Types for Deterministic Concurrency

by

Tachio Terauchi

B.S. (Columbia University) 2000
M.S. (University of California, Berkeley) 2004

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Alexander Aiken, Co-Chair
Professor George Necula, Co-Chair
Professor Rastislav Bodik
Professor Leo Harrington

Fall 2006

The dissertation of Tachio Terauchi is approved:

Professor Alexander Aiken, Co-Chair — Date

Professor George Necula, Co-Chair — Date

Professor Rastislav Bodik — Date

Professor Leo Harrington — Date

University of California, Berkeley

Fall 2006

Types for Deterministic Concurrency

by

Tachio Terauchi

Abstract

Types for Deterministic Concurrency

by

Tachio Terauchi

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Alexander Aiken, Co-Chair

Professor George Necula, Co-Chair

This thesis describes a method for ensuring partial determinism in concurrent programs. The idea is presented in two settings: sequential processes communicating via channels and shared references and functional programming with references.

Professor Alexander Aiken, Co-Chair Date

Professor George Necula, Co-Chair Date

Acknowledgements

This thesis would not have been possible without the guidance of my adviser Alex Aiken. I would also like to thank other faculty and students with whom I have interacted during my time at Berkeley. I am especially grateful to the members of the programming languages group and the reconfigurable computing group.

I would also like to thank my friends for their support. I thank Mike Wang for many interesting conversations about science that do not (always) involve computers.

To my parents

Contents

List of Figures

Chapter 1

Introduction

Deterministic programs are easier to debug and verify than non-deterministic programs, both for testing (or simulation) and for formal methods. However, most concurrent models of computation permit non-determinism from interleaving accesses on shared resources. This thesis presents an approach to make concurrent programs deterministic, at least partially. We demonstrate the idea in two settings:

- A set of sequential processes running concurrently at their own speeds and communicating with each other via channels and shared references (Chapter 2).
- A functional programming language with references (Chapter 3).

Concurrency in functional languages occurs at the granularity of expressions, i.e., syntactically independent expressions can be evaluated in parallel. This model of concurrency is often called data flow parallelism.

In both cases, the programming language itself actually permits non-determinism. We formulate an algorithm (i.e., a static analysis) that checks whether an input program is deterministic. This approach allows the language to remain in a simple and familiar form while providing an option of provable determinism. We present

the static analysis as a type system, which makes for an interesting comparison with type-based concepts such as the capability calculus, monads, and linear types.

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. 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.

Sometimes, ad hoc notions such as race freedom and atomicity are used to reason about possibly mal-behaving concurrency. They can enforce, for example, that there is no point in the program where $x := 1$ and $x := 2$ could both occur. 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 current software practice. We claim that 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 concurrency over sequentiality.

Non-deterministic programs are difficult to debug because one cannot just run the program again to reproduce a bug. 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 [GvdP00; BvdP02].

1.2 Utility of Non-Determinism

It seems rare for a real world application to require true non-determinism in the sense that the program's functionality specification requires the program to behave differently under the identical conditions. However, non-determinism sometimes appears as an artifact of programming convenience. While this thesis provides a step toward making provable deterministic concurrency practical, it may be difficult to completely avoid non-determinism in reality. Our system actually checks for *partial* determinism in which the programmer can control the degree of non-determinism.

Chapter 2

Deterministic Communicating Processes

In current software practice, the dominant style of concurrency is to express programs as sets of communicating processes. Each process runs sequentially at its own speed (i.e., asynchronously) and communicates with the other processes via shared resources such as channels, locks, and basic reference cells. Many popular programming models, including message passing programming 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 convert a sequential language to this concurrent model.

However, writing bug-free programs in this model is notoriously difficult due to asynchronous accesses on shared resources, which introduce non-determinism. This chapter presents a system that can automatically detect partial determinism in programs communicating via a mix of different kinds of communication resources: rendezvous channels, output buffered channels, input buffered channels, and shared reference cells. Our system can detect more programs to be deterministic than previous

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

Figure 2.1: The syntax of the small concurrent language.

approaches [Kah74; NS97; KPT99; Kön00; ET05]. Section 2.2.2 shows a few examples that can be checked by our system: producer consumer, token ring, and barrier synchronization.

2.1 Preliminaries

We focus on the simple concurrent language shown in Figure 2.1. A program, p, is a parallel composition of finitely many processes. A process, s, is a sequential statement consisting of the usual imperative features as well as channel communication operations. Here, $\texttt{!}(e_1, e_2)$ means writing the value of e_2 to the channel e_1, and $\texttt{?}(e, x)$ means storing the value read from the channel e to the variable x. The variables are process-local, and so the only means of communication are channel reads and writes. We use meta-variables x, y, z, etc. for variables and c, d, etc. for channels.

The language cannot dynamically create channels or spawn new processes, but these restrictions are imposed only to keep the main presentation to the novel features of the system. Section 2.3 shows how to handle dynamic channels and processes.

2.1.1 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.

A channel is called *rendezvous* if the sender and the receiver must wait for each other to communicate. Therefore, if c and d are rendezvous channels, then the following program is deterministic[1] because $(x, y) = (1, 2)$ when the process terminates:

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

In contrast, buffered channels allow the sender to proceed without waiting for the receiver. Therefore, the above program is non-deterministic if c is buffered because $!(c, 1)$ does not need to wait for the reader $?(c, x)$, and therefore (x, y) could be $(1, 2)$ or $(1, 3)$.

We call a buffered channel *output buffered* if there is only one buffer for the channel whereas we call a buffered channel *input buffered* if each process has its own buffer (intuitively, output buffered channels are buffered at the sender side whereas input buffered channels are buffered at the receiver side). Therefore, $!(c, 1); !(c, 2) \;||\; ?(c, x) \;||\; ?(c, y)$ is deterministic if c is input buffered but not if c is output buffered or rendezvous. Input buffered channels are the basis of Kahn process networks [Kah74].

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, $!(c, 1); !(c, 2) \;||\; ?(c, x)$ is not deterministic because $!(c, 2)$ may or may-not overwrite 1 in the buffer before $?(c, x)$ reads the buffer. The program is

[1] Here, we use the term informally. Determinism is formally defined in Section 2.1.2.

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

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

is deterministic if c is a reference cell and d is rendezvous because both reads of c happen after $!(c,2)$ overwrites the buffer. But the program is not deterministic if c is output buffered.

2.1.2 Operational Semantics

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

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

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

Figure 2.2 shows the reduction rules. Programs are equivalent up to re-ordering of parallel processes, e.g., $p_1||p_2 = p_2||p_1$. If p is an empty program (i.e., p contains 0 processes), then $p'||p = p'$. Also, we let $s = s; \texttt{skip} = \texttt{skip}; s$. Note that the rules only reduce the left-most processes, and so we rely on process re-ordering to reduce other processes. The rules **IF1**, **IF2**, **WHILE1**, and **WHILE2** do not

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

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

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

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

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

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

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

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

Figure 2.2: The operational semantics of the small concurrent language.

involve channel communication and are self-explanatory. **ASSIGN** is also a process-local reduction because variables are local. Here, $S[i \mapsto h]$ means $\{j \mapsto S(j) \mid j \neq i \wedge j \in \mathit{dom}(S)\} \cup \{i \mapsto h\}$. We use the same notation for other mappings.

UNBUF handles communication over rendezvous channels. The predicate $\neg \mathit{buffered}(c)$ says c is unbuffered (and therefore rendezvous). Note that the written value e_2' is immediately transmitted to the reader. **BUF1** and **BUF2** handle communication over buffered channels, which include output buffered channels, in-

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

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

Figure 2.3: Buffer operations.

put 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 2.3. Buffer operations $B.read(c,i)$ 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 2.1.1. *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 2.1.2. *Let Y be a set of channels. We say that P is deterministic with respect to Y if for each process index i, there exists a (possibly infinite) sequence h_i such that for any $P \rightarrow^* (B, S, p)$ that communicates only over channels in Y, $S(i)$ is a prefix of h_i.*

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

Lemma 2.1.3. *If P is partially confluent with respect to Y then P is deterministic with respect to Y.*

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

2.2 Calculus of Capabilities

We now present a system to ensure partial confluence. We cast our system as a *capability calculus* [CWM99]. 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 informally describe the general idea. To simplify matters, we begin this initial discussion with rendezvous channels and total confluence. Given a program, the goal is to ensure that for each channel c, at most one process can write c and at most one process can read c at any point in time. To this end, we introduce capabilities $r(c)$ and $w(c)$ such that a process needs $r(c)$ to read from c and $w(c)$ to write to c. Capabilities are distributed to the processes at the start of the program and are not allowed be duplicated.

Recall the following confluent program from Section 2.1:

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

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 due to c being 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* [Boy03; TA05] 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 2.1 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. In Chapter 3, we show that fractional capabilities can be derived in a principled way from ordering dependencies.

We now formally present our capability calculus. Let

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

A *capability set* C is a function from **Capabilities** to rational numbers in the range $[0,1]$. If c is rendezvous, output buffered, or input buffered, $C(w(c)) = 1$ (resp. $C(r(c)) = 1$) means that the capability to write (resp. read) c is in C. Read capabilities are not needed for input buffered channels because each process has its

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

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

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

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

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

Figure 2.4: The capability calculus: sequential reductions.

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:

$$hasWcap(C, c) \Leftrightarrow C(w(c)) = 1$$

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

To denote capability merging and splitting, we define:

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

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

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

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

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

Figure 2.5: The capability calculus: communication reductions.

Figure 2.4 and Figure 2.5 show the reduction rules of the capability calculus. The reduction rules (technically, labeled transition rules) are similar to those of operational semantics with the following differences.

Each concurrent process is prefixed by a capability set C representing the current capabilities held by the process. The rules in Figure 2.4 do not utilize capabilities (i.e., capabilities are only passed sequentially) and are self-explanatory. Figure 2.5 shows how capabilities are utilized at communication points. **UNBUF′** sends capabilities C from the writer process to the reader process and sends capabilities C' from the reader process to the writer process. **UNBUF′** checks whether the right capabilities are present by $\mathit{hasWcap}(C_i, c) \wedge \mathit{hasRcap}(C_j, c)$. The label ℓ records whether the check succeeds. Because we are interested in partial confluence with respect to some set Y of channels, we only check the capabilities if $c \in Y$. To this end, the predicate $\mathit{confch}()$ parameterizes the system so that $\mathit{confch}(c)$ iff $c \in Y$.

BUF1′ and **BUF2′** handle buffered communication. Recall that the writer can

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

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

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

Definition 2.2.1. *Let* $\underline{P} = (X, B, S, 1.C_1.s_1 || \ldots || n.C_n.s_n)$. *Let* $C = \sum_{i=1}^{n} C_i + \sum_{c \in dom(X)} X(c)$. *We say* $\underline{P}$ *is well-formed if for all* $cap \in dom(C)$, $C(cap) = 1$.

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

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

We now state the main claim of this section.

Theorem 2.2.3. *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 erase($\underline{P}$) $= P$. Then P is partially confluent with respect to Y.*

2.2.1 Static Checking of Capabilities

Theorem 2.2.3 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*($\underline{P}$) $= P$. [2] 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.

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

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 **StaticCapabilities** to $[0, 1]$. We call such a mapping a *static capability set*. The rendezvous channel type can be read as follows: the channel communicates values of type τ, any writer of the channel sends capabilities Ψ_1, and any reader of the channel sends capabilities Ψ_2. For an output buffered channel, because readers cannot send capabilities, only one static capability set, Ψ, is present in its type. For an input buffered channel, the sequence $\langle \Psi_1, \ldots, \Psi_n \rangle$ lists capabilities

[2] 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.

such that each process i gets Ψ_i from a read. Because a value stored in a reference cell may be read arbitrarily many times, our type system cannot statically reason about processes passing capabilities through reference cells, and so a reference cell type does not carry any static capability set.

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

$$\Psi_1 + \Psi_2 = \{\mathit{cap} \mapsto \Psi_1(\mathit{cap}) + \Psi_2(\mathit{cap}) \mid \mathit{cap} \in \mathbf{StaticCapabilities}\}$$
$$\Psi_1 - \Psi_2 = \Psi_3 \quad \text{if } \Psi_1 = \Psi_3 + \Psi_2$$

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

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

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

($\mathit{0}$ is the constant zero function $\lambda x.0$.) Similarly, $\mathit{writeRecv}(\tau)$ (resp. $\mathit{readRecv}(\tau, i)$)

is the set of capabilities received by the writer (resp. the reader process i):

$$writeRecv(\tau) = readSend(\tau)$$

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

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,

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

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

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 2.3 discusses a way to relax this restriction.)

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

Expressions are type-checked as follows:

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

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

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

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

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

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

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

Figure 2.6: Type checking rules.

Figure 2.6 shows the type checking rules for statements. The judgments are of the form $\Gamma, i, \Psi \vdash s : \Psi'$ where i is the index of the process 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, $\mathit{confch}(\tau, \Gamma)$ is defined as:

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

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

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

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

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

Theorem 2.2.4. *Let Y be a set of channels and let $confch(c) \Leftrightarrow c \in Y$. Let $M = \{\underline{P} \mid \exists Env. Env \vdash \underline{P}\}$. Then M is capability-respecting with respect to Y.*

Theorem 2.2.4 together with Theorem 2.2.3 implies that to check if P is confluent, it suffices to find a well-typed $\underline{P}$ such that $P = erase(\underline{P})$. More formally,

Corollary 2.2.5. *Let Y be a set of channels and let $confch(c) \Leftrightarrow c \in Y$. P is partially-confluent and deterministic with respect to Y if there exists $\underline{P}$ and Env such that $P = erase(\underline{P})$ and $Env \vdash \underline{P}$.*

The problem of finding $\underline{P}$ and *Env* such that $P = \mathit{erase}(\underline{P})$ and $\mathit{Env} \vdash \underline{P}$ can be reduced to a linear inequality constraint satisfaction problem. The reduction is divided into two phases. In the first phase, we look for a valid type derivation but ignoring static capabilities. More formally, we look for $\underline{P'}$ and *Env*′ such that $P = \mathit{erase}(\underline{P'})$ and $\mathit{Env}' \vdash \underline{P'}$ by assuming $\forall \Psi, \tau. \mathit{hasWcap}(\Psi, \tau) = \mathit{hasRcap}(\Psi, \tau) = \mathit{true}$. This is a simple type inference problem that can be solved, for example, by the standard union-find approach. (Note that it suffices to let each $\Psi = 0$ and each $C = 0$.)

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 $P = \mathit{erase}(\underline{P})$, $\mathit{Env} \vdash \underline{P}$, and $\forall c. \Gamma[c] = \Gamma'[c]$ where $\mathit{Env} = (\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi_1, \ldots, \Psi_n, W)$ and $\mathit{Env'} = (\Gamma', \Gamma'_1, \ldots, \Gamma'_n, \Psi'_1, \ldots, \Psi'_n, W')$. Because every $\mathit{hdl}(\tau)$ appearing in the type derivation is known, the problem reduces to 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 $\mathit{hasWcap}(\Psi, \tau)$ and $\mathit{hasRcap}(\Psi, \tau)$.) 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, which can then be solved efficiently by a linear programming algorithm.

2.2.2 Examples

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

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

is a simple but common communication pattern of sender and receiver processes being fixed for each channel; no capabilities need to be passed between processes. The type system can prove confluence by assigning the starting capabilities $0[w(\rho) \mapsto 1]$ to process 1 and $0[r(\rho) \mapsto 1]$ to process 2 where $c : ch(\rho, int, 0)$.

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:

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

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

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

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

The type system can prove confluence by assigning the following types (assume e_1, e_2, and e_3 are of type int): $c_1 : ch(\rho_{c1}, int)$, $c_2 : ch(\rho_{c2}, int)$, $c_3 : ch(\rho_{c3}, int)$, 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.

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.[3] 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 [AG98]. Note that reference cells c_1 and c_2 are not used for synchronization and therefore need no syntactic restriction.

2.3 Extensions

We discuss extensions to our system that handle aliasing and dynamic creation of channels and processes.

2.3.1 Regions

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

[3] This can be done without changing the implementation. See *named barriers* in [AG98].

2.3.2 Existential Abstraction and Linear Types

Another way to resolve aliasing is to existentially abstract capabilities as in $\exists\rho.\tau \otimes \Psi$. Any type containing a capability set must be handled linearly[4] to prevent the duplication of capabilities. The capabilities are recovered by opening the existential package. Existential abstraction can encode linearly typed channels [NS97; KPT99] (for rendezvous channels) as: $\exists\rho.ch(\rho, \tau, 0, 0) \otimes 0[w(\rho) \mapsto 1, r(\rho) \mapsto 1]$. Note that the type encapsulates both a channel and the capability to access the channel. This encoding allows transitions to and from linearly typed channels to the capabilities world, e.g., it is possible to use once a linearly-typed channel multiple times. An analogous approach has been applied to express updatable recursive data structures in the capability calculus [WM00].

2.3.3 Dynamically Created Channels

Dynamically created channels can be handled in much the same way heap allocated objects are handled in the capability calculus [CWM99] (we only show the rule for the case where c is rendezvous):

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

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

[4]Actually, a more relaxed sub-structural type system like the one in Chapter 3 is preferred for handling fractional capabilities.

2.3.4 Dynamically Spawned Processes

Dynamic spawning of processes can be typed as follows:

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

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

2.4 Issues on Memory Model

The operational semantics uses the strict consistency memory model where every operation on the buffer B is serialized. While we do not give a formal proof, it is easy to see that our type system works even with weak consistency which is often argued to be more efficient for distributed systems.

Weak consistency allows non-synchronizing operations to be unserialized. As discussed Section 2.2.2, the type system does not consider a reference cell access to be a synchronization operation. Hence, as in the standard implementation of weak consistency, reference writes and reads can be carried out locally such that changes are broadcasted only at the next synchronization operation. In fact, the capability set tells which writes must be broadcasted. Such information may prove useful to further reduce the communication cost.

2.5 Proofs

Lemma 2.1.3 *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. Let i be a process index. For contradiction, suppose that there exist reductions $P \to^* (B_1, S_1, p_1)$ and $P \to^* (B_2, S_2, p_2)$ both communicating only over channels in Y such that neither $S_1(i)$ nor $S_2(i)$ is a prefix of the other. But since P is partially confluent, there exists Q such that $(B_1, S_1, p_1) \to^* Q$ and $(B_2, S_2, p_2) \to^* Q$. But by inspection of the reduction rules, such a Q cannot exist. □

Lemma 2.5.1 (Diamond Property)**.** *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 $\mathit{erase}(\underline{P}) = P$. Suppose $P \to P_1$ communicating only over channels in Y and $P \to P_2$. Then either $P_1 = P_2$ or there exists Q such that $P_2 \to Q$ communicating only over channels in Y and $P_1 \to Q$.*

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

- $\mathit{erase}(\underline{P_1}) = P_1$ and $\mathit{erase}(\underline{P_2}) = P_2$, 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 $P \to P_1$ is a sequential reduction (i.e., **IF1**, **IF2**, **WHILE1**, **WHILE2**, or **ASSIGN**) is trivial. Suppose $P \to P_1$ is:

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

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 $P \to P_2$ does not communicate over c, then the result follows trivially. So suppose $P \to P_2$ communicates over c. Let i' and j' be the process indexes 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 $P \to P_1$ 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 $P_1 = (B', S, i.s||p)$. Suppose c is output buffered or input buffered. By the assumption, $c \in Y$. If $P \to P_2$ does not communicate over c, then the result follows trivially. Also, if $P \to P_2$ is a read from c, then because buffers are FIFO, the result follows trivially. So suppose $P \to P_2$ writes c. Let i' be the process indexes such that $\underline{P} \xrightarrow{\ell_2} \underline{P_2}$ 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 $P \to P_1$ reads an output buffered channel is similar. The case where $P \to P_1$ reads an input buffered channel follows trivially from the fact that input buffers are process local.

Suppose $P \to P_1$ 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 $P_1 = (B', S, i.s||p)$. Suppose c is a reference cell. By assumption, $c \in Y$. If $P \to P_2$ does not communicate over c, then the result follows trivially. So suppose $P \to P_2$ communicates over c (read or write). Let i' be the process indexes such that $\underline{P} \xrightarrow{\ell_2} \underline{P_2}$ 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 $P \to P_2$ is in fact a write). Therefore, $P_1 = P_2$.

Suppose $P \to P_1$ is:

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

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 $P \to P_2$ does not communicate over c, then the result follows trivially. Also, if $P \to P_2$ is a read from c then because reading a reference cell is non-destructive, the result follows trivially. So suppose $P \to P_2$ writes to c. But because $\ell_1 = \ell_2 = \mathit{true}$, $\underline{P}$ being well-formed implies that $P \to P_2$ is, in fact, not a write to c. □

Theorem 2.2.3 *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.*

Proof. Note that for any $\underline{P} \in M$, if $\mathit{erase}(\underline{P}) \to^* Q$ then there exists $\underline{Q} \in M$ such that $\mathit{erase}(\underline{Q}) = Q$. Therefore, the proof is a standard inductive application of the diamond property (Lemma 2.5.1). □

Lemma 2.5.2. *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 2.5.3. *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 2.5.4. $\Gamma, i, \Psi \vdash (s_1; s_2); s_3 : \Psi'$ *iff* $\Gamma, i, \Psi \vdash s_1; (s_2; s_3) : \Psi'$.

Proof. By inspection of the type checking rules. □

Theorem 2.2.4 *Let Y be a set of channels and let $confch(c) \Leftrightarrow c \in Y$. Let $M = \{\underline{P} \mid \exists Env.Env \vdash \underline{P}\}$. Then M is capability-respecting with respect to Y.*

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

(1) $\underline{P}$ is well-formed, and

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

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

Let Q be a state such that $erase(\underline{P}) \rightarrow Q$. Let $P = erase(\underline{P})$. Let $(\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi_1, \ldots, \Psi_n, W) = Env$. We show that there exist $\underline{Q}$, $\Psi'_1, \ldots, \Psi'_n$, and W' such that $(\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$, $\underline{P} \xrightarrow{\ell} \underline{Q}$, and if ℓ is not empty then $\ell = true$. (So in fact, Env and Env' share the same $\Gamma, \Gamma_1, \ldots, \Gamma_n$, indicating the flow-insensitivity of our system.) We prove by case analysis on $P \rightarrow Q$.

Throughout this proof, we implicitly use Lemma 2.5.4 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

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

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

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

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

Therefore, by Lemma 2.5.2, $\Gamma_i, i, \Psi_{i1} \vdash s : \Psi_{i5}$ for some Ψ_{i5}. And so, $\Gamma_i, i, \Psi_i \vdash s_1; s : \Psi_{i5}$. Let $\Psi'_j = \Psi_j$ for each $j \in \{1 \ldots n\}$ and $W' = W$. Then it follows that $(\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

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 $erase(\underline{Q}) = Q$ and $\underline{P} \rightarrow \underline{Q}$. By assumption,

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

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

From $\Gamma_i \vdash S(i)$, $\Gamma_i \vdash e : \Gamma_i(x)$, and Lemma 2.5.3, it follows that $\Gamma_i \vdash S'(i)$. Let $\Psi'_j = \Psi_j$ for each $j \in \{1, \ldots, n\}$ and $W' = W$. Then it follows that $(\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

Suppose $P \to Q$ is

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

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,

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

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

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

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

where $\Psi_{i1} = \Psi_i - \mathit{writeSend}(\tau) + \mathit{writeRecv}(\tau)$ and $\Psi_{j1} = \Psi_j - \mathit{readSend}(\tau') + \mathit{readRecv}(\tau', j)$. Lemma 2.5.3 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 $(\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi_1', \ldots, \Psi_n', W') \vdash \underline{Q}$.

Suppose $\mathit{confch}(c)$ holds. Then $\mathit{confch}(\tau, \Gamma_i)$ and $\mathit{confch}(\tau', \Gamma_j)$. Therefore,

$\mathit{hasWcap}(\Psi_i, \tau)$ and $\mathit{hasRcap}(\Psi_j, \tau')$. Therefore, $\mathit{hasWcap}(C_i, c)$ and $\mathit{hasRcap}(C_j, c)$. Thus it follows that $\underline{P} \stackrel{\mathit{true}}{\rightarrow} \underline{Q}$. On the other hand, if $\neg \mathit{confch}(c)$, then $\underline{P} \stackrel{\mathit{true}}{\rightarrow} \underline{Q}$ trivially.

Suppose $P \rightarrow Q$ is

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

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,

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

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

where $\Psi_{i1} = \Psi_i - \mathit{writeSend}(\tau) + \mathit{writeRecv}(\tau)$, and Γ_i 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 2.5.3 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 $(\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi_1', \ldots, \Psi_n', W') \vdash \underline{Q}$.

Suppose $\mathit{confch}(c)$ holds. Then $\mathit{confch}(\tau, \Gamma_i)$, and so $\mathit{hasWcap}(\Psi_i, \tau)$. Therefore,

$\mathit{hasWcap}(C, c)$. Thus it follows that $\underline{P} \xrightarrow{\mathit{true}} \underline{Q}$. On the other hand, if $\neg \mathit{confch}(c)$, then $\underline{P} \xrightarrow{\mathit{true}} \underline{Q}$ trivially.

Suppose $P \to Q$ is

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

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,

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

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

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 2.5.3 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 $(\Gamma, \Gamma_1, \ldots, \Gamma_n, \Psi'_1, \ldots, \Psi'_n, W') \vdash \underline{Q}$.

Suppose $\mathit{confch}(c)$ holds. Then $\mathit{confch}(\tau, \Gamma_i)$, and so $\mathit{hasRcap}(\Psi_i, \tau)$. Therefore, $\mathit{hasRcap}(C, c)$. Thus it follows that $\underline{P} \xrightarrow{\mathit{true}} \underline{Q}$. On the other hand, if $\neg \mathit{confch}(c)$, then

$\underline{P} \stackrel{true}{\rightarrow} \underline{Q}$ trivially. □

Chapter 3

Deterministic Functional Programming with References

The recent emergence of multicore chips (chip multiprocessors) has caused a resurgence of research interest in data flow parallelism [SMSO03; BVCG04]. Data flow languages are the ideal language for programming data flow machines [DM75; AW77; McG82; AN90; FC95; GDF+97]. Unlike the communicating processes model studied in Chapter 2, pure data flow languages rely exclusively on *joins* for synchronization. Two concurrently running expressions e_1 and e_2 are said to be joined at e if e depends on the values of both e_1 and e_2 (e.g., $e = e_1 + e_2$). In short, data dependence governs synchronization.

When expressions are purely functional, determinism follows from the standard confluence property of functional languages. The problem arises when expressions contain side effects, such as writes to references. For example, $e_1 + e_2$ is nondeterministic if $e_1 = (\texttt{write}\, x\, 1; \texttt{read}\, x)$ and $e_2 = (\texttt{write}\, x\, 2; \texttt{read}\, x)$ where $\texttt{write}\, e\, e'$ writes e' to the reference e and $\texttt{read}\, e$ is the value stored in the reference e.

Ensuring determinism in this setting is equivalent to the problem of incorpo-

$$\begin{array}{rcl} e & := & x \mid i \mid \lambda x.e \mid e\ e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid e \otimes e' \mid \pi_i(e) \\ & \mid & \texttt{write}\ e_1\ e_2\ e_3 \mid \texttt{read}\ e\ e' \mid \texttt{ref}\ e \mid \texttt{join}\ e\ e' \mid \bullet \end{array}$$

Figure 3.1: The syntax of the language λ_{wit}.

rating writable references in purely functional languages. This problem has been studied extensively and has a number of solutions [JW93; LS97; ORH93; Wad98; GH90; AvGP93] with monads [Mog91; ORH93; JW93; LS97] being arguably the most popular. However, these approaches completely sequentialize reference accesses and therefore destroy useful parallelism. This chapter presents an approach that ensures determinism (or more formally, confluence) without imposing strict sequentiality.

The underlying idea is similar to the capability calculus discussed in Chapter 2. However, whereas these capabilities are an implicit concept, here we need something to be explicitly presented as data because data dependence is the only means for synchronization. To this end, we introduce the notion of *witnesses* for creating dependences between reference accesses. We also take a deeper look at dependences which allows us to make a completeness argument, i.e., the system is the best we can do under certain assumptions.

The rest of the chapter is organized as follows. Section 3.1 formally introduces witnesses. Section 3.2 defines a semantic condition called *witness race freedom* for correct usage of witnesses and a proof of its sufficiency. Section 3.3 presents an algorithm for checking witness race freedom. The algorithm is derived as a type inference algorithm for a substructural type system.

3.1 Preliminaries

Figure 3.1 gives the syntax of λ_{wit}, a simple functional language with references and witnesses. The syntax of λ_{wit} has the usual features of a functional language:

integers i, function abstractions $\lambda x.e$, function applications $e\ e'$, variable bindings $\texttt{let}\ x = e\ \texttt{in}\ e'$, pairs $e \otimes e'$, and projections $\pi_i(e)$ where $i = 1$ or $i = 2$. Bindings $\texttt{let}\ x = e\ \texttt{in}\ e'$ can be recursive, i.e., x may appear in e. Three expression kinds work with references: reference writes $\texttt{write}\ e_1\ e_2\ e_3$, reference reads $\texttt{read}\ e\ e'$, and reference creations $\texttt{ref}\ e$. A read $\texttt{read}\ e\ e'$ has a *witness* parameter e' along with a reference parameter e such that the reference e is not read until seeing the witness e'. (Section 3.2 defines the formal meaning of "seeing the witness.") Similarly, a write $\texttt{write}\ e_1\ e_2\ e_3$ writes the expression e_2 to the reference e_1 after seeing the witness e_3. After completion of the read, $\texttt{read}\ e\ e'$ returns a pair of the read value and a witness. Similarly, $\texttt{write}\ e_1\ e_2\ e_3$ returns a witness after the write.

Before describing the formal semantics of λ_{wit}, we describe novel properties of λ_{wit} informally by examples.

Programs in λ_{wit} can use witnesses to order reference accesses. For example, the following program returns 2 regardless of the evaluation order because the read requires a witness of the write:

$$\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in}\ \texttt{let}\ w = (\texttt{write}\ x\ 2\ \bullet)\ \texttt{in}\ \texttt{read}\ x\ w$$

(The symbol $\bullet$ is used for dummy witnesses.) On the other hand, λ_{wit} does not guarantee correctness. For example, the following λ_{wit} program has no ordering between the read and the write and hence may return 1 or 2 depending on the evaluation order:

$$\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in}\ \texttt{let}\ w = (\texttt{write}\ x\ 2\ \bullet)\ \texttt{in}\ \texttt{read}\ x\ \bullet$$

The expression kind $\texttt{join}ee'$ joins two witnesses by waiting until it sees the witness e and the witness e' and returning a witness. For example, the following program returns the pair $1 \otimes 1$ regardless of the evaluation order because the write waits until

$$
\begin{array}{rcl}
E & := & D \cup \{a \mapsto E\} \mid [\,] \mid E\ e \mid e\ E \mid E \otimes e \mid e \otimes E \mid \pi_i(E) \\
& \mid & \texttt{write}\ E\ e\ e' \mid \texttt{write}\ e\ E\ e' \mid \texttt{write}\ e\ e'\ E \mid \texttt{read}\ e\ E \mid \texttt{read}\ E\ e \\
& \mid & \texttt{ref}\ E \mid \texttt{join}\ E\ e \mid \texttt{join}\ e\ E
\end{array}
$$

App	$(S, E[(\lambda x.e)\ e']) \Rightarrow (S, E[e[a/x]] \uplus \{a \mapsto e'\})$
Let	$(S, E[\texttt{let}\ x = e\ \texttt{in}\ e']) \Rightarrow (S, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\})$
Pair	$(S, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow (S, E[e_i])$
Write	$(S, E[\texttt{write}\ \ell\ e\ \bullet]) \Rightarrow (S[\ell \leftarrow a], E[\bullet] \uplus \{a \mapsto e\})$
Read	$(S, E[\texttt{read}\ \ell\ \bullet]) \Rightarrow (S, E[S(\ell) \otimes \bullet])$
Ref	$(S, E[\texttt{ref}\ e]) \Rightarrow (S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\})$
Join	$(S, E[\texttt{join}\ \bullet\ \bullet]) \Rightarrow (S, E[\bullet])$
Arrive	$(S, E[a] \uplus \{a \mapsto e\}) \Rightarrow (S, E[e] \uplus \{a \mapsto e\})$ where $e \in V$
GC	$(S, D \uplus D') \Rightarrow (S, D)$ where $\diamond \notin \mathit{dom}(D') \wedge \mathit{dom}(D') \cap \mathit{free}(D) = \emptyset$

Figure 3.2: The semantics of λ_{wit}.

it sees witnesses of both reads:

$$
\begin{array}{l}
\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in} \\
\quad \texttt{let}\ y = (\texttt{read}\ x\ \bullet)\ \texttt{in}\ \texttt{let}\ z = (\texttt{read}\ x\ \bullet)\ \texttt{in} \\
\quad\quad \texttt{let}\ w = (\texttt{write}\ x\ 2\ (\texttt{join}\ \pi_2(y)\ \pi_2(z)))\ \texttt{in} \\
\quad\quad\quad \pi_1(y) \otimes \pi_1(z)
\end{array}
$$

Note that the two reads may be evaluated in any order. In general, witnesses are first class values and hence can be passed to and returned from a function, captured in function closures, and even written to and read from a reference. Witnesses are a simple feature that can be used to order reference accesses in a straightforward manner.

In the rest of this section, we describe the semantics of λ_{wit} so that we can formally define when a λ_{wit} program is confluent. Figure 3.2 shows the semantics of λ_{wit} defined via reduction rules of the form $(S, D) \Rightarrow (S', D')$ where S, S' are *reference stores* and D, D' are *expression stores*. A reference store is a function from a set of *reference locations* ℓ to *ports* a, and an expression store is a function from a set of ports to

expressions. Here, expressions include any expression from the source syntax extended with reference locations and ports. Given a program e, evaluation of e starts from the initial state $(\emptyset, \{\diamond \mapsto e\})$ where the symbol $\diamond$ denotes the special *root port*. Ports are used for evaluation sharing.[1] The reduction rules are parametrized by the evaluation contexts E. For an expression e, $\mathit{free}(e)$ is the set of free variables, ports, and reference locations of e. For an expression store D, $\mathit{free}(D) = \mathit{dom}(D) \cup \bigcup_{e \in \mathit{ran}(D)} \mathit{free}(e)$.

We briefly describe the reduction rules from top-to-bottom. The rule **App** corresponds to a function application. For functions F and F', $F \uplus F'$ denotes $F \cup F'$ if $\mathit{dom}(F) \cap \mathit{dom}(F') = \emptyset$ and is undefined otherwise. **App** creates a fresh port a and stores e' at a. **Let** is similar. **Pair** projects the ith element of the pair. **Write** creates a fresh port a, stores the expression e' at the port a, and stores the port a at the reference location ℓ. We use $S[\ell \leftarrow a]$ as a shorthand for $\{\ell' \mapsto S(\ell') \mid \ell' \in \mathit{dom}(S) \wedge \ell' \neq \ell\} \cup \{\ell \mapsto a\}$. We use the dummy witness symbol $\bullet$ as the run-time representation of any witness because, operationally, a witness is like a dataflow token in dataflow machines. **Read** reads from the reference location ℓ and, as noted above, returns the value paired with a witness. **Ref** creates a fresh location ℓ and a fresh port a, initializes a to the expression e and ℓ to the port a. **Join** takes two witnesses and returns one witness.

Arrive is somewhat non-standard. Here V is the set of "safe to duplicate" expressions. Partly for the sake of the static-checking algorithm to be presented later, we fix V to *values* generated by the following grammar:

$$v := x \mid i \mid a \mid \bullet \mid \ell \mid v \otimes v' \mid \lambda x.e$$

Arrive says that if e is safe to duplicate, then we can replace a by e; we say a safe to duplicate expression has *arrived* at port a. In essence, while standard operational

[1] In the literature, top-level `let`-bound variables often double as variables and ports.

semantics for functional languages [Plo75; Lau93] implicitly combine **Arrive** with other rules, we separate **Arrive** for increased freedom in the evaluation order. Lastly, **GC** garbage-collects unreachable (from the root port $\diamond$) portions of the expression store.

Here is an example of a λ_{wit} evaluation:

$$
\begin{aligned}
&(\emptyset, \{\diamond \mapsto (\lambda x.\texttt{read}\ x\ \bullet)\ \texttt{ref}\ 1\}) \\
&\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto (\lambda x.\texttt{read}\ x\ \bullet)\ \ell, a \mapsto 1\}) && \textbf{Ref} \\
&\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto \texttt{read}\ a'\ \bullet, a \mapsto 1, a' \mapsto \ell\}) && \textbf{App} \\
&\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto \texttt{read}\ \ell\ \bullet, a \mapsto 1, a' \mapsto \ell\}) && \textbf{Arrive} \\
&\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto a \otimes \bullet, a \mapsto 1, a' \mapsto \ell\}) && \textbf{Read} \\
&\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto 1 \otimes \bullet, a \mapsto 1, a' \mapsto \ell\}) && \textbf{Arrive} \\
&\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto 1 \otimes \bullet\}) && \textbf{GC}
\end{aligned}
$$

The semantics is non-deterministic and therefore also allows other reduction sequences for the same program. For example, we may take an **App** step immediately instead of first creating a new reference by a **Ref** step:

$$(\emptyset, \{\diamond \mapsto (\lambda x.\texttt{read}\ x\ \bullet)\ \texttt{ref}\ 1\}) \Rightarrow (\emptyset, \{\diamond \mapsto \texttt{read}\ a\ \bullet, a \mapsto \texttt{ref}\ 1\}) \quad \textbf{App}$$

Before defining confluence, we point out several important properties of this semantics. The evaluation contexts E do not extend to subexpressions of a λ abstraction, i.e., we do not reduce under λ. The evaluation contexts also do not extend to subexpressions of an expression $\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'$, but e and e' may become available for evaluation via applications of the **Let** rule. As with call-by-value evaluation or call-by-need evaluation, evaluation of an expression is shared. For example, in the program $(\lambda x.x \otimes x)\ e$, the expression e is evaluated at most once.

The semantics of λ_{wit} has strictly more freedom in evaluation order than both call-

by-value and call-by-need. In particular, call-by-need evaluation can be obtained by using the same reduction rules but restricting the evaluation contexts to the following

$$\begin{array}{rcl} E & := & D \cup \{\diamond \mapsto E\} \mid [\,] \mid E\ e \mid \pi_i(E) \mid \texttt{write}\ E\ e\ e' \mid \texttt{write}\ \ell\ e\ E \\ & \mid & \texttt{read}\ E\ e \mid \texttt{read}\ \ell\ E \mid \texttt{join}\ E\ e \mid \texttt{join} \bullet E \end{array}$$

Call-by-value evaluation can be obtained by adding the following contexts to the evaluation contexts of the call-by-need evaluation

$$E \ := \ \ldots \mid (\lambda x.e)\ E \mid E \otimes e \mid v \otimes E \mid \texttt{write}\ \ell\ E \bullet \mid \texttt{ref}\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e$$

in addition to restricting the rule **App** to the case $e' \in V$, the rule **Let** to the case $e \in V$, the rule **Pair** to the case $e_1, e_2 \in V$, the rule **Write** to the case $e' \in V$, and the rule **Ref** to the case $e \in V$.[2] Note that both lazy writes and strict writes are possible in λ_{wit}.

Having defined the semantics, we can formally define when a λ_{wit} program is confluent. To this end, we define *observational equivalence* as the smallest reflexive and transitive relation $D \approx D'$ on expression stores satisfying:

- $D \approx D[a/a']$ where $a \notin \mathit{free}(D)$
- $D \approx D[\ell/\ell']$ where $\ell \notin \mathit{free}(D)$

That is, expression stores are observationally equivalent if they are equivalent up to consistent renaming of free ports and reference locations. Let $\Rightarrow^*$ be a sequence of zero or more $\Rightarrow$.

Definition 3.1.1 (Confluence)**.** *A program state (S, D) is confluent if for any two states (S_1, D_1) and (S_2, D_2) such that $(S, D) \Rightarrow^* (S_1, D_1)$ and $(S, D) \Rightarrow^*$*

[2]Strictly speaking, the context $\texttt{let}\ x = E\ \texttt{in}\ e$ is not in the semantics of λ_{wit}. But λ_{wit} can simulate the behavior via a **Let** step and then reducing $e[a/x]$ which is now in an evaluation context.

(S_2, D_2), there exist two states (S_1', D_1') and (S_2', D_2') such that $(S_1, D_1) \Rightarrow^* (S_1', D_1')$, $(S_2, D_2) \Rightarrow^* (S_2', D_2')$, and $D_1' \approx D_2'$. A program e is confluent if its initial state $(\emptyset, \{\diamond \mapsto e\})$ is confluent.

Note that the definition does not require any relation between reference location stores S_1' and S_2'. So, for example, a program that writes but never reads would be confluent. As shown before, λ_{wit} contains programs that are not confluent. Indeed, the difference between call-by-need and call-by-value is enough to demonstrate non-confluence:

$$(\lambda x.\texttt{read}\, x\, \bullet)\ (\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in}\ \texttt{let}\ y = (\texttt{write}\, x\, 2\, \bullet)\ \texttt{in}\ x)$$

The above program evaluates to the pair $1 \otimes \bullet$ under call-by-need and to the pair $2 \otimes \bullet$ under call-by-value. No further reductions can make the two states observationally equivalent. (Here we implicitly read back the top-level expression from the root port instead of showing the actual expression stores for brevity.)

We have shown earlier that witnesses can aid in writing confluent programs by directly ordering reference accesses. Witnesses are first class values and hence can be treated like other expressions. For example, the program below captures a witness in a function which itself returns a witness to ensure that reads and writes happen in a correct order:

$$\begin{array}{l}
\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in} \\
\quad \texttt{let}\ w = \texttt{write}\, x\, 2\, \bullet\ \texttt{in} \\
\quad\quad \texttt{let}\ f = \lambda y.\texttt{read}\, x\, w\ \texttt{in} \\
\quad\quad\quad \texttt{let}\ z = (f\ 0) \otimes (f\ 0)\ \texttt{in} \\
\quad\quad\quad\quad \texttt{let}\ w = \texttt{write}\, x\, 3\ \texttt{join}\ \pi_2(\pi_1(z))\ \pi_2(\pi_2(z))\ \texttt{in}\ z
\end{array}$$

Here, a witness of the first write is captured in the function f. Hence both reads from

the two calls to f see a witness of the write. A witness of each read is returned by f, and the last write waits until it sees witnesses from both reads. Therefore, the result of the program is $(2 \otimes \bullet) \otimes (2 \otimes \bullet)$ regardless of the evaluation order. Note that the two calls to f, and thus the reads in the calls, can occur in either order.

3.2 Witness Race Freedom

As discussed in Section 3.1, witnesses aid in writing confluent programs in the presence of reference accesses but do not enforce confluence. In this section, we give a sufficient condition for guaranteeing confluence.

Intuitively, our goal is to ensure that reads and writes happen in some race-free order by partially ordering them via witnesses. We now make this intuition more precise. First, we formally define what we mean by the phrase "reference access A sees a witness of reference access B" that we have used informally up to this point.

A *trace graph* is a program trace with all information other than reads, writes, and witnesses elided. There are three kinds of nodes in a trace graph: read nodes $\mathit{read}(\ell)$, write nodes $\mathit{write}(\ell)$, and the join node *join*. Read and write nodes are parametrized by a reference location ℓ. There is a directed edge (A, B) from node A to node B if B directly sees a witness of A. A trace graph $(\mathbb{V}, \mathbb{E})$ is constructed as the program

evaluates a modified semantics:

$$
\begin{array}{ll}
\textbf{Write} & (S, E[\texttt{write}\,\ell\,e\,A]) \Rightarrow (S[\ell \leftarrow a], E[B] \uplus \{a \mapsto e\}) \\
& \quad \mathbb{V} := \mathbb{V} \cup \{B\} \text{ where } B \text{ is a new } \mathit{write}(\ell) \text{ node} \\
& \quad \mathbb{E} := \mathbb{E} \cup \{(A, B)\} \\
\textbf{Read} & (S, E[\texttt{read}\,\ell\,A]) \Rightarrow (S, E[S(\ell) \otimes B]) \\
& \quad \mathbb{V} := \mathbb{V} \cup \{B\} \text{ where } B \text{ is a new } \mathit{read}(\ell) \text{ node} \\
& \quad \mathbb{E} := \mathbb{E} \cup \{(A, B)\} \\
\textbf{Join} & (S, E[\texttt{join}\,A\,B]) \Rightarrow (S, E[C]) \\
& \quad \mathbb{V} := \mathbb{V} \cup \{C\} \text{ where } C \text{ is a new } \mathit{join} \text{ node} \\
& \quad \mathbb{E} := \mathbb{E} \cup \{(A, C), (B, C)\}
\end{array}
$$

Note that we now use nodes as witnesses instead of $\bullet$. The line below each reduction rule shows the graph update action corresponding to that rule. The other rules remain unmodified and hence have no graph update actions. An evaluation starts with $\mathbb{V} = \mathbb{E} = \emptyset$ and performs the corresponding graph update when taking a **Write** step, a **Read** step, or a **Join** step. A trace graph and the annotated semantics are only needed to state the semantic condition for confluence and are not needed in the actual execution of a λ_{wit} program.

We can now define what it means for a node A to *see a witness* of a node B, a notion we have used informally until now.

Definition 3.2.1. *Given a trace graph, we say that a node A sees a witness of a node B if there is a path from B to A in the trace graph. We write $B \rightsquigarrow A$.*

The following is a trivial observation:

Theorem 3.2.2. *If $B \rightsquigarrow A$ in a trace graph, then the reference access corresponding to B must have happened before the reference access corresponding to A in the evaluation that generated the trace graph.*

Clearly, any trace graph is acyclic.

Having defined trace graphs and the $\rightsquigarrow$ relation, we are now ready to state the semantic condition for confluence. We note that a program could produce different trace graphs depending on the choice of reductions, even when those trace graphs are from terminating evaluations. Furthermore, it is not necessarily the case that such a program is non-confluent. Therefore, instead of trying to argue about confluence by comparing different trace graphs, we shall define a condition that can be checked by observing each individual trace graph in isolation.

We write $A : \mathit{nodetype}$ to mean a node A of type $\mathit{nodetype}$. If we have $A : \mathit{read}(\ell)$ and $B : \mathit{write}(\ell)$, then we want either $A \rightsquigarrow B$ or $B \rightsquigarrow A$ to ensure that A always happens before B or B always happens before A because otherwise we may get a read-write race condition due to non-determinism. Also, for any $A : \mathit{read}(\ell)$, if there are two nodes $B_1, B_2 : \mathit{write}(\ell)$ such that neither $B_1 \rightsquigarrow B_2$ nor $B_2 \rightsquigarrow B_1$ (so we do not know which write occurs first) and A could happen after both B_1 and B_2, then we want $C : \mathit{write}(\ell)$ such that $C \rightsquigarrow A$, $B_1 \rightsquigarrow C$ and $B_2 \rightsquigarrow C$, because otherwise the read at A might depend on whether the evaluation chose to do B_1 first or B_2 first, i.e., we have another kind of race-condition. Perhaps somewhat surprisingly, satisfying these two conditions turns out to be sufficient to ensure confluence.

We now formalizes this discussion. For $B : \mathit{write}(\ell)$, we use the shorthand $B \overset{!}{\rightsquigarrow} A$ if for any $C : \mathit{write}(\ell)$ such that $C \rightsquigarrow A$, we have $C \rightsquigarrow B$. It follows that for any $A : \mathit{read}(\ell)$, there exists at most one $B : \mathit{write}(\ell)$ such that $B \overset{!}{\rightsquigarrow} A$. The second condition above is equivalent to requiring that for any $A : \mathit{read}(\ell)$, either there is no $B : \mathit{write}(\ell)$ such that $B \rightsquigarrow A$ or there is a $B : \mathit{write}(\ell)$ such that $B \overset{!}{\rightsquigarrow} A$

Definition 3.2.3 (Witness Race Freedom)**.** *We say that a trace graph* $(\mathbb{V}, \mathbb{E})$ *is witness race free if for every location* ℓ*,*

- *for every* $A : \mathit{read}(\ell) \in \mathbb{V}$ *and* $B : \mathit{write}(\ell) \in \mathbb{V}$*, either* $A \rightsquigarrow B$ *or* $B \rightsquigarrow A$*, and*

- *for every* $A : \mathit{read}(\ell) \in \mathbb{V}$*, either there is no* $B : \mathit{write}(\ell) \in \mathbb{V}$ *such that* $B \rightsquigarrow A$ *or there is a* $B : \mathit{write}(\ell) \in \mathbb{V}$ *such that* $B \overset{!}{\rightsquigarrow} A$*.*

We say that a program e *is witness race free if every trace graph of* e *is witness race free.*

Theorem 3.2.4. *If* e *is witness race free then* e *is confluent.*

The proof appears in Section 3.4.

While witness race freedom is a sufficient condition, it is not necessary. For example, if for each reference location writes happen to never change the location's value, then the program is trivially confluent regardless of the order of reads and writes. Another example is a program using implicit order of evaluation: e.g., in λ_{wit}, expressions are not reduced under λ so a function body is evaluated only after a call. Hence a program that stores a function in a reference location, reads the reference location to call the function, and then writes in the same reference location from the body of the called function is confluent because the write always happens after the read despite the write not seeing the witness of the read.

Nevertheless, witness race freedom is "almost complete" in a sense that if the only way to order two reference accesses is to make one see a witness of the other, and if we cannot assume anything about what expressions are written and how the contents are used, then it is the weakest condition guaranteeing confluence. In particular, if the trace graphs are the only information available about a program, then witness race freedom becomes a necessary condition.

Because witness race freedom is an entirely semantic condition, the result in this section can be extended to most other functional program transformations. However, the static checking algorithm described in Section 3.3 is not as forgiving, which is why we have restricted the set of program transformations to that of the semantics of λ_{wit}. For example, the checking algorithm is unsound for call-by-name.

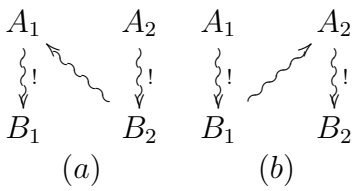

Figure 3.3: Possible orderings between pairs of reads and writes in a witness race free trace graph.

3.3 Types for Statically Checking Witness Race Freedom

While the concept of witnesses is straightforward, it may nevertheless be desirable to have an automated way of checking whether an arbitrary λ_{wit} program is witness race free. Witness race freedom may be checked directly by checking every program trace, which is computationally infeasible. Instead, we exploit a special property of witness-race-free trace graphs to design a sound algorithm that can verify a large subset of witness-race-free λ_{wit} programs.

The key observation is that any witness-race-free trace graph contains for each reference location ℓ a subgraph that we shall call a *read-write pipeline with bottlenecks*. We shall design an algorithm that detects these subgraphs instead of directly checking the witness race freedom condition. Consider a witness-race-free trace graph. Suppose there are $A_1, A_2 : \mathit{write}(\ell)$ and $B_1, B_2 : \mathit{read}(\ell)$ such that $A_1 \neq A_2$, $A_1 \overset{!}{\rightsquigarrow} B_1$ and $A_2 \overset{!}{\rightsquigarrow} B_2$. Due to witness race freedom, it must be the case that $B_2 \rightsquigarrow A_1$ or $A_1 \rightsquigarrow B_2$. If the former is the case, we have the relation as depicted in Figure 3.3 (a). Suppose that the latter is the case. Then, since $A_2 \overset{!}{\rightsquigarrow} B_2$, it must be the case that $A_1 \rightsquigarrow A_2$. Consider A_2 and B_1. Due to witness race freedom again, it must be the case that either $A_2 \rightsquigarrow B_1$ or $B_1 \rightsquigarrow A_2$. But if $A_2 \rightsquigarrow B_1$, then since $A_1 \overset{!}{\rightsquigarrow} B_1$, it must be the case that $A_2 \rightsquigarrow A_1$. But this is impossible since $A_2 \rightsquigarrow A_1 \rightsquigarrow A_2$ forms a

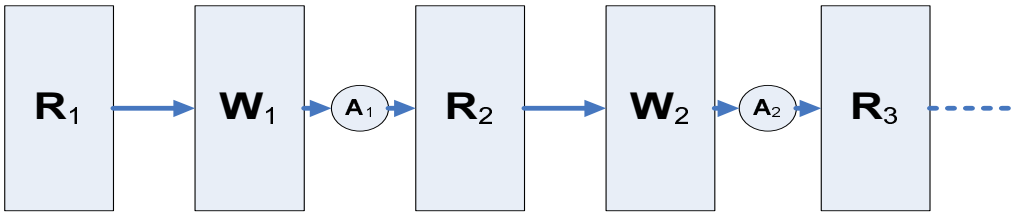

Figure 3.4: A read-write pipeline with bottlenecks for a reference location ℓ.

cycle. So it must be the case that $B_1 \rightsquigarrow A_2$, and we have the relation as depicted in Figure 3.3 (b).

Further reasoning along this line of thought reveals that for a witness-race-free trace graph, for any reference location ℓ, the nodes in the set $X = \{A : \mathit{write}(\ell) \mid \exists B : \mathit{read}(\ell).A \overset{!}{\rightsquigarrow} B\}$ are totally ordered (with $\rightsquigarrow$ as the ordering relation) and that these nodes partition all $\mathit{read}(\ell)$ nodes and $\mathit{write}(\ell)$ nodes in a way depicted in Figure 3.4 where $X = \{A_1, \ldots, A_n\}$. In the figure, each $\mathbb{R}_i$ and $\mathbb{W}_i$ is a collection of nodes. No $\mathbb{R}_i$ contains a $\mathit{write}(\ell)$ node and no $\mathbb{W}_i$ contains a $\mathit{read}(\ell)$ nodes. Each A_i is one $\mathit{write}(\ell)$ node. An arrow from X to Y means that there is a path from each $\mathit{write}(\ell)$ node or $\mathit{read}(\ell)$ node in X to each $\mathit{write}(\ell)$ node or $\mathit{read}(\ell)$ node in Y, except if a $\mathbb{W}_i$ contains no such node, then there is a path from each $\mathit{read}(\ell) \in \mathbb{R}_i$ to A_i. Each $\mathbb{R}_i$ for $i \neq 1$ must contain at least one $\mathit{read}(\ell)$. Arrows just imply the presence of paths, and hence there can be more paths than the ones implied by the arrows, e.g., paths to/from nodes that are not in the diagram, paths to and from nodes in the same collection, and even paths relating the collections in the diagram such as one that goes directly from $\mathbb{R}_i$ to A_i, bypassing $\mathbb{W}_i$.

The graph in Figure 3.4 can be described formally as a subgraph of the trace graph satisfying certain properties.

Definition 3.3.1. *Given a trace graph* $(\mathbb{V}, \mathbb{E})$ *and a reference location* ℓ*, we call its subgraph* G_ℓ *a read-write pipeline with bottlenecks if* G_ℓ *consists of collections of nodes* $\mathbb{R}_1, \mathbb{R}_2, \ldots, \mathbb{R}_n$ *and* $\mathbb{W}_1, \mathbb{W}_2, \ldots, \mathbb{W}_n$ *with the following properties:*

- $\{A \mid A : \mathit{read}(\ell) \in \mathbb{V}\} \subseteq \bigcup_{i=1}^{n} \mathbb{R}_i$,
- $\{A \mid A : \mathit{write}(\ell) \in \mathbb{V}\} \subseteq \bigcup_{i=1}^{n} \mathbb{W}_i$,
- $\mathbb{R}_1, \ldots, \mathbb{R}_n, \mathbb{W}_1, \ldots, \mathbb{W}_n$, *restricted to* $\mathit{write}(\ell)$ *nodes and* $\mathit{read}(\ell)$ *nodes are pairwise disjoint,*
- *for each* $A : \mathit{read}(\ell) \in \mathbb{R}_i$ *and* $B : \mathit{write}(\ell) \in \mathbb{W}_i$, $A \rightsquigarrow B$,
- *for each* $\mathbb{R}_i$ *such that* $i \neq 1$*, there exists at least one* $A : \mathit{read}(\ell) \in \mathbb{R}_i$*, and*
- *there exists* $A : \mathit{write}(\ell) \in \mathbb{W}_i$ *for all* $i \neq n$ *such that for all* $B : \mathit{read}(\ell) \in \mathbb{R}_{i+1}$ *and all* $C : \mathit{write}(\ell) \in \mathbb{W}_i$, $A \overset{!}{\rightsquigarrow} B$ *and* $C \rightsquigarrow A$.

Here, each collection $\mathbb{R}_i$ and $\mathbb{W}_i$ corresponds to the collection of nodes marked by the same name in Figure 3.4 but with each node A_i included in the collection $\mathbb{W}_i$. The "bottlenecks" are the A_i's. Note that a trace graph $(\mathbb{V}, \mathbb{E})$ actually contains a read-write pipeline with bottlenecks per each reference location ℓ as a subgraph G_ℓ (but the subgraphs may not be disjoint because the paths may involve other locations and share join nodes).

The following theorem formalizes our earlier informal discussion.

Theorem 3.3.2. *A trace graph* $(\mathbb{V}, \mathbb{E})$ *is witness race free if and only if it has a read-write pipeline with bottlenecks for every* ℓ*.*

Proof. If

Suppose $(\mathbb{V}, \mathbb{E})$ has a read-write pipeline with bottlenecks for every ℓ. Let $\mathbb{R}_1, \mathbb{R}_2, \ldots, \mathbb{R}_n$ and $\mathbb{W}_1, \mathbb{W}_2, \ldots, \mathbb{W}_n$ be the collection of the nodes making up the read-write pipeline

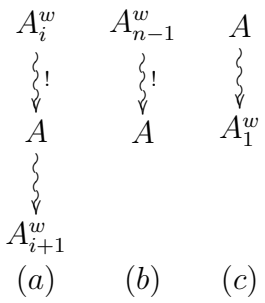

Figure 3.5: The three cases for partitioning read nodes with respect to A_i^w nodes.

with bottlenecks for ℓ. Let $A : \mathit{read}(\ell), B : \mathit{write}(\ell) \in \mathbb{V}$. Then $A \in \mathbb{R}_i$ and $B \in \mathbb{W}_j$ for some i, j. If $i \leq j$, then $A \rightsquigarrow B$. Otherwise, $i > j$ and $B \rightsquigarrow A$. Let $A : \mathit{read}(\ell), B : \mathit{write}(\ell) \in \mathbb{V}$ and $B \rightsquigarrow A$. Then $A \in \mathbb{R}_i$ for some $i \neq 0$. So there is $C : \mathit{write}(\ell) \in \mathbb{R}_{i-1}$ such that $C \overset{!}{\rightsquigarrow} A$.

Only If

Suppose $(\mathbb{V}, \mathbb{E})$ is witness race free. Let ℓ be a reference location. Let $A_1^w, A_2^w, \ldots, A_{n-1}^w$ be the $\mathit{write}(\ell)$ nodes such that for each A_i^w, there is some $B : \mathit{read}(\ell)$ such that $A_i^w \overset{!}{\rightsquigarrow} B$. As discussed above, $A_1^w, A_2^w, \ldots, A_{n-1}^w$ are in some total order. Without loss of generality, we assume that $A_1^w \rightsquigarrow A_2^w \rightsquigarrow \ldots \rightsquigarrow A_{n-1}^w$.

We construct the collections $\mathbb{R}_1, \mathbb{R}_2, \ldots, \mathbb{R}_n$ and $\mathbb{W}_1, \mathbb{W}_2, \ldots, \mathbb{W}_n$ as follows. First, we add the node A_i^w to the collection $\mathbb{W}_i$ for each $i \neq n$. Let $A : \mathit{read}(\ell)$. Suppose there is some $B : \mathit{write}(\ell)$ where $B \rightsquigarrow A$. Then $A_i^w \overset{!}{\rightsquigarrow} A$ for some i. As discussed above, it must be the case that $A \rightsquigarrow A_{i+1}^w$ if $i \neq n-1$. So we put A in the collection $\mathbb{R}_{i+1}$. Figure 3.5 depicts this case when (a) $i \neq n-1$ and (b) $i = n-1$.

Otherwise, there is no $B : \mathit{write}(\ell)$ such that $B \rightsquigarrow A$. Hence $A \rightsquigarrow C$ for any $C : \mathit{write}(\ell)$. In particular, $A \rightsquigarrow A_1^w$. So we put A in the collection $\mathbb{R}_1$. This case corresponds to the diagram (c) in Figure 3.5.

At this point, we have successfully partitioned $\mathit{read}(\ell)$ nodes with respect to A_i^w's.

What remains are the *write*(ℓ) nodes that are not among the A_i^w's. Let A : *write*(ℓ) be such a node. Suppose that there is no B : *read*(ℓ) such that $A \rightsquigarrow B$. Then it must be the case that for all B : *read*(ℓ), $B \rightsquigarrow A$. In particular, for all B : *read*(ℓ) $\in \mathbb{R}_n$, $B \rightsquigarrow A$. So we put such a node A in the collection $\mathbb{W}_n$. Otherwise, there exists B : *read*(ℓ) such that $A \rightsquigarrow B$. Let i be the largest such that there is no C : *read*(ℓ) $\in \mathbb{R}_i$ with $A \rightsquigarrow C$. Note that such i always exists since no *read*(ℓ) nodes in the collection $\mathbb{R}_1$ can be reached from a *write*(ℓ) node. So, for each C : *read*(ℓ) $\in \mathbb{R}_i$, we have $C \rightsquigarrow A$. And since i is the largest, there exists D : *read*(ℓ) $\in \mathbb{R}_{i+1}$ such that $A \rightsquigarrow D$. Therefore, $A \rightsquigarrow A_i^w$. Hence we can put the node A in the collection $\mathbb{W}_i$. □

Corollary 3.3.3. *A λ_{wit} program e is witness race free if and only if every trace graph of e has a read-write pipeline with bottlenecks for every reference location ℓ.*

3.3.1 Regions

Corollary 3.3.3 reduces the problem of deciding whether a program e is witness race free to the problem of deciding if every trace graph of e has a read-write pipeline with bottlenecks for every reference location ℓ. Therefore it suffices to design an algorithm for solving the latter problem. But before we do so, we make a slight change to λ_{wit} to make the problem more tractable. In λ_{wit}, there is a read-write pipeline with bottlenecks for each reference location ℓ, but distinguishing dynamically allocated reference locations individually is difficult for a compile-time algorithm. Therefore, we add *regions* to the language so that programs can explicitly group reference locations that are to be tracked together.

Figure 3.6 shows λ_{wit}^{reg}, λ_{wit} extended with regions. The syntax contains two new expression kinds: `letreg` x e which creates a new region and `ref` e e' which places the newly created reference in region e'; `ref` e e' replaces `ref` e. Figure 3.7 gives the semantics of λ_{wit}^{reg} which differs from λ_{wit} in two small ways. First, a state now

$$
\begin{array}{rcl}
e & := & x \mid i \mid \lambda x.e \mid e\ e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid e \otimes e' \mid \pi_i(e) \mid \texttt{write}\ e_1\ e_2\ e_3 \mid \texttt{read}\ e\ e' \\
& \mid & \texttt{ref}\ e\ e' \mid \texttt{join}\ e\ e' \mid \bullet \mid \texttt{letreg}\ x\ e
\end{array}
$$

Figure 3.6: The syntax of λ_{wit}^{reg}.

$$
\begin{array}{rcl}
E & := & D \cup \{a \mapsto E\} \mid [\,] \mid E\ e \mid e\ E \mid E \otimes e \mid e \otimes E \mid \pi_i(E) \\
& \mid & \texttt{write}\ E\ e\ e' \mid \texttt{write}\ e\ E\ e' \mid \texttt{write}\ e\ e'\ E \mid \texttt{read}\ e\ E \mid \texttt{read}\ E\ e \\
& \mid & \texttt{ref}\ E\ e \mid \texttt{ref}\ e\ E \mid \texttt{join}\ E\ e \mid \texttt{join}\ e\ E
\end{array}
$$

App	$(R, S, E[(\lambda x.e)\ e']) \Rightarrow (R, S, E[e[a/x]] \uplus \{a \mapsto e'\})$
Let	$(R, S, E[\texttt{let}\ x = e\ \texttt{in}\ e']) \Rightarrow (R, S, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\})$
Pair	$(R, S, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow (R, S, E[e_i])$
Write	$(R, S, E[\texttt{write}\ \ell\ e\ \bullet] \Rightarrow (R, S[\ell \leftarrow a], E[\bullet] \uplus \{a \mapsto e\})$
Read	$(R, S, E[\texttt{read}\ \ell\ \bullet]) \Rightarrow (R, S, E[S(\ell) \otimes \bullet])$
Ref	$(R, S, E[\texttt{ref}\ e\ r]) \Rightarrow (R, S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\})$
Join	$(R, S, E[\texttt{join}\ \bullet\ \bullet]) \Rightarrow (R, S, E[\bullet])$
LetReg	$(R, S, E[\texttt{letreg}\ x\ e]) \Rightarrow (R \uplus \{r\}, S, E[e[a/x]] \uplus \{a \mapsto r\})$
Arrive	$(R, S, E[a] \uplus \{a \mapsto e\}) \Rightarrow (R, S, E[e] \uplus \{a \mapsto e\})$ where $e \in V$
GC	$(R, S, D \uplus D') \Rightarrow (R, S, D)$ where $\diamond \notin \mathit{dom}(D') \wedge \mathit{dom}(D') \cap \mathit{free}(D) = \emptyset$

Figure 3.7: The semantics of λ_{wit}^{reg}.

contains a set of regions R. We use symbols r, r', r_i, etc. to denote regions. Regions are safe to duplicate, i.e., $r \in V$. The R's are used only for ensuring that the newly created region r at a **LetReg** step is fresh. (We overload the symbol $\uplus$ such that $R \uplus R' = R \cup R'$ if $R \cap R' = \emptyset$ and is undefined otherwise.) Note that evaluation contexts E do not extend to the subexpressions of $\texttt{letreg}\ x\ e$. The second difference is that a **Ref** step now takes a region r along with the initializer e to indicate that the newly created reference location ℓ belongs to the region r. Note that the semantics does not actually associate the reference location ℓ and the region r, and therefore grouping of reference locations via regions is entirely conceptual.[3]

[3]Regions are traditionally coupled with some semantic meaning such as memory management [TT94; GMJ+02]. It is possible to extend λ_{wit}^{reg} to do similar things with its regions.

Regions force programs to abide by witness race freedom at the granularity of regions instead of at the granularity of individual reference locations. That is, instead of *read*(ℓ) nodes and *write*(ℓ) nodes, we use *read*(r) nodes and *write*(r) nodes. Formally, a trace graph for λ^{reg}_{wit} is constructed by the following graph construction semantics:

Write $(R, K, S, E[\texttt{write}\ \ell\ e\ A]) \Rightarrow (R, K, S[\ell \leftarrow a], E[B] \uplus \{a \mapsto e\})$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where B is a new *write*($K(\ell)$) node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$

Read $(R, K, S, E[\texttt{read}\ \ell\ A]) \Rightarrow (R, K, S, E[S(\ell) \otimes B])$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where B is a new *read*($K(\ell)$) node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$

Join $(R, K, S, E[\texttt{join}\ A\ B]) \Rightarrow (R, K, S, E[C])$
$\mathbb{V} := \mathbb{V} \cup \{C\}$ where C is a new *join* node
$\mathbb{E} := \mathbb{E} \cup \{(A, C), (B, C)\}$

Ref $(R, K, S, E[\texttt{ref}\ e\ r]) \Rightarrow (R, K \uplus \{\ell \mapsto r\}, S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\})$

In the rules above, K is a mapping from reference locations to regions. The mapping K starts empty at the beginning of evaluation. Other reductions rules are unmodified except that the function K is passed from left to right in the obvious way.

Since there is less information available in a λ^{reg}_{wit} trace graph than in a λ_{wit} trace graph, the witness race freedom condition is more conservative for λ^{reg}_{wit}. That is, we still need the condition that for any A : *write*(r) and B : *read*(r), either $A \rightsquigarrow B$ or $B \rightsquigarrow A$. But we need to tighten the second condition so that for any A : *read*(r) if there are B_1, B_2 : *write*(r) such that $B_1 \rightsquigarrow A$ and $B_2 \rightsquigarrow A$, then either $B_1 \rightsquigarrow B_2$ or $B_2 \rightsquigarrow B_1$. This condition is strictly more conservative than for λ_{wit}, which only requires some C : *write*(r) such that $C \overset{!}{\rightsquigarrow} A$ in such a situation. The reason for this conservativeness is that we do not know from a trace graph of λ^{reg}_{wit} whether B_1 and

B_2 both write to the same reference location.

Formally, witness race freedom for λ^{reg}_{wit} can be defined as follows.

Definition 3.3.4 (Witness Race Freedom for λ^{reg}_{wit})**.** *We say that a λ^{reg}_{wit} trace graph $(\mathbb{V}, \mathbb{E})$ is witness race free if for every region r,*

- *for every $A : read(r) \in \mathbb{V}$ and $B : read(r) \in \mathbb{V}$, either $A \rightsquigarrow B$ or $B \rightsquigarrow A$, and*
- *for every $A : read(r) \in \mathbb{V}$ and $B_1, B_2 : write(r) \in \mathbb{V}$ such that $B_1 \rightsquigarrow A$ and $B_2 \rightsquigarrow A$, we have $B_1 \rightsquigarrow B_2$ or $B_2 \rightsquigarrow B_1$.*

Theorem 3.3.5. *If a λ^{reg}_{wit} program e is witness race free, then e is confluent.*

Proof. For any evaluation of e, carry out the same reduction sequence with the trace graph building action of λ_{wit}, i.e., the trace graph G generated is at the granularity of reference locations. Then it is easy to see that if G satisfies the above two conditions, G also satisfies the two conditions of Theorem 3.2.4. □

It is easy to see that Definition 3.3.4 is the weakest possible restriction to the original witness race freedom under the region abstraction because for any λ^{reg}_{wit} trace graph that is not witness race free, one can easily find a non-confluent program that produces the graph.

In a witness-race-free trace graph for λ^{reg}_{wit}, the read-write pipeline with bottlenecks for a region r consisting of the collections $(\mathbb{R}_1, \ldots, \mathbb{R}_n, \mathbb{W}_1, \ldots, \mathbb{W}_n)$ has the following property: each set $\{A \mid A : write(r) \in \mathbb{W}_i\}$ for $i \neq n$ can be totally ordered (with $\rightsquigarrow$ as the ordering relation). The theorem below is immediate from Corollary 3.3.3 under this additional property.

Theorem 3.3.6. *A λ^{reg}_{wit} program e is witness race free if and only if every trace graph of e has a read-write pipeline with bottlenecks for every region r.*

This additional property helps in designing a static checking algorithm.

3.3.2 From Network Flow to Types

Now our goal is to design an algorithm for statically checking if every trace graph of a λ^{reg}_{wit} program e has a read-write pipeline with bottlenecks for every region r. Our approach exploits a *network flow* property of read-write pipelines with bottlenecks. Consider a trace graph as a network of nodes with each edge (A, B) able to carry any non-negative flow from A to B. (Recall edges are directed.) As usual with network flow, we require that the total incoming flow equal the total outgoing flow for every node in the graph. Now, let us add a *virtual source* node A_S and connect it to every node B by adding an edge (A_S, B). We assign incoming flow 1 to A_S. Then it is not hard to see that if there exists a read-write pipeline with bottlenecks for the region r then there exists flow assignments such that every *read*(r) node and *write*(r) node gets a positive flow and every $A : \mathit{write}(r) \in \mathbb{W}_i$ for $i \neq n$ gets a flow equal to 1.

It turns out that the converse also holds. That is, given a trace graph, if there is a flow assignment such that each *read*(r) node and *write*(r) node gets a positive flow and each $A : \mathit{write}(r)$ that has some $B : \mathit{read}(r)$ such that $B \not\rightsquigarrow A$ gets a flow equal to 1, then there is a read-write pipeline with bottlenecks for the region r. By Theorem 3.3.6, this implies that there exists such a flow assignment for every region r if and only if the trace graph is witness race free. Because edges in a trace graph are traces of witnesses, our idea is to assign a type to a witness such that the type contains flow assignments for each (static) region. We use this idea to design a type system such that a well-typed program is guaranteed to be witness race free.

Formally, a witness type W is a function from the set of *static region identifiers* **RegIDs** to rational numbers in the range $[0, 1]$, i.e., $W : \mathbf{RegIDs} \to [0, 1]$. The rational number $W(\rho)$ indicates the flow amount for the static region identifier ρ in the witness type W. We use the notation $\{\rho_1 \mapsto q_1, \ldots, \rho_n \mapsto q_n\}$ to mean a witness type W such that $W(\rho) = q_i$ if $\rho = \rho_i$ for some $1 \leq i \leq n$ and $W(\rho) = 0$ otherwise.

$$\tau \quad := \quad \mathit{int} \mid \tau \xrightarrow{q} \tau' \mid \tau \otimes \tau' \mid \mathit{ref}(\tau, \tau', \rho) \mid \mathit{reg}(\rho) \mid W$$

Figure 3.8: The type language.

(We use the symbols q, q_i, q', etc for non-negative rational numbers, including those larger than 1.)

The rest of the types are defined in Figure 3.8. Types include integer types *int*, function types $\tau \xrightarrow{q} \tau'$, pair types $\tau \otimes \tau'$, reference types $\mathit{ref}(\tau, \tau', \rho)$, and region types $\mathit{reg}(\rho)$. The non-negative rational number q in $\tau \xrightarrow{q} \tau'$ represents the number of times the function can be called. We allow the symbols q, q', etc to take the valuation ∞ to imply that the function can be called arbitrarily many times. We use the following arithmetic relation: $q + \infty = \infty$, $q \times \infty = \infty$ for $q \neq 0$, and $0 \times \infty = 0$.

Figure 3.9 and Figure 3.10 show the main type judgment rules. Our type system belongs to the family of substructural type systems, which includes linear types. We discuss the rules from top-to-bottom and left-to-right, except for **Sub** which we defer to the end. **Var** and **Int** are standard. **Dummy** gives a dummy witness $\bullet$ an empty witness type; note that $\emptyset(\rho) = 0$ for any static region identifier ρ.

Source uses additive arithmetic over types defined in Figure 3.11. The rule adds W_3 amount of flow from the virtual source nodes (i.e., nodes A_S from the first paragraph of this section) to W_2. In the type judgment relation $\Gamma; W \vdash e : \tau$, the witness type W represents the flow the expression e receives from the virtual source nodes. Therefore, **Source** says that assuming that we took W_1 flow from the virtual source nodes in the precondition, we are now taking W_3 more.

In **Abs**, we multiply the left hand side of the judgments by the number of times that the function can be used. Multiplication over type environments Γ is defined as follows:

$$(\Gamma, x{:}\tau) \times q = (\Gamma \times q), x{:}(\tau \times q)$$

$$\frac{\Gamma; W \vdash e : \tau \qquad \tau \geq \tau'}{\Gamma; W \vdash e : \tau'} \textbf{ Sub} \qquad \frac{\Gamma(x) = \tau}{\Gamma; W \vdash x : \tau} \textbf{ Var}$$

$$\frac{}{\Gamma; W \vdash i : \mathit{int}} \textbf{ Int} \qquad \frac{}{\Gamma; W \vdash \bullet : \emptyset} \textbf{ Dummy}$$

$$\frac{\Gamma; W_1 \vdash e : W_2}{\Gamma; W_1 + W_3 \vdash e : W_2 + W_3} \textbf{ Source}$$

$$\frac{\Gamma, x{:}\tau; W \vdash e : \tau'}{\Gamma \times q; W \times q \vdash \lambda x.e : \tau \xrightarrow{q} \tau'} \textbf{ Abs}$$

$$\frac{\Gamma; W \vdash e : \tau \xrightarrow{q} \tau' \qquad \Gamma'; W' \vdash e' : \tau \qquad q \geq 1}{\Gamma + \Gamma'; W + W' \vdash e\ e' : \tau'} \textbf{ App}$$

$$\frac{\Gamma; W \vdash e : \tau \qquad \Gamma'; W' \vdash e' : \tau'}{\Gamma + \Gamma'; W + W' \vdash e \otimes e' : \tau \otimes \tau'} \textbf{ Pair}$$

$$\frac{\Gamma; W \vdash e : \tau_1 \otimes \tau_2}{\Gamma; W \vdash \pi_i(e) : \tau_i} \textbf{ Proj}$$

$$\frac{\Gamma; W \vdash e : \tau \qquad \Gamma'; W' \vdash e' : \mathit{reg}(\rho)}{\Gamma + \Gamma'; W + W' \vdash \texttt{ref}\ e\ e' : \mathit{ref}\,(\tau, \tau, \rho)} \textbf{ Ref}$$

Figure 3.9: Type judgment rules I.

So for example, if $\lambda x.e$ captures a witness as a free variable y and that $\Gamma(y) = W$, then $(\Gamma \times q)(y) = W \times q$. Thus if the function body requires W amount of flow in the witness, then we actually require $W \times q$ amount of flow because the function may be called q times.

In **App**, the precondition $q \geq 1$ says that the number of times the function can be used must be at least 1. The left hand side of the two judgments in the precondition are added so that we can compute the combined flow required for the expressions e

$$\frac{\begin{array}{cc}\Gamma_1; W_1 \vdash e_1 : \mathit{ref}(\tau, \tau', \rho) & \Gamma_2; W_2 \vdash e_2 : \tau' \\ \Gamma_3; W_3 \vdash e_3 : W & W(\rho) \geq 1\end{array}}{\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3 \vdash \texttt{write}\, e_1\, e_2\, e_3 : W}\ \textbf{Write}$$

$$\frac{\Gamma; W_1 \vdash e : \mathit{ref}(\tau, \tau', \rho) \qquad \Gamma'; W_2 \vdash e' : W \qquad W(\rho) > 0}{\Gamma + \Gamma'; W_1 + W_2 \vdash \texttt{read}\, e\, e' : W \otimes \tau}\ \textbf{Read}$$

$$\frac{\begin{array}{c}\Gamma, x : \mathit{reg}(\rho); W + \{\rho \mapsto q\} \vdash e : \tau \\ q \leq 1 \qquad \rho \notin \mathit{free}(\Gamma) \cup \mathit{free}(W) \cup \mathit{free}(\tau)\end{array}}{\Gamma; W \vdash \texttt{letreg}\, x\, e : \tau}\ \textbf{LetRegion}$$

$$\frac{\Gamma; W_1 \vdash e : W \qquad \Gamma; W_2 \vdash e' : W'}{\Gamma + \Gamma'; W_1 + W_2 \vdash \texttt{join}\, e\, e' : W + W'}\ \textbf{Join}$$

$$\frac{\Gamma; W \vdash e'[e/x] : \tau \qquad e \in V \qquad x \notin \mathit{free}(e)}{\Gamma; W \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : \tau}\ \textbf{LetA}$$

$$\frac{\begin{array}{c}\Gamma, x : \tau; W \vdash e : \tau \qquad \Gamma', x : \tau; W' \vdash e' : \tau' \\ \tau \geq \tau \times \infty \text{ if } x \in \mathit{free}(e)\end{array}}{\Gamma + \Gamma'; W + W' \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : \tau'}\ \textbf{LetB}$$

Figure 3.10: Type judgment rules II.

and e'. Addition over type environments is defined as follows:

$$(\Gamma, x : \tau) + (\Gamma', x : \tau') = (\Gamma + \Gamma'), x : (\tau + \tau')$$

Pair and **Proj** are self-explanatory.

In a reference type $\mathit{ref}(\tau, \tau', \rho)$, the static region identifier ρ identifies the region where the reference belongs while the type τ is the read type of the reference and the type τ' is the write type of the reference. Initially the read and write types are the same as seen in **Ref**. **Write** matches the type of the to-be-assigned expression e_2 with the write type of the reference while **Read** uses the read type of the reference type. We require $W(\rho) \geq 1$ at **Write** and $W(\rho) > 0$ at **Read**; both correspond to the flow requirement for writes and reads. The reason for read-type/write-type

$$
\begin{array}{l}
\mathit{reg}(\rho) + \mathit{reg}(\rho) = \mathit{reg}(\rho) \\
\mathit{int} + \mathit{int} = \mathit{int} \\
\tau \xrightarrow{q} \tau' + \tau \xrightarrow{q'} \tau' = \tau \xrightarrow{q+q'} \tau' \\
\tau_1 \otimes \tau_2 + \tau_3 \otimes \tau_4 = (\tau_1 + \tau_3) \otimes (\tau_2 + \tau_4) \\
\mathit{ref}\,(\tau_1, \tau, \rho) + \mathit{ref}\,(\tau_2, \tau, \rho) = \mathit{ref}\,(\tau_1 + \tau_2, \tau, \rho) \\
W + W' = \{\rho \mapsto W(\rho) + W'(\rho) \mid \rho \in \mathbf{RegIDs}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{reg}(\rho) \times q = \mathit{reg}(\rho) \\
\mathit{int} \times q = \mathit{int} \\
\tau \xrightarrow{q'} \tau' \times q = \tau \xrightarrow{q' \times q} \tau' \\
\tau \otimes \tau' \times q = (\tau \times q) \otimes (\tau' \times q) \\
\mathit{ref}\,(\tau, \tau', \rho) \times q = \mathit{ref}\,(\tau \times q, \tau', \rho) \\
W \times q = \{\rho \mapsto W(\rho) \times q \mid \rho \in \mathbf{RegIDs}\}
\end{array}
$$

Figure 3.11: Arithmetic over types.

separation is subtle. Consider the following expression where the expressions e_1 and e_3 are witnesses and the expression e_2 is a region:

$$\texttt{let}\ x = (\texttt{ref}\ e_1\ e_2)\ \texttt{in}\ \texttt{let}\ w = (\texttt{write}\ x\ e_3\ \bullet)\ \texttt{in}\ \texttt{read}\ x\ w$$

Suppose we just have read types so that the type system uses read types at instances **Write** as well as at instances **Read**. Then the type system is unsound (even without **Sub**) for the following reason. The type system may assign some flow W to the occurrence of the variable x at the write and some flow W' to the occurrence of the variable x at the read. But there is no constraint to force $W = W'$, so the type system can let $W' > W$ while keeping the sum $W + W'$ fixed, i.e., we get more flow from a reference than what was assigned to the reference. Separating read and write types prevents this problem because addition and multiplication do not act on write types.

LetRegion introduces a fresh static region identifier ρ. The witness type $\{\rho \mapsto q\}$

$$\frac{}{\tau \geq \tau} \qquad \frac{\tau_1 \leq \tau_1' \quad \tau_2 \geq \tau_2' \quad q \geq q'}{\tau_1 \xrightarrow{q} \tau_2 \geq \tau_1' \xrightarrow{q'} \tau_2'} \qquad \frac{\tau_1 \geq \tau_1' \quad \tau_2 \geq \tau_2'}{\tau_1 \otimes \tau_2 \geq \tau_1' \otimes \tau_2'}$$

$$\frac{\tau_1 \geq \tau_1' \qquad \tau_2 \leq \tau_2'}{\mathit{ref}\,(\tau_1, \tau_2, \rho) \geq \mathit{ref}\,(\tau_1', \tau_2', \rho)} \qquad \frac{W(\rho) \geq W(\rho) \text{ for all } \rho \in \mathbf{RegIDs}}{W \geq W'}$$

Figure 3.12: Subtyping.

represents the virtual source node for the new region. We constrain $q \leq 1$ to ensure that we do not use more than 1 unit total from the source.

Join combines two witnesses by adding their types.

There are two rules, **LetA** and **LetB**, for the expression kind `let` x `=` e `in` e'. **LetA** is less conservative and should be used whenever x occurs more than once in e' and $e \in V$. This rule corresponds to the usual substitution interpretation of let-based predicative polymorphism with the value restriction. **LetB** is used if $e \notin V$ or x occurs at most once in e'. Here, $\mathit{free}(\tau)$ is the set of static region identifiers in the type τ where $\mathit{free}(W) = \{\rho \mid W(\rho) \neq 0\}$, and $\mathit{free}(\Gamma) = \bigcup_{\tau \in ran(\Gamma)} \mathit{free}(\tau)$.

Finally, we return to **Sub**. The subtyping relation is defined in Figure 3.12. As usual, argument types of function types are contravariant. Write types of reference types are also contravariant; this treatment of reference subtyping is identical to that of a type-based formulation of Andersen's points-to analysis [FFSA98]. Intuitively, the rule **Sub** expresses the observation that the flow graph property may be relaxed so that the sum of the outgoing flow can be less than the sum of the incoming flow, i.e., if we could find a flow assignment satisfying the required flow constraints at reads and writes under this relaxed condition, then we still have a read-write pipeline with bottlenecks.

We say that a λ_{wit}^{reg} program e is well-typed if $\emptyset; \emptyset \vdash e : \tau$ for some type τ. The following theorem states that the type system is sound.

Theorem 3.3.7. *If a λ_{wit}^{reg} program e is well-typed, then e is witness race free.*

The proof appears in Section 3.5.

We point out a few of the positive properties of this type system. If a program contains no reads or writes and can be typed by a standard Hindley-Milner polymorphic type system, then it can also be typed by our type system; we may use the qualifier ∞ for all function types and use 0 for all flows. In general, we can give the ∞ qualifier to the function type of any function that does not capture a witness (directly or indirectly). We can also assign 0 to any flow for a region r that does not flow into a reference operating on the region r.

The type system is quite expressive. In particular, it is able to type all of the confluent examples that were used up to this point in the chapter (with straightforward modification to translate λ_{wit} programs into λ_{wit}^{reg}). In fact, assuming that $write(r)$ nodes in each collection $\mathbb{W}_i$ are totally ordered for each r, the type system is complete for the first-order fragment (i.e., no higher order functions) with no recursion and no storing of witnesses in references.

The limitations of the type system are the standard ones: let-based predicative polymorphism, flow-insensitivity of reference types, and unsoundness under call-by-name semantics; the latter is a typical limitation of a non-linear substructural type system. Another limitation is that the type system enforces $write(r)$ nodes in every collection $\mathbb{W}_i$ to be totally ordered for each r whereas witness race freedom permits an absence of ordering for the case $i = n$; we believe that this is a minor limitation.

3.3.3 Inference

We next present the type inference algorithm. By Theorem 3.3.7, this results in an algorithm for statically checking witness race freedom.

At a high-level, our type system is a standard Hindley-Milner type system with some additional rational arithmetic constraints. Therefore we could perform infer-

$$
\begin{array}{l}
\mathit{Fresh}(\mathit{int}) = \mathit{int} \\
\mathit{Fresh}(\sigma{\rightarrow}\sigma) = \mathit{Fresh}(\sigma) \xrightarrow{\beta} \mathit{Fresh}(\sigma) \text{ where } \beta \text{ is fresh} \\
\mathit{Fresh}(\sigma \otimes \sigma') = \mathit{Fresh}(\sigma) \otimes \mathit{Fresh}(\sigma) \\
\mathit{Fresh}(\mathit{ref}\,(\sigma, \sigma', \rho)) = \mathit{ref}\,(\mathit{Fresh}(\sigma), \mathit{Fresh}(\sigma'), \rho) \\
\mathit{Fresh}(\mathit{reg}(\rho)) = \mathit{reg}(\rho) \\
\mathit{Fresh}(I) = \{\rho \mapsto \alpha \mid \rho \in I\} \text{ where } \alpha \text{ is fresh}
\end{array}
$$

Figure 3.13: *Fresh*.

$$
\frac{\Gamma, W \vdash_b e^I : \tau, \mathcal{C}}{\Gamma, W + \mathit{Fresh}(I) \vdash_a e^I : \tau + \mathit{Fresh}(I), \mathcal{C}} \qquad \frac{\Gamma, W \vdash_b e^\sigma : \tau, \mathcal{C} \qquad \sigma \notin \text{type } I}{\Gamma, W \vdash_a e^\sigma : \tau, \mathcal{C}}
$$

Figure 3.14: Type inference $\vdash_a$.

ence by employing a standard type inference technique to solve all type-structural constraints while generating rational arithmetic constraints on the side, and then solving the generated arithmetic constraints separately. Unfortunately, the arithmetic constraints may be non-linear since they involve the multiplication of variables. Because there is no efficient algorithm for solving general non-linear rational arithmetic constraints, we need to dive into lower-level details of the type system.

Let us separate type inference into two phases. The first phase carries out type inference after erasing all rational numbers from the type system. That is, the types inferred in this phase are:

$$
\sigma \quad := \quad \mathit{int} \mid \sigma{\rightarrow}\sigma' \mid \sigma \otimes \sigma' \mid \mathit{ref}\,(\sigma, \sigma', \rho) \mid \mathit{reg}(\rho) \mid I
$$

where a type I is a subset of **RegIDs**. Intuitively, each type I represents the non-0 domain of some witness type W. The first phase can be carried out by a standard Hindley-Milner type inference, albeit with regions, which is no harder than type variables. We omit the details of this phase. We may safely reject the program if the

$$\frac{\tau = \mathit{Fresh}(\sigma)}{\{x{:}\tau\}, \emptyset \vdash_b x^{\sigma} : \tau, \emptyset} \qquad \frac{}{\emptyset, \emptyset \vdash_b i : \mathit{int}, \emptyset} \qquad \frac{}{\emptyset, \emptyset \vdash_b \bullet : \emptyset, \emptyset}$$

$$\frac{\Gamma, W \vdash_a e : \tau, \mathcal{C} \qquad \beta \text{ is fresh}}{\Gamma \times \beta, W \times \beta \vdash_b \lambda x.e : \Gamma(x) \xrightarrow{\beta} \tau, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \qquad \tau = \mathit{Fresh}(\sigma) \qquad \tau' = \mathit{Fresh}(\sigma') \\ \beta \text{ is fresh} \qquad \Gamma_3 = \Gamma_1 + \Gamma_2 / \mathcal{C}_3 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \geq \tau \xrightarrow{\beta} \tau', \beta \geq 1, \tau_2 \geq \tau\}\end{array}}{\Gamma_3, W_1 + W_2 \vdash_b e_1^{\sigma \to \sigma'}\ e_2 : \tau', \mathcal{C}}$$

$$\frac{\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \qquad \Gamma_3 = \Gamma_1 + \Gamma_2 / \mathcal{C}_3}{\Gamma_3, W_1 + W_2 \vdash_b e_1 \otimes e_2 : \tau_1 \otimes \tau_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$\frac{\Gamma, W \vdash_a \pi_i(e) : \tau, \mathcal{C} \qquad \tau_1 = \mathit{Fresh}(\sigma) \qquad \tau_2 = \mathit{Fresh}(\sigma')}{\Gamma, W \vdash_b \pi_i(e^{\sigma_1 \otimes \sigma_2}) : \tau_i, \mathcal{C} \cup \{\tau \geq \tau_1 \otimes \tau_2\}}$$

$$\frac{\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \qquad \Gamma_3 = \Gamma_1 + \Gamma_2 / \mathcal{C}_3}{\Gamma_3, W_1 + W_2 \vdash_b \texttt{ref}\ e_1\ e_2^{\mathit{reg}(\rho)} : \mathit{ref}(\tau_1, \tau_2, \rho), \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

Figure 3.15: Type inference $\vdash_b$ II.

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ \Gamma_3, W_3 \vdash_a e_2 : \tau_3, \mathcal{C}_3 \qquad \tau = \mathit{Fresh}(\sigma) \qquad \tau' = \mathit{Fresh}(\sigma') \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \geq \mathit{ref}(\tau, \tau', \rho), \tau_2 \geq \tau', \tau_3(\rho) \geq 1\} \\ \Gamma = \Gamma_1 + \Gamma_2 + \Gamma_3/\mathcal{C}_3 \qquad W = W_1 + W_2 + W_3\end{array}}{\Gamma, W \vdash_b \texttt{write}\, e_1^{\mathit{ref}(\sigma,\sigma',\rho)}\, e_2\, e_3 : \tau_3, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ \tau = \mathit{Fresh}(\sigma) \qquad \tau' = \mathit{Fresh}(\sigma') \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \geq \mathit{ref}(\tau, \tau', \rho), \tau_2(\rho) > 0\} \\ \Gamma_3 = \Gamma_1 + \Gamma_2/\mathcal{C}_3\end{array}}{\Gamma_3, W_1 + W_2 \vdash_b \texttt{read}\, e_1^{\mathit{ref}(\sigma,\sigma',\rho)}\, e_2 : \tau \otimes \tau_2, \mathcal{C}}$$

$$\frac{\Gamma, W \vdash_a e : \tau, \mathcal{C}}{\Gamma, W \vdash_b \texttt{letreg}\, x^{\mathit{reg}(\rho)}\, e : \tau, \mathcal{C} \cup \{W(\rho) \leq 1\}}$$

$$\frac{\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \qquad \Gamma_3 = \Gamma_1 + \Gamma_2/\mathcal{C}_3}{\Gamma_3, W_1 + W_2 \vdash_b \texttt{join}\, e_1\, e_2 : \tau_1 + \tau_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 K \\ x \notin \mathit{free}(e_1) \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \geq \Gamma_2(x)\} \qquad \Gamma_3 = \Gamma_1 + \Gamma_2/\mathcal{C}_3\end{array}}{\Gamma_3, W_1 + W_2 \vdash_b \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ x \in \mathit{free}(e_1) \qquad \Gamma_3 = \Gamma_1 + \Gamma_2/\mathcal{C}_3 \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \geq \Gamma_1(x) \times \infty, \tau_1 \geq \Gamma_2(x)\}\end{array}}{\Gamma_3, W_1 + W_2 \vdash_b \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2, \mathcal{C}}$$

Figure 3.16: Type inference $\vdash_b$ II.

$$
\begin{array}{l}
reg(\rho) + reg(\rho) = reg(\rho)\emptyset \\
int + int = int\emptyset \\
\tau_1 \xrightarrow{q} \tau_1' + \tau_2 \xrightarrow{q'} \tau_2' = \tau_1 \xrightarrow{q+q'} \tau_1' / \{\tau_1 = \tau_2, \tau_1' = \tau_2'\} \\
\tau_1 \otimes \tau_2 + \tau_3 \otimes \tau_4 = \tau \otimes \tau' / \mathcal{C} \cup \mathcal{C}' \text{ where } \begin{array}{l} \tau_1 + \tau_3 = \tau / \mathcal{C} \\ \tau_2 + \tau_4 = \tau' / \mathcal{C}' \end{array} \\
ref\,(\tau_1, \tau_3, \rho) + ref\,(\tau_2, \tau_4, \rho) = ref\,(\tau', \tau_3, \rho) / \{\tau_3 = \tau_4\} \cup \mathcal{C} \text{ where } \tau_1 + \tau_2 = \tau' / \mathcal{C} \\
W + W' = \\
\quad \{\rho \mapsto W(\rho) + W'(\rho) \mid \rho \in dom(W) \wedge \rho \in dom(W')\} \\
\quad \cup \{\rho \mapsto W(\rho) \mid \rho \in dom(W) \wedge \rho \notin dom(W')\} \\
\quad \cup \{\rho \mapsto W'(\rho) \mid \rho \in dom(W') \wedge \rho \notin dom(W)\} / \emptyset
\end{array}
$$

Figure 3.17: Addition over types for type inference.

first phase fails. Otherwise we annotate each subexpression e by its inferred type σ: e^{σ}. In the second phase, we use the annotated program to generate the appropriate rational arithmetic constraints via bottom-up type-inference. Let e be an annotated program. Then the generated constraints for e is $\mathcal{C}$ where $\Gamma, W \vdash_a e : \tau, C$ for some Γ, W, and τ.

The second-phase type inference rules are separated into two kinds, $\vdash_a$ (Figure 3.14) and $\vdash_b$ (Figure 3.15 and Figure 3.16), which must occur in strictly interleaving manner. The purpose of $\vdash_a$ is to account for the type judgment rule **Source**, whereas $\vdash_b$ accounts for all other rules.

We should note that, strictly speaking, types τ appearing in the algorithm are different from the ones in the type judgment rules. That is, instead of rational numbers, the types τ in the algorithm are qualified by *rational number variables* α, β, γ, etc. Also the domain of a witness type W is not the entire **RegIDs** set but only some subset of it. In other words, a witness type W is a partial function from **RegIDs** to rational number variables. Also, because we will be adding types containing different rational number variables, we augment addition of types to emit constraints as shown in Figure 3.17. We also re-define the addition of type environments such that

$\Gamma_1 + \Gamma_2 = \Gamma/\mathcal{C}$ where for each $x \in dom(\Gamma_1) \cup dom(\Gamma_2)$, there exists $\mathcal{C}_x$ such that

- $\Gamma(x) = \Gamma_1(x) + \Gamma_2(x)\ \mathcal{C}_x$ if $x \in dom(\Gamma_1)$ and $x \in dom(\Gamma_1)$,
- $\Gamma(x) = \Gamma_i(x)$ and $\mathcal{C}_x = \emptyset$ if $x \in dom(\Gamma_i)$ and $x \notin dom(\Gamma_j)$ for $i, j \in \{1, 2\}$,

and $\mathcal{C} = \bigcup_{x \in dom(\Gamma_1) \cup dom(\Gamma_2)} \mathcal{C}_x$. Without loss of generality, we assume that bound variables are distinct.

We omit annotations when they are not used (i.e., we say e instead of e^σ, etc.). There are only two cases for $\vdash_a$. The first case is for expressions that were given a witness type I in the first phase. In this case, we add $\mathit{Fresh}(I)$ to τ and W to account for a possible application of **Source**. *Fresh* is defined in Figure 3.13. The second case is for expressions that were not given a witness type. In this case, we simply pass the result of the subderivation $\vdash_b$ up.

We discuss a few representative $\vdash_b$ rules. Note that $\vdash_b$ rules are syntax directed. In the case of a variable x^σ, we create a fresh τ from σ and pass $\{x : \tau\}, \emptyset \vdash_b x^\sigma : \tau, \emptyset$ up to the parent derivation. (Recall our type inference is bottom-up.) The case for integers and dummy witnesses are trivial. In the case of an abstraction $\lambda x.e$, we multiply Γ and W passed from the subderivation by β. In the case of a function application $e_1^{\sigma \to \sigma'}\ e_2$, we add the constraints $\{\tau_1 \geq \tau \xrightarrow{\beta} \tau', \beta \geq 1, \tau_2 \geq \tau\}$ to connect arguments and returns as well as requiring β to be at least 1. Note that the type rule **Sub** is implicitly incorporated in the constraints. In the case of $\texttt{write}\, e_1^{\mathit{ref}(\sigma,\sigma',\rho)}\, e_2\, e_3$, we add the constraint $\tau_2(\rho) \geq 1$ to match the type rule **Write**. The first phase guarantees that $\rho \in dom(\tau_3)$. In the case $\texttt{letreg}\ x^{\mathit{reg}(\rho)}\ e$, the constraint $W(\rho) \leq 1$ is effective only when $\rho \in dom(W)$ as $\rho \notin dom(W)$ implies that the region was not used at all. There is no case corresponding to the type rule **LetA**. Prior to running the algorithm, we replace each occurrence of the expression $\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'$ in the program by the expression $e'[e/x]$ whenever $e \in V$, $x \notin \mathit{free}(e)$, and x occurs more than once in e'.

$$\{r : \mathit{reg}(\rho)\}; \emptyset \vdash_a \texttt{ref}\ 1\ r : \mathit{ref}(\mathit{int}, \mathit{int}, \rho), \emptyset$$

$$\Gamma; \{\rho \mapsto \gamma_1 + \gamma_2\} \vdash_a \texttt{write}\ x\ 2\ \bullet : \{\rho \mapsto \gamma_1 + \gamma_2\}, \{\gamma_1 \geq 1\}$$
where $\Gamma = \{x : \mathit{ref}(\mathit{int}, \mathit{int}, \rho)\}$

$$\Gamma; W \vdash_a \lambda y.\texttt{read}\ x\ w : \mathit{int} \xrightarrow{\beta_1} \mathit{int} \otimes \{\rho \mapsto \alpha_1 + \gamma_3\}, \mathcal{C}$$
where $\Gamma = \{x : \mathit{ref}(\mathit{int}, \mathit{int}, \rho), w : \{\rho \mapsto \alpha_1 \times \beta_1\}\}$
and $W = \{\rho \mapsto \gamma_3 \times \beta_1\}$
and $\mathcal{C} = \{\alpha_1 + \gamma_3 > 0\}$

$$\Gamma; \emptyset \vdash_a (f\ 0) \otimes (f\ 0) : \tau, \mathcal{C}$$
where $\Gamma = \{f : \mathit{int} \xrightarrow{\beta_2+\beta_3} \mathit{int} \otimes \{\rho \mapsto \alpha_2\}\}$
and $\tau = (\mathit{int} \otimes \{\rho \mapsto \alpha_2\}) \otimes (\mathit{int} \otimes \{\rho \mapsto \alpha_2\})$
and $\mathcal{C} = \{\beta_2 \geq 1, \beta_3 \geq 1\}$

$$\Gamma; \{\rho \mapsto \gamma_4\} \vdash_a \texttt{write}\ x\ 3 \ldots : \{\rho \mapsto \alpha_3 + \alpha_4 + \gamma_4\}, \mathcal{C}$$
where $\Gamma = \{x : \mathit{ref}(\mathit{int}, \mathit{int}, \rho), z : \tau\}$
and $\tau = (\mathit{int} \otimes \{\rho \mapsto \alpha_3\}) \otimes (\mathit{int} \otimes \{\rho \mapsto \alpha_4\})$
and $\mathcal{C} = \{\alpha_3 + \alpha_4 + \gamma_4 \geq 1\}$

Figure 3.18: Generated constraints.

As an example, consider the following program (a λ^{reg}_{wit} version of the last example from Section 3.1):

$$\begin{array}{l}
\texttt{letreg}\ r \\
\quad \texttt{let}\ x = \texttt{ref}\ 1\ r\ \texttt{in} \\
\quad\quad \texttt{let}\ w = \texttt{write}\ x\ 2\ \bullet\ \texttt{in} \\
\quad\quad\quad \texttt{let}\ f = \lambda y.\texttt{read}\ x\ w\ \texttt{in} \\
\quad\quad\quad\quad \texttt{let}\ z = (f\ 0) \otimes (f\ 0)\ \texttt{in} \\
\quad\quad\quad\quad\quad \texttt{let}\ w = \texttt{write}\ x\ 3\ \texttt{join}\ \pi_2(\pi_1(z))\ \pi_2(\pi_2(z))\ \texttt{in}\ z
\end{array}$$

Suppose the first phase assigns r the type $\mathit{reg}(\rho)$. Assume each `let`-bound variable is treated monomorphically. The second phase generates the constraints shown in

Figure 3.18 for the `let`-bound expressions (slightly simplified for readability). The final constraints, after some simplification, is as follows:

$$\gamma_1 \geq 1, 1 \geq \gamma_1 + \gamma_2 + \gamma_3 \times \beta_1 + \gamma_4, \beta_1 \geq \beta_2 + \beta_3,$$
$$\beta_2 \geq 1, \beta_3 \geq 1, \gamma_1 + \gamma_2 \geq \alpha_1 \times \beta_1, \alpha_1 + \gamma_3 > 0,$$
$$\alpha_1 + \gamma_3 \geq \alpha_2, \alpha_3 + \alpha_4 + \gamma_4 \geq 1, \alpha_2 \geq \alpha_3, \alpha_2 \geq \alpha_4$$

The constraints are satisfiable, e.g., by the substitution

$$\beta_1 = 2 \quad \beta_2 = \beta_3 = \gamma_1 = 1 \quad \alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.5 \quad \gamma_2 = \gamma_3 = \gamma_4 = 0$$

In general, a program e is well-typed if and only if the constraints $\mathcal{C}$ generated by type inference are satisfiable. So it suffices to show that the satisfaction problem for any $\mathcal{C}$ generated by the type inference algorithm can be solved.

To this end, we first observe that because of the first phase, any constraint $\tau \geq \tau' \in \mathcal{C}$ can be reduced to a set of rational arithmetic constraints of the form $p \geq p'$ where p, p' are rational polynomials. The troublesome non-linearity comes from $\Gamma \times \beta$ and $W \times \beta$ in the $\lambda x.e$ case. Let us focus our attention on the set $\mathcal{B}$ of variables used in such multiplications. (We have used the meta-variable β instead of α just for this case in the pseudo-code to make it clear that these variables are special.) We can show that the following holds:

Theorem 3.3.8. *Let $p \rhd p' \in \mathcal{C}$ where $\rhd \in \{\geq, >\}$. If $\beta \in \mathcal{B}$ occurs in the polynomial p, then it must be the case that $\rhd = \geq$, $p = \beta$, and that the polynomial p' consists only of symbols in the set $\{+, \times, 1, \infty\} \cup \mathcal{B}$.*

Proof. Let $\Gamma, W \vdash_a e : \tau, \mathcal{C}$. For any $\tau' \in \Gamma$, $\times$ and $+$ only appear at the top-level, i.e., not within argument and return types of a function type. Secondly, we can show by induction that within the type τ, $\times$ only appears in negative positions. More

precisely, for any $p \in Pos(\tau)$, the polynomial p contains no $\times$ where Pos is defined as follows:

$$
\begin{aligned}
&Pos(\tau \xrightarrow{p} \tau') = Neg(\tau) \cup Pos(\tau') \cup \{p\} \\
&Pos(\tau \otimes \tau') = Pos(\tau) \cup Pos(\tau') \\
&Pos(ref\,(\tau, \tau', \rho)) = Pos(\tau) \cup Neg(\tau') \\
&Pos(W) = ran(W) \\
&Neg(int) = Pos(int) = Neg(reg(\rho)) = Pos(reg(\rho)) = \emptyset \\
&Neg(\tau \xrightarrow{p} \tau') = Pos(\tau) \cup Neg(\tau') \\
&Neg(\tau \otimes \tau') = Neg(\tau) \cup Neg(\tau') \\
&Neg(ref\,(\tau, \tau', \rho)) = Neg(\tau) \cup Pos(\tau') \\
&Neg(W) = \emptyset
\end{aligned}
$$

Third, for any $+$ that appears in a positive position of τ, i.e. in some $p \in Pos(\tau)$, the polynomial p does not contain any $\beta \in \mathcal{B}$. Then the result follows from inspection of the subtyping rules. □

The theorem implies that we can compute all assignments to the variables in $\mathcal{B}$ by computing the minimum satisfying assignment for $\mathcal{C}' = \{\beta \geq p \mid \beta \in \mathcal{B}\} \subseteq \mathcal{C}$. It is easy to see that such an assignment always exists. (Recall that the range is non-negative.) This problem can be solved in quadratic time by an iterative method in which all variables are initially set to 0, and at each iteration the new values for the variables are computed by taking the maximum of the right hand polynomials evaluated at the current values. It is possible to show that if the minimum satisfying assignment for a variable β is some $q < \infty$, then the iterative method finds q for β in $2|\mathcal{C}'|$ iterations. Hence any variable changing after the $2|\mathcal{C}'|$th iteration can be safely set to ∞. All variables are then guaranteed to converge within $3|\mathcal{C}'|$ iterations. Because each iteration examines every constraint, the overall time is at most quadratic

App	$(L, E[(\lambda x.e)\ e']) \Rightarrow_{alt} (L, E[e[a/x]] \uplus \{a \mapsto e'\})$
Let	$(L, E[\texttt{let}\ x = e\ \texttt{in}\ e']) \Rightarrow_{alt} (L, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\})$
Pair	$(L, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow_{alt} (L, E[e_i])$
Write	$(L, E[\texttt{write}\ \langle \ell, a' \rangle\ e\ T]) \Rightarrow_{alt} (L, E[\mathbf{write}(T, \ell, a)] \uplus \{a \mapsto e\})$
Read	$(L, E[\texttt{read}\ \langle \ell, a \rangle\ T]) \Rightarrow_{alt} (L, E[\mathbf{read}(T, \ell, a) \otimes T])$
Ref	$(L, E[\texttt{ref}\ e]) \Rightarrow_{alt} (L \uplus \{\ell\}, E[\langle \ell, a \rangle] \uplus \{a \mapsto e\})$
Join	$(L, E[\texttt{join}\ T\ T']) \Rightarrow_{alt} (L, E[\mathbf{join}(T, T')])$
Arrive	$(L, E[a] \uplus \{a \mapsto e\}) \Rightarrow_{alt} (L, E[e] \uplus \{a \mapsto e\})$ where $e \in V$
GC	$(L, D \uplus D') \Rightarrow_{alt} (L, D)$ where $\diamond \notin dom(D') \wedge dom(D') \cap free(D) = \emptyset$

Figure 3.19: The alternative semantics for λ_{wit}.

in the size of $\mathcal{C}'$.

Substituting the computed assignments for $\mathcal{B}$ in $\mathcal{C}$ results in linear rational constraints, which can be solved efficiently by a linear programming algorithm.

3.4 Proof of Theorem 3.2.4

Theorem 3.2.4 *If e is witness race free then e is confluent.*

We prove the theorem by showing that the λ_{wit} semantics is equivalent to an alternative semantics provided that e is witness race free. The alternative semantics is confluent for any program, witness-race-free or not. Hence it follows that e is confluent with the λ_{wit} semantics.

Figure 3.19 shows this alternative semantics. Evaluation contexts E are unchanged. L is some set of reference locations. The new expression kind $\langle \ell, a \rangle$ is merely a pair of a reference location and a port, and is also a value (i.e., $\langle \ell, a \rangle \in V$). A *table* T is a function from reference locations to ports. The symbol $\bullet$ is interpreted

as a table such that $\bullet(\ell) = \bot$ for all ℓ. Operations on tables are defined as follows:

$$\begin{array}{lcl}
\mathbf{write}(T, \ell, a) & = & \{\ell' \mapsto a' \mid \ell' \mapsto a' \in T \wedge \ell' \neq \ell\} \cup \{\ell \mapsto a\} \\
\mathbf{read}(T, \ell, a) & = & \begin{cases} a & \text{if } T(\ell) = \bot \\ T(\ell) & \text{if } T(\ell) \neq \bot \end{cases} \\
\mathbf{join}(T, T') & = & \{\ell \mapsto T(\ell) \mid T'(\ell) = T(\ell) \vee T'(\ell) = \bot\} \\
 & & \cup\{\ell \mapsto T'(\ell) \mid T(\ell) = \bot\} \\
 & & \cup\{\ell \mapsto \bot \mid T(\ell) = T'(\ell) = \bot \vee T(\ell) \neq T'(\ell)\}
\end{array}$$

Two states (L, D) and (L', D') are defined to be observationally equivalent if $\mathbf{Erase}(D) \approx \mathbf{Erase}(D')$ where $\mathbf{Erase}(D)$ is D but with each occurrence of a table replaced by $\bullet$ and each occurrence of $\langle \ell, a \rangle$ replaced by ℓ. There are no reference stores in the alternative semantics, i.e., the alternative semantics is trivially side effect free. Therefore, while we will not prove it formally, it is not hard to see that the alternative semantics is always confluent, i.e., for any states (L, D), (L_1, D_1) and (L_2, D_2) such that $(L, D) \Rightarrow^*_{alt} (L_1, D_1)$ and $(L, D) \Rightarrow^*_{alt} (L_2, D_2)$, there exist states (L'_1, D'_1) and (L'_2, D'_2) such that $(L_1, D_1) \Rightarrow^*_{alt} (L'_1, D'_1)$, $(L_2, D_2) \Rightarrow^*_{alt} (L'_2, D'_2)$, and $\mathbf{Erase}(D'_1) \approx \mathbf{Erase}(D'_2)$. Hence, it suffices to prove that if a program e is witness race free then for any evaluation $(\emptyset, D = \{\diamond \mapsto e\}) \Rightarrow^* (S, D_1)$ there is an evaluation $(\emptyset, D) \Rightarrow^*_{alt} (L, D_2)$ such that for any evaluation $(L, D_2) \Rightarrow^*_{alt} (L', D'_2)$ there is an evaluation $(S, D_1) \Rightarrow^* (S', D'_1)$ such that $D'_1 \approx \mathbf{Erase}(D'_2)$.

We augment the graph constructing semantics slightly by adding information about the written port to each *write*(ℓ) node:

Write $(S, E[\texttt{write}\ \ell\ e\ A]) \Rightarrow (S[\ell \leftarrow a], E[B] \uplus \{a \mapsto e\})$
$\mathbb{V}$:=$\mathbb{V} \cup \{B\}$ where B is a new *write*(ℓ, a) node; $\mathbb{E}$:=$\mathbb{E} \cup \{(A, B)\}$

For observational equivalence, we ignore this information, i.e., formally, each node is

replaced by $\bullet$.

Our idea is to show that for a witness-race-free program e, $\Rightarrow$ and $\Rightarrow_{alt}$ can simulate each other while maintaining the following relationship.

Definition 3.4.1. $(S, D_1) \sim_{(\mathbb{V},\mathbb{E})} (L, D_2)$ *if*

- $L = dom(S)$,
- $D_1 = \mathbf{Erase}(D_2)$,
- *if* ℓ *has not been written, i.e., there is no write*(ℓ) *nodes in* $\mathbb{V}$*, then for any occurrence of* $\langle \ell, a \rangle$ *in* D_2*,* $a = S(\ell)$*, and*
- *for any table* T *occurring in* D_2*,* $T(\ell) = a \neq \bot$ *off there exists a node* B : *write*(ℓ, a) *such that* $B \overset{!}{\rightsquigarrow} A$ *where* A *is the node associated with* T*.*

The phrase *node associated with a table* used in the last sentence is defined as follows: the table T at the expression store $D_2 = E[T]$ is associated with the node A at the expression store $D_1 = \mathbf{Erase}(E)[A]$.

We now state the main claim which carries out the aforementioned simulation.

Lemma 3.4.2. *Let a program* e *be witness race free. Then there exists a set* Ω_e *such that* $((\emptyset, \{\diamond \mapsto e\}), (\emptyset, \emptyset), (\emptyset, \{\diamond \mapsto e\})) \in \Omega_e$ *and furthermore, for any triple* $((S, D_1), (\mathbb{V}, \mathbb{E}), (L, D_2)) \in \Omega_e$*,*

- $(S, D_1) \sim_{(\mathbb{V},\mathbb{E})} (L, D_2)$,
- $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S, D_1)$ *with the trace graph* $(\mathbb{V}, \mathbb{E})$*,*
- *if* $(S, D_1) \Rightarrow (S', D_1')$ *with the corresponding trace graph action updating the trace graph from the state* $(\mathbb{V}, \mathbb{E})$ *to the state* $(\mathbb{V}', \mathbb{E}')$*, then there exists a state* (L', D_2') *such that* $(L, D_2) \Rightarrow_{alt} (L', D_2')$ *and* $((S', D_1'), (\mathbb{V}', \mathbb{E}'), (L', D_2')) \in \Omega_e$*, and*

- *if* $(L, D_2) \Rightarrow_{alt} (L', D_2')$*, then there exists a state* (S', D_1') *and a trace graph* $(\mathbb{V}', \mathbb{E}')$ *such that* $(S, D_1) \Rightarrow (S', D_1')$ *with the corresponding trace graph action updating the trace graph from the trace graph* $(\mathbb{V}, \mathbb{E})$ *to the trace graph* $(\mathbb{V}', \mathbb{E}')$ *and* $((S', D_1'), (\mathbb{V}', \mathbb{E}'), (L', D_2')) \in \Omega_e$*.*

The first condition says that the two states in a triple in Ω_e are $\sim$-related with the trace graph in the triple. The second condition says that these states can be reached while generating the trace graph. The third and fourth conditions are the simulation steps, i.e., showing that a step in $\Rightarrow$ can be simulated by a step in $\Rightarrow_{alt}$ and vice versa.

Proof. Our proof constructs Ω_e inductively. For the base case, it is easy to see that $(\emptyset, \{\diamond \mapsto e\}) \sim_{(\emptyset,\emptyset)} (\emptyset, \{\diamond \mapsto e\})$ and $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (\emptyset, \{\diamond \mapsto e\})$ with the trace graph $(\emptyset, \emptyset)$. Therefore, we may set $\Omega_e = \{((\emptyset, \{\diamond \mapsto e\}), (\emptyset, \emptyset), (\emptyset, \{\diamond \mapsto e\}))\}$ initially.

The inductive case is split into case by reduction kinds. Let

$$((S, D_1), (\mathbb{V}, \mathbb{E}), (L, D_2)) \in \Omega_e$$

App, **Let**, **Pair**, **Arrive**, **GC**

Suppose $D_1 = E_1[(\lambda x.e)\ e']$ and we took a **App** step so that

$$(S, E_1[(\lambda x.e_1)\ e_1']) \Rightarrow (S, E_1[e_1[a/x]] \uplus \{a \mapsto e_1'\})$$

Note that the trace graph $(\mathbb{V}, \mathbb{E})$ is not updated by this reduction. Let us take a $\Rightarrow_{alt}$ version of **App** from the state (L, D_2) so that

$$(L, D_2 = E_2[(\lambda x.e_2)\ e_2']) \Rightarrow_{alt} (L, E_2[e_2[a/x]] \uplus \{a \mapsto e_2'\})$$

where $\mathbf{Erase}(E_2) = E_1$. Such E_2 exists since $D_1 = \mathbf{Erase}(D_2)$. Also, it is easy to see

that

$$(S, E_1[e_1[a/x]] \uplus \{a \mapsto e_1'\}) \sim_{(\mathbb{V},\mathbb{E})} (L, E_2[e_2[a/x]] \uplus \{a \mapsto e_2'\})$$

So we add $((S, E_1[e_1[a/x]] \uplus \{a \mapsto e_1'\}), (\mathbb{V}, \mathbb{E}), (L, E_2[e_2[a/x]] \uplus \{a \mapsto e_2'\}))$ to Ω_e. The converse case where we take the **App** step from the state (L, D_2) is analogous. A similar argument works for the case a **Let** step, a **Pair** step, an **Arrive** step, or a **GC** step is taken from the state (S, D_1) or the state (L, D_2).

Write

Suppose $D_1 = E_1[\texttt{write}\ \ell\ e_1\ A]]$ and we took a **Write** step so that

$$(S, E_1[\texttt{write}\ \ell\ e_1\ A]]) \Rightarrow (S[\ell \leftarrow a], D_1' = (E_1[B] \uplus \{a_1 \mapsto e\}))$$

with $\mathbb{V}' = \mathbb{V} \cup \{B\}$ and $\mathbb{E}' = \mathbb{E} \cup \{(A, B)\}$ where $B : \mathit{write}(\ell, a)$. Let us take a $\Rightarrow_{\mathit{alt}}$ version of **Write** from (L, D_2) so that

$$(L, D_2 = E_2[\texttt{write}\ \langle \ell, a' \rangle\ e_2\ T]) \Rightarrow_{\mathit{alt}} (L, D_2' = (E_2[\mathbf{write}(T, \ell, a)] \uplus \{a \mapsto e_2\}))$$

where $\mathbf{Erase}(E_2) = E_1$. Clearly, $D_1' = \mathbf{Erase}(D_2')$. For any table T in the expression store D_2 other than the table $T' = \mathbf{write}(T, \ell, a)$ at the context $E_2[\ ]$, it is easy to see that the fourth condition from the definition of $\sim_{(\mathbb{V}',\mathbb{E}')}$ holds since there is no node reachable from the newly added node B. So consider the port $T'(\ell')$. If $\ell' = \ell$, then $T'(\ell) = a$, which is consistent with the condition since $B \overset{!}{\rightsquigarrow} B$ and $B : \mathit{write}(\ell, a)$. On the other hand, if $\ell' \neq \ell$, then $T'(\ell') = T(\ell)$, and again this is consistent with the condition because the node A is associated with the table T and for any $C : \mathit{write}(\ell', a')$, $C \overset{!}{\rightsquigarrow} B$ if and only if $C \overset{!}{\rightsquigarrow} A$. Therefore, $(S[\ell \leftarrow a], D_1) \sim_{(\mathbb{V}',\mathbb{E}')} (L, D_2)$. So we add the triple $((S[\ell \leftarrow a], D_1), (\mathbb{V}', \mathbb{E}'), (L, D_2))$ to the set Ω_e. The converse

case where we take a **Write** step from the state (L, D_2) is analogous.

Ref

Suppose $D_1 = E_1[\texttt{ref}\ e_1]$ and we took a **Ref** step so that

$$(S, D_1 = E_1[\texttt{ref}\ e_1]) \Rightarrow (S \uplus \{\ell \mapsto a\}, D_1' = (E_1[\ell] \uplus \{a \mapsto e_1\}))$$

Note that the trace graph $(\mathbb{V}, \mathbb{E})$ is not updated by this reduction. Let us take a $\Rightarrow_{alt}$ version of **Ref** from the state (L, D_2) so that

$$(L, D_2 = E_2[\texttt{ref}\ e_2]) \Rightarrow_{alt} (L \uplus \{\ell\}, D_2' = (E_2[\langle \ell, a\rangle] \uplus \{a \mapsto e_2\}))$$

where $\mathbf{Erase}(E_2) = E_1$. Clearly, $D_1' = \mathbf{Erase}(D_2')$. Also, $L \uplus \{\ell\} = \mathit{dom}(S \uplus \{\ell \mapsto a\})$. The reference location ℓ has not been written and $a = (S \uplus \{\ell \mapsto a\})(\ell)$. Therefore $(S \uplus \{\ell \mapsto a\}, D_1) \sim_{(\mathbb{V}', \mathbb{E}')} (L \uplus \{\ell\}, D_2)$. So we add the triple $((S \uplus \{\ell \mapsto a\}, D_1), (\mathbb{V}, \mathbb{E}), (L \uplus \{\ell\}, D_2))$ to Ω_e. The converse case where we take a **Ref** step from the state (L, D_2) is analogous.

Read

Suppose $D_1 = E_1[\texttt{read}\ \ell\ A]$ and we took a **Read** step so that

$$(S, D_1 = E_1[\texttt{read}\ \ell\ A]) \Rightarrow (S, D_1' = E_1[S(\ell) \otimes B])$$

with $\mathbb{V}' = \mathbb{V} \cup \{B\}$ and $\mathbb{E}' = \mathbb{E} \cup \{(A, B)\}$ where $B : \mathit{read}(\ell)$. Let us take a $\Rightarrow_{alt}$ version of **Read** from the state (L, D_2) such that

$$(L, D_2 = E_2[\texttt{read}\ \langle \ell, a\rangle\ T]) \Rightarrow_{alt} (L, D_2' = E_2[\mathbf{read}(T, \ell, a) \otimes T])$$

where $\mathbf{Erase}(E_2) = E_1$. Now consider the table $\mathbf{read}(T, \ell, a)$. If the reference location

ℓ has not been written, i.e., there are no *write*(ℓ) nodes in the vertex set $\mathbb{V}$, then we have $S(\ell) = a$. Also, $T(\ell) = \bot$ since otherwise there is some C : *write*(ℓ) such that $C \overset{!}{\rightsquigarrow} A$, which contradicts the statement we just made. Hence $\mathbf{read}(T, \ell, a) = a$. Otherwise, the reference location ℓ has been written, i.e., there exists C : *write*(ℓ) in the vertex set $\mathbb{V}$. By witness race freedom, it must be the case that either $C \rightsquigarrow B$ or $B \rightsquigarrow C$, but since B is a newly added node, it must be the case that $C \rightsquigarrow B$. Therefore, again by witness race freedom, it must be the case that there exists a node C' : *write*(ℓ, a') for some port a' such that $C' \overset{!}{\rightsquigarrow} B$. Obviously, $C' \overset{!}{\rightsquigarrow} A$. Therefore $\mathbf{read}(T, \ell, a) = a'$. Suppose for contradiction that $S(\ell) = a'' \neq a'$. Then there must be a node C'' : *write*(ℓ, a'') such that this node was added after C' was added. But by witness race freedom, it must be the case that $C'' \rightsquigarrow B$. Hence $C'' \rightsquigarrow C'$ by the definition of $\overset{!}{\rightsquigarrow}$. But this implies that C'' was added before C' was added, a contradiction. Hence $S(\ell) = a$ and $D_1' = \mathbf{Erase}(D_2')$. Lastly, for any C, $C \overset{!}{\rightsquigarrow} A$ if and only if $C \overset{!}{\rightsquigarrow} B$. Therefore $(S, D_1) \sim_{(\mathbb{V}', \mathbb{E}')} (L, D_2)$. So we add the triple $((S, D_1), (\mathbb{V}', \mathbb{E}'), (L, D_2))$ to the set Ω_e. The converse case where we take a **Read** step from the state (L, D_2) is analogous.

Join

Suppose $D_1 = E_1[\texttt{join}\ A\ B]$ and we took a **Join** step so that

$$(S, E_1[\texttt{join}\ A\ B]) \Rightarrow (S, E_1[C])$$

with $\mathbb{V}' = \mathbb{V} \cup \{C\}$ and $\mathbb{E}' = \mathbb{E} \cup \{(A, C), (B, C)\}$ where C : *join*. Let us take a $\Rightarrow_{alt}$ version of **Join** from the state (L, D_2) so that

$$(L, D_2 = E_2[\texttt{join}\ T\ T']) \Rightarrow_{alt} (L, D_2' = E_2[\mathbf{join}(T, T')])$$

where $\mathbf{Erase}(E_2) = E_1$. Clearly, $D_1' = \mathbf{Erase}(D_2')$. Consider the table $\mathbf{join}(T, T')(\ell)$.

Because C is a new node, there exists $C' : \mathit{write}(\ell, a)$ such that $C' \overset{!}{\rightsquigarrow} C$ iff either

1. $C' \overset{!}{\rightsquigarrow} A$ and $C' \overset{!}{\rightsquigarrow} B$,
2. $C' \overset{!}{\rightsquigarrow} A$ and there exists no $C'' : \mathit{write}(\ell, a)$ such that $C'' \neq C'$ and $C'' \overset{!}{\rightsquigarrow} B$, or
3. $C' \overset{!}{\rightsquigarrow} B$ and there exists no $C'' : \mathit{write}(\ell, a)$ such that $C'' \neq C'$ and $C'' \overset{!}{\rightsquigarrow} A$.

For case 1, $T(\ell) = T'(\ell) = a$. For case 2, $T(\ell) = a$ and $T'(\ell) = \bot$. For case 3, $T(\ell) = \bot$ and $T'(\ell) = a$. In all three cases, $\mathbf{join}(T, T')(\ell) = a$. Therefore $(S, D_1) \sim_{(\mathbb{V}', \mathbb{E}')} (L, D_2)$. So we add the triple $((S, D_1), (\mathbb{V}', \mathbb{E}'), (L, D_2))$ to the set Ω_e. The converse case where we take a **Join** step from the state (L, D_2) is analogous. □

The preceding lemma implies that for a witness-race-free e, any evaluation $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S, D)$ has a corresponding simulation $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^*_{alt} (L, D_{alt})$ such that any evaluation $(L, D_{alt}) \Rightarrow^*_{alt} (L', D'_{alt})$ has a corresponding simulation $(S, D) \Rightarrow^* (S', D')$ such that $D' \approx \mathbf{Erase}(D'_{alt})$.[4]

So suppose $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S_1, D_1)$ and $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S_2, D_2)$. Then there are $(L_1, D_{1,alt})$ and $(L_2, D_{2,alt})$ such that

- $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^*_{alt} (L_1, D_{1,alt})$,
- $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^*_{alt} (L_2, D_{2,alt})$,
- for any evaluation $(L_1, D_{1,alt}) \Rightarrow^*_{alt} (L'_1, D'_{1,alt})$, there is (S'_1, D'_1) such that $(S_1, D_1) \Rightarrow^* (S'_1, D'_1)$ and $D'_1 \approx \mathbf{Erase}(D'_{1,alt})$, and
- for any evaluation $(L_2, D_{2,alt}) \Rightarrow^*_{alt} (L'_2, D'_{2,alt})$, there is (S'_2, D'_2) such that $(S_2, D_2) \Rightarrow^* (S'_2, D'_2)$ and $D'_2 \approx \mathbf{Erase}(D'_{2,alt})$.

[4] In fact, Lemma 3.4.2 implies that there is one that maintains $\approx$ relation at every step of the simulation. But $\approx$ at the end of the simulation is sufficient to prove the theorem.

$$
\begin{array}{rcl}
E & := & D \cup \{a \mapsto E\} \mid [\,] \mid E\ e \mid e\ E \mid E \otimes e \mid e \otimes E \mid \pi_i(E) \\
& \mid & \texttt{write}\ E\ e\ e' \mid \texttt{write}\ e\ E\ e' \mid \texttt{write}\ e\ e'\ E \mid \texttt{read}\ e\ E \mid \texttt{read}\ E\ e \\
& \mid & \texttt{ref}\ E\ e \mid \texttt{ref}\ e\ E \mid \texttt{join}\ E\ e \mid \texttt{join}\ e\ E
\end{array}
$$

Figure 3.20: Evaluation contexts for flow annotated λ^{reg}_{wit} semantics.

But since $\Rightarrow_{alt}$ is confluent, there are states $(L'_1, D'_{1,alt})$ and $(L'_2, D'_{2,alt})$ such that $\mathbf{Erase}(D'_{1,alt}) \approx \mathbf{Erase}(D'_{2,alt})$. Hence there are (S'_1, D'_1) and (S'_2, D'_2) such that $(S_1, D_1) \Rightarrow^* (S'_1, D'_1)$, $(S_2, D_2) \Rightarrow^* (S'_2, D'_2)$, and $\mathbf{Erase}(D'_1) \approx \mathbf{Erase}(D'_2)$, i.e., e is confluent even with $\Rightarrow$.

3.5 Proof of Theorem 3.3.7

Theorem 3.3.7 *If a λ^{reg}_{wit} program e is well-typed, then e is witness race free.*

We prove the result by showing that any evaluation of e generates a witness-race-free trace graph. By Theorem 3.3.2, it suffices to show that the evaluation builds a read-write pipeline with bottlenecks for every r. Then, as discussed in Section 3.3.2, it suffices to show that for each r there exists flow assignments to the trace graph with a source node having flow 1 such that every *read*(r) node gets a positive flow (i.e., flow > 0) and every *write*(r) node gets a flow equal to 1.

To this end, we annotate the semantics with flow information such that generated trace graph is constructed together with flow assignments. The resulting semantics $\Rightarrow_{flow}$ shown in Figure 3.20 and Figure 3.21 has the following differences from the unannotated version.

1. R's are now mapping from some set of regions to rational numbers in the range $[0, 1]$. Intuitively, $R(r)$ represents the amount of flow remaining in the source node for r.

App $(R, K, S, E[(\lambda x.e)\ e']) \Rightarrow_{\mathit{flow}} (R, K, S, E[e[a/x]] \uplus \{a \mapsto e'\})$

Let $(R, K, S, E[\texttt{let}\ x = e\ \texttt{in}\ e']) \Rightarrow_{\mathit{flow}} (R, K, S, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\})$

Pair $(R, K, S, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow_{\mathit{flow}} (R, K, S, E[e_i])$

Write $(R, K, S, E[\texttt{write}\ \ell\ e\ \langle A, P\rangle] \Rightarrow_{\mathit{flow}} (R, K, S[\ell \leftarrow a], E[\langle B, P\rangle] \uplus \{a \mapsto e\})$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where B is a new $\mathit{write}(K(\ell))$ node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$
for each r flow $P(r)$ from A to B

Read $(R, K, S, E[\texttt{read}\ \ell\ \langle A, P\rangle]) \Rightarrow_{\mathit{flow}} (R, K, S, E[S(\ell) \otimes \langle B, P\rangle])$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where B is a new $\mathit{read}(K(\ell))$ node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$
for each r flow $P(r)$ from A to B

Ref $(R, K, S, E[\texttt{ref}\ e\ r]) \Rightarrow_{\mathit{flow}} (R, K \uplus \{\ell \mapsto r\}, S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\})$

Join $(R, K, S, E[\texttt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]) \Rightarrow_{\mathit{flow}} (R, K, S, E[\langle C, P + P'\rangle])$
$\mathbb{V} := \mathbb{V} \cup \{C\}$ where C is a new *join* node
$\mathbb{E} := \mathbb{E} \cup \{(A, C), (B, C)\}$
for each r flow $P(r)$ from A to C
for each r flow $P'(r)$ from B to C

LetReg $(R, K, S, E[\texttt{letreg}\ x\ e]) \Rightarrow_{\mathit{flow}} (R \uplus \{r \mapsto 1\}, K, S, E[e[a/x]] \uplus \{a \mapsto r\})$

Arrive $(R, K, S, E[a] \uplus \{a \mapsto e\}) \Rightarrow_{\mathit{flow}} (R, K, S, E[e_1] \uplus \{a \mapsto e_2\})$
where $e \in V \wedge e = e_1 + e_2$

GC $(R, K, S, D \uplus D') \Rightarrow_{\mathit{flow}} (R, K, S, D)$
where $\diamond \notin \mathit{dom}(D') \wedge \mathit{dom}(D') \cap \mathit{free}(D) = \emptyset$

Source $(R + P', K, S, E[\langle A, P\rangle]) \Rightarrow_{\mathit{flow}} (R, K, S, E[\langle A, P + P'\rangle])$
for each r flow $P(r)$ from the source node to A

Figure 3.21: Flow annotated λ^{reg}_{wit} semantics.

2. Each witness (i.e., a graph node) is paired with a *packet* P. A packet P is a function from regions to rational numbers in the range $[0, 1]$. Intuitively, the pair $\langle A, P\rangle$ implies that $P(r)$ amount of flow for region r is flowing from the node A.

3. **Arrive** "splits" the expression e into expressions e_1 and e_2 instead of duplicating e. Splitting is defined using the additive arithmetic $e_1 + e_2 = e$ given in

$$
\begin{array}{l}
e + e = e \text{ where } e = x, i, a, r, \ell, \text{ or } \lambda x.e \\
e_1 \otimes e_2 + e_1' \otimes e_2' = (e_1 + e_2) \otimes (e_1' + e_2') \\
\langle A, P_1 \rangle + \langle A, P_2 \rangle = \langle A, P_1 + P_2 \rangle \\
P_1 + P_2 = \{r \mapsto P_1(r) + P_2(r) \mid r \in \textbf{Regions}\}
\end{array}
$$

Figure 3.22: Additive arithmetic of $e \in V$.

Figure 3.22.[5] Note that the addition $P_1 + P_2$ is also used at the **Join** rule to combine two packets.

4. Flow assignments are made at **Read**, **Write**, and **Join**.

5. The new reduction rule **Source** is introduced to account for flow from the source node to an arbitrary node.

Any $\Rightarrow_{\mathit{flow}}$ evaluation sequence has a corresponding unannotated evaluation sequence, i.e., replace packets by $\bullet$ and bypass **Source** steps. Conversely, any $\Rightarrow$ evaluation sequence has at least one corresponding flow-annotated evaluation sequence. However, this correspondence is not a bijection since different $\Rightarrow_{\mathit{flow}}$ evaluation sequences may correspond to the same $\Rightarrow$ evaluation sequence. In particular, even for a well-typed-program, there may be a $\Rightarrow_{\mathit{flow}}$ evaluation sequence which does not have the proper flow requirement, i.e., some $\mathit{read}(r)$ node gets flow ≤ 0 or some $\mathit{write}(r)$ node gets flow < 1.

Our goal is to show that for any $\Rightarrow$ evaluation sequence of a well-typed program e, there is a corresponding $\Rightarrow_{\mathit{flow}}$ evaluation sequence with proper flow assignments. The proof is by subject reduction, i.e., a $\Rightarrow$ step from a *well-typed state* X always has (a) corresponding $\Rightarrow_{\mathit{flow}}$ step(s) taking X to another well-typed state Y. As we shall see, we can always assign proper flow as long as we are in well-typed states. But before we define well-typed states, we need to extend the type system to type

[5] We do not need to split below the body of $\lambda x.e$ since there cannot be any packets captured in a λ abstraction.

non-source expression kinds:

$$\frac{\text{for each } r \in \mathit{dom}(\Gamma), P(r) = W'(\rho) \text{ where } \Gamma(r) = \mathit{reg}(\rho)}{\Gamma; W \vdash \langle A, P\rangle : W'} \textbf{ Packet}$$

$$\frac{\Gamma(r) = \mathit{reg}(\rho)}{\Gamma; W \vdash r : \mathit{reg}(\rho)} \textbf{ Region} \qquad \frac{\Gamma(\ell) = \mathit{ref}\,(\tau, \tau', \rho)}{\Gamma; W \vdash \ell : \mathit{ref}\,(\tau, \tau', \rho')} \textbf{ Loc}$$

$$\frac{\Gamma(a) = \tau}{\Gamma; W \vdash a : \tau} \textbf{ Port}$$

Also, we need to extend arithmetic over type environments such that

$$(\Gamma, e\!:\!\tau) + (\Gamma', e\!:\!\tau') = (\Gamma + \Gamma'), e\!:\!(\tau + \tau')$$

and

$$(\Gamma, e\!:\!\tau) \times q = (\Gamma \times q), e\!:\!(\tau \times q)$$

for any expression e that is either a port, a variable, a reference location, or a region.

We are now ready to define well-typed states.

Definition 3.5.1. *The state (R, K, S, D) is well-typed under the environment $\Gamma; W$ (written $\Gamma; W \vdash (R, S, K, D)$) if*

- $\mathit{dom}(R) = \mathit{dom}(\Gamma) \cap \textbf{Regions}$,
- $\mathit{dom}(S) = \mathit{dom}(K) = \mathit{dom}(\Gamma) \cap \textbf{Locations}$,
- $\mathit{dom}(D) = \mathit{dom}(\Gamma) \cap \textbf{Ports}$,
- *For any region $r \in \mathit{dom}(R)$, $R(r) \geq W(\rho)$ where $\Gamma(r) = \mathit{reg}(\rho)$,*
- *For any reference location $\ell \in \mathit{dom}(K)$, $\Gamma(K(\ell)) = \mathit{reg}(\rho)$ where $\Gamma(\ell) = \mathit{ref}\,(\tau, \tau', \rho)$ for some τ and τ', and*

- *Suppose* $D = \{a_1 \mapsto e_1, \ldots, a_n \mapsto e_n\}$ *and* $S = \{\ell_1 \mapsto a'_1, \ldots, \ell_m \mapsto a'_m\}$*, then there exists environments* $\Gamma_1; W_1, \ldots, \Gamma_n; W_n, \Gamma'_1; W'_1, \ldots, \Gamma'_m; W'_m$ *such that*
  - $\Gamma = \sum_{i=1}^{n} \Gamma_i + \sum_{i=1}^{m} \Gamma'_i$,
  - $W = \sum_{i=1}^{n} W_i + \sum_{i=1}^{m} W'_i$,
  - *for each port* a_i, $\Gamma_i; W_i \vdash D(a_i) : \Gamma(a_i)$*, and*
  - *for each reference location* ℓ_i, $\Gamma'_i; W'_i \vdash S(\ell_i) : \tau$ *where* $\Gamma(\ell_i) = ref\,(\tau, \tau', \rho)$ *for some types* τ *and* τ' *such that* $\tau \leq \tau'$ *and a static region identifier* ρ *such that* $\Gamma(K(\ell)) = reg(\rho)$.

It becomes cumbersome later when we repeatedly pick just one $\Gamma_i; W_i$ or $\Gamma'_i; W'_i$, and so for convenience, given $\Gamma; W \vdash (R, S, K, D \uplus \{a \mapsto e\})$, we often say "$\Gamma'; W'$ is the environment for $\{a \mapsto e\}$" to mean that the environment $\Gamma'; W'$ is one of the environments $\Gamma_i; W_i$ from the definition above such that $\Gamma_i; W_i \vdash e : \Gamma(a)$. Similarly, given $\Gamma; W \vdash (R, S \uplus \{\ell \mapsto a\}, K, D)$, we often say "$\Gamma'; W'$ is the environment for $\{\ell \mapsto a\}$" to mean that the environment $\Gamma'; W'$ is one of the environments $\Gamma'_i; W'_i$ from the definition above such that $\Gamma'_i; W'_i \vdash a : \Gamma(\ell)$.

Before we prove the main subject reduction lemma (Lemma 3.5.6), we need a few side lemmas. The following lemma is needed to "convert" an application of a **Source** type rule to a step of a **Source** $\Rightarrow_{flow}$ rule.

Lemma 3.5.2. *Suppose* $\Gamma; W + W' \vdash (R, K, S, D \uplus \{a \mapsto E[\langle A, P\rangle]\})$ *such that one application of the type rule* **Source** *with* W' *is applied at the context* $E[\ ]$ *following the type rule* **Packet** *in the type judgment for the expression* $E[\langle A, P\rangle]$*, i.e.,*

$$\dfrac{\dfrac{\cdots}{\Gamma'; W'' \vdash \langle A, P\rangle : W_P}\ \textbf{Packet}}{\Gamma'; W'' + W' \vdash \langle A, P\rangle : W_P + W'}$$

appears in the judgment for the expression $E[\langle A, P\rangle]$ *in* $\Gamma; W + W' \vdash (R, K, S, D \uplus$

$\{a \mapsto E[\langle A, P\rangle]\})$. *Then there exists* P' *such that* $\Gamma; W \vdash (R', S, K, D \uplus \{a \mapsto E[\langle A, P + P'\rangle]\})$ *where* $R = R' + P'$ *without using the type rule* **Source** *at the context* $E[\,]$.

Proof. Choose a packet P' such that for each region r, $P'(r) = W'(\rho)$ where $\Gamma(r) = \mathit{reg}(\rho)$ and $P'(r) = 0$ if $r \notin \mathit{dom}(\Gamma)$. Suppose the environment $\Gamma_i; W_i$ is the environment for $\{a \mapsto E[\langle A, P\rangle]\}$ in the environment $\Gamma; W + W'$, i.e., $\Gamma_i; W_i \vdash E[\langle A, P\rangle] : \tau$ for some type τ. Then it is easy to see that $\Gamma_i; W_i - W' \vdash E[\langle A, P + P'\rangle] : \tau$ by not using the type rule **Source** at the context $E[\,]$. Hence $\Gamma; W \vdash (R', S, K, D \uplus \{a \mapsto E[\langle A, P + P'\rangle]\})$. □

The following side lemma shows that any expression $e \in V$ can be "split" according to a splitting of its types. This lemma is used only to prove Lemma 3.5.4.

Lemma 3.5.3. *Suppose* $e \in V$, $\Gamma; W \vdash e : \tau$, *and* $\tau = \tau_1 + \tau_2$. *Then there exists expressions* e_1 *and* e_2, *and environments* $\Gamma_1; W_1$ *and* $\Gamma_2; W_2$ *such that* $e = e_1 + e_2$, $\Gamma = \Gamma_1 + \Gamma_2$, $W = W_1 + W_2$, $\Gamma_1; W_1 \vdash e_1 : \tau_1$, *and* $\Gamma_2; W_2 \vdash e_2 : \tau_2$.

Proof. By structural induction on the expression e. The case $e = i$ is trivial. For the case the expression e is x, a, ℓ, or r, we let $e_1 = e_2 = e$ and split the type $\Gamma(e)$ accordingly (the rest of the environment Γ may be split in any way), and if $\Gamma(e) = W'$ for some witness type W', then we also split the witness W in case the type rule **Source** was used in the judgment $\Gamma; W \vdash e : \tau$.

For the case $e = \langle A, P\rangle$, τ_1 and τ_2 are witness types. Therefore there exist packets P_1 and P_2 such that $P = P_1 + P_2$, $\Gamma_1; W_1 \vdash \langle A, P_1\rangle : \tau_1$, and $\Gamma_2; W_2 \vdash \langle A, P_2\rangle : \tau_2$ where $W_1 + W_2 = W$ and $\Gamma_1 + \Gamma_2 = \Gamma$. Here, W_1 and W_2 are split according to the uses of **Source** in typing τ_1 and τ_2. Γ_1 and Γ_2 can be split in any way.

The case $e = v \otimes v'$ follows from the induction and the type rule **Pair**. I.e., since $\tau = \tau' \otimes \tau''$ for some τ' and τ'', we split v and v' according to how τ' and τ'' are

split, and add the environments of each half to obtain $\Gamma_1; W_1$ and $\Gamma_2; W_2$. Details are omitted but are straightforward.

For the case $e = \lambda x.e'$, as discussed before, we only consider the case when e' contains no packets because no evaluation produces a λ abstraction containing a packet. So we may let $e_1 = e_2 = \lambda x.e$. Since $\tau_1 = \tau' \xrightarrow{q_1} \tau''$ and $\tau_2 = \tau' \xrightarrow{q_2} \tau''$ for some τ', τ'', q_1, and q_2, we may split Γ into $\Gamma_1 = \Gamma \times q_1$ and $\Gamma_2 = \Gamma \times q_2$, and split W into $W_1 = W \times q_1$ and $W_2 = W \times q_2$. Checking the arithmetic is straightforward. □

The following side lemma is required when we take **Arrive** steps in the subject reduction argument.

Lemma 3.5.4. *Suppose* $\Gamma; W \vdash (R, S, K, E[a] \uplus \{a \mapsto e\})$ *and* $e \in V$*, then there exist expressions* e_1*,* e_2 *and a type environment* Γ' *such that* $e = e_1 + e_2$ *and* $\Gamma'; W \vdash (R, S, K, E[e_1] \uplus \{a \mapsto e_2\})$.

Proof. Let $D = E[a] \uplus \{a \mapsto e\}$. Let $n + m$ environments $\Gamma_1; W_1, \ldots, \Gamma_n; W_n$, $\Gamma'_1; W'_1, \ldots, \Gamma'_m; W'_m$ be such that $\Gamma = \sum_{i=1}^{n} \Gamma_i + \sum_{i=1}^{m} \Gamma'_i$, $W = \sum_{i=1}^{n} W_i + \sum_{i=1}^{m} W'_i$, for each a_i, $\Gamma_i; W_i \vdash D(a_i) : \Gamma(a_i)$, and for each ℓ_i, $\Gamma'_i; W'_i \vdash S(\ell_i) : \Gamma(\ell_i)$. Suppose $a = a_i$ and $E[\,] = \{a_j \mapsto E'[\,]\} \cup D'$. Note that $i \neq j$.

Then by Lemma 3.5.3 and by induction on the structure of E and the type judgment rules, it follows that there exist expressions e_1 and e_2 such that $D(a_i) = e_1 + e_2$ where there exist $\Gamma_{i_1}, \Gamma_{i_2}, \tau_1, \tau_2, W_{i_1}$, and W_{i_2} such that

- $\Gamma_i = \Gamma_{i_1} + \Gamma_{i_2}$,
- $W_i = W_{i_1} + W_{i_2}$,
- $\Gamma_{i_1}; W_{i_1} \vdash e_1 : \tau - \tau_2$, and
- $\Gamma_j - \{a : \tau_2\} + \Gamma_{i_2}; W_j + W_{i_2} \vdash E'[e_2] : \Gamma(a_j)$

Let $\Gamma' = \Gamma - \Gamma_i - \Gamma_j + \Gamma_{i_1} + (\Gamma_j - \{a{:}\tau_2\} + \Gamma_{i_2}) = \Gamma - \{a{:}\tau_2\}$. Then it follows that

$$\Gamma'; W \vdash (R, S, K, E[e_1] \uplus \{a \mapsto e_2\})$$

□

The following lemma allows us to elide the type checking rule **LetA** when proving the main lemma.

Lemma 3.5.5. *Let e and e' be programs such that e' is identical to e except that an occurrence of the expression* `let` x `=` e_1 `in` e_2 *is replaced by the expression $e_2[e_1/x]$ where $e_1 \in V$. Then, there is an evaluation of e' that generates a trace graph G iff there is an evaluation of e that generates G.*

Proof. By inspection of the reduction rules. □

Based on Lemma 3.5.5, we assume in the next lemma that no type judgment uses **LetA**. Let **Erase** be an operation that replaces each occurrence of $\langle A, P\rangle$ by A. We are now ready to state and prove the main lemma.

Lemma 3.5.6. *Suppose*

$$\Gamma; W \vdash (R, S, K, D)$$

and

$$(dom(R), S, \mathbf{Erase}(D)) \Rightarrow (R_o, S_o, D_o)$$

Then there exist R_f, K_f, S_f, D_f, Γ', and W' such that

- $(R, S, K, D) \Rightarrow_{\mathit{flow}} (R_f, S_f, K_f, D_f)$,
- $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$,
- $dom(R_f) = R_o$,

- $S_o = S_f$, *and*
- $\mathbf{Erase}(D_o) = D_f$.

Furthermore, if the above $\Rightarrow_{\mathit{flow}}$ *step is a* **Write** *step writing to location* ℓ *then* $P(K_f(\ell)) \geq 1$ *where* P *is the packet at the write, and if it is a* **Read** *step reading from* ℓ *then* $P(K_f(\ell)) > 0$ *where* P *is the packet at the read.*

This lemma implies that there is a proper flow assignment for a trace graph constructed from reducing a well-typed state.

Proof. We prove the lemma by case analysis on $\Rightarrow$.

App

We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(\{a' \mapsto E[(\lambda x.e)\ e']\} \uplus D')) \Rightarrow (R_o, S_o, D_o)$$

where $\{a' \mapsto E[(\lambda x.e)\ e']\} \uplus D') = D$, $R_o = \mathit{dom}(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto e'\})$. Pick a **App** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = \{a' \mapsto E[(\lambda x.e)\ e'] \uplus D'\}) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto e'\})$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[(\lambda x.e)\ e']\}$ in $\Gamma; W \vdash (R, S, K, D)$ such that

$$\frac{\Gamma_2; W_2 \vdash \lambda x.e : \tau \xrightarrow{q} \tau' \qquad \Gamma_3; W_3 \vdash e' : \tau \qquad q \geq 1}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash (\lambda x.e)\ e' : \tau'}$$

appears in the subderivation at the context $E[\,]$. Then because $q \geq 1$, by inspection of the type checking rules, it follows that $\Gamma_2, a{:}\tau; W_2 \vdash e[a/x] : \tau'$.

We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2, a:\tau; W_1 + W_2$ for $\{a' \mapsto E[e[a/x]]\}$ and by using the environment $\Gamma_3; W_3$ for $\{a \mapsto e'\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

Let

We have

$$(dom(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[\texttt{let } x \texttt{ = } e \texttt{ in } e']\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e'[a/x]]\} \uplus D' \uplus \{a \mapsto e[a/x]\})$. The case $x \notin free(e)$ identical to the **App** case. So suppose $x \in free(e)$. Pick a **Let** step for $\Rightarrow_{flow}$ such that

$$(R, K, S, D = (\{a' \mapsto E[\texttt{let } x \texttt{ = } e \texttt{ in } e']\} \uplus D')) \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e'[a/x]]\} \uplus D' \uplus \{a \mapsto e[a/x]\})$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Since $x \in free(e)$, Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[\texttt{let } x \texttt{ = } e \texttt{ in } e']\}$ in $\Gamma; W \vdash (R, S, K, D)$ such that

$$\frac{\Gamma_2, x:\tau; W_2 \vdash e : \tau \qquad \Gamma_3, x:\tau; W_3 \vdash e' : \tau' \qquad \tau \geq \tau \times \infty}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \texttt{let } x \texttt{ = } e \texttt{ in } e' : \tau'}$$

appears in the subderivation at the context $E[\ ]$. Then by inspection of the type checking rules, it follows that $\Gamma_2, a:\tau; W_2 \vdash e[a/x] : \tau$ and $\Gamma_3, a:\tau; W_3 \vdash e'[a/x] : \tau'$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_2, a:\tau; W_2$ for $\{a \mapsto e[a/x]\}$ and using the environment $\Gamma_1 + \Gamma_3, a:$

$\tau; W_1 + W_3$ for $\{a' \mapsto E[e'[a/x]]\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$ since $\tau \geq \tau \times \infty$ implies that $\tau + \tau = \tau$.

Pair

We have

$$(dom(R), S, \mathbf{Erase}(D = (\{a \mapsto E[\pi_i(e_1 \otimes e_2)]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a \mapsto E[e_i]\} \uplus D')$. Pick a **Pair** step for $\Rightarrow_{flow}$ such that

$$(R, K, S, D = \{a \mapsto E[\pi_i(e_1 \otimes e_2)] \uplus D'\}) \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a \mapsto E[e_i]\} \uplus D')$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2; W_1 + W_2$ be the environment for $\{a \mapsto E[\pi_i(e_1 \otimes e_2)]\}$ in $\Gamma; W \vdash (R, S, K, D)$ such that

$$\frac{\Gamma_2, W_2 \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2}{\Gamma_2; W_2 \vdash \pi_i(e_1 \otimes e_2) : \tau_i}$$

appears in the subderivation at the context $E[\ ]$. Then by inspection of the type checking rules, it follows that $\Gamma_2; W_2 \vdash e_i : \tau_i$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2; W_1 + W_2$ for $\{a \mapsto E[e_i]\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

Write

We have

$$(dom(R), S = (S' \uplus \{\ell \mapsto a''\}), \mathbf{Erase}(D = (\{a' \mapsto E[\texttt{write}\ \ell\ e\ \langle A, P\rangle]\} \uplus D'))) \\ \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = (S' \uplus \{\ell \mapsto a\})$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[\langle B, P\rangle]\} \uplus D' \uplus \{a \mapsto e\})$.

Suppose in $\Gamma; W \vdash (R, S, K, D = (\{a' \mapsto E[\texttt{write}\ \ell\ e\ \langle A, P\rangle]\} \uplus D'))$ the type rule **Source** is applied at $E[\texttt{write}\ell e[\ ]]$ with some witness type W_6. Then by Lemma 3.5.2, there exists P' such that $\Gamma; W - W_6 \vdash (R - P', S, K, \{a' \mapsto E[\texttt{write}\,\ell\, e\, \langle A, P + P'\rangle]\} \uplus D')$. Pick a **Source** step followed by a **Write** step for $\Rightarrow_{flow}$ such that

$$(R, K, S = (S' \uplus \{\ell \mapsto a''\}), D = (\{a' \mapsto E[\texttt{write}\ \ell\ e\ \langle A, P\rangle]\} \uplus D')) \\ \Rightarrow_{flow} (R - P', S' \uplus \{\ell \mapsto a''\}, K, \{a' \mapsto E[\texttt{write}\ \ell\ e\ \langle A, P + P'\rangle]\} \uplus D') \\ \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R - P'$, $K_f = K$, $S_f = (S' \uplus \{\ell \mapsto a\})$, $D_f = (\{a' \mapsto E[\langle B, P + P'\rangle]\} \uplus D' \uplus \{a \mapsto e\})$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3 + \Gamma_4; W_1 + W_2 + W_3 + W_4$ be the environment for $\{a' \mapsto E[\texttt{write}\ell e\langle A, P + P'\rangle]\}$ in $\Gamma; W - W_6 \vdash (R - P', S, K, \{a' \mapsto E[\texttt{write}\ell e\langle A, P + P'\rangle]\} \uplus D')$ such that

$$\frac{\begin{array}{cc} \Gamma_2; W_2 \vdash \ell : \mathit{ref}\,(\tau, \tau', \rho) & \Gamma_3; W_3 \vdash e : \tau' \\ \Gamma_4; W_4 \vdash \langle A, P + P'\rangle : W_5 & W_5(\rho) \geq 1 \end{array}}{\Gamma_2 + \Gamma_3 + \Gamma_4; W_2 + W_3 + W_4 \vdash \texttt{write}\ \ell\ e\ \langle A, P + P'\rangle : W_5}$$

appears in the subderivation at the context $E[\ ]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S' the same as the environment $\Gamma; W - W_6$ and by using the environment $\Gamma_3; W_3$ for $\{a \mapsto e\}$, the environment $\Gamma_1 + \Gamma_4; W_1 + W_4$ for $\{a' \mapsto E[\langle B, P + P'\rangle]\}$, and

the environment $a : \tau; \emptyset$ for $\{\ell \mapsto a\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$ since $\tau' \geq \tau$. Also, since we eliminated the type rule **Source** for the expression $\langle A, P + P' \rangle$ at the context $E[\texttt{write}\, \ell\, e\, [\,]]$, it must be the case that $(P + P')(r) \geq 1$ where $\Gamma(r) = \mathit{reg}(\rho)$. Since $\Gamma_2; W_2 \vdash \ell : \mathit{ref}\,(\tau, \tau', \rho)$ implies that $K_f(\ell) = r$, we have $(P + P')(K_f(\ell)) \geq 1$ as required.

Read

We have

$$\begin{aligned}&(\mathit{dom}(R), S = (S' \uplus \{\ell \mapsto a\}), \mathbf{Erase}(D = (\{a' \mapsto E[\texttt{read}\, \ell\, \langle A, P \rangle]\} \uplus D'))) \\ &\quad \Rightarrow (R_o, S_o, D_o)\end{aligned}$$

where $R_o = \mathit{dom}(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[a \otimes \langle B, P \rangle]\} \uplus D')$.

Suppose in $\Gamma; W \vdash (R, S, K, D = (\{a' \mapsto E[\texttt{read}\, \ell\, \langle A, P \rangle]\} \uplus D'))$, the type rule **Source** is applied at the context $E[\texttt{read}\, \ell\, [\,]]$ with some witness type W_5. Then by Lemma 3.5.2, there exists a packet P' such that $\Gamma; W - W_5 \vdash (R - P', S, K, \{a' \mapsto E[\texttt{read}\, \ell\, \langle A, P + P' \rangle]\} \uplus D')$. Pick a **Source** step followed by a **Read** step for $\Rightarrow_{\mathit{flow}}$ such that

$$\begin{aligned}&(R, K, S = (S' \uplus \{\ell \mapsto a\}), D = (\{a' \mapsto E[\texttt{read}\, \ell\, \langle A, P \rangle]\} \uplus D')) \\ &\quad \Rightarrow_{\mathit{flow}} (R - P', S, K, \{a' \mapsto E[\texttt{read}\, \ell\, \langle A, P + P' \rangle]\} \uplus D') \\ &\quad \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)\end{aligned}$$

where $R_f = R - P'$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[a \otimes \langle B, P + P' \rangle]\} \uplus D')$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[\texttt{read}\ell\langle A, P + P' \rangle]\}$ in $\Gamma; W - W_5 \vdash (R - P', S, K, \{a' \mapsto E[\texttt{read}\, \ell\, \langle A, P + P' \rangle]\} \uplus D')$ such that

$$\frac{\Gamma_2; W_2 \vdash \ell : \mathit{ref}\,(\tau, \tau', \rho) \qquad \Gamma_3; W_3 \vdash \langle A, P + P' \rangle : W_4 \qquad W_4(\rho) > 0}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \texttt{read}\, \ell\, \langle A, P + P' \rangle : W_4 \otimes \tau}\ \textbf{Read}$$

appears in the subderivation at the context $E[\,]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S' the same as the environment $\Gamma; W - W_5$. Let $\Gamma_6; W_6$ be the environment for $\{\ell \mapsto a\}$ in $\Gamma; W - W_5 \vdash (R - P', S, K, \{a' \mapsto E[\texttt{read}\ \ell\ \langle A, P + P'\rangle]\} \uplus D')$. Then in the environment $\Gamma'; W'$, we use the environment $\Gamma_6 - \{a : \tau\}; W_6$ for $\{\ell \mapsto a\}$ and the environment $\Gamma_1 + \Gamma_3 + \{a : \tau\}; W_1 + W_3$ for $\{a' \mapsto E[a \otimes \langle B, P + P'\rangle]\}$. This implies that $\Gamma'(\ell) = (\Gamma - \Gamma_2)(\ell) \geq \mathit{ref}\,(\tau_o - \tau, \tau'_o, \rho)$ where $\Gamma(\ell) = \mathit{ref}\,(\tau_o, \tau'_o, \rho)$. Therefore,$\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$. Also, since we eliminated the type rule **Source** at the expression $\langle A, P + P'\rangle$ at the context $E[\texttt{read}\ \ell\ [\,]]$, it must be the case that $(P + P')(r) > 0$ where $\Gamma(r) = \mathit{reg}(\rho)$. Since $\Gamma_2; W_2 \vdash \ell : \mathit{ref}\,(\tau, \tau', \rho)$ implies that $K_f(\ell) = r$, we have $(P + P')(K_f(\ell)) \geq 1$ as required.

Ref

We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[\texttt{ref}\ e\ r]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = \mathit{dom}(R)$, $S_o = (S \uplus \{\ell \mapsto a\})$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[\ell]\} \uplus D' \uplus \{a \mapsto e\})$. Pick a **Ref** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = (\{a' \mapsto E[\texttt{ref}\ e\ r]\} \uplus D')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = (K \uplus \{\ell \mapsto r\})$, $S_f = (S \uplus \{\ell \mapsto a\})$, $D_f = (\{a' \mapsto E[\ell]\} \uplus D' \uplus \{a \mapsto e\})$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[\texttt{ref}\ e\ r]\}$ in $\Gamma; W \vdash (R, S, K, \{a' \mapsto E[\texttt{ref}\ e\ r]\} \uplus D')$ such that

$$\frac{\Gamma_2; W_2 \vdash e : \tau \qquad \Gamma_3; W_3 \vdash r : \mathit{reg}(\rho)}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \texttt{ref}\ e\ r : \mathit{ref}\,(\tau, \tau, \rho)}$$

appears in the subderivation at the context $E[\,]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_2; W_2$ for $\{a \mapsto e\}$, the environment $\Gamma_1, \ell : \mathit{ref}(\tau, \tau, \rho); W_1$ for $\{a' \mapsto E[\ell]\}$, and the environment $a : \tau; \emptyset$ for $\{\ell \mapsto a\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

Join

We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(D = (\{a \mapsto E[\mathtt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = \mathit{dom}(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a \mapsto E[\langle C, P + P'\rangle]\} \uplus D')$. Pick a **Join** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = (\{a \mapsto E[\mathtt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]\} \uplus D')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a \mapsto E[\langle C, P + P'\rangle]\} \uplus D')$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a \mapsto E[\mathtt{join}\langle A, P\rangle\langle B, P'\rangle]\}$ in $\Gamma; W \vdash (R, S, K, \{a \mapsto E[\mathtt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]\} \uplus D')$ such that

$$\frac{\Gamma_2; W_2 \vdash \langle A, P\rangle : W_4 \qquad \Gamma_2; W_3 \vdash \langle B, P'\rangle : W_5}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \mathtt{join}\ \langle A, P\rangle\ \langle B, P'\rangle : W_4 + W_5}$$

appears in the subderivation at the context $E[\,]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ for $\{a \mapsto E[\langle C, P + P'\rangle]\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

LetReg

We have

$$(dom(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[\texttt{letreg}\ x\ e]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R \uplus \{r \mapsto 1\})$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto r\})$. Pick a **LetReg** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = (\{a' \mapsto E[\texttt{letreg}\ x\ e]\} \uplus D')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R \uplus \{r \mapsto 1\}$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto r\})$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2; W_1 + W_2$ be the environment for $\{a \mapsto E[\texttt{letreg}\ x\ e]\}$ in $\Gamma; W \vdash (R, S, K, \{a \mapsto E[\texttt{letreg}\ x\ e]\} \uplus D')$ such that

$$\frac{\Gamma_2, x : \mathit{reg}(\rho); W_2 + \{\rho \mapsto q\} \vdash e : \tau \qquad q \leq 1 \qquad \rho \notin \mathit{free}(\Gamma_2, W_2, \tau)}{\Gamma_2; W_2 \vdash \texttt{letreg}\ x\ e : \tau}$$

appears in the subderivation at the context $E[\,]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2 + \{a : \mathit{reg}(\rho)\}; W_1 + W_2 + \{\rho \mapsto 1\}$ for $\{a' \mapsto E[e[a/x]]\}$ and the environment $r : \mathit{reg}(\rho); \emptyset$ for $\{a \mapsto r\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$ since $q \leq 1$.

Arrive

We have

$$(dom(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[a]\} \uplus \{a \mapsto e\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $e \in V$, $R_o = dom(R \uplus \{r \mapsto 1\})$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e]\} \uplus \{a \mapsto$

$e\} \uplus D')$.

By Lemma 3.5.4, there exist expressions e_1, e_2, and a type environment Γ' such that $\Gamma'; W \vdash (R, S, K, \{a' \mapsto E[e_1]\} \uplus \{a \mapsto e_2\} \uplus D')$. Pick an **Arrive** step for $\Rightarrow_{\mathit{flow}}$ with the above e_1 and e_2, i.e.,

$$(R, K, S, D = (\{a' \mapsto E[a]\} \uplus \{a \mapsto e\} \uplus D')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e_1]\} \uplus \{a \mapsto e_2\} \uplus D')$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$. Let $W' = W$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

GC

We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(D = (D' \uplus D''))) \Rightarrow (R_o, S_o, D_o)$$

where $\diamond \notin \mathit{dom}(D'')$, $\mathit{dom}(D'') \cap \mathit{free}(D') = \emptyset$, $R_o = \mathit{dom}(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(D')$. Pick a **GC** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = (D' \uplus D'')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = D'$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

We construct an environment $\Gamma'; W'$ by subtracting the portions for the expression store D'' from the environment $\Gamma; W$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

□

Finally, to start off subject reduction, we need the initial state to be well-typed.

Lemma 3.5.7. *If e is a well-typed program, then $\emptyset; \emptyset \vdash (\emptyset, \emptyset, \emptyset, \{\diamond \mapsto e\})$. That is,*

the initial state is well-typed.

Combining Lemma 3.5.6 and Lemma 3.5.7, it follows that any trace graph of a well-typed program has a proper flow assignment.

Chapter 4

Related Work

In both Chapter 2 and Chapter 3, the key step was to derive a sufficient condition for ensuring determinism in terms of dependencies among accesses on shared resources. Because an access A dependent on an access B is guaranteed to happen after B, we are able to prove, for example, a program is deterministic if the assignment $x := 1$ is dependent on the assignment $x := 2$ and there are no other resource accesses in the program.

Hence, the task of the static analysis was reduced to proving that necessary dependences exist. In this sense, our system is different in spirit from approaches that obtain determinism as a corollary of well-studied concepts such as linear types and monads. In the rest of the chapter, we show that our system is actually able to prove more programs to be deterministic by showing that these other approaches can be expressed as the restricted cases of our system (as already seen in Section 2.1.1 for linearly typed process algebra).

4.1 Communicating Processes

We compare the system presented in Chapter 2 with related work on communicating processes.

Kahn process networks [Kah74] restrict communication to input buffered channels with a unique sender process to guarantee determinism. Edwards et al. [ET05] 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 [NS97; KPT99]. Section 2.3 discusses how to express linearly typed channels in our system. König presents a type system that can be parameterized to check partial confluence in the π-calculus [Kön00]. Her system corresponds to the restricted case of our system where each (rendezvous) channel is given a type of the form $ch(\rho, \tau, \mathit{0}[w(\rho) \mapsto 1], \mathit{0}[r(\rho) \mapsto 1])$, i.e., each channel sends its own write capability at writes and sends its own read capability at reads. Therefore, for example, while her system can check the confluence of $!(c,1); ?(c,x)||?(c,x); !(c,2)$, it cannot check the confluence of $!(c,1); !(c,2)||?(c,x); ?(c,x)$.

Boyland has proposed a system similar to ours, but for a language without channels (i.e., shared references are the only means for inter-process communication) [Boy03]. His system can only reason about synchronization at process joins, and therefore cannot infer confluence of channel-communicating processes.

The literature on process algebra has popularized the asynchronous π calculus [Bou92; HT92]. Here, the term *asynchronous* is different from the notion we have been in this thesis. (The processes in vanilla π calculus are asynchronous in the sense that they do not run in lock step.) Essentially, the asynchronous π calculus is

the 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). Since the asynchronous π calculus is a subset of the π calculus, it can be handled by our system. In fact, our system is more powerful than other approaches even in this restricted setting. For example, our approach can prove that the following program is confluent:

$$!(c, 1) \mid\mid ?(d, x); !(c, 2) \mid\mid ?(c, y); \texttt{spawn}(!(d, 0)); ?(c, y)$$

4.2 Functional Languages with References

We compare the system presented in Chapter 3 with related work on functional programs with references.

Adding side effects to a functional language is an old problem with many proposed solutions. Here we compare our technique against two of the more prominent approaches: linear types [Wad98; GH90; AvGP93] and monads [Mog91; ORH93; JW93; LS97].

In approaches based on linear types [Wad98; GH90; AvGP93], there is an explicit *world* program value (or one world per region for languages with regions) that conceptually represents the current program state. By requiring each world have a linear type, the type system ensures that the world can be updated in place.

Such a system can be expressed in our type system by restricting every flow to 1, every witness to contain only one flow, and designating one dummy witness to serve the role of the "starting" witness (or for regions, one dummy witness per region). Thus our approach is more expressive than such an approach. Note that this also implies that every function type can be restricted to have either 1 or ∞ as the qualifier. It is easy to see that the program is well-typed under this restriction if and only if it is well-typed by the linear type system. The restriction limits programs to

manipulate witnesses only in a linear fashion. In practice, this implies that there can be no parallel reads, no dead witnesses, no redundant witnesses, and no duplication of values containing witnesses.

Monadic approaches [Mog91; ORH93; JW93; LS97] are inspired by category theory. In programming languages, monads are simply used to create ordering dependencies much like witnesses or states in the linear types based approach. Like linear types, monads enforce strictly sequential order. Our system can implement monadic primitives as follows (for concrete comparison, we use *state monads* [LS97]):

$$\texttt{newVar} = \lambda x.\lambda y.\texttt{let } z = (\texttt{ref } x\ \pi_2(y))\ \texttt{in } y \otimes z$$
$$\texttt{readVar} = \lambda x.\lambda y.\texttt{let } z = (\texttt{read } x\ \pi_1(y))\ \texttt{in } (\pi_2(z) \otimes \pi_2(y)) \otimes \pi_1(z)$$
$$\texttt{writeVar} = \lambda x.\lambda w.\lambda y.\texttt{let } z = (\texttt{write } x\ w\ \pi_1(y))\ \texttt{in } (z \otimes \pi_2(y)) \otimes w$$
$$\texttt{>>=} = \lambda f.\lambda g.\lambda x.\texttt{let } y = (f\ x)\ \texttt{in } g\ \pi_2(y)\ \pi_1(y)$$
$$\texttt{returnST} = \lambda x.\lambda y.y \otimes x$$
$$\texttt{runST}\ e = \texttt{letreg } x\ \pi_2(e\ (\bullet \otimes x))$$

The idea behind these definitions is to implement each state monad of the type $ST(\alpha, \tau)$ as a function that takes a witness and the region α as arguments and returns a witness, the region α, and a value of the type τ. It is easy to see that if a state monad program is well-typed by the monadic type system, then it is also well-typed with our type system using the above definitions for the monadic primitives. Thus, our approach is more expressive than monads.

The monadic approach shares essentially the same limitations as linear types; for example, side effects are restricted to a linear, sequential order. (In fact, it is not too hard to see that we can actually implement monadic primitives with the linear types restriction with only slightly longer code.) On the other hand, a monadic type system has an engineering advantage as it only requires Hindley-Milner type inference.

Research in data flow languages has also proposed adding references in ways that do not guarantee determinism, such as M-structures [BNA91].

4.3 Other Models of Concurrency

We discuss deterministic concurrency in computation models other than the two models discussed above.

Purely functional languages without references are confluent. Researchers have worked on applying purely functional languages to parallel computing for decades, but unfortunately, with little success in software practice. We cite a survey and an annotated bibliography containing over 400 references on this topic [Ham94; Sch93].

Current research in data flow architectures seems to focus on compiling sequential languages, such as C, rather than data flow languages or functional languages [SMSO03; BVCG04]. While determinism is trivially guaranteed with sequential programs, it is not clear whether data flow architectures will be able to run sequential programs faster than traditional microprocessors. In fact, there is some evidence suggesting otherwise [BAG05].

Unlike in an asynchronous model of concurrency (which includes all of the models discussed thus far in this thesis), program components run in lock step in a synchronous model. The synchronous model dominates hardware design practice, and is also often used to model control systems where synchronous languages like Esterel [BG92; Ber00] and Lustre [CPHP87; HCRP91] have been designed specifically for that purpose. Many of the issues discussed in this thesis seem to disappear in the synchronous model due to the absence of timing issues. Nevertheless, determinism is important for synchronous concurrency, and is often checked with the help of simulation and human inspection [SSV05].

An exhaustive approach for checking partial confluence has been proposed in

which the confluence for every state of the program is individually checked by following the transitions from that state [GvdP00; BvdP02]. These methods, which have been shown effective for programs with a small number of states, are language independent and hence potentially work with any computation models. These methods are designed specifically to drive state space reduction, and hence have a somewhat different aim than this thesis.

Chapter 5

Conclusions

This thesis presented a system to ensure partial determinism in two settings: asynchronously running sequential processes communicating via channels and shared references (Chapter 2) and functional programming languages with references (Chapter 3). The underlying ideas behind both systems are as follows:

- Identify a sufficient condition to ensure confluence based on dependencies between shared resource accesses.
- Formulate a type-based analysis for enforcing this condition.

In Chapter 2, dependencies are induced by the sequential composition (i.e., s_1 happens before s_2 in $s_1; s_2$), and the semantics of channels (e.g., the write to an output buffered channel happens before the corresponding read), whereas Chapter 3 introduced witnesses to explicitly create the needed dependencies. We showed that our approach is powerful enough to prove more programs partially deterministic than previous approaches.

Bibliography

[AG98] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, San Diego, California, January 1998.

[AN90] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transaction on Computers*, 39(3):300–318, 1990.

[AvGP93] Peter Achten, John H. G. van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 1–17. Springer-Verlag, 1993.

[AW77] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[BAG05] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. Dataflow: A complement to superscalar. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 177–186, Austin, TX, March 20-22 2005.

[Ber00] Gerard Berry. The foundations of Esterel. pages 425–454, 2000.

[BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BNA91] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In J. Hughes, editor, *FPCA '91 Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568, Cambridge, MA, August 1991. Springer-Verlag.

[Bou92] Gerard Boudol. Asynchrony and the pi-calculus. Technical Report 1702, INRIA Sophia Antipolis, May 1992.

[Boy03] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, Tenth International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, San Diego, CA, June 2003. Springer-Verlag.

[BVCG04] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 14–26, New York, NY, USA, 2004. ACM Press.

[BvdP02] Stefan Blom and Jaco van de Pol. State space reduction by proving confluence. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 596–609, Copenhagen, Denmark, July 2002.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, January 1987.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, January 1999.

[DM75] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, New York, NY, USA, 1975. ACM Press.

[ET05] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM International Conference On Embedded Software*, pages 264–272, Jersey City, NJ, September 2005.

[FC95] John T. Feo and David C. Cann. A report on the Sisal language project. pages 205–222, 1995.

[FFSA98] Manuel Fähndrich, Jefrrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.

[GDF+97] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar, and Patrick Miller. The Sisal model of functional programming and its implementation. In *PAS '97: Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis*, page 112, Washington, DC, USA, 1997. IEEE Computer Society.

[GH90] Juan C. Guzman and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, Philadelphia, Pennsylvania, June 1990.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[GvdP00] Jan Friso Groote and Jaco van de Pol. State space reduction using partial tau-confluence. In *Proceedings of 25th International Symposium on the Mathematical Foundations of Computer Science 2000*, pages 383–393, Bratislava, Slovakia, August 2000.

[Ham94] Kevin Hammond. Parallel functional programming: An introduction. In *International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 1994. World Scientific.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, pages 21–51. Springer-Verlag, 1992.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974.

[Kön00] Barbara König. Analysing input/output-capabilities of mobile processes with a generic type system. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 403–414, Geneva, Switzerland, July 2000.

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, September 1999.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993.

[LS97] John Launchbury and Amr Sabry. Monadic state: Axiomatization of type safety. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 227–238, Amsterdam, The Netherlands, June 1997.

[McG82] James R. McGraw. The VAL language: Description and analysis. *ACM Trans. Program. Lang. Syst.*, 4(1):44–82, 1982.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[NS97] Uwe Nestmann and Martin Steffen. Typing confluence. In *Proceedings of FMICS '97*, pages 77–101, 1997.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56, Charleston, South Carolina, January 1993.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Sch93] Wolfgang Schreiner. Parallel functional programming — An annotated bibliography. Technical Report 93-24, Johannes Kepler University, Linz, Austria, 1993.

[SMSO03] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291, Washington, DC, USA, 2003. IEEE Computer Society.

[SSV05] Patrick Schaumont, Sandeep Shukla, and Ingrid Verbauwhede. Extended abstract: A race-free hardware modeling language. In *Third ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2005)*, pages 255–256, July 2005.

[TA05] Tachio Terauchi and Alexander Aiken. Witnessing side-effects. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 105–115, Tallinn, Estonia, September 2005. ACM.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[Wad98] Philip Wadler. Linear types can change the world! In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 63–74, Baltimore, Maryland, September 1998.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. In *International Workshop on Types in Compilation*, Montreal, Canada, September 2000.