A Capability Calculus for Concurrency and Determinism

TACHIO TERAUCHI
Tohoku University
and
ALEX AIKEN
Stanford University

This paper presents a static system for checking determinism (technically, partial confluence) of communicating concurrent processes. Our approach automatically detects partial confluence in programs communicating via a mix of different kinds of communication methods: rendezvous channels, buffered channels, broadcast channels, and reference cells. Our system reduces the partial confluence checking problem in polynomial time (in the size of the program) to the problem of solving a system of rational linear inequalities, and is thus efficient.

Categories and Subject Descriptors: F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.3.3 [**Language Classifications**]: Concurrent programming structures

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Determinism, Capabilities, Type Systems

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 automatically proves more such programs to be deterministic than previous methods [Kahn 1974; König 2000; Edwards and Tardieu 2005], and can encode linear type systems which can also be used to guarantee determinism [Nestmann and Steffen 1997; Kobayashi et al. 1999]. Our system is able to handle programs communicating via a mix of different communication methods: rendezvous channels, output buffered channels, input buffered channels (broadcast channels), and reference cells. Section 3.2 shows a few examples that can be checked by our system: producer consumer, token ring, and barrier synchronization.

We cast our system as a *capability calculus* [Crary et al. 1999]. The capability

This is a revised and extended version of a paper presented at the Concurrency Theory, [17th International] Conference, CONCUR 2006 pp.218-232, 2006. This work was supported by KAKENHI 20700019 and 2024001.

calculus was originally proposed as a framework for reasoning about resources in sequential computation and inspired a number of such applications [Smith et al. 2000; DeLine and Fähndrich 2001; Foster et al. 2002]. Recently, the framework has been extended to reason about determinism in concurrent programs [Boyland 2003; Terauchi and Aiken 2005]. A key innovation of these systems is that they allow concurrent imperative operations. However, these systems 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. The problem of finding a flow assignment can be reduced in polynomial time (in the size of the program) to the problem of solving a system of rational linear inequalities, for which many efficient algorithms, such as the simplex algorithm and interior point methods, exist. Therefore, in contrast to the more rigid, set-oriented approach taken in the original capability calculus [Crary et al. 1999], not only are fractional capabilities more flexible, but also more efficient for automatic inference.

This article is a revised and extended version of a conference paper with the same title [Terauchi and Aiken 2006]. The main additions are a stronger notion of determinism which guarantees (among other things) deterministic termination/non-termination, type inference details including a prototype implementation, the $|$ operator, compatibility with a weak consistency memory model, and the proofs.

1.1 Utility of Concurrency and Determinism

Concurrency makes programs faster. For example, in the expression $(a + b) \times (c + d)$, $a + b$ and $c + d$ can be computed in parallel, thus overlapping the time to compute one addition with the other. The importance of concurrency for speed has increased in recent years as many predict that we will soon reach the limits of sequential computation, if we have not already [Sutter and Larus 2005]. Also, concurrency is convenient when program parts must be placed at distant locations, or when the application can be naturally designed as a union of concurrently running components, as often is the case in embedded systems [Edwards and Tardieu 2005]. However, concurrency is difficult to get right. For example, the assignments $x := 1$ and $x := 2$ may not occur in parallel as x could be either 1 or 2 after the assignments.

In current software practice, a predominant style of concurrency is to express programs as sets of communicating processes. Each process runs sequentially at its own speed (i.e., asynchronously[1]) and communicates with other processes via shared resources such as channels, locks, and basic reference cells. Many popular programming models, including message passing and shared memory programming, are derivatives of this general model. The model is popular partly because of its resemblance to sequential programming. Usually, few syntactic changes are needed to extend a sequential language to this concurrent model. Also, the absence of a global synchronizing clock makes the model ideal for situations requiring the program parts to be placed at distant locations.

However, writing bug-free programs in this model is notoriously difficult due to asynchronous accesses to shared resources, which introduce non-determinism.

[1]We use the term in the sense of asynchronous circuits and systems, and not in the sense of asynchronous process algebra [Honda and Tokoro 1992; Boudol 1992].

For this reason, some advocate alternatives to this programming model [Lee 2006]. Instead, we believe that it is possible to keep this convenient model of programming while (mostly) guaranteeing determinism, and we make a step toward this goal in this paper.

Sometimes, notions such as race freedom and atomicity are used to reason about possibly misbehaving concurrency. These properties can enforce, for example, that there is no point in the program where $x := 1$ and $x := 2$ could simultaneously execute. In contrast, determinism is an extensional property that can be stated purely in terms of program semantics, that is, the program behaves semantically the same under the same conditions. Neither race freedom nor atomicity implies determinism, nor vice versa.

Determinism is the norm in sequential programming which dominates the current software practice. We believe most concurrent programs are also expected to behave deterministically. This assumption appears to be reasonable not only for non-interactive applications (i.e., "batch processes") such as scientific computing, but also for interactive applications such as web servers because one would expect a web server, in some state, to behave deterministically given a series of user requests (which could come from multiple users). At the least, non-determinism is not the reason for choosing concurrent over sequential computation.

Non-deterministic programs are difficult to debug because one cannot reproduce the bug by just running the program again from the same initial state. Determinism also helps formal methods. For example, in model checking, determinism implies that there is no need to explore more than one (abstract) program path because following any other paths would lead to an equivalent result [Groote and van de Pol 2000; Blom and van de Pol 2002].

1.2 Utility of Non-Determinism

It seems rare for a real world application to require true non-determinism in the sense that a program's specification requires the program to behave differently from previous executions in an identical environment. However, harmless non-determinism sometimes appears as an artifact of programming convenience.

While this paper makes a step toward making provable deterministic concurrency practical, it may be difficult to completely eliminate non-determinism in reality. Our analysis is preliminary in that it often prohibits harmless instances of non-determinism. For example, internal actions of a program that do not affect the program output are arguably harmless and allowed to be non-deterministic, but our analysis demands determinism even for such actions. Our analysis does allow some degree of non-determinism by letting the user specify the communication channels that could be used for non-deterministic inputs and outputs. We leave as future work understanding and accommodating a wider variety of harmless non-deterministic programming idioms.

2. A SIMPLE CONCURRENT LANGUAGE

We focus on the simple concurrent language shown in Figure 1. A program, p, is a parallel composition of finitely many processes. A process, $i.s$, is a sequential statement s prefixed by a process index i. Process indices are used to connect processes to their input buffers and stores. We drop process indices when it is

$$
\begin{array}{rcll}
p & ::= & i.s & (\textit{process}) \\
 & | & p_1 \,||\, p_2 & (\textit{parallel composition}) \\
e & ::= & c & (\textit{channel}) \\
 & | & x & (\textit{local variable}) \\
 & | & n & (\textit{integer constant}) \\
 & | & e_1 \; op \; e_2 & (\textit{integer operation})
\end{array}
\qquad
\begin{array}{rcll}
s & ::= & s_1; s_2 & (\textit{sequence}) \\
 & | & \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 & (\textit{branch}) \\
 & | & \texttt{while}\ e\ \texttt{do}\ s & (\textit{loop}) \\
 & | & \texttt{skip} & (\textit{skip}) \\
 & | & x\ \texttt{:=}\ e & (\textit{assignment}) \\
 & | & \texttt{!}(e_1, e_2) & (\textit{write channel}) \\
 & | & \texttt{?}(e, x) & (\textit{read channel})
\end{array}
$$

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

convenient to do so. A sequential statement consists 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 4.3 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[2] because $(x, y) = (1, 2)$ when the process terminates. This is because $\texttt{!}(d, 2)$ waits for $\texttt{!}(c, 1)$ to complete, which in turn waits for the reader $\texttt{?}(c, x)$ to be available, and this is only after the write $\texttt{!}(d, 3)$ is completed, and therefore, the first write to d is always 3 and the second write to d is always 2.

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

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 because both read 1 from their input channel, but not if c is output buffered or rendezvous. Input buffered channels are often called broadcast channels, and are the basis of Kahn process networks [Kahn 1974].

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

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

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

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

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

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

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

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

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

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

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

is deterministic if c is any other channel kind. On the other hand,

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

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.

A store is a mapping from process indices 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 $l :: l'$ denote an append of lists l and l' (l and l' can be a singleton list). 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 $(h, e_1 \; op \; e_2) \Downarrow n$ if $(h, e_1) \Downarrow n_1$ and $(h, e_2) \Downarrow n_2$ and $n = n_1 [\![op]\!] n_2$ where $[\![op]\!]$ is the standard semantics of the

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

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

Fig. 3. Buffer operations.

operator *op*.

Figure 2 shows the reduction rules. The sequential composition operator ; is associative. Also, we let $s = s; \texttt{skip} = \texttt{skip}; s$. The parallel composition operator $||$ is commutative and associative, e.g., $p_1 \mid\mid p_2 \mid\mid p_3 = p_2 \mid\mid p_3 \mid\mid p_1$. Note that the rules only reduce the left-most processes, and so we rely on process re-ordering to reduce other processes. We assume that the process indices are disjoint in any program p. 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 $\{j \mapsto S(j) \mid j \neq i \wedge j \in dom(S)\} \cup \{i \mapsto h\}$. We use the same notation for other mappings.

UNBUF handles communication over rendezvous channels. The predicate $\neg buffered(c)$ 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 $buffered(c)$ says that c is a buffered channel. We write $B.write(c,e_2')$ for the buffer B after e_2' is written to the channel c, and $B.read(c,i)$ 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 $B(c) = nil$ 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 $enq(q,e)$ be q after e is enqueued. Let $deq(q)$ 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 $B.read(c,i)$ and $B.write(c,e)$ 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 2.1. 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 2.2. 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,

(1) $S(i)$ is a prefix of h_i, and
(2) if $S(i)$ is shorter than h_i, then there exists (B', S', p') such that $(B, S, p) \rightarrow^* (B', S', p')$ communicating only over channels in Y and $S'(i)$ is longer than $S(i)$.

Note that, because of (1), $S'(i)$ is a prefix of h_i in (2). Determinism implies that for any single process, interaction with the rest of the program is deterministic. Condition (2) also implies that, if some processes deadlock they always deadlock.

Determinism and partial confluence are related in the following way.

LEMMA 2.3. *If P is partially confluent with respect to Y then P is deterministic with respect to Y.*

We leave the proof of this lemma to the appendix.

The definition of partial confluence is borrowed from [Kobayashi et al. 1999]. Because the operational semantics lacks explicit visible behavior, the definition of determinism is somewhat different (if superficially) from the standard definitions used in the literature [Hansen and Valmari 2006; Wang and Kwiatkowska 2006]. However, the definition is sufficient for programs taking inputs or interacting with an environment. An environment can be modelled by designating a process as an "observation process" whose store contents can be observed. Behavior of an environment can be modelled by using operators with unknown (but deterministic) semantics. Then, determinism implies the determinism of the transition system modulo the observables and stutter equivalence, that is, there is no ambiguity in the observation (if the program does not communicate over channels in Y).

Note that if a program taking inputs (in the form of initial configuration or interactively from the environment) is deterministic with respect to Y, then it simply means that it has a deterministic input to output (in the form of the final result of the program or interactively as communication to the environment) relationship when communicating only over channels in Y, i.e., it does not mean that given any input the program always appears to behave identically.

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 [Terauchi and Aiken 2005].)

We informally describe the general idea. To simplify matters, we begin this initial discussion with rendezvous channels and total confluence (i.e., confluence over all channels). 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:

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

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 $!(d,3)$ is guaranteed to occur before the communication on c because channel c is rendezvous. If c is buffered (recall that the program is not confluent in this case), then $!(c,1)$ may occur before $!(d,3)$. 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 $1.!(c,1);!(c,2)||2.?(c,x)$ is not deterministic although process 1 is the only writer and process 2 is the only reader. We use *fractional capabilities* [Boyland 2003; Terauchi and Aiken 2005] 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:

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

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 [Terauchi and Aiken 2005].

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

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

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

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

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

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

Fig. 4. The capability calculus: sequential reductions.

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:

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

To denote capability merging and splitting, we define:

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

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 the 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 and therefore are almost identical to the operational semantics reduction rules. Note that capabilities are not passed between processes. 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 boolean value 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 $c \in Y$.

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

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

$$\frac{\begin{array}{c}(S(i), e_1) \Downarrow c \quad (S(i), e_2) \Downarrow e_2' \quad \mathit{buffered}(c) \\ S \quad B' = B.\mathit{write}(c, e_2') \\ \ell = (c \in Y \Rightarrow \mathit{hasWcap}(C, c)) \quad C' = 0 \text{ if } c \text{ is a reference cell}\end{array}}{(X, B, S, i.C.!(e_1, e_2); s || p) \xrightarrow{\ell} (X[c \mapsto X(c) + C'], B', S, i.(C - C').s || p)} \textbf{BUF1'}$$

$$\frac{\begin{array}{c}(S(i), e) \Downarrow c \quad \mathit{buffered}(c) \\ (B', e') = B.\mathit{read}(c, i) \quad S' = S[i \mapsto S(i) :: (x, e')] \\ \ell = (c \in Y \Rightarrow \neg \mathit{hasRcap}(C, c)) \quad C' = 0 \text{ if } c \text{ is a reference cell}\end{array}}{(X, B, S, i.C.?(e, x); s || p) \xrightarrow{\ell} (X[c \mapsto X(c) - C'], B', S', i.(C + C').s || p)} \textbf{BUF2'}$$

Fig. 5. The capability calculus: communication reductions.

capabilities stored in each channel. Due to difficulty in reasoning statically and compatibility with realistic memory models, our system does not allow passing capabilities through reference cells (see Section 4.2 for the details).

We now formally state when our capability calculus guarantees partial confluence. Let $\mathit{erase}(X, B, S, i_1.C_1.s_1 || \ldots || i_n.C_n.s_n) = (B, S, i_1.s_1 || \ldots || i_n.s_n)$, 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.

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

Definition 3.1. Let $\underline{P} = (X, B, S, i_1.C_1.s_1 || \ldots || i_n.C_n.s_n)$. Let $C = \sum_{i=1}^{n} C_i + \sum_{c \in dom(X)} X(c)$. 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 3.2. Let Y be a set of channels and 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 3.3. *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.

The theorem is a straightforward application of the following lemma.

LEMMA 3.4. *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 $erase(\underline{P}) = P$. Suppose $P \rightarrow P_1$ communicating only over channels in Y and $P \rightarrow P_2$. Then either $P_1 = P_2$ or there exists Q such that $P_2 \rightarrow Q$ communicating only over channels in Y and $P_1 \rightarrow Q$.*

We leave the proof of this lemma to the appendix. We now prove Theorem 3.3.

PROOF. Note that for any $\underline{P} \in M$, if $erase(\underline{P}) \rightarrow^* Q$ then there exists $\underline{Q} \in M$ such that $erase(\underline{Q}) = Q$. Therefore, the proof is a standard proof by induction using Lemma 3.4. □

3.1 Static Checking of Capabilities

Theorem 3.3 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 $erase(\underline{P}) = P$. [3] 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.

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

Meta-variables ρ, ρ', etc. are *channel handles*. Let **Handles** be the set of channel handles. Let **StaticCapabilities** $= \{w(\rho), r(\rho) \mid \rho \in \textbf{Handles}\}$. Meta-variables Ψ, Ψ', etc. are mappings from some subset of **StaticCapabilities** to $[0, 1]$. We call such a mapping a *static capability set*. If $cap \notin dom(\Psi)$, then we let $\Psi(cap) = 0$ for all $cap \in$ **StaticCapabilities**. 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. Recall that our capability calculus does not allow passing of capabilities via reference cells. Therefore, a reference cell type does not carry any static capability set.

Addition and subtraction of static capabilities is analogous to those of (actual) capabilities:

$$
\begin{array}{l}
\Psi_1 + \Psi_2 = \{cap \mapsto \Psi_1(cap) + \Psi_2(cap) \mid cap \in dom\,\Psi_1 \cup dom\,\Psi_2\} \\
\Psi_1 - \Psi_2 = \Psi_3 \quad \text{if } \Psi_1 = \Psi_3 + \Psi_2
\end{array}
$$

[3] 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.

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

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

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

(0 is equivalent to the constant zero function $\lambda x.0$, i.e., $0(cap) = 0$ for all $cap \in$ **Capabilities**.) Similarly, $writeRecv(\tau)$ (resp. $readRecv(\tau, i)$) is the set of capabilities received by the writer (resp. the reader process i):

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

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, $hasWcap(\Psi, \tau)$ and $hasRcap(\Psi, \tau)$ are the static analog of $hasWcap(C, c)$ and $hasRcap(C, c)$. More formally,

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

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 $hdl(\Gamma(c)) \neq hdl(\Gamma(d))$, i.e., each handle ρ uniquely identifies a channel. (Section 4.3 discusses a way to relax this restriction.)

We sometimes write $\Gamma[c]$ to mean $hdl(\Gamma(c))$. Expressions are type-checked as follows:

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

Figure 6 shows the type checking rules for statements. The judgements are of the form $\Gamma, i, \Psi \vdash s : \Psi'$ where i is the index of the process that 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, $confch(\tau, \Gamma)$ is defined as:

$$confch(\tau, \Gamma) \Leftrightarrow \exists c.(\Gamma[c] = hdl(\tau) \wedge c \in Y)$$

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

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

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

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

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

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

Fig. 6. Type checking rules.

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 $r(c) \in \mathit{dom}(C)$, $\Psi(r(\Gamma[c])) = C(r(c))$.

An *environment* for $\underline{P}$ consists of a type environment Γ for typing the channels and variables, the starting static capability Ψ_i for each process i, and the mapping W from handles to static capabilities representing X. Because variables are process local, without the loss of generality, we assume that each process uses a disjoint set of variables.

Definition 3.5. Let $\underline{P} = (X, B, S, i_1.C_1.s_1 || \ldots || i_n.C_n.s_n)$. We write $(\Gamma, \Psi_1, \ldots, \Psi_n, W)\underline{P}$ if

(1) For each c, $\Gamma \vdash B(c)$.
(2) For each i, $\Gamma \vdash S(i)$, $\Gamma \vdash C_i : \Psi_i$, and $\Gamma, i, \Psi_i \vdash s_i : \Psi_i'$ for some Ψ_i'.
(3) For each c, $\Gamma \vdash X(c) : W(\Gamma[c])$, i.e., W is a static representation of X.
(4) Let $\Psi_{total} = \sum_{i=1}^{n} \Psi_i + \sum_{\rho \in dom(W)} W(\rho)$. Then for each $\mathit{cap} \in \mathit{dom}(\Psi_{total})$, $\Psi_{total}(\mathit{cap}) = 1$, i.e., there are no duplicated capabilities.
(5) 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 3.6. *Let* $M = \{\underline{P} \mid \exists \mathit{Env}.\mathit{Env} \vdash \underline{P}\}$. *Then* M *is capability-respecting with respect to* Y.

We leave the proof of this theorem to the appendix. The theorem is a kind of a progress theorem, that is, reducing a well-typed state (i.e., a state in M) results in a well-typed state. In addition, we need to show that the reduction happens under the appropriate capabilities to show that the state is capability-respecting. Theorem 3.6 together with Theorem 3.3 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 3.7. *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 $\mathit{Env} \vdash \underline{P}$.*

Note that while the theorem implies that the static capability calculus is sound for the dynamic capability calculus, it is not complete. An instance of incompleteness is flow-insensitivity of capability-passing discussed in Section 3.2. In general, the problem whether P is (dynamically) capability respecting with respect to Y is undecidable and so any decidable static system is necessarily incomplete.

3.2 Examples

Producer-Consumer:. Let c be an output buffered channel. The following program 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.

$$1.\texttt{while}\ 1\ \texttt{do}\ \texttt{!}(c, 1)\ ||\ 2.\texttt{while}\ 1\ \texttt{do}\ \texttt{?}(c, x)$$

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:

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

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:

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

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 $!(d_i, 0); \mathbf{BR}$ 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 modelled by $!(d'_i, 0); \mathbf{BR}'$, 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*): $c_1 : ch(\rho_{c1}, int)$, $c_2 : ch(\rho_{c2}, int)$, $c_3 : ch(\rho_{c3}, int)$, and for each $i \in \{1, 2, 3\}$,

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

The initial static capability set for each process i is $0[w(\rho_{c_i}) \mapsto 1, w(\rho_{d_i}) \mapsto 1, w(\rho_{d'_i}) \mapsto 1]$. 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. The modified program is still (dynamically) capability respecting because there is no such limitation to the dynamic capability system.

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.[4] 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 [Aiken and Gay 1998]. Note that reference cells c_1 and c_2 are not used for synchronization and therefore need no syntactic restriction.

4. DISCUSSION

4.1 Type Inference

The problem of finding $\underline{P}$ and *Env* such that $P = erase(\underline{P})$ and $Env \vdash \underline{P}$ can be translated to the problem of solving a system of rational linear inequalities. The translation is divided into two phases. In the first phase, we look for a valid type derivation while ignoring static capabilities. More formally, we look for $\underline{P'}$ and Env' such that $P = erase(\underline{P'})$ and $Env' \vdash \underline{P'}$ by assuming $\forall \Psi, \tau. hasWcap(\Psi, \tau) = hasRcap(\Psi, \tau) = true$. This is a simple type inference problem that can be solved, for example, by a standard union-find based approach. (Note that it suffices to let each $\Psi = 0$ and each $C = 0$.) That is, we instantiate types appearing during the type derivation by type variables α's, and try to find a satisfying assignment that satisfies the set of constraints of the form $\alpha = ch(\rho, \sigma, \ldots)$ and $\alpha = int$ where the type language σ is defined as $\sigma ::= \alpha \mid int \mid ch(\rho, \sigma, \ldots)$. If the constraints

[4]This can be done without changing the implementation. See *named barriers* in [Aiken and Gay 1998].

cannot be satisfied at this phase, it indicates that P fails to satisfy a basic type safety property (e.g., trying to use an integer typed value as a channel) or contains channel handle aliases (see Section 4.3 for extensions to resolve this issue).

In the second phase, we use the channel handles computed in the first phase to complete the typing. More formally, we look for $\underline{P}$ and Env such that $P = erase(\underline{P})$, $Env \vdash \underline{P}$, and $\forall c.\Gamma[c] = \Gamma'[c]$ where $Env = (\Gamma, \Psi_1, \ldots, \Psi_n, W)$ and $Env' = (\Gamma', \Psi'_1, \ldots, \Psi'_n, W')$. Because every $hdl(\tau)$ appearing in the type derivation is known, the problem can be solved by finding a satisfying assignments (for Ψ's) for the constraints of the form $\sum \Psi \geq \sum \Psi'$, $\Psi(\rho) = 1$, and $\Psi(\rho) > 0$. (The latter two forms are induced by $hasWcap(\Psi, \tau)$ and $hasRcap(\Psi, \tau)$.) Let $\rho_1, \ldots, \rho_n$ be the channel handles appearing in Γ'. We represent each Ψ as a mapping from $\rho_1, \ldots, \rho_n$ to n fresh rational number variables $q_1, \ldots q_n$ such that $\Psi(\rho_i) = q_i$. Then, a capability constraint $\sum \Psi \geq \sum \Psi'$ can be represented by n rational inequality constraints $\sum q_1 \geq \sum q'_1, \ldots, \sum q_n \geq \sum q'_n$. Similarly, $\Psi(\rho) = 1$ becomes a constraint of the form $q = 1$ where $\Psi(\rho) = q$, and $\Psi(\rho) > 0$ becomes a constraint of the form $q > 0$ where $\Psi(\rho) = q$. Because the range of a static capability is $[0, 1]$, we also assert $1 \geq q \geq 0$ for each q. These rational inequality constraints can be generated in time polynomial in the size of P (in contrast to the number of states of P, which is often exponential or worse in the size of P for software programs), which can then be solved efficiently by linear programming algorithms, such as the simplex algorithm and interior point methods. Note the importance of being able to use rational numbers not just for reference cells but for all channel kinds, which allows us to reduce the problem into a system of rational linear inequalities instead of, say, a harder problem like integer programming. In fact, it is possible to show that the type inference problem is NP-hard if we restrict capabilities to just integers instead of rationals [Kobayashi 2007].

THEOREM 4.1. *Suppose the type system is altered such that for any capability set Ψ appearing in the type derivation, $\Psi(\rho) \in \{0, 1\}$ for all ρ. Then the set $\{P \mid \exists Env, \underline{P}. Env \vdash \underline{P} \wedge P = erase(\underline{P})\}$ is NP-hard.*

This two phased approach is similar to the type inference algorithm in our previous work [Terauchi and Aiken 2005], only simpler thanks to the absence of multiplication (which can actually be eliminated in quadratic time due to the structure of the constraints generated and thus is not a significant problem).

Megacz has implemented a prototype of the type inference algorithm [Megacz 2006] for the purpose of verifying determinism of FLEET programs. FLEET is an experimental computer architecture with a high degree of asynchronous concurrency [Coates et al. 2001]. The prototype implementation was also able to check all of the examples used in the paper.

4.2 Memory Model Issues

Both the operational semantics (Section 2.2) and the dynamic capability calculus (Section 3) use the *strict consistency* memory model where every operation on the buffer is serialized. While we do not give a formal proof, our system works even with a *weak consistency* memory model for reference cells. Weak consistency allows non-synchronizing operations to be unserialized, and therefore is often considered more efficient than strict consistency for distributed computer architectures. Our

system does not consider a reference cell access to be a synchronizing operation (and thus does not allow capability passing via reference cells). And therefore, our system does not require strict consistency for reference cells. Here is the intuitive justification: the only issue is when one process, say i, writes to a reference cell while another process, say j, tries to read from it. But then the process i must have capability $= 1$ for the reference cell, and so the the process j cannot read it because it does not have the read capability. Therefore, the fact that capabilities can only be passed via synchronizing communication is enough to guarantee soundness even if reference cells are only weakly consistent.

As in the standard implementation of a weak consistency memory model, reference writes can be carried out locally such that changes are not necessarily broadcast before the next synchronization operation. In fact, the inferred capability sets from the type system give a conservative approximation of which writes must be broadcast by the time next synchronizing operation happens. Such information may prove useful to reducing the communication cost in weak consistency memory model implementations.

Note that our system is of course sound even with a strict memory model that could synchronize all reference cell operations. Indeed, the conference paper [Terauchi and Aiken 2006] uses such a dynamic system.

4.3 Extensions

We discuss extensions to our system which include coping with aliasing, encoding linear types, and handling dynamic channel and process creation.

Regions. Aliasing becomes an issue when channels are used as values, e.g., as in a π-calculus program (indeed, π-calculus is so minimalistic that it requires channel passing to express even the most basic programming idioms such as reference cells and integers), though fortunately, this seems rare in applications such as FLEET programs [Coates et al. 2001] and embedded systems [Edwards and Tardieu 2005]. 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 $ch(\rho, \ldots)$ at the cost of sharing $w(\rho)$ (and $r(\rho)$) for all the channels in the region ρ. There are two approaches to regions. One is to use a conservative may-alias analysis to automatically infer regions. Another approach is to let the programmer explicitly manage regions. The former may be more convenient, but the latter may give more control to the programmer when combined with expressive quantified and recursive types [Crary et al. 1999; Walker and Morrisett 2000]

Existential Abstraction and Linear Types. Another way to resolve aliasing is to existentially abstract capabilities as $\exists\rho.\tau \otimes \Psi$. Existentially packing a capability set results in subtracting the capabilities from the capabilities of the process packing them. The capabilities are recovered by opening the existential package, i.e., the packed capabilities are added to the capabilities of the process that opens them.

Existential types are equivalent up to renaming of bound channel handles, i.e., $\exists\rho.\tau \otimes \Psi = \exists\rho'.\tau[\rho'/\rho] \otimes \Psi[\rho'/\rho]$ when $\rho' \notin \mathit{freehandles}(\exists\rho.\tau \otimes \Psi)$ where $\mathit{freehandles}(\tau)$ is the set of free channel handles of τ. Following our previous

work [Terauchi and Aiken 2005], any type containing a capability set must be handled carefully to prevent the duplication of capabilities. Therefore, we extend the addition relation we used for capabilities to types as shown below.

$$\frac{}{\exists\rho.\tau_1 \otimes \Psi_1 + \exists\rho.\tau_2 \otimes \Psi_2 = \exists\rho.(\tau_1 + \tau_2) \otimes (\Psi_1 + \Psi_2)}$$

$$\frac{\tau = \mathit{int}}{\tau + \tau = \tau} \quad \frac{\tau = \mathit{ch}(\rho, \tau', \Psi_1, \Psi_2)}{\tau + \tau = \tau} \quad \frac{\tau = \mathit{ch}(\rho, \tau', \Psi)}{\tau + \tau = \tau}$$

$$\frac{\tau + \tau' = \tau}{\mathit{ch}(\rho, \tau) + \mathit{ch}(\rho, \tau') = \mathit{ch}(\rho, \tau)}$$

$$\frac{\tau + \tau' = \tau}{\mathit{ch}(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle) + \mathit{ch}(\rho, \tau', \langle \Psi_1, \ldots, \Psi_n \rangle) = \mathit{ch}(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle)}$$

$\tau_1 + \tau_2$ is undefined otherwise.

We can encode a linearly typed channel [Nestmann and Steffen 1997; Kobayashi et al. 1999] as: $\hat{ch}(\tau) = \exists\rho.\mathit{ch}(\rho, \tau, 0, 0) \otimes 0[w(\rho) \mapsto 1, r(\rho) \mapsto 1]$ (for rendezvous channels). Note that the type encapsulates both the channel and the capability to access the channel. This reduces the aliasing problem because all linear (rendezvous) channels communicating a value of the type τ can be given that same type $\hat{ch}(\tau)$. If necessary, we can split the type into the reader type $?\hat{ch}(\tau) = \exists\rho.\mathit{ch}(\rho, \tau, 0, 0) \otimes 0[r(\rho) \mapsto 1]$ and the writer type $!\hat{ch}(\tau) = \exists\rho.\mathit{ch}(\rho, \tau, 0, 0) \otimes 0[w(\rho) \mapsto 1]$. Note that the reader type and the writer type add up to the linear channel type, i.e., $\hat{ch}(\tau) = ?\hat{ch}(\tau) + !\hat{ch}(\tau)$. Furthermore, this encoding allows transitions to and from linearly typed channels to the capabilities world, e.g., it is possible to use a linearly-typed channel multiple times. An analogous approach has been applied to express updatable recursive data structures in the capability calculus [Walker and Morrisett 2000].

Dynamically Created Channels. Dynamically created channels can be handled in much the same way heap allocated objects are handled in the capability calculus [Crary et al. 1999] (we only show the rule for the case where c is rendezvous). We borrow the syntax from π-calculus: $\nu c.s$ creates a new channel named c, and then executes s.

$$\frac{\rho \notin \mathit{freehandles}(\Psi) \cup \mathit{freehandles}(\Psi') \cup \mathit{freehandles}(\Gamma) \quad \Gamma, c : \mathit{ch}(\rho, \tau, \Psi_1, \Psi_2), i, \Psi + 0[w(\rho) \mapsto 1][r(\rho) \mapsto 1] \vdash s : \Psi'}{\Gamma, i, \Psi \vdash \nu c.s : \Psi'}$$

where $\mathit{freehandles}(\Gamma) = \bigcup_{\tau \in ran(\Gamma)} \mathit{freehandles}(\tau)$. 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 $\rho \notin \mathit{freehandles}(\Psi) \cup \mathit{freehandles}(\Psi') \cup \mathit{freehandles}(\Gamma)$ but we also must remove the capabilities $0[w(\rho) \mapsto 1][r(\rho) \mapsto 1]$.

Dynamically Spawned Processes. Dynamic spawning of processes can be typed as follows. (For simplicity, we assume that the local store of the parent process is

copied for the spawned process.)

$$\frac{\Gamma, m, \Psi_2 \vdash s : \Psi' \quad m \notin \{1, \ldots, n\}}{\Gamma, i, \Psi_1 + \Psi_2 \vdash \texttt{spawn}(s) : \Psi_1}$$

where n is the number of static (i.e., not dynamically spawned) processes. We update $\mathit{readRecv}(\tau, i)$ as follows.

$$\mathit{readRecv}(\tau, i) = \begin{cases} \Psi_i & \text{if } \tau = \mathit{ch}(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle) \\ 0 & \text{if } \tau = \mathit{ch}(\rho, \tau, \langle \Psi_1, \ldots, \Psi_n \rangle) \wedge i \notin \{1, \ldots, n\} \\ \mathit{writeSend}(\tau) & \text{otherwise} \end{cases}$$

Note that the premise $m \notin \{1, \ldots, n\}$ in the type rule is sufficient because $\mathit{readRecv}(\tau, i) = 0$ for any $i \notin \{1, \ldots, n\}$.

Note that the spawned process may take capabilities from the parent process.

We can also express the classical | operator which combines process spawning and process joining such that $s; (s_1|s_2); s'$ means do s, then do s_1 and s_2 in parallel, then do s'.

$$\frac{\Gamma, m, \Psi_1 \vdash s_1 : \Psi_1' \quad \Gamma, i, \Psi_2 \vdash s_2 : \Psi_2' \quad m \notin \{1, \ldots, n\}}{\Gamma, i, \Psi_1 + \Psi_2 \vdash s_1|s_2 : \Psi_1' + \Psi_2'}$$

Again, we assume that the local store of the parent process is copied for the spawned processes.

With dynamic channel creation, it is possible to model | with `spawn`() and sequential composition.

$$s_1|s_2 = \nu c.\texttt{spawn}(s_1; \texttt{!}(c, 0)); s_2; \texttt{?}(c, x)$$

where x is a variable that does not appear anywhere else in the program. Note that the type derivation with this encoding matches the type rule for |. (Below, $\Gamma' = \Gamma, c : \mathit{ch}(\rho, \mathit{int}, \Psi_1', 0)$).

$$\frac{\dfrac{\Gamma', i, \Psi_1 + \Psi_2 + 0[w(\rho) \mapsto 1][r(\rho) \mapsto 1] \vdash \texttt{spawn}(s_1; \texttt{!}(c, 0)) : \Psi_2 + 0[r(\rho) \mapsto 1] \quad \Gamma', i, \Psi_2 + 0[r(\rho) \mapsto 1] \vdash s_2; \texttt{?}(c, x) : \Psi_1' + \Psi_2' \quad m \notin \{1, \ldots, n\}}{\Gamma', i, \Psi_1 + \Psi_2 \vdash \texttt{spawn}(s_1; \texttt{!}(c, 0)); s_2; \texttt{?}(c, x) : \Psi_1' + \Psi_2'}}{\Gamma, i, \Psi_1 + \Psi_2 \vdash \nu c.\texttt{spawn}(s_1; \texttt{!}(c, 0)); s_2; \texttt{?}(c, x) : \Psi_1' + \Psi_2'}$$

where $\Gamma, m, \Psi_1 \vdash s_1 : \Psi_1'$ for $m \notin \{1, \ldots, n\}$ and $\Gamma, i, \Psi_2 \vdash s_2 : \Psi_2'$.

4.4 Limitations

We discuss some limitations of our work. As mentioned in Section 4.1, the dynamic capability calculus is undecidable[5], and so any sound static approximation for the dynamic capability calculus would be incomplete.

One source of incompleteness is the flow-insensitivity of capability passing mentioned Section 3.2. Precisely, this means that a channel can only be used to transmit the same capabilities regardless of when the communication happens. In practice,

[5]This is easy to see via a reduction from the halting problem, e.g., for an arbitrary sequence code s that does not contain c, $1.s; \texttt{!}(c, 1) \parallel 2.\texttt{!}(c, 2)$ is capability respecting with respect to c iff s does not halt.

this implies that a channel used to synchronize processes sharing resource in one way cannot be used to synchronize them in another way. For instance, if an output buffered channel is used to allow a certain shared resource used by process 1 to be used by process 2 temporarily, then the same output buffered channel cannot be used by process 2 to give the capability to access the resource to some other process.

Other sources of incompleteness are quite standard for a typical (non-dependent) type-based analysis, e.g., our analysis is insensitive to branch conditions (and results of arithmetic expressions in general), and, so for example,

$$1.\texttt{if}\ 0\ \texttt{then}\ !(c,0)\ \texttt{else}\ \texttt{skip}\ ||\ 2.!(c,1)$$

would be reported to be non-deterministic.

Another source of incompleteness is due to the incompleteness of the dynamic capability calculus relative to partial confluence. For example, even the dynamic capability calculus is insensitive to the integer values sent across a channel, and so $1.!(c,0)\ ||\ 2.!(c,0)$ where c is a buffered channel is not capability respecting with respect to c even though it is partially confluent with respect to c. Also, the capability calculus cannot reason about any determinism relying on commutativity of arithmetic operations.

Finally, the criteria that every aspect of the program must be deterministic may be too strict for some real world applications.

We do not currently have the experience with realistic applications that would allow us to judge how serious these limitations are in practice. Assuming some or all of these issues needed to be addressed, We believe that it should be possible to combine our analysis with more powerful (and perhaps more expensive) techniques such as state exploration based analyses, and relax the capability requirements to allow some degree of non-determinism when it is harmless.

5. RELATED WORK

We discuss previous approaches to checking and inferring determinism of communicating concurrent processes. Kahn process networks [Kahn 1974] restrict communication to input buffered channels with a unique sender process to guarantee determinism. Edwards et al. [Edwards and Tardieu 2005] 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 [Nestmann and Steffen 1997; Kobayashi et al. 1999]. Section 4.3 discusses how to express linearly typed channels in our system. A fundamental difference between these "one-use-per-channel" approaches and our approach is that our system allows a channel to be used multiple times, just that at any point in time, only one process can write to it (and read from it, for a rendezvous or output buffered channel).

König presents a type system that can be parameterized to check partial confluence of π-calculus programs [König 2000]. König's system corresponds to the restricted case of our system where each (rendezvous) channel is given a type of the form $ch(\rho, \tau, 0[w(\rho) \mapsto 1], 0[r(\rho) \mapsto 1])$, i.e., each channel must

send its own write capability at writes and must send its own read capability at reads. Therefore, for example, while their system can check the confluence of $!(c,1);?(c,x)||?(c,x);!(c,2)$, it cannot check the confluence of $!(c,1);!(c,2)||?(c,x);?(c,x)$.

The literature on process algebra has popularized the asynchronous π-calculus [Boudol 1992; Honda and Tokoro 1992], which is often studied in close conjunction with linear type systems [Kobayashi et al. 1999; Yoshida et al. 2004]. Here, the term asynchronous is different from the notion we have been using in this paper. (The processes in vanilla π-calculus are asynchronous in the sense of asynchronous circuits and systems because they do not run in lock step.) Essentially, the asynchronous π-calculus is π-calculus with the following restriction: a write to a channel cannot be sequentially followed by an action (i.e., $!(e,e');s$ is not allowed). Because the asynchronous π-calculus is a subset of the (full) π-calculus, and our system can handle the fragment of π-calculus without the non-deterministic choice ($+$) operator, the fragment of the asynchronous π-calculus without $+$ can be handled by our system. In fact, our system is quite powerful even in this restricted setting. For example, our approach can check that the following program is confluent.

$$!(c,1)||?(d,x);!(c,2)||?(c,y);(!(d,0)|?(c,y))$$

More exhaustive approaches for checking determinism and confluence have been proposed in which the determinism and confluence are checked by explicitly exploring the program states [Groote and van de Pol 2000; Blom and van de Pol 2002; Hansen and Valmari 2006]. These approaches are precise, and are language independent and thus potentially work with any model of concurrent computation. However, an issue with these approaches is that their worst-case running time is at least linear in the number of run-time states of the program, which can be much larger than the size of the program (indeed, it is common for a software program to have a super-exponential number of states), whereas our algorithm can run in time polynomial in the size of the program. Recently, a compositional approach has been proposed which may alleviate the problem [Wang and Kwiatkowska 2006]. Also, these methods have a somewhat different aim from our work because they are designed specifically to drive state space reduction instead of just checking determinism, i.e., in some sense, the price for exhaustive state space exploration is paid by state space reduction.

This work was motivated by our previous work on inferring confluence of functional languages with global mutable states [Terauchi and Aiken 2005] (see also [Boyland 2003]). These systems can only reason about synchronization at join points, and therefore cannot infer confluence of channel-communicating processes. These systems were in turn motivated by the original capability calculus system [Crary et al. 1999], which was designed to reason about resource usage in sequential programs. Whereas the original capability calculus uses rather rigid and complex set-oriented rules for manipulating (e.g., splitting and joining) capabilities, fractional capabilities are more flexible and cheaper to reason with because capability manipulations are based on rational linear arithmetic.

We note that the idea that capabilities can be passed at channel communication points is analogous to ideas underlying various type systems for channel-communicating processes [Kobayashi et al. 1995; Gordon and Jeffrey 2001; 2002;

Igarashi and Kobayashi 2004], for various applications ranging from authentication protocol to deadlock freedom.

6. CONCLUSIONS

We have presented a system for inferring partial confluence of concurrent programs communicating via a mix of different kinds of communication methods. We have cast our system as a capability calculus where fractional capabilities can be passed at synchronizing channel communications, and have presented a type system for statically inferring partial confluence by finding an appropriate capability passing strategy in the calculus, which can be reduced in polynomial time to the problem of solving a system of rational linear inequalities.

7. ACKNOWLEDGEMENTS

We thank Adam Megacz for implementing a type inference prototype [Megacz 2006] as well as suggesting the | operator extension (Section 4.3), a feature missing from the conference paper [Terauchi and Aiken 2006] that was needed for expressing some FLEET [Coates et al. 2001] programs. We also thank Naoki Kobayashi for useful discussions, especially his observation that allowing rationals for every kind of capability provably decreases the computational complexity of type inference, assuming P $\neq$ NP.

REFERENCES

2002. *Types and Effects for Asymmetric Cryptographic Protocols*. IEEE Computer Society, Cape Breton, Nova Scotia, Canada.

Aiken, A. and Gay, D. 1998. Barrier inference. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, 342–354.

Blom, S. and van de Pol, J. 2002. State space reduction by proving confluence. In *Proceedings of the 14th International Conference on Computer Aided Verification*. Copenhagen, Denmark, 596–609.

Boudol, G. 1992. Asynchrony and the pi-calculus. Tech. Rep. 1702, INRIA Sophia Antipolis. May.

Boyland, J. 2003. Checking interference with fractional permissions. In *Static Analysis, Tenth International Symposium*. San Diego, CA, 55–72.

Coates, W. S., Lexau, J. K., Jones, I. W., Fairbanks, S. M., and Sutherland, I. E. 2001. Fleetzero: An asynchronous switching experiment. In *7th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2001)*. IEEE Computer Society, Salt Lake City, UT, 173–.

Crary, K., Walker, D., and Morrisett, G. 1999. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas, 262–275.

DeLine, R. and Fähndrich, M. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah, 59–69.

Edwards, S. A. and Tardieu, O. 2005. Shim: a deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM International Conference On Embedded Software*. Jersey City, NJ, 264–272.

Foster, J. S., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany.

Gordon, A. D. and Jeffrey, A. 2001. Typing correspondence assertions for communication protocols. *Theoretical Computer Science 45*.

Groote, J. F. and van de Pol, J. 2000. State space reduction using partial tau-confluence. In *Proceedings of 25th International Symposium on the Mathematical Foundations of Computer Science 2000*. Bratislava, Slovakia, 383–393.

Hansen, H. and Valmari, A. 2006. Operational determinism and fast algorithms. In *CONCUR 2006 - Concurrency Theory, 17th International Conference*. Vol. 4137. Springer, Bonn, Germany, 188–202.

Honda, K. and Tokoro, M. 1992. On asynchronous communication semantics. In *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*. Springer-Verlag, 21–51.

Igarashi, A. and Kobayashi, N. 2004. A generic type system for the pi-calculus. *Theoretical Compututer Science 311,* 1-3, 121–163.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In *Information processing*. Stockholm, Sweden, 471–475.

Kobayashi, N. 2007. Personal communication.

Kobayashi, N., Nakade, M., and Yonezawa, A. 1995. Static analysis of communication for asynchronous concurrent programming languages. In *Static Analysis, Second International Symposium*. Glasgow, Scotland, 225–242.

Kobayashi, N., Pierce, B. C., and Turner, D. N. 1999. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems 21,* 5 (Sept.), 914–947.

König, B. 2000. Analysing input/output-capabilities of mobile processes with a generic type system. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*. Geneva, Switzerland, 403–414.

Lee, E. A. 2006. The problem with threads. Tech. Rep. UCB/EECS-2006-1, EECS Department, University of California, Berkeley. January 10.

Megacz, A. 2006. CCCD implementation. http://research.cs.berkeley.edu/project/cccd-impl/README.

Nestmann, U. and Steffen, M. 1997. Typing confluence. In *Proceedings of FMICS '97*. 77–101.

Smith, F., Walker, D., and Morrisett, G. 2000. Alias Types. In *9th European Symposium on Programming*, G. Smolka, Ed. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, Berlin, Germany, 366–381.

Sutter, H. and Larus, J. 2005. Software and the concurrency revolution. *Queue 3,* 7, 54–62.

Terauchi, T. and Aiken, A. 2005. Witnessing side-effects. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*. ACM, Tallinn, Estonia, 105–115.

Terauchi, T. and Aiken, A. 2006. A capability calculus for concurrency and determinism. In *CONCUR 2006 - Concurrency Theory, 17th International Conference*. Vol. 4137. Springer, Bonn, Germany, 218–232.

Walker, D. and Morrisett, G. 2000. Alias Types for Recursive Data Structures. In *International Workshop on Types in Compilation*. Montreal, Canada.

Wang, X. and Kwiatkowska, M. 2006. Compositional state space reduction using untangled actions. In *EXPRESS'06 13th International Workshop on Expressiveness in Concurrency*. Bonn, Germany, 16–28.

Yoshida, N., Berger, M., and Honda, K. 2004. Strong normalisation in the pi -calculus. *Information and Computation 191,* 2, 145–202.

A. PROOFS

Lemma A.1. *Suppose* $(B, S, p) \rightarrow^* (B', S', p')$ *and* $i \in dom(S)$. *Then* $S(i)$ *is a prefix of* $S'(i)$.

Proof. By induction on the reduction sequence. □

LEMMA 2.3. *If P is partially confluent with respect to Y then P is deterministic with respect to Y.*

PROOF. Suppose P is partially confluent with respect to Y. We show (1). Let i be a process index. For contradiction, suppose that there exist reductions $P \rightarrow^* (B_1, S_1, p_1)$ and $P \rightarrow^* (B_2, S_2, p_2)$ both communicating only over channels in Y such that neither $S_1(i)$ nor $S_2(i)$ is a prefix of the other. But because P is partially confluent, there exists Q such that $(B_1, S_1, p_1) \rightarrow^* Q$ and $(B_2, S_2, p_2) \rightarrow^* Q$. But by Lemma A.1, such a Q cannot exist.

We now show (2). For contradiction, suppose (2) does not hold. Then it must be the case that there exist reductions $P \rightarrow^* (B_1, S_1, p_1)$ and $P \rightarrow^* (B_2, S_2, p_2)$ both communicating only over channels in Y such that $S_1(i)$ is shorter than $S_2(i)$ (and so $S_1(i)$ is a prefix of $S_2(i)$ because of (1)) and there exists no (B_1', S_1', p_1') such that $(B_1, S_1, p_1) \rightarrow^* (B_1', S_1', p_1')$ communicating only over channels in Y and $S_1'(i)$ is longer than $S_1(i)$. But because P is partially confluent, there exists Q such that $(B_1, S_1, p_1) \rightarrow^* Q$ and $(B_2, S_2, p_2) \rightarrow^* Q$ communicating only over channels in Y. But then Q is such a (B_1', S_1', p_1'). □

LEMMA 3.4. *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 $erase(\underline{P}) = P$. Suppose $P \rightarrow P_1$ communicating only over channels in Y and $P \rightarrow P_2$. Then either $P_1 = P_2$ or there exists Q such that $P_2 \rightarrow Q$ communicating only over channels in Y and $P_1 \rightarrow Q$.*

PROOF. We prove the result by case analysis on $P \rightarrow P_1$. Note that because $\underline{P} \in M$, there exist $\underline{P_1}$ and $\underline{P_2}$ such that

—$erase(\underline{P_1}) = P_1$ and $erase(\underline{P_2}) = P_2$, and

—$\underline{P} \xrightarrow{\ell_1} \underline{P_1}$ and $\underline{P} \xrightarrow{\ell_2} \underline{P_2}$ and neither ℓ_1 nor ℓ_2 is *false*.

The case where $P \rightarrow P_1$ is a sequential reduction (i.e., **IF1**, **IF2**, **WHILE1**, **WHILE2**, or **ASSIGN**) is trivial because a sequential reduction affects only one process and does not interfere with other processes. Suppose $P \rightarrow P_1$ is:

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

where $P = (B, S, i.!(e_1, e_2); s_1 || j.?(e_3, x); s_2 || p)$ and $P_1 = (B, S', i.s_1 || j.s_2 || p)$. By assumption, $c \in Y$. If $P \rightarrow P_2$ does not communicate over c, then the result follows trivially. So suppose $P \rightarrow P_2$ communicates over c. Let i' and j' be the process indices such that $\underline{P} \xrightarrow{\ell_2} \underline{P_2}$ is between the writer process i' and the reader process j'. But because $\ell_1 = \ell_2 = \mathit{true}$, $\underline{P}$ being well-formed implies that $i = i'$ and $j = j'$. Therefore, $P_1 = P_2$.

Suppose $P \rightarrow P_1$ is:

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

where $P = (B, S, i.!(e_1, e_2); s||p)$ and $P_1 = (B', S, i.s||p)$. Suppose c is output buffered or input buffered. By the assumption, $c \in Y$. If $P \rightarrow P_2$ does not communicate over c, then the result follows trivially. Also, if $P \rightarrow P_2$ is a read from c, then because buffers are FIFO, the result follows trivially. So suppose $P \rightarrow P_2$ writes c. Let i' be the process indices such that $\underline{P} \stackrel{\ell_2}{\rightarrow} \underline{P_2}$ is a write by process i'. But because $\ell_1 = \ell_2 = \mathit{true}$, $\underline{P}$ being well-formed implies that $i = i'$. Therefore, $P_1 = P_2$.

The case where $P \rightarrow P_1$ reads an output buffered channel is similar. The case where $P \rightarrow P_1$ reads an input buffered channel follows trivially from the fact that input buffers are process local.

Suppose c is a reference cell. By assumption, $c \in Y$. If $P \rightarrow P_2$ does not communicate over c, then the result follows trivially. So suppose $P \rightarrow P_2$ communicates over c (read or write). Let i' be the process indices such that $\underline{P} \stackrel{\ell_2}{\rightarrow} \underline{P_2}$ is a write or a read by process i'. But because $\ell_1 = \ell_2 = \mathit{true}$, $\underline{P}$ being well-formed implies that $i = i'$ (so $P \rightarrow P_2$ is in fact a write). Therefore, $P_1 = P_2$.

Suppose $P \rightarrow P_1$ is:

$$\frac{(S(i), e) \Downarrow c \quad \mathit{buffered}(c) \quad (B', e') = B.\mathit{read}(c, i) \quad S' = S[i \mapsto S(i) :: (x, e')]}{(B, S, i.?(e, x); s||p) \rightarrow (B', S', i.s||p)}$$

where $P = (B, S, i.?(e, x); s||p)$ and $P_1 = (B', S', i.s||p)$. Suppose c is a reference cell. By assumption, $c \in Y$. If $P \rightarrow P_2$ does not communicate over c, then the result follows trivially. Also, if $P \rightarrow P_2$ is a read from c then because reading a reference cell is non-destructive, the result follows trivially. So suppose $P \rightarrow P_2$ writes to c. But because $\ell_1 = \ell_2 = \mathit{true}$, $\underline{P}$ being well-formed implies that $P \rightarrow P_2$ is, in fact, not a write to c. □

LEMMA A.2. *If* $\Gamma, i, \Psi_1 \vdash s : \Psi_2$ *and* $\Psi_1' \geq \Psi_1$, *then* $\Gamma, i, \Psi_1' \vdash s : \Psi_2'$ *for some* $\Psi_2' \geq \Psi_2$

PROOF. By structural induction on the type derivation. □

LEMMA A.3. *If* $\Gamma \vdash h$, $\Gamma \vdash e : \tau$, *and* $(h, e) \Downarrow e'$, *then* $\Gamma \vdash e' : \tau$.

PROOF. By structural induction on the type derivation. □

LEMMA A.4. $\Gamma, i, \Psi \vdash (s_1; s_2); s_3 : \Psi'$ *iff* $\Gamma, i, \Psi \vdash s_1; (s_2; s_3) : \Psi'$.

PROOF. By inspection of the type checking rules. □

THEOREM 3.6. *Let* $M = \{\underline{P} \mid \exists \mathit{Env}.\mathit{Env} \vdash \underline{P}\}$. *Then* M *is capability-respecting with respect to* Y.

PROOF. It suffices to show that if $\mathit{Env} \vdash \underline{P}$, then

(1) $\underline{P}$ is well-formed, and
(2) for any state Q such that $\mathit{erase}(\underline{P}) \rightarrow Q$, there exists $\underline{Q}$ and Env' such that $\mathit{Env}' \vdash \underline{Q}$, $\mathit{erase}(\underline{Q}) = Q$, $\underline{P} \stackrel{\ell}{\rightarrow} \underline{Q}$, and if ℓ is not empty then $\ell = \mathit{true}$.

Suppose $\mathit{Env} \vdash \underline{P}$. From the second, the third, and the fourth conditions of $\mathit{Env} \vdash \underline{P}$, it follows that $\underline{P}$ is well-formed. Thus it suffices to show that (2) holds.

Let Q be a state such that $\mathit{erase}(\underline{P}) \rightarrow Q$. Let $P = \mathit{erase}(\underline{P})$. Let $(\Gamma, \Psi_1, \ldots, \Psi_n, W) = \mathit{Env}$. We show that there exist $\underline{Q}$, $\Psi'_1, \ldots, \Psi'_n$, and W' such that $(\Gamma, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$, $\underline{P} \xrightarrow{\ell} \underline{Q}$, and if ℓ is not empty then $\ell = \mathit{true}$. (So in fact, Env and Env' share the same Γ, indicating the flow-insensitivity of our system.) We prove by case analysis on $P \rightarrow Q$.

Throughout this proof, we implicitly use Lemma A.4 to convert a type derivation for any sequential composition $s_1; s_2; \ldots; s_n$ to a derivation for $s_1; (s_2; \ldots; s_n)$. Also, note that typability is invariant under process re-ordering and sequential `skip` compositions (i.e., $s = s; \texttt{skip} = \texttt{skip}; s$).

Suppose $P \rightarrow Q$ is

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

where $P = (B, S, i.(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2); s||p)$ and $Q = (B, S, i.s_1; s||p)$. Let $(X, B, S, i.C_i.((\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2); s||p') = \underline{P}$. Let $\underline{Q} = (X, B, S, i.C_i.s_1; s||p')$. Note that $\mathit{erase}(\underline{Q}) = Q$ and $\underline{P} \rightarrow \underline{Q}$. By assumption,

$$\frac{\Gamma \vdash e : \mathit{int} \quad \Gamma, i, \Psi_i \vdash s_1 : \Psi_{i1} \quad \Gamma, i, \Psi_i \vdash s_2 : \Psi_{i2} \quad \Psi_{i1} \geq \Psi_{i3} \quad \Psi_{i2} \geq \Psi_{i3}}{\Gamma, i, \Psi_i \vdash \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 : \Psi_{i3}}$$

$$\frac{\Gamma, i, \Psi_i \vdash \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 : \Psi_{i3} \quad \Gamma, i, \Psi_{i3} \vdash s : \Psi_{i4}}{\Gamma, i, \Psi_i \vdash (\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2); s : \Psi_{i4}}$$

Therefore, by Lemma A.2, $\Gamma, i, \Psi_{i1} \vdash s : \Psi_{i5}$ for some Ψ_{i5}. And so, $\Gamma, i, \Psi_i \vdash s_1; s : \Psi_{i5}$. Let $\Psi'_j = \Psi_j$ for each $j \in \{1 \ldots n\}$ and $W' = W$. Then it follows that $(\Gamma, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

The cases where $P \rightarrow Q$ is **IF2**, **WHILE1**, or **WHILE2** can be proven in a similar manner.

Suppose $P \rightarrow Q$ is

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

where $P = (B, S, i.x := e; s||p)$ and $Q = (B, S', i.s||p)$. Let $(X, B, S, i.C_i.x := e; s||p') = \underline{P}$. Let $\underline{Q} = (X, B, S', i.C_i.s||p')$. Note that $\mathit{erase}(\underline{Q}) = Q$ and $\underline{P} \rightarrow \underline{Q}$. By assumption,

$$\frac{\Gamma \vdash e : \Gamma(x)}{\Gamma, i, \Psi_i \vdash x := e : \Psi_i}$$

$$\frac{\Gamma, i, \Psi_i \vdash x := e : \Psi_i \quad \Gamma, i, \Psi_i \vdash s : \Psi_{i1}}{\Gamma, i, \Psi_i \vdash x := e; s : \Psi_{i1}}$$

From $\Gamma \vdash S(i)$, $\Gamma \vdash e : \Gamma(x)$, and Lemma A.3, it follows that $\Gamma \vdash S'(i)$. Let $\Psi'_j = \Psi_j$ for each $j \in \{1, \ldots, n\}$ and $W' = nW$. Then it follows that $(\Gamma, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

Suppose $P \rightarrow Q$ is

$$\frac{\begin{array}{c}(S(i), e_1) \Downarrow c \quad (S(i), e_2) \Downarrow e'_2 \quad (S(j), e_3) \Downarrow c \\ \neg \mathit{buffered}(c) \quad S' = S[j \mapsto S(j) :: (x, e'_2)]\end{array}}{(B, S, i.!(e_1, e_2); s_1||j.?(e_3, x); s_2||p) \rightarrow (B, S', i.s_1||j.s_2||p)}$$

where $P = (B, S, i.!(e_1, e_2); s_1||j.?(e_3, x); s_2||p)$ and $Q = (B, S', i.s_1||j.s_2||p)$. Let $(X, B, S, i.C_i.!(e_1, e_2); s_1||j.C_j.?(e_3, x); s_2||p') = \underline{P}$. By assumption,

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathit{valtype}(\tau) \quad \mathit{confch}(\tau, \Gamma) \Rightarrow \mathit{hasWcap}(\Psi_i, \tau)}{\Gamma, i, \Psi_i \vdash !(e_1, e_2) : \Psi_{i1}}$$

$$\frac{\Gamma, i, \Psi_i \vdash !(e_1, e_2) : \Psi_{i1} \quad \Gamma, i, \Psi_{i1} \vdash s : \Psi_{i2}}{\Gamma, i, \Psi_i \vdash !(e_1, e_2); s : \Psi_{i2}}$$

$$\frac{\Gamma \vdash e_3 : \tau' \quad \Gamma(x) = \mathit{valtype}(\tau') \quad \mathit{confch}(\tau', \Gamma) \Rightarrow \mathit{hasRcap}(\Psi_j, \tau')}{\Gamma, j, \Psi_j \vdash ?(e_3, x) : \Psi_{j1}}$$

$$\frac{\Gamma, j, \Psi_j \vdash ?(e_3, x) : \Psi_{j1} \quad \Gamma, j, \Psi_{j1} \vdash s : \Psi_{j2}}{\Gamma, j, \Psi_j \vdash ?(e_3, x); s : \Psi_{j2}}$$

where $\Psi_{i1} = \Psi_i - \mathit{writeSend}(\tau) + \mathit{writeRecv}(\tau)$ and $\Psi_{j1} = \Psi_j - \mathit{readSend}(\tau') + \mathit{readRecv}(\tau', j)$. Lemma A.3 implies that $\Gamma(c) = \tau = \tau'$. Let C and C' be capability sets such that $\Gamma \vdash C : \mathit{writeSend}(\tau)$ and $\Gamma \vdash C' : \mathit{readSend}(\tau)$. Let $\underline{Q} = (B, S', i.(C_i - C + C').s_1||j.(C_j + C - C').s_2||p)$. Note that $\mathit{erase}(\underline{Q}) = Q$. Let $\Psi'_i = \Psi_{i1}$, $\Psi'_j = \Psi_{j1}$, and $\Psi'_k = \Psi_k$ for each $k \in \{1, \ldots, n\} \setminus \{i, j\}$. Because c must be rendezvous, $\Psi'_i + \Psi'_j = \Psi_i + \Psi_j$. Also, $\Gamma \vdash (C_i - C + C') : \Psi'_i$ and $\Gamma \vdash (C_j + C - C') : \Psi'_j$. Therefore it follows that $(\Gamma, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

Suppose $c \in Y$. Then $\mathit{confch}(\tau, \Gamma)$ and $\mathit{confch}(\tau', \Gamma)$, and so $\mathit{hasWcap}(\Psi_i, \tau)$ and $\mathit{hasRcap}(\Psi_j, \tau')$. Therefore, $\mathit{hasWcap}(C_i, c)$ and $\mathit{hasRcap}(C_j, c)$. Thus it follows that $\underline{P} \xrightarrow{\mathit{true}} \underline{Q}$. On the other hand, if $c \notin Y$, then $\underline{P} \xrightarrow{\mathit{true}} \underline{Q}$ trivially.

Suppose $P \to Q$ is

$$\frac{(S(i), e_1) \Downarrow c \quad (S(i), e_2) \Downarrow e'_2 \quad \mathit{buffered}(c) \quad B' = B.\mathit{write}(c, e'_2)}{(B, S, i.!(e_1, e_2); s||p) \to (B', S, i.s||p)}$$

where $P = (B, S, i.!(e_1, e_2); s||p)$ and $Q = (B', S, i.s||p)$. Let $(X, B, S, i.C.!(e_1, e_2); s||p) = \underline{P}$. By assumption,

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathit{valtype}(\tau) \quad \mathit{confch}(\tau, \Gamma) \Rightarrow \mathit{hasWcap}(\Psi_i, \tau)}{\Gamma, i, \Psi_i \vdash !(e_1, e_2) : \Psi_{i1}}$$

$$\frac{\Gamma, i, \Psi_i \vdash !(e_1, e_2) : \Psi_{i1} \quad \Gamma, i, \Psi_{i1} \vdash s : \Psi_{i2}}{\Gamma, i, \Psi_i \vdash !(e_1, e_2); s : \Psi_{i2}}$$

where $\Psi_{i1} = \Psi_i - \mathit{writeSend}(\tau) + \mathit{writeRecv}(\tau)$, and Γ and Ψ_i are from the definition of $\Gamma \vdash \underline{P}$. Let C' be a capability set such that $\Gamma \vdash C' : \mathit{writeSend}(\tau)$. Let $\underline{Q} = (X[c \mapsto X(c) + C'], B', S, i.(C - C').s||p)$. Note that $\mathit{erase}(\underline{Q}) = Q$. Let $W' = W[\Gamma[c] \mapsto W(\Gamma[c]) + \mathit{writeSend}(\tau)]$. Note that $\forall d.\Gamma \vdash X[c \mapsto X(c) + C'](d) : W'(\Gamma[d])$. Lemma A.3 implies that $\Gamma(c) = \tau$. Let $\Psi'_i = \Psi_{i1}$ and $\Psi'_j = \Psi_j$ for each $j \in \{1, \ldots, n\} \setminus \{i\}$. Because c is not rendezvous, $\mathit{writeRecv}(\tau) = 0$. Therefore, $\Psi_i + \sum_{\rho \in dom(W)} W(\rho) = \Psi'_i + \sum_{\rho \in dom(W')} W'(\rho)$. Also, $\Gamma \vdash (C_i - C) : \Psi'_i$. Also, for each d output buffered, $W'(\Gamma[d]) = |B'(d)| \times \mathit{writeSend}(\Gamma(d))$, and for each d

input buffered, $W'(\Gamma[d]) = \sum_{j=1}^{n} |B'(d).j| \times \mathit{readRecv}(\Gamma(d), j)$. Therefore it follows that $(\Gamma, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

Suppose $c \in Y$. Then $\mathit{confch}(\tau, \Gamma)$, and so $\mathit{hasWcap}(\Psi_i, \tau)$. Therefore, $\mathit{hasWcap}(C, c)$. Thus it follows that $\underline{P} \overset{\mathit{true}}{\rightarrow} \underline{Q}$. On the other hand, if $c \notin Y$, then $\underline{P} \overset{\mathit{true}}{\rightarrow} \underline{Q}$ trivially.

Suppose $P \rightarrow Q$ is

$$\frac{(S(i), e) \Downarrow c \quad \mathit{buffered}(c) \quad (B', e') = B.\mathit{read}(c, i) \quad S' = S[i \mapsto S(i) :: (x, e')]}{(B, S, i.?(e, x); s||p) \rightarrow (B', S', i.s||p)}$$

where $P = (B, S, i.?(e, x); s||p)$ and $Q = (B', S', i.s||p)$. Let $(X, B, S, i.C.?(e, x); s||p) = \underline{P}$. By assumption,

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \mathit{valtype}(\tau) \quad \mathit{confch}(\tau, \Gamma) \Rightarrow \mathit{hasRcap}(\Psi_i, \tau)}{\Gamma, i, \Psi_i \vdash ?(e, x) : \Psi_{i1}}$$

$$\frac{\Gamma, i, \Psi_i \vdash ?(e, x) : \Psi_{i1} \quad \Gamma, i, \Psi_{i1} \vdash s : \Psi_{i2}}{\Gamma, i, \Psi_i \vdash ?(e, x); s : \Psi_{i2}}$$

where $\Psi_{i1} = \Psi_i - \mathit{readSend}(\tau) + \mathit{readRecv}(\tau, i)$. Let C' be a capability set such that $\Gamma \vdash C' : \mathit{readRecv}(\tau, i)$. Let $\underline{Q} = (X[c \mapsto X(c) - C'], B', S', i.(C + C').s||p)$. Note that $\mathit{erase}(\underline{Q}) = Q$. Let $W' = W[\Gamma[c] \mapsto W(\Gamma[c]) - \mathit{readRecv}(\tau, i)]$. Note that $\forall d.\Gamma \vdash X[c \mapsto X(c) - C'](d) : W'(\Gamma[d])$. Lemma A.3 implies that $\Gamma(c) = \tau$. Let $\Psi'_i = \Psi_{i1}$ and $\Psi'_j = \Psi_j$ for each $j \in \{1, \ldots, n\} \setminus \{i\}$. Because c is not rendezvous, $\mathit{readSend}(\tau) = 0$. Therefore, $\Psi_i + \sum_{\rho \in dom(W)} W(\rho) = \Psi'_i + \sum_{\rho \in dom(W')} W'(\rho)$. Also, for each d output buffered, $W'(\Gamma[d]) = |B'(d)| \times \mathit{writeSend}(\Gamma(d))$, and for each d input buffered, $W'(\Gamma[d]) = \sum_{j=1}^{n} |B'(d).j| \times \mathit{readRecv}(\Gamma(d), j)$. Therefore it follows that $(\Gamma, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

Suppose $c \in Y$. Then $\mathit{confch}(\tau, \Gamma)$, and so $\mathit{hasRcap}(\Psi_i, \tau)$. Therefore, $\mathit{hasRcap}(C, c)$. Thus it follows that $\underline{P} \overset{\mathit{true}}{\rightarrow} \underline{Q}$. On the other hand, if $c \notin Y$, then $\underline{P} \overset{\mathit{true}}{\rightarrow} \underline{Q}$ trivially. □

THEOREM 4.1. *Suppose the type system is altered such that for any capability set Ψ appearing in the type derivation, $\Psi(\rho) \in \{0, 1\}$ for all ρ. Then the set $\{P \mid \exists.\mathit{Env}, \underline{P}.\mathit{Env} \vdash \underline{P} \wedge P = \mathit{erase}(\underline{P})\}$ is NP-hard.*

PROOF. We prove the result by reduction from one-in-three 3-SAT, a restricted version of 3-SAT where a satisfying assignment must set exactly one literal in each clause (instead of at least one) to 1. One-in-three 3-SAT is known to be NP-complete.

Let $\phi = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ where each clause c_i is of the form $l_1 \vee l_2 \vee l_3$ where each l_i is a literal, i.e., either a variable or a negation of a variable. Let $\{a_1, \ldots, a_n\}$ be the set of variables appearing in ϕ. For each variable a_i, let a_i and $\overline{a_i}$ be distinct output buffered channels, representing the literals a and $\neg a$, respectively. Let *truech* be a distinct output buffered channel.

Let p be the following program consisting of $n+1$ processes:

$$\begin{array}{rcl} p & = & !(\mathit{truech}, 0); !(a_1, 0); !(\overline{a_1}, 0) \\ & || & ?(a_1, x); ?(\overline{a_1}, x); !(\mathit{truech}, 0); !(a_2, 0); !(\overline{a_2}, 0) \\ & || & \ldots || ?(a_{n-1}, x); ?(\overline{a_{n-1}}, x); !(\mathit{truech}, 0); !(a_n, 0); !(\overline{a_n}, 0) \\ & || & ?(a_n, x); ?(\overline{a_n}, x); !(\mathit{truech}, 0) \end{array}$$

For each literal l, let l^* denote its representative channel, i.e., for a variable a, $a* = a$ and $(\neg a)^* = \overline{a}$. For each clause $c = l_1 \vee l_2 \vee l_3$, let p_c be the process defined as follows.

$$p_c = ?(l_1^*, x); ?(l_2^*, x)?(l_3^*, x); !(\mathit{truech}, 0)$$

Let $p_\phi = p \,||\, p_{c_1} \,||\, p_{c_2} \,||\, \ldots \,||\, p_{c_m}$. Let $P = (B, \{x \mapsto 0\}, p_\phi)$ where $B(c)$ is an empty queue for all channels c. We claim that ϕ is one-in-three 3SAT satisfiable iff there exists Env and $\underline{P}$ such that $\mathit{Env} \vdash \underline{P}$ and $P = \mathit{erase}(\underline{P})$ where the set of channels to be checked for partial confluence is $\{\mathit{truech}\}$ (i.e., $c \in Y \Leftrightarrow c = \mathit{truech}$). To see this, note that the typability of p implies that for each variable a, exactly one of a and $\overline{a}$ sends the $w(\mathit{truech})$ capability on channel write. Thus p makes the channels a and $\overline{a}$ represent the boolean value of the variable a. For each clause c, the typability of p_c ensures that exactly one of its literals are set to 1, i.e., receives the $w(\mathit{truech})$ capability on channel read. Thus $\Gamma \vdash P$ implies that ϕ is one-in-three 3SAT satisfiable, and vice versa. □