A Capability Calculus for Concurrency and Determinism [⋆]

Tachio Terauchi[1] and Alex Aiken[2]

[1] EECS Department, University of California, Berkeley
[2] Computer Science Department, Stanford University

Abstract. We present a capability calculus for checking partial confluence of channel-communicating concurrent processes. Our approach automatically detects more programs to be partially confluent than previous approaches and is able to handle a mix of different kinds of communication channels, including shared reference cells.

1 Introduction

Deterministic programs are easier to debug and verify than non-deterministic programs, both for testing (or simulation) and for formal methods. However, sometimes programs are written as communicating concurrent processes, for speed or for ease of programming, and therefore are possibly non-deterministic. In this paper, we present a system that can automatically detect more programs to be deterministic than previous methods [7, 10, 8, 9, 5]. Our system is able to handle programs communicating via a mix of different kinds of channels: rendezvous, output buffered, input buffered, and shared reference cells. Section 3.2 shows a few examples that can be checked by our system: producer consumer, token ring, and barrier synchronization. The companion technical report contains the omitted proofs [12].

We cast our system as a *capability calculus* [4]. The capability calculus was originally proposed as a framework for reasoning about resources in sequential computation, but has recently been extended to reason about determinism in concurrent programs [3, 11]. However, these systems can only reason about synchronization at join points, and therefore cannot verify determinism of channel-communicating processes. This paper extends the capability calculus to reason about synchronization due to channel communications. A key insight comes from our previous work [11] which showed that confluence can be ensured in a principled way from ordering dependencies between the side effects; dependencies are enforced by finding a flow assignment (which can be interpreted as *fractional capabilities* [3]) in the dependence graph.

2 Preliminaries

We focus on the simple concurrent language shown in Figure 1. A program, p, is a parallel composition of finitely many processes. A process, s, is a sequential

[⋆] This research was supported in part by NSF Grant No. CCR-0326577.

$$
\begin{array}{rcll}
p & ::= & s_1 || s_2 || \ldots || s_n & (\textit{program}) \\
e & ::= & c & (\textit{channel}) \\
 & | & x & (\textit{local variable}) \\
 & | & n & (\textit{integer constant}) \\
 & | & e_1 \; op \; e_2 & (\textit{integer operation})
\end{array}
\qquad
\begin{array}{rcll}
s & ::= & s_1 ; s_2 & (\textit{sequence}) \\
 & | & \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 & (\textit{branch}) \\
 & | & \texttt{while}\ e\ \texttt{do}\ s & (\textit{loop}) \\
 & | & \texttt{skip} & (\textit{skip}) \\
 & | & x\ \texttt{:=}\ e & (\textit{assignment}) \\
 & | & \texttt{!}(e_1, e_2) & (\textit{write channel}) \\
 & | & \texttt{?}(e, x) & (\textit{read channel})
\end{array}
$$

Fig. 1. The syntax of the small concurrent language.

statement consisting of the usual imperative features as well as channel communication operations. Here, $\texttt{!}(e_1, e_2)$ means writing the value of e_2 to the channel e_1, and $\texttt{?}(e, x)$ means storing the value read from the channel e to the variable x. The variables are process-local, and so the only means of communication are channel reads and writes. We use meta-variables x, y, z, etc. for variables and c, d, etc. for channels.

The language cannot dynamically create channels or spawn new processes, but these restrictions are imposed only to keep the main presentation to the novel features of the system. Section 3.3 shows that techniques similar to previous work in the capability calculus can be used to handle dynamic channels and processes.

2.1 Channel Kinds

The literature on concurrency includes several forms of channels with distinct semantics. We introduce these channel kinds and show how they affect determinism.

If c and d are *rendezvous* channels, then the following program is deterministic[3] because $(x, y) = (1, 2)$ when the process terminates:

$$\texttt{!}(c, 1); \texttt{!}(d, 2) \;||\; \texttt{!}(d, 3); \texttt{?}(c, x) \;||\; \texttt{?}(d, y); \texttt{?}(d, y)$$

The same program is non-deterministic if c is *output buffered* because $\texttt{!}(c, 1)$ does not need to wait for the reader $\texttt{?}(c, x)$, and therefore (x, y) could be $(1, 2)$ or $(1, 3)$.

While all the processes share one output buffer per channel, each process has its own input buffer per channel. Therefore, $\texttt{!}(c, 1); \texttt{!}(c, 2) \;||\; \texttt{?}(c, x) \;||\; \texttt{?}(c, y)$ is deterministic if c is input buffered but not if c is output buffered or rendezvous. Input buffered channels are the basis of Kahn process networks [7].

We also consider a buffered channel whose buffer is overwritten by every write but never modified by a read. Such a channel is equivalent to a reference cell. If c is a reference cell, $\texttt{!}(c, 1); \texttt{!}(c, 2) \;||\; \texttt{?}(c, x)$ is not deterministic because $\texttt{!}(c, 2)$ may or may-not overwrite 1 in the buffer before $\texttt{?}(c, x)$ reads the buffer. The program is deterministic if c is any other channel kind. On the other hand,

[3] Here, we use the term informally. Determinism is formally defined in Section 2.2.

$$\frac{(S(i),e) \Downarrow n \qquad\qquad n \neq 0}{(B,S,i.(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2);s||p) \rightarrow (B,S,i.s_1;s||p)}\ \textbf{IF1}$$

$$\frac{(S(i),e) \Downarrow 0}{(B,S,i.(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2);s||p) \rightarrow (B,S,i.s_2;s||p)}\ \textbf{IF2}$$

$$\frac{(S(i),e) \Downarrow n \qquad\qquad n \neq 0}{(B,S,i.(\texttt{while}\ e\ \texttt{do}\ s_1);s||p) \rightarrow (B,S,i.s_1;(\texttt{while}\ e\ \texttt{do}\ s_1);s||p)}\ \textbf{WHILE1}$$

$$\frac{(S(i),e) \Downarrow 0}{(B,S,i.(\texttt{while}\ e\ \texttt{do}\ s_1);s||p) \rightarrow (B,S,i.s||p)}\ \textbf{WHILE2}$$

$$\frac{(S(i),e) \Downarrow e' \qquad S' = S[i \mapsto S(i) :: (x,e')]}{(B,S,i.x\ \texttt{:=}\ e;s||p) \rightarrow (B,S',i.s||p)}\ \textbf{ASSIGN}$$

$$\frac{\begin{array}{ccc} (S(i),e_1) \Downarrow c & (S(i),e_2) \Downarrow e_2' & (S(j),e_3) \Downarrow c \\ \neg \mathit{buffered}(c) & \multicolumn{2}{c}{S' = S[j \mapsto S(j) :: (x,e_2')]} \end{array}}{(B,S,i.\texttt{!}(e_1,e_2);s_1||j.\texttt{?}(e_3,x);s_2||p) \rightarrow (B,S',i.s_1||j.s_2||p)}\ \textbf{UNBUF}$$

$$\frac{(S(i),e_1) \Downarrow c \quad (S(i),e_2) \Downarrow e_2' \quad \mathit{buffered}(c) \quad B' = B.\mathit{write}(c,e_2')}{(B,S,i.\texttt{!}(e_1,e_2);s||p) \rightarrow (B',S,i.s||p)}\ \textbf{BUF1}$$

$$\frac{\begin{array}{cc} (S(i),e) \Downarrow c & \mathit{buffered}(c) \\ (B',e') = B.\mathit{read}(c,i) & S' = S[i \mapsto S(i) :: (x,e')] \end{array}}{(B,S,i.\texttt{?}(e,x);s||p) \rightarrow (B',S',i.s||p)}\ \textbf{BUF2}$$

Fig. 2. The operational semantics of the small concurrent language.

$\texttt{!}(c,1);\texttt{!}(c,2);\texttt{!}(d,3);\texttt{?}(c,x)\ ||\ \texttt{?}(d,x);\texttt{?}(c,y)$ is deterministic if c is a reference cell and d is rendezvous because both reads of c happen after $\texttt{!}(c,2)$ overwrites the buffer. But the program is not deterministic if c is output buffered.

2.2 Operational Semantics

The operational semantics of the language is defined as a series of reductions from states to states. A state is represented by the triple (B,S,p) where B is a buffer, S is a store, and p is a program such that each concurrent process in p is indexed by a process number, i.e., $p ::= 1.s_1||2.s_2||\ldots||n.s_n$. Indexes are used to connect a process to its input buffer and its store.

A store is a mapping from process indexes to histories of assignments where a *history* is a sequence of pairs (x,e), meaning e was assigned to x. We use meta-variables h, h', etc. for histories. Let :: be append. A lookup in a history is defined as: $(h :: (x,e))(x) = e$ and $(h :: (y,e))(x) = h(x)$ if $y \neq x$. We use history instead of memory for the purpose of defining determinism.

Expressions are evaluated entirely locally. The semantics of expressions are defined as: $(h,c) \Downarrow c$, $(h,x) \Downarrow h(x)$, $(h,n) \Downarrow n$, and $(h,e_1\ op\ e_2) \Downarrow e_1'\ op\ e_2'$ if $(h,e_1) \Downarrow e_1'$ and $(h,e_2) \Downarrow e_2'$.

$$B.\mathit{write}(c,e) = \begin{cases} B[c \mapsto \mathit{enq}(B(c),e)] & \text{if } c \text{ is output buffered} \\ B[c \mapsto \langle \mathit{enq}(q_1,e),\ldots,\mathit{enq}(q_n,e)\rangle] & \text{if } c \text{ is input buffered} \\ \quad \text{where } B(c) = \langle q_1,\ldots,q_n\rangle & \\ B[c \mapsto e] & \text{if } c \text{ is a reference cell} \end{cases}$$

$$B.\mathit{read}(c,i) = \begin{cases} (B[c \mapsto q'], e) & \\ \quad \text{where } B(c) = q \text{ and } (q',e) = \mathit{deq}(q) & \text{if } c \text{ is output buffered} \\ (B[c \mapsto \langle q_1,\ldots,q_i',\ldots,q_n\rangle], e) & \\ \quad \text{where } B(c) = \langle q_1,\ldots,q_i,\ldots,q_n\rangle & \text{if } c \text{ is input buffered} \\ \qquad\qquad (q_i', e) = \mathit{deq}(q_i) & \\ (B, B(c)) & \text{if } c \text{ is a reference cell} \end{cases}$$

Fig. 3. Buffer operations.

Figure 2 shows the reduction rules. Programs are equivalent up to re-ordering of parallel processes, e.g., $p_1||p_2 = p_2||p_1$. If p is an empty program (i.e., p contains 0 processes), then $p'||p = p'$. Also, we let $s = s;\texttt{skip} = \texttt{skip};s$. Note that the rules only reduce the left-most processes, and so we rely on process re-ordering to reduce other processes. The rules **IF1**, **IF2**, **WHILE1**, and **WHILE2** do not involve channel communication and are self-explanatory. **ASSIGN** is also a process-local reduction because variables are local. Here, $S[i \mapsto h]$ means $\{j \mapsto S(j) \mid j \neq i \wedge j \in \mathit{dom}(S)\} \cup \{i \mapsto h\}$. We use the same notation for other mappings.

UNBUF handles communication over rendezvous channels. The predicate $\neg\mathit{buffered}(c)$ says c is unbuffered (and therefore rendezvous). Note that the written value e_2' is immediately transmitted to the reader. **BUF1** and **BUF2** handle communication over buffered channels, which include output buffered channels, input buffered channels, and reference cells. The predicate $\mathit{buffered}(c)$ says that c is a buffered channel. We write $B.\mathit{write}(c, e_2')$ for the buffer B after e_2' is written to the channel c, and $B.\mathit{read}(c,i)$ for the pair (B', e') where e' is the value process i read from channel c and B' is the buffer after the read.

Formally, a buffer B is a mapping from channels to buffer contents. If c is a rendezvous channel, then $B(c) = \mathit{nil}$ indicating that c is not buffered. If c is output buffered, then $B(c) = q$ where q is a FIFO queue of values. If c is input buffered, then $B(c) = \langle q_1, q_2, \ldots, q_n\rangle$, i.e., a sequence of FIFO queues where each q_i represents the buffer content for process i. If c is a reference cell, then $B(c) = e$ for some value e. Let $\mathit{enq}(q,e)$ be q after e is enqueued. Let $\mathit{deq}(q)$ be the pair (q', e) where q' is q after e is dequeued. Buffer writes and reads are defined as shown in Figure 3. Note that $B.\mathit{read}(c,i)$ and $B.\mathit{write}(c,e)$ are undefined if c is rendezvous.

We write $P \rightarrow^* Q$ for 0 or more reduction steps from P to Q. We define partial confluence and determinism.

Definition 1. *Let Y be a set of channels. We say that P is partially confluent with respect to Y if for any $P \rightarrow^* P_1$ communicating only over channels in Y, and for any $P \rightarrow^* P_2$, there exists a state Q such that $P_2 \rightarrow^* Q$ communicating only over channels in Y and $P_1 \rightarrow^* Q$.*

Definition 2. *Let Y be a set of channels. We say that P is deterministic with respect to Y if for each process index i, there exists a (possibly infinite) sequence h_i such that for any $P \rightarrow^* (B, S, p)$ that communicates only over channels in Y, $S(i)$ is a prefix of h_i.*

Determinism implies that for any single process, interaction with the rest of the program is deterministic. Determinism and partial confluence are related in the following way.

Lemma 1. *If P is partially confluent with respect to Y then P is deterministic with respect to Y.*

Note that the definitions are sufficient for programs interacting with the environment because an environment can be modeled as a process using integer operators with unknown (but deterministic) semantics.

3 Calculus of Capabilities

We now present a capability calculus for ensuring partial confluence. While capability calculi are typically presented as a type system in the literature, we take a different approach and present the capability calculus as a dynamic system. We then construct a type system to statically reason about the dynamic capability calculus. This approach allows us to distinguish approximations due to the type abstraction from approximations inherent in the capability concept. (We have taken a similar approach in previous work [11].)

We informally describe the general idea. To simplify matters, we begin this initial discussion with rendezvous channels and total confluence. Given a program, the goal is to ensure that for each channel c, at most one process can write c and at most one process can read c at any point in time. To this end, we introduce capabilities $r(c)$ and $w(c)$ such that a process needs $r(c)$ to read from c and $w(c)$ to write to c. Capabilities are distributed to the processes at the start of the program and are not allowed be duplicated.

Recall the following confluent program from Section 2:

$$1.\texttt{!}(c, 1); \texttt{!}(d, 2) \mid\mid 2.\texttt{!}(d, 3); \texttt{?}(c, x) \mid\mid 3.\texttt{?}(d, y); \texttt{?}(d, y)$$

Note that for both c and d, at most one process can read and at most one process can write at any point in time. However, because both process 1 and process 2 write to d, they must somehow share $w(d)$. A novel feature of our capability calculus is the ability to pass capabilities between processes. The idea is to let capabilities be passed when the two processes synchronize, i.e., when the processes communicate over a channel. In our example, we let process 2 have $w(d)$ at the start of the program. Then, when process 1 and process 2 communicate over c, we pass $w(d)$ from process 2 to process 1 so that process 1 can write to d.

An important observation is that capability passing works in this example because $\texttt{!}(d, 3)$ is guaranteed to occur before the communication on c due to c being rendezvous. If c is buffered (recall that the program is not confluent in this case), then $\texttt{!}(c, 1)$ may occur before $\texttt{!}(d, 3)$. Therefore, process 1 cannot obtain $w(d)$ from process 2 when c is written because process 2 may still need $w(d)$ to write on d. In general, for a buffered channel, while the read is guaranteed to occur after the write, there is no ordering dependency in the other direction, i.e., from the read to the write. Therefore, capabilities can be passed from the writer to the reader but not vice versa, whereas capabilities can be passed in both directions when communicating over a rendezvous channel.

Special care is needed for reference cells. If c is a reference cell, the program $1.\texttt{!}(c, 1); \texttt{!}(c, 2)||2.\texttt{?}(c, x)$ is not deterministic although process 1 is the only writer and process 2 is the only reader. We use *fractional capabilities* [3, 11] such that a read capability is a fraction of the write capability. Capabilities can be split into multiple fractions, which allows concurrent reads on the same reference cell, but must be re-assembled to form the write capability. Fractional capabilities can be passed between processes in the same way as other capabilities. Recall the following confluent program from Section 2 where c is a reference cell and d is rendezvous:

$$1.\texttt{!}(c, 1); \texttt{!}(c, 2); \texttt{!}(d, 3); \texttt{?}(c, x) \mid\mid 2.\texttt{?}(d, x); \texttt{?}(c, y)$$

Process 1 must start with the capability to write c. Because both processes read from c after communicating over d, we split the capability for c such that one half of the capability stays in process 1 and the other half is passed to process 2 via d. As a result, both processes obtain the capability to read from c. We have shown previously that fractional capabilities can be derived in a principled way from ordering dependencies [11].

We now formally present our capability calculus. Let

$$\begin{aligned}\textbf{Capabilities} = &\{w(c), r(c) \mid c \text{ is rendezvous or output buffered}\} \\ &\cup\{w(c) \mid c \text{ is input buffered}\} \cup \{w(c) \mid c \text{ is a reference cell}\}\end{aligned}$$

A *capability set* C is a function from **Capabilities** to rational numbers in the range $[0, 1]$. If c is rendezvous, output buffered, or input buffered, $C(w(c)) = 1$ (resp. $C(r(c)) = 1$) means that the capability to write (resp. read) c is in C. Read capabilities are not needed for input buffered channels because each process has its own buffer. For reference cells, $C(w(c)) = 1$ means that the capability to write is in C, whereas $C(w(c)) > 0$ means that the capability to read is in C. To summarize, we define the following predicates:

$$\begin{aligned}&\mathit{hasWcap}(C, c) \Leftrightarrow C(w(c)) = 1 \\ &\mathit{hasRcap}(C, c) \Leftrightarrow \begin{cases} C(r(c)) = 1 & \text{if } c \text{ is rendezvous or output buffered} \\ \mathit{true} & \text{if } c \text{ is input buffered} \\ C(w(c)) > 0 & \text{if } c \text{ is reference cell} \end{cases}\end{aligned}$$

$$\frac{(S(i), e) \Downarrow n \qquad\qquad n \neq 0}{(X, B, S, i.C.(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2); s||p) \rightarrow (X, B, S, i.C.s_1; s||p)} \textbf{ IF1}'$$

$$\frac{(S(i), e) \Downarrow 0}{(X, B, S, i.C.(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2); s||p) \rightarrow (X, B, S, i.C.s_2; s||p)} \textbf{ IF2}'$$

$$\frac{(S(i), e) \Downarrow n \qquad\qquad n \neq 0}{\begin{array}{l}(X, B, S, i.C.(\texttt{while } e \texttt{ do } s_1); s||p) \\ \qquad \rightarrow (X, B, S, i.C.s_1; (\texttt{while } e \texttt{ do } s_1); s||p)\end{array}} \textbf{ WHILE1}'$$

$$\frac{(S(i), e) \Downarrow 0}{(X, B, S, i.C.(\texttt{while } e \texttt{ do } s_1); s||p) \rightarrow (X, B, S, i.C.s||p)} \textbf{ WHILE2}'$$

$$\frac{(S(i), e) \Downarrow e' \qquad S' = S[i \mapsto S(i) :: (x, e')]}{(X, B, S, i.C.x \texttt{ := } e; s||p) \rightarrow (X, B, S', i.C.s||p)} \textbf{ ASSIGN}'$$

Fig. 4. The capability calculus: sequential reductions.

To denote capability merging and splitting, we define:

$$C_1 + C_2 = \{cap \mapsto C_1(cap) + C_2(cap) \mid cap \in \textbf{Capabilities}\}$$

We define $C_1 - C_2 = C_3$ if $C_1 = C_3 + C_2$. (We avoid negative capabilities.)

Figure 4 and Figure 5 show the reduction rules of the capability calculus. The reduction rules (technically, labeled transition rules) are similar to those of operational semantics with the following differences.

Each concurrent process is prefixed by a capability set C representing the current capabilities held by the process. The rules in Figure 4 do not utilize capabilities (i.e., capabilities are only passed sequentially) and are self-explanatory. Figure 5 shows how capabilities are utilized at communication points. **UNBUF′** sends capabilities C from the writer process to the reader process and sends capabilities C' from the reader process to the writer process. **UNBUF′** checks whether the right capabilities are present by $\mathit{hasWcap}(C_i, c) \wedge \mathit{hasRcap}(C_j, c)$. The label ℓ records whether the check succeeds. Because we are interested in partial confluence with respect to some set Y of channels, we only check the capabilities if $c \in Y$. To this end, the predicate $\mathit{confch}()$ parameterizes the system so that $\mathit{confch}(c)$ iff $c \in Y$.

BUF1′ and **BUF2′** handle buffered communication. Recall that the writer can pass capabilities to the reader. **BUF1′** takes capabilities C' from the writer process and stores them in X. **BUF2′** takes capabilities C' from X and gives them to the reader process. The mapping X from channels to capability sets maintains the capabilities stored in each channel.

We now formally state when our capability calculus guarantees partial confluence. Let $\mathit{erase}((X, B, S, 1.C_1.s_1|| \ldots ||n.C_n.s_n)) = (B, S, 1.s_1|| \ldots ||n.s_n)$, i.e., $\mathit{erase}()$ erases all capability information from the state. We use meta-variables P, Q, R, etc. for states in the operational semantics and underlined meta-variables $\underline{P}$, $\underline{Q}$, $\underline{R}$, etc. for states in the capability calculus.

$$\frac{\begin{array}{ccc} (S(i), e_1) \Downarrow c & (S(i), e_2) \Downarrow e_2' & (S(j), e_3) \Downarrow c \\ \neg \mathit{buffered}(c) & \multicolumn{2}{c}{S' = S[j \mapsto S(j) :: (x, e_2')]} \\ \multicolumn{3}{c}{\ell = (\mathit{confch}(c) \Rightarrow (\mathit{hasWcap}(C_i, c) \wedge \mathit{hasRcap}(C_j, c)))} \end{array}}{\begin{array}{l} (X, B, S, i.C_i.\,!(e_1, e_2); s_1 || j.C_j\,?(e_3, x); s_2 || p) \\ \quad \xrightarrow{\ell} (X, B, S', i.(C_i - C + C').s_1 || j.(C_j + C - C').s_2 || p) \end{array}} \ \textbf{UNBUF}'$$

$$\frac{\begin{array}{ccc} (S(i), e_1) \Downarrow c & (S(i), e_2) \Downarrow e_2' & \mathit{buffered}(c) \\ B' = B.\mathit{write}(c, e_2') & \multicolumn{2}{c}{\ell = (\mathit{confch}(c) \Rightarrow \mathit{hasWcap}(C, c))} \end{array}}{(X, B, S, i.C.\,!(e_1, e_2); s || p) \xrightarrow{\ell} (X[c \mapsto X(c) + C'], B', S, i.(C - C').s || p)} \ \textbf{BUF1}'$$

$$\frac{\begin{array}{ccc} (S(i), e) \Downarrow c & \mathit{buffered}(c) & (B', e') = B.\mathit{read}(c, i) \\ S' = S[i \mapsto S(i) :: (x, e')] & \multicolumn{2}{c}{\ell = (\mathit{confch}(c) \Rightarrow \neg \mathit{hasRcap}(C, c))} \end{array}}{(X, B, S, i.C.?(e, x); s || p) \xrightarrow{\ell} (X[c \mapsto X(c) - C'], B', S', i.(C + C').s || p)} \ \textbf{BUF2}'$$

Fig. 5. The capability calculus: communication reductions.

A *well-formed state* is a state in the capability calculus that does not carry duplicated capabilities. More formally,

Definition 3. *Let* $\underline{P} = (X, B, S, 1.C_1.s_1 || \ldots || n.C_n.s_n)$. *Let* $C = \sum_{i=1}^{n} C_i + \sum_{c \in \mathit{dom}(X)} X(c)$. *We say* $\underline{P}$ *is well-formed if for all* $\mathit{cap} \in \mathit{dom}(C)$, $C(\mathit{cap}) = 1$.

We define *capability-respecting states*. Informally, $\underline{P}$ is capability respecting with respect to a set of channels Y if for any sequence of reductions from $\mathit{erase}(\underline{P})$, there exists a strategy to pass capabilities between the processes such that every communication over the channels in Y occurs under the appropriate capabilities. More formally,

Definition 4. *Let* Y *be a set of channels and let* $\mathit{confch}(c) \Leftrightarrow c \in Y$. *Let* M *be a set of states in the capability calculus.* M *is said to be capability-respecting with respect to* Y *if for any* $\underline{P} \in M$,

– $\underline{P}$ *is well-formed, and*
– *for any state* Q *such that* $\mathit{erase}(\underline{P}) \rightarrow Q$, *there exists* $\underline{Q} \in M$ *such that,* $\mathit{erase}(\underline{Q}) = Q$, $\underline{P} \xrightarrow{\ell} \underline{Q}$, *and if* ℓ *is not empty then* $\ell = \mathit{true}$.

We now state the main claim of this section.

Theorem 1. *Let* P *be a state. Suppose there exists* M *such that* M *is capability-respecting with respect to* Y *and there exists* $\underline{P} \in M$ *such that* $\mathit{erase}(\underline{P}) = P$. *Then* P *is partially confluent with respect to* Y.

3.1 Static Checking of Capabilities

Theorem 1 tells us that to ensure that P is partially confluent, it is sufficient to find a capability-respecting set containing some $\underline{P}$ such that $\mathit{erase}(\underline{P}) = P$. [4]

[4] It is not a necessary condition, however. For example, $!(c, 1) || !(c, 1) || ?(c, x) || ?(c, x)$ is confluent but does not satisfy the condition.

Ideally, we would like to use the largest capability-respecting set, but such a set is not recursive (because it is reducible from the halting problem). Instead, we use a type system to compute a safe approximation of the set.

We define four kinds of channel types, one for each channel kind.

$$\begin{array}{rll} \tau ::= & ch(\rho, \tau, \Psi_1, \Psi_2) & (\textit{rendezvous}) \\ | & ch(\rho, \tau, \Psi) & (\textit{output buffered}) \\ | & ch(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle) & (\textit{input buffered}) \\ | & ch(\rho, \tau) & (\textit{reference cell}) \\ | & \mathit{int} & (\textit{integers}) \end{array}$$

Meta-variables ρ, ρ', etc. are *channel handles*. Let **Handles** be the set of channel handles. Let $\textbf{StaticCapabilities} = \{w(\rho), r(\rho) \mid \rho \in \textbf{Handles}\}$. Meta-variables Ψ, Ψ', etc. are mappings from **StaticCapabilities** to $[0, 1]$. We call such a mapping a *static capability set*. The rendezvous channel type can be read as follows: the channel communicates values of type τ, any writer of the channel sends capabilities Ψ_1, and any reader of the channel sends capabilities Ψ_2. For an output buffered channel, because readers cannot send capabilities, only one static capability set, Ψ, is present in its type. For an input buffered channel, the sequence $\langle \Psi_1, \ldots, \Psi_n \rangle$ lists capabilities such that each process i gets Ψ_i from a read. Because a value stored in a reference cell may be read arbitrarily many times, our type system cannot statically reason about processes passing capabilities through reference cells, and so a reference cell type does not carry any static capability set.

Additions and subtractions of static capabilities are analogous to those of (actual) capabilities:

$$\begin{array}{l} \Psi_1 + \Psi_2 = \{\mathit{cap} \mapsto \Psi_1(\mathit{cap}) + \Psi_2(\mathit{cap}) \mid \mathit{cap} \in \textbf{StaticCapabilities}\} \\ \Psi_1 - \Psi_2 = \Psi_3 \quad \text{if } \Psi_1 = \Psi_3 + \Psi_2 \end{array}$$

We say $\Psi_1 \geq \Psi_2$ if there exists Ψ_3 such that $\Psi_1 = \Psi_2 + \Psi_3$.

For channel type τ, $\mathit{hdl}(\tau)$ is the handle of the channel, and $\mathit{valtype}(\tau)$ is the type of the communicated value. That is, $\mathit{hdl}(ch(\rho, \ldots)) = \rho$ and $\mathit{valtype}(ch(\rho, \tau, \ldots)) = \tau$. Also, $\mathit{writeSend}(\tau)$ (resp. $\mathit{readSend}(\tau)$) is the set of capabilities sent by a writer (resp. reader) of the channel. More formally,

$$\begin{array}{l} \mathit{writeSend}(ch(\rho, \tau, \Psi_1, \Psi_2)) = \Psi_1 \\ \mathit{writeSend}(ch(\rho, \tau, \Psi)) = \Psi \\ \mathit{writeSend}(ch(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle)) = \sum_{i=1}^{n} \Psi_i \\ \mathit{writeSend}(ch(\rho, \tau)) = \mathit{0} \\ \mathit{readSend}(\tau) = \begin{cases} \Psi_2 & \text{if } \tau = ch(\rho, \tau', \Psi_1, \Psi_2) \\ \mathit{0} & \text{otherwise} \end{cases} \end{array}$$

($\mathit{0}$ is the constant zero function $\lambda x.0$.) Similarly, $\mathit{writeRecv}(\tau)$ (resp. $\mathit{readRecv}(\tau, i)$) is the set of capabilities received by the writer (resp. the reader process i):

$$\begin{array}{l} \mathit{writeRecv}(\tau) = \mathit{readSend}(\tau) \\ \mathit{readRecv}(\tau, i) = \begin{cases} \Psi_i & \text{if } \tau = ch(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle) \\ \mathit{writeSend}(\tau) & \text{otherwise} \end{cases} \end{array}$$

$$\frac{\Gamma, i, \Psi \vdash s_1 : \Psi_1 \quad \Gamma, i, \Psi_1 \vdash s_2 : \Psi_2}{\Gamma, i, \Psi \vdash s_1; s_2 : \Psi_2}\ \textbf{SEQ}$$

$$\frac{\Gamma \vdash e : \mathit{int} \quad \Gamma, i, \Psi \vdash s_1 : \Psi_1 \quad \Gamma, i, \Psi \vdash s_2 : \Psi_2 \quad \Psi_1 \geq \Psi_3 \quad \Psi_2 \geq \Psi_3}{\Gamma, i, \Psi \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Psi_3}\ \textbf{IF}$$

$$\frac{\Gamma \vdash e : \mathit{int} \quad \Gamma, i, \Psi_1 \vdash s : \Psi_2 \quad \Psi_2 \geq \Psi_1 \quad \Psi \geq \Psi_1}{\Gamma, i, \Psi \vdash \texttt{while}\ e\ \texttt{do}\ s : \Psi_2}\ \textbf{WHILE}$$

$$\frac{}{\Gamma, i, \Psi \vdash \texttt{skip} : \Psi}\ \textbf{SKIP} \qquad \frac{\Gamma \vdash e : \Gamma(x)}{\Gamma, i, \Psi \vdash x \texttt{ := } e : \Psi}\ \textbf{ASSIGN}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathit{valtype}(\tau) \quad \mathit{confch}(\tau, \Gamma) \Rightarrow \mathit{hasWcap}(\Psi, \tau)}{\Gamma, i, \Psi \vdash \texttt{!}(e_1, e_2) : \Psi - \mathit{writeSend}(\tau) + \mathit{writeRecv}(\tau)}\ \textbf{WRITE}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \mathit{valtype}(\tau) \quad \mathit{confch}(\tau, \Gamma) \Rightarrow \mathit{hasRcap}(\Psi, \tau)}{\Gamma, i, \Psi \vdash \texttt{?}(e, x) : \Psi - \mathit{readSend}(\tau) + \mathit{readRecv}(\tau, i)}\ \textbf{READ}$$

Fig. 6. Type checking rules.

Note that the writer of the input buffered channel $ch(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle)$ must be able to send the sum of all capabilities to be received by each process (i.e., $\sum_{i=1}^{n} \Psi_i$), whereas the reader receives only its own share (i.e., Ψ_i).

For channel type τ, $\mathit{hasWcap}(\Psi, \tau)$ and $\mathit{hasRcap}(\Psi, \tau)$ are the static analog of $\mathit{hasWcap}(C, c)$ and $\mathit{hasRcap}(C, c)$. More formally,

$$\mathit{hasWcap}(\Psi, \tau) \Leftrightarrow \Psi(w(\mathit{hdl}(\tau))) = 1$$

$$\mathit{hasRcap}(\Psi, \tau) \Leftrightarrow \begin{cases} \Psi(r(\mathit{hdl}(\tau))) = 1 & \text{if } \tau \text{ is rendezvous or output buffered} \\ \mathit{true} & \text{if } \tau \text{ is input buffered} \\ \Psi(w(\mathit{hdl}(\tau))) > 0 & \text{if } \tau \text{ is reference cell} \end{cases}$$

A *type environment* Γ is a mapping from channels and variables to types such that for each channel c and d,

– the channel type kind of $\Gamma(c)$ coincides with the channel kind of c, and
– if $c \neq d$ then $\mathit{hdl}(\Gamma(c)) \neq \mathit{hdl}(\Gamma(d))$, i.e., each handle ρ uniquely identifies a channel. (Section 3.3 discusses a way to relax this restriction.)

We sometimes write $\Gamma[c]$ to mean $\mathit{hdl}(\Gamma(c))$.

Expressions are type-checked as follows:

$$\frac{}{\Gamma \vdash c : \Gamma(c)} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \mathit{int}} \qquad \frac{\Gamma \vdash e_1 : \mathit{int} \quad \Gamma \vdash e_2 : \mathit{int}}{\Gamma \vdash e_1\ \mathit{op}\ e_2 : \mathit{int}}$$

Figure 6 shows the type checking rules for statements. The judgments are of the form $\Gamma, i, \Psi \vdash s : \Psi'$ where i is the index of the process where s appears in, Ψ the capabilities before s, and Ψ' the capabilities after s. **SEQ**, **IF**, **WHILE**,

SKIP, and **ASSIGN** are self-explanatory. **WRITE** handles channel writes and **READ** handles channel reads. Here, $\mathit{confch}(\tau, \Gamma)$ is defined as:

$$\mathit{confch}(\tau, \Gamma) \Leftrightarrow \exists c.\Gamma[c] = \mathit{hdl}(\tau) \wedge \mathit{confch}(c)$$

We write $\Gamma \vdash B(c)$ if the buffer $B(c)$ is well-typed, i.e., $\Gamma \vdash e : \mathit{valtype}(\Gamma(c))$ for each value e stored in the buffer $B(c)$. We write $\Gamma \vdash h$ if the history h is well-typed, i.e, $\Gamma \vdash h(x) : \Gamma(x)$ for each $x \in \mathit{dom}(h)$. We write $\Gamma \vdash C : \Psi$ if Ψ represents C, i.e., for each $w(c) \in \mathit{dom}(C)$, $\Psi(w(\Gamma[c])) = C(w(c))$ and for each $r(c) \in \mathit{dom}(C)$, $\Psi(r(\Gamma[c])) = C(r(c))$.

Let $\underline{P} = (B, X, S, 1.C_1.s_1 || \dots || n.C_n.s_n)$. An *environment* for $\underline{P}$ consists of a type environment Γ for typing the channels, a type environment Γ_i for typing each process i, the starting static capability Ψ_i for each process i, and the mapping W from handles to static capabilities that represents X. We say $\underline{P}$ is well-typed under the environment $(\Gamma, \Gamma_1, \dots, \Gamma_n, \Psi_1, \dots, \Psi_n, W)$, written $(\Gamma, \Gamma_1, \dots, \Gamma_n, \Psi_1, \dots, \Psi_n, W) \vdash \underline{P}$, if

- For each c, $\Gamma \vdash B(c)$.
- For each i, $\Gamma_i \supseteq \Gamma$, $\Gamma_i \vdash S(i)$, $\Gamma \vdash C_i : \Psi_i$, and $\Gamma_i, i, \Psi_i \vdash s_i : \Psi'_i$ for some Ψ'_i.
- For each c, $\Gamma \vdash X(c) : W(\Gamma[c])$, i.e., W is a static representation of X.
- Let $\Psi_{total} = \sum_{i=1}^{n} \Psi_i + \sum_{\rho \in \mathit{dom}(W)} W(\rho)$. Then for each $\mathit{cap} \in \mathit{dom}(\Psi_{total})$, $\Psi_{total}(\mathit{cap}) = 1$, i.e., there are no duplicated capabilities.
- For all output buffered channels c, $W(\Gamma[c]) = |B(c)| \times \mathit{writeSend}(\Gamma(c))$. For all input buffered channels c, $W(\Gamma[c]) = \sum_{i=1}^{n} |B(c).i| \times \mathit{readRecv}(\Gamma(c), i)$.

In the last condition, $|B(c)|$ denotes the length of the queue $B(c)$, and $|B(c).i|$ denotes the length of the queue for process i (for input buffered channels). The condition ensures that there are enough capabilities in X for buffered reads. We now state the main claim of this section.

Theorem 2. *Let Y be a set of channels and let $\mathit{confch}(c) \Leftrightarrow c \in Y$. Let $M = \{\underline{P} \mid \exists \mathit{Env}.\mathit{Env} \vdash \underline{P}\}$. Then M is capability-respecting with respect to Y.*

Theorem 2 together with Theorem 1 implies that to check if P is confluent, it suffices to find a well-typed $\underline{P}$ such that $P = \mathit{erase}(\underline{P})$. More formally,

Corollary 1. *Let Y be a set of channels and let $\mathit{confch}(c) \Leftrightarrow c \in Y$. P is partially-confluent and deterministic with respect to Y if there exists $\underline{P}$ and Env such that $P = \mathit{erase}(\underline{P})$ and $\mathit{Env} \vdash \underline{P}$.*

The problem of finding $\underline{P}$ and Env such that $P = \mathit{erase}(\underline{P})$ and $\mathit{Env} \vdash \underline{P}$ can be formulated as linear inequality constraints satisfaction problem. The details are similar to the type inference algorithm from our previous work [11]. The constraints can be generated in time polynomial in the size of P, which can then be solved efficiently by a linear programming algorithm.

3.2 Examples

Producer Consumer: Let c be an output buffered channel. The program $1.\texttt{while } 1 \texttt{ do } \texttt{!}(c,1) \mid\mid 2.\texttt{while } 1 \texttt{ do } \texttt{?}(c,x)$ is a simple but common communication pattern of sender and receiver processes being fixed for each channel; no capabilities need to be passed between processes. The type system can prove confluence by assigning the starting capabilities $\mathit{0}[w(\rho) \mapsto 1]$ to process 1 and $\mathit{0}[r(\rho) \mapsto 1]$ to process 2 where $c : \mathit{ch}(\rho, \mathit{int}, \mathit{0})$.

Token Ring: Let c_1, c_2, c_3 be rendezvous and d be output buffered. The program below models a token ring where processes 1, 2, and 3 take turns writing to d:

$$\begin{array}{l}
1.\texttt{while } 1 \texttt{ do } (\texttt{?}(c_3,x); \texttt{!}(d,1); \texttt{!}(c_1,0)) \\
\mid\mid 2.\texttt{while } 1 \texttt{ do } (\texttt{?}(c_1,x); \texttt{!}(d,2); \texttt{!}(c_2,0)) \\
\mid\mid 3.\texttt{!}(c_3,0); \texttt{while } 1 \texttt{ do } (\texttt{?}(c_2,x); \texttt{!}(d,3); \texttt{!}(c_3,0)) \\
\mid\mid 4.\texttt{while } 1 \texttt{ do } \texttt{?}(d,y)
\end{array}$$

Recall that variables x and y are process local. The type system can prove confluence by assigning the channel d the type $\mathit{ch}(\rho_d, \mathit{int}, \mathit{0})$ and each c_i the type $\mathit{ch}(\rho_{c_i}, \mathit{int}, \mathit{0}[w(\rho_d) \mapsto 1], \mathit{0})$, which says that a write to c_i sends $w(d)$ to the reader. The starting capabilities are $\mathit{0}[r(\rho_{c_3}) \mapsto 1, w(\rho_{c_1}) \mapsto 1]$ for process 1, $\mathit{0}[r(\rho_{c_1}) \mapsto 1, w(\rho_{c_2}) \mapsto 1]$ for process 2, $\mathit{0}[r(\rho_{c_2}) \mapsto 1, w(\rho_{c_3}) \mapsto 1, w(\rho_d) \mapsto 1]$ for process 3, and $\mathit{0}[r(\rho_d) \mapsto 1]$ for process 4.

Barrier Synchronization: Let c_1, c_2, c_3 be reference cells. Let d_1, d_2, d_3, d'_1, d'_2, d'_3 be input buffered channels. Consider the following program:

$$\begin{array}{l}
1.\texttt{while } 1 \texttt{ do } (\texttt{!}(c_1,e_1); \texttt{!}(d_1,0); \mathbf{BR}; \texttt{?}(c_1,y); \texttt{?}(c_2,z); \texttt{?}(c_3,w); \texttt{!}(d'_1,0); \mathbf{BR}') \\
\mid\mid 2.\texttt{while } 1 \texttt{ do } (\texttt{!}(c_2,e_2); \texttt{!}(d_2,0); \mathbf{BR}; \texttt{?}(c_1,y); \texttt{?}(c_2,z); \texttt{?}(c_3,w); \texttt{!}(d'_2,0); \mathbf{BR}') \\
\mid\mid 3.\texttt{while } 1 \texttt{ do } (\texttt{!}(c_3,e_3); \texttt{!}(d_3,0); \mathbf{BR}; \texttt{?}(c_1,y); \texttt{?}(c_2,z); \texttt{?}(c_3,w); \texttt{!}(d'_3,0); \mathbf{BR}')
\end{array}$$

Here, $\mathbf{BR} = \texttt{?}(d_1,x); \texttt{?}(d_2,x); \texttt{?}(d_3,x)$ and $\mathbf{BR}' = \texttt{?}(d'_1,x); \texttt{?}(d'_2,x); \texttt{?}(d'_3,x)$. The program is an example of barrier-style synchronization. Process 1 writes to c_1, process 2 writes to c_2, process 3 writes to c_3, and then the three processes synchronize via a barrier so that none of the processes can proceed until all are done with their writes. Note that $\texttt{!}(d_i,0); \mathbf{BR}$ models the barrier for each process i. After the barrier synchronization, each process reads from all three reference cells before synchronizing themselves via another barrier, this time modeled by $\texttt{!}(d'_i,0); \mathbf{BR}'$, before the next iteration of the loop.

The type system can prove confluence by assigning the following types (assume e_1, e_2, and e_3 are of type *int*): $c_1 : \mathit{ch}(\rho_{c1}, \mathit{int})$, $c_2 : \mathit{ch}(\rho_{c2}, \mathit{int})$, $c_3 : \mathit{ch}(\rho_{c3}, \mathit{int})$, and for each $i \in \{1,2,3\}$,

$$\begin{array}{l}
d_i : \mathit{ch}(\rho_{d_i}, \mathit{int}, \langle \mathit{0}[w(\rho_{c_i}) \mapsto \frac{1}{3}], \mathit{0}[w(\rho_{c_i}) \mapsto \frac{1}{3}], \mathit{0}[w(\rho_{c_i}) \mapsto \frac{1}{3}]\rangle) \\
d'_i : \mathit{ch}(\rho_{d'_i}, \mathit{int}, \langle \mathit{0}[w(\rho_{c_1}) \mapsto \frac{1}{3}], \mathit{0}[w(\rho_{c_2}) \mapsto \frac{1}{3}], \mathit{0}[w(\rho_{c_3}) \mapsto \frac{1}{3}]\rangle)
\end{array}$$

The initial static capability set for each process i is $\mathit{0}[w(\rho_{c_i}) \mapsto 1, w(\rho_{d_i}) \mapsto 1, w(\rho_{d'_i}) \mapsto 1]$. Note that fractional capabilities are passed at barrier synchronization points to enable reads and writes on c_1, c_2, and c_3.

Type inference fails if the program is changed so that d_1, d_2, d_3 are also used for the second barrier (in place of d'_1, d'_2, d'_3) because while the first write to d_i must send the capability to read c_i, the second write to d_i must send to each process j the capability to access c_j, and there is no single type for d_i to express this behavior. This demonstrates the *flow-insensitivity* limitation of our type system, i.e., a channel must send and receive the same capabilities every time it is used.

However, if synchronization points are syntactically identifiable (as in this example) then the program is easily modified so that flow-insensitivity becomes sufficient by using distinct channels at each syntactic synchronization point.[5] In our example, the first barrier in each process matches the other, and the second barrier in each process matches the other. Synchronizations that are not syntactically identifiable are often considered as a sign of potential bugs [1]. Note that reference cells c_1 and c_2 are not used for synchronization and therefore need no syntactic restriction.

3.3 Extensions

We discuss extensions to our system.

Regions: Aliasing becomes an issue when channels are used as values, e.g., like in a π calculus program. For example, our type system does not allow two different channels c and d to be passed to the same channel because two different channels cannot be given the same handle. One way to resolve aliasing is to use *regions* so that each ρ represents a set of channels. Then, we may give both c and d the same type $\mathit{ch}(\rho, \dots)$ at the cost of sharing $w(\rho)$ (and $r(\rho)$) for all the channels in the region ρ.

Existential Abstraction and Linear Types: Another way to resolve aliasing is to existentially abstract capabilities as in $\exists\rho.\tau \otimes \Psi$. Any type containing a capability set must be handled linearly[6] to prevent the duplication of capabilities. The capabilities are recovered by opening the existential package. Existential abstraction can encode linearly typed channels [10, 8] (for rendezvous channels) as: $\exists\rho.\mathit{ch}(\rho, \tau, \mathit{0}, \mathit{0}) \otimes \mathit{0}[w(\rho) \mapsto 1, r(\rho) \mapsto 1]$. Note that the type encapsulates both a channel and the capability to access the channel. This encoding allows transitions to and from linearly typed channels to the capabilities world, e.g., it is possible to use once a linearly-typed channel multiple times. An analogous approach has been applied to express updatable recursive data structures in the capability calculus [13].

Dynamically Created Channels: Dynamically created channels can be handled in much the same way heap allocated objects are handled in the capability

[5] This can be done without changing the implementation. See *named barriers* in [1].

[6] Actually, a more relaxed sub-structural type system is preferred for handling fractional capabilities [11].

calculus [4] (we only show the rule for the case where c is rendezvous):

$$\frac{\rho \text{ is not free in the conclusion} \quad \Gamma \cup \{c \mapsto ch(\rho, \tau, \Psi_1, \Psi_2)\}, i, \Psi + \mathit{0}[w(\rho) \mapsto 1][r(\rho) \mapsto 1] \vdash s : \Psi'}{\Gamma, i, \Psi \vdash \nu c.s : \Psi'}$$

Existential abstraction allows dynamically created channels to leave their lexical scope. An alternative approach is to place the newly created channel in an existing region. In this case, we can remove the hypothesis "ρ is not free in the conclusion", but we also must remove the capabilities $\mathit{0}[w(\rho) \mapsto 1][r(\rho) \mapsto 1]$.

Dynamically Spawned Processes: Dynamic spawning of processes can be typed as follows:

$$\frac{\Gamma, i, \Psi_2 \vdash s : \Psi'}{\Gamma, i, \Psi_1 + \Psi_2 \vdash \texttt{spawn}(s) : \Psi_1}$$

(For simplicity, we assume that the local store of the parent process is copied for the spawned process. Details for handling input buffered channels are omitted.) Note that the spawned process may take capabilities from the parent process.

4 Related Work

We discuss previous approaches to inferring partial confluence. Kahn process networks [7] restrict communication to input buffered channels with a unique sender process to guarantee determinism. Edwards et al. [5] restricts communication to rendezvous channels with a unique sender process and a unique receiver process to model deterministic behavior of embedded systems. These models are the easy cases for our system where capabilities are not passed between processes.

Linear type systems can infer partial confluence by checking that each channel is used at most once [10, 8].[7] Section 3.3 discusses how to express linearly typed channels in our system. König presents a type system that can be parameterized to check partial confluence in the π-calculus [9]. Her system corresponds to the restricted case of our system where each (rendezvous) channel is given a type of the form $ch(\rho, \tau, \mathit{0}[w(\rho) \mapsto 1], \mathit{0}[r(\rho) \mapsto 1])$, i.e., each channel sends its own write capability at writes and sends its own read capability at reads. Therefore, for example, while her system can check the confluence of $\texttt{!}(c, 1); \texttt{?}(c, x) || \texttt{?}(c, x); \texttt{!}(c, 2)$, it cannot check the confluence of $\texttt{!}(c, 1); \texttt{!}(c, 2) || \texttt{?}(c, x); \texttt{?}(c, x)$.

A more exhaustive approach for checking partial confluence has been proposed in which the confluence for every state of the program is individually checked by following the transitions from that state [6, 2]. These methods are designed specifically to drive state space reduction, and hence have somewhat a different aim from our work. They have been shown effective for programs with a small number of states.

This work was motivated by our previous work on inferring confluence in functional languages with side effects [11] (see also [3]). These systems can only reason about synchronization at join points, and therefore cannot infer confluence of channel-communicating processes.

[7] [8] uses asynchronous π calculus, and so is not entirely comparable with our work.

5 Conclusions

We have presented a system for inferring partial confluence of concurrent programs communicating via a mix of different kinds of communication channels. We casted our system as a capability calculus where fractional capabilities can be passed at channel communications, and presented a type system for statically inferring partial confluence by finding an appropriate capability passing strategy in the calculus.

References

1. A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, San Diego, California, Jan. 1998.
2. S. Blom and J. van de Pol. State space reduction by proving confluence. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 596–609, Copenhagen, Denmark, July 2002.
3. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis, Tenth International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, San Diego, CA, June 2003. Springer-Verlag.
4. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, Jan. 1999.
5. S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM International Conference On Embedded Software*, pages 264–272, Jersey City, NJ, Sept. 2005.
6. J. F. Groote and J. van de Pol. State space reduction using partial tau-confluence. In *Proceedings of 25th International Symposium on the Mathematical Foundations of Computer Science 2000*, pages 383–393, Bratislava, Slovakia, Aug. 2000.
7. G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974.
8. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, Sept. 1999.
9. B. König. Analysing input/output-capabilities of mobile processes with a generic type system. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 403–414, Geneva, Switzerland, July 2000.
10. U. Nestmann and M. Steffen. Typing confluence. In *Proceedings of FMICS '97*, pages 77–101, 1997.
11. T. Terauchi and A. Aiken. Witnessing side-effects. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 105–115, Tallinn, Estonia, Sept. 2005. ACM.
12. T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. Technical Report UCB/EECS-2006-84, University of California, Berkeley, 2006.
13. D. Walker and G. Morrisett. Alias types for recursive data structures. In *International Workshop on Types in Compilation*, Montreal, Canada, Sept. 2000.