read channel of type $\langle A\rangle^r$ to which the result of the computation at a fresh thread is sent

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

As an example, consider future box box M which evaluates M at a fresh thread. First x' is bound to a global value box M, which is then transmitted to the fresh thread. By evaluating M at the fresh thread, we obtain a global value z, which is finally written to write channel y'_w. Note that even though y_w is actually bound to a global value (namely a write channel), we need to explicitly mark it as a global value by introducing another variable y'_w because it is used inside spawn (see the rule Spawn). Typically we use a term of the form letbox $x' =$ box x in M to inform the type system that variable x has a primitive type and is thus bound to a global value (because the type system does not automatically keep track of the locality of variables of primitive type).

6.2 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. However, for communications in the other direction, 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

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

the parent thread sends the child thread a^w, through which a write channel of type $\langle A\rangle^w$ is sent back to the parent thread.

We design $\mathsf{spawn}^{\mathsf{BI}}$, 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 $\mathsf{box}\ \lambda x_r : \langle A\rangle^r . M$, 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

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

Note that even though y_w is not marked explicitly as a global value, channel write $x'_w\,!\,y_w$ still typechecks by the rule W! because write channels have primitive types.

A minor drawback of $\mathsf{spawn}^{\mathsf{BI}}$ is 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):

$$
\begin{array}{l}
\mathsf{letbox}\ x_M = \mathsf{box}\ M\ \mathsf{in} \\
\mathsf{letbox}\ x_N = \mathsf{box}\ N\ \mathsf{in} \\
\mathsf{spawn}\ \{x_M\,!\,x_N\}
\end{array}
$$

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

6.3 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 $\mathsf{inl}\ a^w$ (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 $\mathsf{inr}\ V$ (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.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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

$$\begin{array}{l}\mathsf{let}\ cell_w = \\ \quad \mathsf{spawn}^{\mathsf{BI}}\ \mathsf{box}\ \lambda cell_r : \langle \mathsf{int} \rangle^w + \mathsf{int} \rangle^r .\ \ \mathsf{let\ rec}\ f = \lambda n : \mathsf{int}. \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{case}\ cell_r\ ?\ \mathsf{of}\ \ \mathsf{inl}\ ch_w \Rightarrow \mathsf{let}\ \_ = ch_w\ !n\ \mathsf{in}\ f\ n \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{inr}\ v \Rightarrow f\ v \\ \qquad\qquad\qquad\qquad\quad \mathsf{in} \\ \qquad\qquad\qquad\qquad f\ 0\end{array}$$

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

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

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.

6.4 Remote evaluation

Remote evaluation (Stamos & 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:

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

The following term starts the server and returns its interface which is a write channel req_w of type $\langle \langle \mathsf{int} \times \langle \mathsf{int} \rangle^w \rangle^w \to \mathsf{unit} \rangle^w$. Since req_w is a global value, any

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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 such a global value, the server binds s to the interface to the service and evaluates M:

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

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 $c_r\,?$.

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

6.5 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 6.4. 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:

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

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

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

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

6.6 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 $\mathsf{int} \to \mathsf{int}$.

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

7 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.

7.1 Local threads

Consider two read channels a^r_1 and a^r_2 (of the same type $\langle A\rangle^r$) belonging to different threads γ_1 and γ_2, respectively. a^r_1 wishes to forward to a^r_2 every incoming message V addressed to it with a channel write $a^w_2\,!\,V$ where a^w_2 is a write channel corresponding to a^r_2. (As write channels are global values, we assume that a^w_2 has already been transmitted to thread γ_1.) An easy solution is to call the following construct forward with arguments a^r_1 and a^w_2 at thread γ_1:

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

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^r_1 and a^w_2:

$$
\begin{array}{l}
\mathsf{forward}' : \langle A\rangle^r \to (\langle A\rangle^w \to \mathsf{unit})_{@\mathsf{G}} \\
\mathsf{forward}' = \lambda x : \langle A\rangle^r.\, \lambda y : \langle A\rangle^w. \\
\quad \mathsf{letbox}\ x' = \mathsf{box}\ x\ \mathsf{in} \qquad (*\ \textit{does not typecheck}\ *) \\
\quad \mathsf{letbox}\ y' = \mathsf{box}\ y\ \mathsf{in} \\
\quad \mathsf{spawn}\{\,\mathsf{let\ rec}\ f = \lambda\_ : \mathsf{unit}.\, \mathsf{let}\ \_ = y'\,!(x'\,?)\ \mathsf{in}\ f\,()\ \mathsf{in} \\
\qquad\quad f\,()\}
\end{array}
$$

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

$$\frac{\textit{fresh}\ \gamma'}{\pi, \{\phi[\![\mathsf{spawn}\ \{M\}]\!], M_1, \ldots, M_n \mid \psi\ @\ \gamma\} \Rightarrow \pi, \{\phi[\![()]\!], M_1, \ldots, M_n \mid \psi\ @\ \gamma\}, \{M \mid \cdot\ @\ \gamma'\}}\ \textit{Spawn}$$

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

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

Fig. 6. Configuration transition rules for local threads.

Unfortunately $\mathsf{forward}'$ 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 $\mathsf{forward}\ a_1^r\ a_2^w$. 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 beyond the scope of this paper.) Hence both references and read channels can be a 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 *solution* which is essentially a thread in $\lambda_{\square}^{\mathsf{PC}}$, while a solution may run multiple *processes* which are essentially local threads in $\lambda_{\square}^{\mathsf{PC}}$; all these processes can interact with any message addressed to the channel.

We introduce a new construct $\mathsf{lthread}\ \{M\}$ 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, $\mathsf{lthread}\ \{M\}$ 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 $\{M_1, \ldots, M_n \mid \psi\ @\ \gamma\}$ where each term M_i, $1 \leqslant i \leqslant n$, 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 $\{M, M_1, \ldots, M_n \mid \psi\ @\ \gamma\}$. 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 $\mathsf{lthread}\ \{M\}$ to create a fresh local thread.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

Now we rewrite forward by exploiting lthread

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

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 6.3 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.

$$\begin{array}{l}
\mathsf{let}\ cell_w = \mathsf{spawn}^{\mathsf{BI}}\ \ \mathsf{box}\ \lambda cell_r{:}\langle\langle\mathsf{int}\rangle^{w} + \mathsf{int}\rangle^{r}.\\
\qquad\qquad\qquad\qquad \mathsf{let}\ (c_r, c_w) = \mathsf{new}_{\langle\mathsf{int}\rangle}\ \mathsf{in}\\
\qquad\qquad\qquad\qquad \mathsf{let\ rec}\ f = \lambda_{:}\mathsf{unit}.\\
\qquad\qquad\qquad\qquad\quad \mathsf{case}\ cell_r\,?\ \mathsf{of}\ \ \mathsf{inl}\ ch_w \Rightarrow \mathsf{lthread}\ \{ch_w\,!(c_r\,?)\}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{inr}\ v \Rightarrow \mathsf{lthread}\ \{c_w\,!v\}\\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\\
\qquad\qquad\qquad\qquad f\,()\\
\\
\quad \mathsf{letbox}\ cell'_w = \mathsf{box}\ cell_w\ \mathsf{in}\\
\quad \mathsf{letbox}\ read = \mathsf{box}\ \lambda_{:}\mathsf{unit}.\ \ \mathsf{let}\ (x_r, x_w) = \mathsf{new}_{\langle\mathsf{int}\rangle}\ \mathsf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{let}\ _ = cell'_w\,!(\mathsf{inl}\ \ x_w)\ \mathsf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad x_r\,?\\
\quad \mathsf{in}\\
\quad \mathsf{letbox}\ write = \mathsf{box}\ \lambda x{:}\mathsf{int}.\,cell'_w\,!(\mathsf{inr}\ x)\ \mathsf{in}\\
\quad (read, write)
\end{array}$$

7.2 Type-safe data parallelism

The characteristic feature of the type system of $\lambda^{\mathsf{PC}}_{\Box}$ 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 $\lambda^{\mathsf{PC}}_{\Box}$, as follows.

Consider a parallel map construct mapP which applies a function f to each element of an array e in parallel (where we assume that e consists only of global values)

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

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 among child threads and thus are safe to use.

Checking whether 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

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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.

If we want f to protect against not only references but also write channels being shared by child threads, we can 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

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

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.

8 Related work

8.1 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 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. Alice (Rossberg *et al.*, 2005) raises runtime exceptions when references are transmitted to remote threads.

$\lambda^{\mathsf{PC}}_{\square}$ 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 $\lambda^{\mathsf{PC}}_{\square}$ relies on the type system to forestall remote references.

8.2 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

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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 $\lambda_{\cdot}^{\mathsf{PC}}$. 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 that in $\lambda_{\cdot}^{\mathsf{PC}}$ 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 $\lambda_{\cdot}^{\mathsf{PC}}$ 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 restriction on function types which severely limits the generality of the calculus: for a function type $\tau \to \sigma$, if σ is sendable, τ must also be sendable, in which case $\tau \to \sigma$ is also automatically regarded as sendable. In our setting, such a restriction means, for example, that no term of type $\mathsf{ref\ int} \to \mathsf{int}$ or $\langle \mathsf{int} \rangle^r \to \mathsf{int}$ is even allowed and that every function of type $\mathsf{int} \to \mathsf{int}$ must be global, neither of which is the case in $\lambda_{\cdot}^{\mathsf{PC}}$. Second, the type system ignores the call-by-value semantics of the underlying lambda-calculus. Specifically it includes a typing rule ($\textsc{Term}_l$) 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, $\lambda_{\cdot}^{\mathsf{PC}}$ imposes no syntactic or semantic restriction on function types and properly accounts for the call-by-value semantics in the presence of mutable references.

8.3 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 (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 (Zhang & Potter, 2002)

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

also syntactically distinguishes between read channels and write channels with an additional requirement that only write channels can be transmitted via channels.

$\lambda_{\boxdot}^{\mathsf{PC}}$ is 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 $\lambda_{\boxdot}^{\mathsf{PC}}$. Thus our decision to distinguish between read channels and write channels has a different motivation from previous work.

8.4 Modal types for parallel and distributed computations

There are a few distributed languages whose type systems are based on the spatial interpretation of modal logic.

Borghuis and Feijs (2000) present a typed lambda-calculus MTSN (Modal Type System for Networks) which assumes stationary services (i.e., stationary code) and mobile data. An indexed modal type $\Box^{\omega}(A \to B)$ represents services transforming data of type A into data of type B at node ω. MTSN is a task description language rather than a programming language, since services are all "black boxes" whose inner workings are unknown. For example, terms of type $tex \to dvi$ all describe procedures to convert *tex* files to *dvi* files. Thus reduction on terms is tantamount to simplifying procedures to achieve a certain task.

Jia and Walker (2004) present a modal language λ_{rpc} which is based on hybrid logic (Braüner, 2004) as every typing judgment explicitly specifies the node where typechecking takes place. The modalities $\Box$ and $\diamond$ are used for terms that can be evaluated at any node and at a certain node, respectively.

Murphy *et al.* (2004) present a modal language *Lambda 5* which addresses both code mobility and resource locality. It is based on modal logic S5 where all judgments are relativized to nodes as in Simpson (1994). A value of type $\Box A$ contains a term that can be evaluated at any node, and a value of type $\diamond A$ contains a *label*, a reference to a local resource. A label may appear at remote nodes, but the type system guarantees that it is dereferenced only at the node where it is valid. Murphy *et al.* (2007) later developed Lambda 5 into a distributed language *ML5* whose type system also prevents local resources from being used at remote nodes while their references may travel between nodes. ML5 uses a typing judgment $M : A_{@\mathrm{w}}$ to mean that M is a term of type A for locality w. As it permits not only constants but also variables as localities, the type system of ML5 is expressive enough to encode modalities $\Box$ and $\diamond$ both.

Our previous work (Park, 2006) was the first departure from the typical spatial interpretation of modal logic in that it uses a new modality $\boxdot$ to focus on values rather than terms. In our previous work, we used two separate typing judgments to distinguish between local values and global values: 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 a global value. The base language $\lambda_{\boxdot}$ in the present work combines the two typing judgments into a single typing judgment $M : A_{@L}$ where locality L, either L or G, indicates whether M evaluates to a local value or a global value. This simplification of typing judgments is essential to

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

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 previous work whereas $\lambda_{\boxdot}$ uses only three typing rules.

In comparison with previous work whose type system relativizes typing judgments to nodes, the type system of $\lambda_{\boxdot}^{\mathsf{PC}}$ lacks the expressive power necessary for allowing references to escape from their host threads, even if it is known that they are not dereferenced at remote threads. This is essential because its typing judgment relies only on two fixed localities: every value is associated with either locality L, meaning that it is valid "here only," or locality G, meaning that it is valid "everywhere." As a result, the type system cannot distinguish remote threads particularly relevant to the current thread (e.g., parent threads or child threads) from remote threads irrelevant to the current thread, and every value traveling between threads must prove to be global regardless of its destination thread. We leave it to the future work to relax this limitation by further extending the type system.

9 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 in parallel computations with higher-order channels as in $\lambda_{\boxdot}^{\mathsf{PC}}$. 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.

Acknowledgements

The authors are grateful to the anonymous reviewers for their helpful comments. 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).

References

Allan, Eric, Chase, David, Hallet, Joe, Luchangco, Victor, Maessen, Jan-Willem, Ryu, Sukyoung, Steele, Jr., Guy L. & Tobin-Hochstadt, Sam (2007) *The Fortress Language Specification Version 1.0Beta*. Technical Report, Sun Microsystems Inc.

Amadio, Roberto (1997) An asynchronous model of locality, failure, and process mobility. In *Proceedings of the Second International Conference on Coordination Languages and Models, LNCS 1282*. Berlin, Germany: Springer-Verlag, pp. 374–391.

Armstrong, Joe (1997) The development of Erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. Amsterdam, The Netherlands: ACM Press, pp. 196–203.

Asanovic, Krste, Bodik, Ras, Catanzaro, Bryan Christopher, Gebis, Joseph, Husbands, Parry, Keutzer, Kurt, Patterson, David, Plishker, William, Shalf, John, Williams, Samuel & Yelick, Katherine (December 2006). *The Landscape of Parallel Computing Research: A View From Berkeley*. Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Baker, Jr., Henry & Hewitt, Carl (1977) The incremental garbage collection of processes. *Sigplan Notice* **12**(8), 55–59.

Blelloch, Guy (1996) Programming parallel algorithms. *Commun. ACM* **39**(3), 85–97.

Borghuis, Tijn & Feijs, Loe (2000) A constructive logic for services and information flow in computer networks. *Comput. J.* **43**(4), 275–289.

Braüner, Torben (2004) Natural deduction for hybrid logic. *J. Logic Comput.* **14**(3), 329–353.

Chakravarty, Manuel, Leshchinskiy, Roman, Peyton Jones, Simon, Keller, Gabriele & Marlow, Simon (2007) Data Parallel Haskell: A status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. Nice, France: ACM, pp. 11–18.

Charles, Philippe, Grothoff, Christian, Saraswat, Vijay, Donawa, Christopher, Kielstra, Allan, Ebcioglu, Kemal, von Praun, Christoph & Sarkar, Vivek (2005) 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*. San Diego, CA: ACM, pp. 519–538.

Cray Inc. (2005) *The Chapel Language Specification Version 0.4*. Technical Report.

El-Ghazawi, Tarek, Carlson, William & Draper, Jesse (2003) *UPC Language Specifications, v1.1.1*

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

Fournet, Cédric, Gonthier, Georges, Lévy, Jean-Jacques, Maranget, Luc & Rémy, Didier (1996) A calculus of mobile agents. In *CONCUR: Seventh International Conference on Concurrency Theory, LNCS 1119*. Pisa, Italy: Springer, pp. 406–421.

Fournet, Cédric, Le Fessant, Fabrice, Maranget, Luc & Schmitt, Alan (2003) JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming, Fourth International School, 2002, LNCS 2638*, Jeuring, Johan, & Peyton Jones, Simon (eds). Oxford, UK: Springer-Verlag, pp. 129–158.

Halstead, Jr., Robert (1985) MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* **7**(4), 501–538.

Hilfinger, Paul, Bonachea, Dan, Datta, Kaushik, Gay, David, Graham, Susan, Liblit, Benjamin, Pike, Geoffrey, Su, Jimmy & Yelick, Katherine (November 2005) *Titanium Language Reference Manual, Version 2.19*. Technical Report No. UCB/EECS-2005-15, EECS Department, University of California, Berkeley.

Hochstein, Lorin, Carver, Jeff, Shull, Forrest, Asgari, Sima, Basili, Victor, Hollingsworth, Jeffrey & Zelkowitz, Marvin (2005) Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. Seattle, WA: IEEE Computer Society, pp. 35–43.

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

Knabe, Frederick (1995) *Language Support for Mobile Agents*. Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University.

Murphy, VII, Tom, Crary, Karl & Harper, Robert (November 2007) Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*. Sophia-Antipolis, France: Springer.

Murphy, VII, Tom, Crary, Karl, Harper, Robert & Pfenning, Frank (2004) A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science*. Turku, Finland: IEEE Press, pp. 286–295.

Nikhil, Rishiyur & Arvind (2001) *Implicit Parallel Programming in pH*. San Francisco, CA: Morgan Kaufmann.

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

Park, Sungwoo (2006) A modal language for the safety of mobile values. In *Proceedings of the Fourth Asian Symposium on Programming Languages and Systems, LNCS 4279*, Kobayashi, Naoki (ed). Sydney, Australia: Springer, pp. 217–233.

Park, Sungwoo (2007) Type-safe higher-order channels in ML-like languages. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. Freiburg, Germany: ACM, pp. 191–202.

Pfenning, Frank & Davies, Rowan (2001) A judgmental reconstruction of modal logic. *Math. Struct. Comput. Sci.* **11**(4), 511–540.

Pierce, Benjamin & Sangiorgi, Davide (1993) Typing and subtyping for mobile processes. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*. Montreal, Canada: IEEE Computer Society, pp. 376–385.

Rossberg, Andreas, Botlan, Didier Le, Tack, Guido, Brunklaus, Thorsten & Smolka, Gert (2005) Alice ML through the looking glass. In *Trends in Functional Programming*, Hans-Wolfgang Loidl (ed), München, Germany: vol. 5. Intellect Books, pp. 79–96.

Schmitt, Alan (2002) Safe dynamic binding in the join calculus. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress—TC1 Stream/Second IFIP International Conference on Theoretical Computer Science*. Montréal, Quebéc, Canada: Kluwer, B.V, pp 563–575.

Schmitt, Alan & Stefani, Jean-Bernard (2003) The M-calculus: A higher-order distributed process calculus. In *Proceedings of the 30th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. New Orleans, LA: ACM, pp. 50–61.

Shavit, Nir & Touitou, Dan (1995) Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. Ottawa, Ontario, Canada: ACM, pp. 204–213.

Simpson, Alex K. (1994) *The Proof Theory and Semantics of Intuitionistic Modal Logic*. Ph.D. Thesis, Department of Philosophy, University of Edinburgh.

Stamos, James & Gifford, David (1990) Remote evaluation. *ACM Trans. Program. Lang. Syst.* **12**(4), 537–564.

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

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

Zhang, Xiaogang & Potter, John (2002) Responsive bisimulation. In *Pages 601–612 of: Proceedings of the Second IFIP International Conference on Theoretical Computer Science*. Montréal Québec, Canada: Kluwer, B.V., pp. 601–612.

Appendix

A Proofs about $\lambda_{\boxdot}$

To simplify proofs, we omit product types, sum types, and the fixed point construct.

Lemma A.1 (*Canonical forms*)
If V is a value of type

$$\begin{array}{l} \mathsf{unit} \\ A \to B \\ \mathsf{ref}\ A \\ \boxdot A \end{array},$$

then V is

$$\begin{array}{l} () \\ \lambda x{:}A.\, M \\ l \\ \mathsf{box}\ M \end{array}.$$

Proof
By case analysis of V. □

Proof of Theorem 2.3
By induction on the structure of $\cdot \mid \Psi \vdash M : A_{@L}$.

If M is already a value, we need no further consideration. Therefore we assume that M is not a value. The cases for those constructs available in the simply typed lambda-calculus remain the same except that we use a judgment $\cdot \mid \Psi \vdash M : A_{@L}$. The case for the rule Prim follows immediately from induction hypothesis on the premise. Therefore the only interesting case is the rule $\boxdot\mathsf{E}$.

$$\text{Case } \mathscr{D} :: \frac{\cdot \mid \Psi \vdash M : \boxdot A_{@\mathsf{L}} \quad x : A_{@\mathsf{G}} \mid \Psi \vdash N : C_{@L}}{\cdot \mid \Psi \vdash \mathsf{letbox}\ x = M\ \mathsf{in}\ N : C_{@L}}\ \boxdot\mathsf{E}$$

Subcase 1: M is a value
$M = \mathsf{box}\ M'$ from $\cdot \mid \Psi \vdash M : \boxdot A_{@\mathsf{L}}$ and Lemma A.1

$$\mathscr{D} :: \frac{\dfrac{\cdot \mid \Psi \vdash M' : A_{@\mathsf{G}}}{\cdot \mid \Psi \vdash \mathsf{box}\ M' : \boxdot A_{@\mathsf{L}}}\ \boxdot\mathsf{I} \qquad \cdot \mid \Psi \vdash N : C_{@L}}{\cdot \mid \Psi \vdash \mathsf{letbox}\ x = \mathsf{box}\ M'\ \mathsf{in}\ N : C_{@L}}\ \boxdot\mathsf{E}$$

because only the rule $\boxdot\mathsf{I}$ is applicable to deduce $\cdot \mid \Psi \vdash \mathsf{box}\ M' : \boxdot A_{@\mathsf{L}}$.

Subcase 1-1: M' is a value
$\mathsf{letbox}\ x = \mathsf{box}\ M'\ \mathsf{in}\ N \mid \psi \to [M'/x]N \mid \psi$

Subcase 1-2: M' is not a value
$\Psi \vdash \psi\ \mathsf{okay}$ assumption
$M' \mid \psi \to M'' \mid \psi'$ by induction hypothesis on $\cdot \mid \Psi \vdash M' : A_{@\mathsf{G}}$
$\mathsf{letbox}\ x = \mathsf{box}\ M'\ \mathsf{in}\ N \mid \psi \to \mathsf{letbox}\ x = \mathsf{box}\ M''\ \mathsf{in}\ N \mid \psi'$

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

Subcase 2: M is not a value

$\Psi \vdash \psi$ okay — assumption

$M \mid \psi \to M' \mid \psi'$ — by induction hypothesis on $\cdot \mid \Psi \vdash M : \Box A_{@\mathsf{L}}$

letbox $x = M$ in $N \mid \psi \to$ letbox $x = M'$ in $N \mid \psi'$ □

Proof of Theorem 2.4

By induction on the structure of $\Gamma, x : A_{@\mathsf{L}} \mid \Psi \vdash M : C_{@L}$ and $\Gamma, x : A_{@\mathsf{G}} \mid \Psi \vdash M : C_{@L}$.

All rules except Var and GVal share the property that typing contexts are not used in premises or that typing contexts in premises extend typing contexts in the conclusion. We consider the rules Var and GVal as follows:

Case $\dfrac{y : C_{@L'} \in \Gamma \quad L \leqslant L'}{\Gamma, x : A_{@L''} \mid \Psi \vdash y : C_{@L}}$ Var where $y \neq x$ and $L'' = \mathsf{G}$ or $L'' = \mathsf{L}$

$\Gamma \mid \Psi \vdash y : C_{@L}$ — from $y : C_{@L'} \in \Gamma$ and the rule Var

$\Gamma \mid \Psi \vdash [N/x]y : C_{@L}$ — from $[N/x]y = y$

Case $\dfrac{L \leqslant \mathsf{L}}{\Gamma, x : A_{@\mathsf{L}} \mid \Psi \vdash x : A_{@L}}$ Var

$\Gamma \mid \Psi \vdash N : A_{@\mathsf{L}}$ — assumption

$\Gamma \mid \Psi \vdash [N/x]x : A_{@\mathsf{L}}$ — from $[N/x]x = N$

Case $\dfrac{L \leqslant \mathsf{G}}{\Gamma, x : A_{@\mathsf{G}} \mid \Psi \vdash x : A_{@L}}$ Var

$\Gamma_{\mathsf{G}} \mid \cdot \vdash V : A_{@\mathsf{L}}$ — assumption

$\Gamma \mid \Psi \vdash V : A_{@\mathsf{G}}$ — by the rule GVal

$\Gamma \mid \Psi \vdash [V/x]x : A_{@\mathsf{G}}$ — from $[V/x]x = V$

$\Gamma \mid \Psi \vdash [V/x]x : A_{@L}$ — by Proposition 2.2

Case $\dfrac{\Gamma_{\mathsf{G}} \mid \cdot \vdash V' : C_{@\mathsf{L}}}{\Gamma, x : A_{@\mathsf{L}} \mid \Psi \vdash V' : C_{@\mathsf{G}}}$ GVal

$\Gamma_{\mathsf{G}} \mid \cdot \vdash [N/x]V' : C_{@\mathsf{L}}$ — from x is not a free variable in V' and $[N/x]V' = V'$

$\Gamma \mid \Psi \vdash [N/x]V' : C_{@\mathsf{G}}$ — by the rule GVal

Case $\dfrac{\Gamma_{\mathsf{G}}, x : A_{@\mathsf{G}} \mid \cdot \vdash V' : C_{@\mathsf{L}}}{\Gamma, x : A_{@\mathsf{G}} \mid \Psi \vdash V' : C_{@\mathsf{G}}}$ GVal

$\Gamma_{\mathsf{G}} \mid \cdot \vdash V : A_{@\mathsf{L}}$ — assumption

$\Gamma_{\mathsf{G}} \mid \cdot \vdash [V/x]V' : C_{@\mathsf{L}}$ — by induction hypothesis

$\Gamma \mid \Psi \vdash [V/x]V' : C_{@\mathsf{G}}$ — by the rule GVal □

Corollary A.2

If $\Gamma \mid \Psi \vdash V : P_{@\mathsf{L}}$ and $\Gamma, x : P_{@\mathsf{G}} \mid \Psi \vdash M : C_{@L}$, then $\Gamma \mid \Psi \vdash [V/x]M : C_{@L}$.

Proof

By Theorem 2.4 and Definition 2.1. □

Proposition A.3

If $\cdot \mid \Psi \vdash (\lambda x{:}A.\, M)\ V : C_{@L}$, then $\cdot \mid \Psi \vdash [V/x]M : C_{@L}$.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

Proof
By induction on the structure of $\cdot \mid \Psi \vdash (\lambda x{:}A.\, M)\ V : C_{@L}$. The proof uses Theorem 2.4 and the inversion property of $\lambda_{\Box}$. □

Proposition A.4
If $\cdot \mid \Psi \vdash \mathsf{letbox}\ x = \mathsf{box}\ V\ \mathsf{in}\ M : C_{@L}$, then $\cdot \mid \Psi \vdash [V/x]M : C_{@L}$.

Proof
By induction on the structure of $\cdot \mid \Psi \vdash \mathsf{letbox}\ x = \mathsf{box}\ V\ \mathsf{in}\ M : C_{@L}$. The proof uses Theorem 2.4 and Lemma 2.5 and the inversion property of $\lambda_{\Box}$. □

Proof of Theorem 2.6
By induction on the structure of $M \mid \psi \to M' \mid \psi'$. □

B Proofs about $\lambda^{\mathsf{P}}_{\Box}$

Proposition B.1 (*Progress*)
Suppose $\cdot \mid \Psi \vdash M : A_{@L}$. Then either
(1) M is a value,
(2) $M = \phi[\![\mathsf{spawn}\ \{N\}]\!]$, or
(3) for any store ψ such that $\Psi \vdash \psi\ \mathsf{okay}$, there exist some term M' and store ψ' such that $M \mid \psi \to M' \mid \psi'$.

Proof
By induction on the structure of $\cdot \mid \Psi \vdash M : A_{@L}$. If M is already a value, we need no further consideration. Assume that M is not a value. The cases for those constructs available in $\lambda_{\Box}$ remain the same, and the case for the rule Spawn is trivial: $M = \mathsf{spawn}\ \{N\} = \phi[\![\mathsf{spawn}\ \{N\}]\!]$ where $\phi = []$. □

Proof of Theorem 4.1
By the case analysis of $\Pi \vdash \pi\ \mathsf{okay}$.

By Proposition B.1, there are only three cases for every $\{M \mid \psi\ @\ \gamma\} \in \pi$. If π consists only of $\{V \mid \psi\ @\ \gamma\}$, we are done. Assume that π contains $\{M \mid \psi\ @\ \gamma\}$ such that M is not a value. Then either M is $\phi[\![\mathsf{spawn}\ \{N\}]\!]$, or there exist some term M' and store ψ' such that $M \mid \psi \to M' \mid \psi'$ where $\Psi \vdash \psi\ \mathsf{okay}$. We analyze these two cases.

Case $\pi = \pi_1, \{\phi[\![\mathsf{spawn}\ \{N\}]\!] \mid \psi\ @\ \gamma\}$
Let $\pi' = \pi_1, \{\phi[\![()]\!] \mid \psi\ @\ \gamma\}, \{N \mid \cdot\ @\ \gamma'\}$ where γ' is a fresh node
$\pi \Rightarrow \pi'$ by the rule *Spawn*

Case $\pi = \pi_1, \{\phi[\![M]\!] \mid \psi\ @\ \gamma\}$ where $M \mid \psi \to M' \mid \psi'$
Let $\pi' = \pi_1, \{\phi[\![M']\!] \mid \psi'\ @\ \gamma\}$
$\pi \Rightarrow \pi'$ by *Red* □

Theorem B.2 (*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 $\Gamma \mid \Psi \vdash [V/x]M : C_{@L}$.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

Proof
By induction on the structure of $\Gamma, x : A_{@\mathsf{L}} \mid \Psi \vdash M : C_{@L}$ and $\Gamma, x : A_{@\mathsf{G}} \mid \Psi \vdash M : C_{@L}$.
The proof extends the proof of Theorem 2.4 with a new case for the rule Spawn. □

Proposition B.3 (*Type preservation*)

Suppose $\cdot \mid \Psi \vdash M : A_{@L}$, $\Psi \vdash \psi$ 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 $\Psi' \vdash \psi'$ okay.

Proof
By induction on the structure of $M \mid \psi \to M' \mid \psi'$. The proof is the same as the proof of Theorem 2.6 except that it uses Theorem B.2 instead of Theorem 2.4 for the substitution property. □

Proof of Theorem 4.2
By the case analysis of $\pi \Rightarrow \pi'$. We show the case for the rule *Red* only which uses Proposition B.3.

Case $\dfrac{M \mid \psi \to M' \mid \psi'}{\pi_1, \{M \mid \psi \mathbin{@} \gamma\} \Rightarrow \pi_1, \{M' \mid \psi' \mathbin{@} \gamma\}}$ *Red*
where $\pi = \pi_1, \{M \mid \psi \mathbin{@} \gamma\}$ and $\pi' = \pi_1, \{M' \mid \psi' \mathbin{@} \gamma\}$
$\cdot \mid \Psi \vdash M : A_{@\mathsf{L}}$ and $\Psi \vdash \psi$ okay and $\gamma : A \in \Pi$ from $\Pi \vdash \pi$ okay
$M \mid \psi \to M' \mid \psi'$ assumption
$\cdot \mid \Psi' \vdash M' : A_{@\mathsf{L}}$ and $\Psi \subset \Psi'$ and $\Psi' \vdash \psi'$ okay for some store typing Ψ'
by Proposition B.3
$\gamma : A \vdash \{M' \mid \psi' \mathbin{@} \gamma\}$ okay by the rule Conf
$\Pi_1 \vdash \pi_1$ okay where $\Pi = \Pi_1, \gamma : A$ from $\Pi \vdash \pi$ okay
$\Pi_1, \gamma : A \vdash \pi_1, \{M' \mid \psi' \mathbin{@} \gamma\}$ okay from $\Pi_1 \vdash \pi_1$ okay and
$\gamma : A \vdash \{M' \mid \psi' \mathbin{@} \gamma\}$ okay
$\Pi' \vdash \pi'$ okay from $\Pi' = \Pi_1, \gamma : A$ and $\pi' = \pi_1, \{M' \mid \psi' \mathbin{@} \gamma\}$
$\Pi \subset \Pi'$ from $\Pi = \Pi'$ □

C Proofs about $\lambda^{\mathsf{PC}}_{\square}$

Lemma C.1 (*Canonical forms*)
If V is a value of type

$$\begin{matrix} \langle A \rangle^r \\ \langle A \rangle^w \end{matrix},$$

then V is

$$\begin{matrix} a^r \\ a^w \end{matrix}.$$

Proof
By the case analysis of V. □

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

Proof of Proposition 5.2

By induction on the structure of $\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash M : A_{@L}$.

If M is already a value, we need no further consideration. Assume that M is not a value. The cases for the constructs available in $\lambda^{\mathsf{P}}_{\square}$ remain the same except that we use $\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash M : A_{@L}$ in place of $\cdot \mid \Psi \vdash M : A_{@L}$. The case for the rule New is trivial: $M = \mathsf{new}_{\langle A \rangle} = \phi[\![\mathsf{new}_{\langle A \rangle}]\!]$ where $\phi = []$. Therefore the only interesting cases are the rules R? and W!. To illustrate the need for induction on the structure of $\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash M : A_{@L}$, we show the case for the rule R?. The case for the rule W! is similar.

Case $\dfrac{\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash N : \langle A \rangle^r{}_{@\mathsf{L}}}{\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash N\,? : A_{@L}}\ \mathsf{R?}$ where $M = N\,?$

Subcase 1: N is a value

$N = a^r$ where $a^r \mapsto A \in \Phi^r$

because only the rule ChanR is applicable to deduce $\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash N : \langle A \rangle^r{}_{@\mathsf{L}}$

$M = \phi[\![a^r\,?]\!]$ where $\phi = []$

Subcase 2: N is written as $\phi[\![\mathsf{new}_{\langle B \rangle}]\!]$, $\phi[\![a^r\,?]\!]$, $\phi[\![a^w\,!V]\!]$, or $\phi[\![\mathsf{spawn}\ \{N'\}]\!]$

$N\,?$ is written as $\phi'[\![\mathsf{new}_{\langle B \rangle}]\!]$, $\phi'[\![a^r\,?]\!]$, $\phi'[\![a^w\,!V]\!]$, or $\phi'[\![\mathsf{spawn}\ \{N'\}]\!]$ where $\phi' = \phi\,?$

Subcase 3: $N \mid \psi \to N' \mid \psi'$

$\Phi^w ; \Phi^r ; \Psi \vdash \psi\ \mathsf{okay}$ assumption

$N\,? \mid \psi \to N'\,? \mid \psi'$ □

Proof of Theorem 5.3

By the case analysis of $\Phi^w ; \Pi \vdash \pi\ \mathsf{okay}$.

By Proposition 5.2, there are only six cases for every $\{M \mid \psi\ @\ \gamma\} \in \pi$. If π consists only of $\{V \mid \psi\ @\ \gamma\}$, $\{\phi[\![a^r\,?]\!] \mid \psi\ @\ \gamma\}$, or $\{\phi[\![a^w\,!V]\!] \mid \psi\ @\ \gamma\}$, we are done. (If π happens to include both $\{\phi[\![a^r\,?]\!] \mid \psi\ @\ \gamma\}$ and $\{\phi'[\![a^w\,!V]\!] \mid \psi'\ @\ \gamma'\}$ for the same channel identifier a, a communication between nodes γ and γ' occurs by the rule *Sync*.)

If there exists $\{M \mid \psi\ @\ \gamma\} \in \pi$ such that $M = \phi[\![\mathsf{spawn}\ \{N\}]\!]$ or $M \mid \psi \to M' \mid \psi'$ for some term M' and store ψ', the proof follows the same pattern as in $\lambda^{\mathsf{P}}_{\square}$. Therefore we consider the case in which there exists $\{\phi[\![\mathsf{new}_{\langle A \rangle}]\!] \mid \psi\ @\ \gamma\} \in \pi$.

Case $\pi = \pi_1, \{\phi[\![\mathsf{new}_{\langle A \rangle}]\!] \mid \psi\ @\ \gamma\}$

Let $\pi' = \pi_1, \{\phi[\![(a^r, a^w)]\!] \mid \psi\ @\ \gamma\}$ where a^r and a^w are fresh

$\pi \Rightarrow \pi'$ by the rule *New* □

Theorem C.2 (*Substitution*)

If $\Gamma \mid \Phi^w ; \Phi^r ; \Psi \vdash N : A_{@\mathsf{L}}$,

then $\Gamma, x : A_{@\mathsf{L}} \mid \Phi^w ; \Phi^r ; \Psi \vdash M : C_{@L}$ implies $\Gamma \mid \Phi^w ; \Phi^r ; \Psi \vdash [N/x]M : C_{@L}$.

If $\Gamma_{\mathsf{G}} \mid \Phi^w ; \cdot ; \cdot \vdash V : A_{@\mathsf{L}}$,

then $\Gamma, x : A_{@\mathsf{G}} \mid \Phi^w ; \Phi^r ; \Psi \vdash M : C_{@L}$ implies $\Gamma \mid \Phi^w ; \Phi^r ; \Psi \vdash [V/x]M : C_{@L}$.

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

Proof

By induction on the structure of $\Gamma, x : A_{@\mathsf{L}} \mid \Phi^w ; \Phi^r ; \Psi \vdash M : C_{@L}$ and $\Gamma, x : A_{@\mathsf{G}} \mid \Phi^w ; \Phi^r ; \Psi \vdash M : C_{@L}$. The proof extends the proof of Theorem B.2 by replacing each judgment of the form $\Gamma \mid \Psi \vdash M : A_{@L}$ by $\Gamma \mid \Phi^w ; \Phi^r ; \Psi \vdash M : A_{@L}$. Among new cases in $\lambda^{\mathsf{PC}}_{\square}$, the cases for the rules New, ChanR, and ChanW are trivial because substitutions do not change terms. The cases for the rules R? and W! are also trivial because typing contexts are the same in both the premise and the conclusion. We consider the case for the rule GVal.

Case $\dfrac{\Gamma_{\mathsf{G}} \mid \Phi^w ; \cdot ; \cdot \vdash V' : C_{@\mathsf{L}}}{\Gamma, x : A_{@\mathsf{L}} \mid \Phi^w ; \Phi^r ; \Psi \vdash V' : C_{@\mathsf{G}}}\ \mathsf{GVal}$

$\Gamma_{\mathsf{G}} \mid \Phi^w ; \cdot ; \cdot \vdash [N/x]V' : C_{@\mathsf{L}}$ from x is not a free variable in V' and $[N/x]V' = V'$

$\Gamma \mid \Phi^w ; \Phi^r ; \Psi \vdash [N/x]V' : C_{@\mathsf{G}}$ by the rule GVal

Case $\dfrac{\Gamma_{\mathsf{G}}, x : A_{@\mathsf{G}} \mid \Phi^w ; \cdot ; \cdot \vdash V' : C_{@\mathsf{L}}}{\Gamma, x : A_{@\mathsf{G}} \mid \Phi^w ; \Phi^r ; \Psi \vdash V' : C_{@\mathsf{G}}}\ \mathsf{GVal}$

$\Gamma_{\mathsf{G}} \mid \Phi^w ; \cdot ; \cdot \vdash V : A_{@\mathsf{L}}$ assumption

$\Gamma_{\mathsf{G}} \mid \Phi^w ; \cdot ; \cdot \vdash [V/x]V' : C_{@\mathsf{L}}$ by induction hypothesis

$\Gamma \mid \Phi^w ; \Phi^r ; \Psi \vdash [V/x]V' : C_{@\mathsf{G}}$ by the rule GVal □

Proof of Proposition 5.4

By induction on the structure of $M \mid \psi \to M' \mid \psi'$. The proof is the same as the proof of Proposition B.3 except that we use $\cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash M : A_{@L}$ in place of $\cdot \mid \Psi \vdash M : A_{@L}$. The proof uses Theorem C.2. □

Proof of Theorem 5.5

By the case analysis of $\pi \Rightarrow \pi'$.

The cases for the rules *Red* and *Spawn* are the same as in Theorem 4.2 except that we use $\Phi^w ; \Pi \vdash \pi$ okay in place of $\Pi \vdash \pi$ okay. Thus, we analyze the cases for the rules *New* and *Sync*. The proof uses a similar strategy as in Theorem 4.2 except that it frequently uses weakening, so we highlight only important points.

Case $\dfrac{\mathit{fresh}\ (a^r, a^w)}{\pi_1, \{\phi[\![\mathsf{new}_{\langle A \rangle}]\!] \mid \psi \ @\ \gamma\} \Rightarrow \pi_1, \{\phi[\![(a^r, a^w)]\!] \mid \psi \ @\ \gamma\}}\ \mathit{New}$

where $\begin{cases} \pi = \pi_1, \{\phi[\![\mathsf{new}_{\langle A \rangle}]\!] \mid \psi \ @\ \gamma\} \\ \pi' = \pi_1, \{\phi[\![(a^r, a^w)]\!] \mid \psi \ @\ \gamma\} \end{cases}$

$\begin{cases} \gamma : C \in \Pi \\ \cdot \mid \Phi^w ; \Phi^r ; \Psi \vdash \phi[\![\mathsf{new}_{\langle A \rangle}]\!] : C_{@\mathsf{L}} \end{cases}$ from $\Phi^w ; \Pi \vdash \pi$ okay

Let $\begin{cases} \Pi = \Pi_1, \gamma : C \\ \Phi'^w = \Phi^w, a^w \mapsto A \\ \Phi'^r = \Phi^r, a^r \mapsto A \end{cases}$

From these assumptions, we can prove $\Phi'^w ; \Pi_1, \gamma : C \vdash \pi_1, \{\phi[\![(a^r, a^w)]\!] \mid \psi \ @\ \gamma\}$ okay which simplifies to $\Phi'^w ; \Pi' \vdash \pi'$ okay where $\Phi^w \subset \Phi'^w$ and $\Pi = \Pi'$.

Case $\dfrac{}{\pi_1, \{\phi_1[\![a^r\ ?]\!] \mid \psi_1 \ @\ \gamma_1\}, \{\phi_2[\![a^w\ !\ V]\!] \mid \psi_2 \ @\ \gamma_2\} \Rightarrow \pi_1, \{\phi_1[\![V]\!] \mid \psi_1 \ @\ \gamma_1\}, \{\phi_2[\![()]\!] \mid \psi_2 \ @\ \gamma_2\}}\ \mathit{Sync}$

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press

$$\text{where } \begin{cases} \pi = \pi_1, \{\phi_1[\![a^r\ ?]\!] \mid \psi_1 \ @\ \gamma_1\}, \{\phi_2[\![a^w\ !V]\!] \mid \psi_2 \ @\ \gamma_2\} \\ \pi' = \pi_1, \{\phi_1[\![V]\!] \mid \psi_1 \ @\ \gamma_1\}, \{\phi_2[\![()]\!] \mid \psi_2 \ @\ \gamma_2\} \end{cases}$$

For this case, the proof exploits $a^r \mapsto B \in {\Phi_1}^r$ and $a^w \mapsto B' \in \Phi^w$.

$$\begin{cases} \gamma_1 : A_1 \in \Pi \\ \cdot \mid \Phi^w ; {\Phi_1}^r ; \Psi_1 \vdash \phi_1[\![a^r\ ?]\!] : A_{1@\mathsf{L}} \end{cases}$$
from $\Phi^w ; \Pi \vdash \pi$ okay

$\cdot \mid \Phi^w ; {\Phi_1}^r ; \Psi_1 \vdash a^r\ ? : B_{@L}$ by the rule R? and $a^r \mapsto B \in {\Phi_1}^r$

$$\begin{cases} \gamma_2 : A_2 \in \Pi \\ \cdot \mid \Phi^w ; {\Phi_2}^r ; \Psi_2 \vdash \phi_2[\![a^w\ !V]\!] : A_{2@\mathsf{L}} \end{cases}$$
from $\Phi^w ; \Pi \vdash \pi$ okay

$\cdot \mid \Phi^w ; {\Phi_2}^r ; \Psi_2 \vdash a^w\ !V : \mathsf{unit}_{@\mathsf{L}}$ by the rule W! and $a^w \mapsto B' \in \Phi^w$

Let $\Pi = \Pi_1, \gamma_1 : A_1, \gamma_2 : A_2$

From these assumptions, we can prove

$$\Phi^w ; \Pi_1, \gamma_1 : A_1, \gamma_2 : A_2 \vdash \pi_1, \{\phi_1[\![V]\!] \mid \psi_1 \ @\ \gamma_1\}, \{\phi_2[\![()]\!] \mid \psi_2 \ @\ \gamma_2\} \ \mathsf{okay}$$

which simplifies to ${\Phi'}^w ; \Pi' \vdash \pi'$ okay where $\Phi^w = {\Phi'}^w$ and $\Pi = \Pi'$. □

https://doi.org/10.1017/S0956796808006989 Published online by Cambridge University Press