THE UNIVERSITY OF CHICAGO

ABSTRACT TRACE ANALYSIS FOR CONCURRENT PROGRAMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
YINGQI XIAO

CHICAGO, ILLINOIS
AUGUST 2010
To my parents, Meng and Mingying, and my wife, Jing.
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... vi

ACKNOWLEDGEMENTS .................................................. viii

ABSTRACT .................................................................. ix

Chapter

1 INTRODUCTION ....................................................... 1
   1.1 The Challenges .................................................... 2
       1.1.1 A Simple Concurrent Language ....................... 2
       1.1.2 A Simple Example ........................................ 4
       1.1.3 A Pipeline Example ...................................... 6
   1.2 This Dissertation ................................................ 9
   1.3 Organization of This Dissertation ......................... 12

2 BACKGROUND AND RELATED WORK ......................... 14
   2.1 Data-Flow Analysis ............................................ 14
       2.1.1 An Example: Reaching-Definition Analysis for Channels .... 15
   2.2 Control-Flow Analysis ....................................... 16
   2.3 $\pi$-calculus ..................................................... 18
   2.4 Related Work ..................................................... 20
       2.4.1 Input and Output Only Channels ..................... 21
       2.4.2 Linear Types for Concurrent Languages ............... 22
       2.4.3 Nielson and Nielson’s Type and Effect System ....... 26
       2.4.4 Session Types ............................................ 29
       2.4.5 Colby’s Abstract Interpretation Approach ........... 30
       2.4.6 Our Previous Work ...................................... 31
3 ABSTRACT TRACE ANALYSIS .......................................................... 35
  3.1 Dynamic Semantics ................................................................. 35
    3.1.1 Properties of Traces .......................................................... 39
  3.2 Abstract Trace Analysis Overview ........................................... 40
  3.3 Type-Sensitive Control-Flow Analysis for CML ........................... 41
    3.3.1 Introduction of 0-CFA ...................................................... 41
    3.3.2 Abstract Values ............................................................. 43
    3.3.3 Type-Sensitive CFA ......................................................... 44
    3.3.4 Properties ................................................................. 45
  3.4 The Extended CFG ............................................................... 45
  3.5 Abstract Trace Analysis ....................................................... 47

4 APPLICATIONS OF ANALYSIS ......................................................... 51
  4.1 Detecting Special Communication Pattern ................................ 51
    4.1.1 Specialized Channel Operations ....................................... 52
    4.1.2 An Example ............................................................... 53
    4.1.3 Communication Topology Analysis .................................... 56
  4.2 Concurrency Analysis ............................................................ 65
    4.2.1 Overview ................................................................. 66
    4.2.2 Ordering Enforced by Control Flow .................................... 67
    4.2.3 Ordering Enforced by Synchronization ................................. 71
    4.2.4 Refinement .............................................................. 73
    4.2.5 Automatic Monitor Recognition and Transformation ............... 73
    4.2.6 Monitor Communication Pattern Recognition ....................... 77
    4.2.7 Monitor Thread Transformation ....................................... 81

5 ANALYSIS CORRECTNESS ............................................................... 84
  5.1 Communication Topology ....................................................... 84
  5.2 Concurrency Analysis ........................................................... 97

6 IMPLEMENTATION .................................................................. 103
  6.1 The Type-Sensitive CFA Algorithm .......................................... 103
  6.2 Abstract Trace Analysis Algorithm ......................................... 108
  6.3 Implementation of Specialized CML Primitives ............................ 110
    6.3.1 Preliminaries ............................................................. 111
    6.3.2 The General Parallel Implementation ................................. 114
    6.3.3 A Reference Implementation of Fan-In and Fan-Out Channels ... 114
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A simple concurrent language</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>A simple example in SCL</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>The flow graph of the simple example</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Simple pipeline</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>The flow graph of the simple pipeline example</td>
<td>8</td>
</tr>
<tr>
<td>1.6</td>
<td>One possible history of five processes</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Syntax of $\pi$-calculus</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Structural congruence of the $\pi$-calculus</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Reduction rules of the $\pi$-calculus</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Syntax of tagged processes</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Typing rules</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Syntax for linear type channels</td>
<td>23</td>
</tr>
<tr>
<td>2.7</td>
<td>Reduction rules of linear types channels</td>
<td>24</td>
</tr>
<tr>
<td>2.8</td>
<td>Typing rules</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>Typing rules</td>
<td>27</td>
</tr>
<tr>
<td>2.10</td>
<td>Concurrent evaluation rules</td>
<td>28</td>
</tr>
<tr>
<td>2.11</td>
<td>Analyzing the CFG $G$ for channel $c$</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Sequential evaluation</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>The join of two abstract values</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>Approximation of channel send and receive sites</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>Simple service</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>A simple service with an abstract client-server protocol</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Data-flow of the server’s request channel</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>A version of Figure 4.1 with specialized communication operations</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Standard data-flow equations</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>A self-concurrent example</td>
<td>68</td>
</tr>
<tr>
<td>4.6</td>
<td>The self-concurrent example in ECFG</td>
<td>69</td>
</tr>
<tr>
<td>4.7</td>
<td>Data-flow equations for concurrency analysis</td>
<td>71</td>
</tr>
<tr>
<td>4.8</td>
<td>A simple example with synchronization edge</td>
<td>73</td>
</tr>
<tr>
<td>4.9</td>
<td>The self-concurrent example in ECFG</td>
<td>74</td>
</tr>
<tr>
<td>4.10</td>
<td>An example of monitor tasks</td>
<td>75</td>
</tr>
<tr>
<td>4.11</td>
<td>Simple monitor</td>
<td>76</td>
</tr>
<tr>
<td>4.12</td>
<td>The optimized version of simple monitor</td>
<td>77</td>
</tr>
<tr>
<td>4.13</td>
<td>The simple monitor after transformation</td>
<td>83</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>5.1</td>
<td>The definition of \texttt{abs}</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>CFA for expressions Part I</td>
<td>104</td>
</tr>
<tr>
<td>6.2</td>
<td>CFA for expressions Part II</td>
<td>105</td>
</tr>
<tr>
<td>6.3</td>
<td>Algorithm for abstract traceflow analysis</td>
<td>109</td>
</tr>
<tr>
<td>6.4</td>
<td>Algorithm for testing for a valid trace</td>
<td>109</td>
</tr>
<tr>
<td>6.5</td>
<td>Algorithm for redundant trace</td>
<td>110</td>
</tr>
<tr>
<td>6.6</td>
<td>The reference implementation of general \texttt{send}</td>
<td>113</td>
</tr>
<tr>
<td>6.7</td>
<td>The reference implementation of general \texttt{recv}</td>
<td>113</td>
</tr>
<tr>
<td>6.8</td>
<td>The specialized implementation of \texttt{send} for fan-in channels</td>
<td>118</td>
</tr>
<tr>
<td>6.9</td>
<td>The specialized implementation of \texttt{recv} for fan-in channels</td>
<td>119</td>
</tr>
<tr>
<td>6.10</td>
<td>The specialized implementation of \texttt{send} for fan-out channels</td>
<td>120</td>
</tr>
<tr>
<td>6.11</td>
<td>The specialized implementation of \texttt{recv} for fan-out channels</td>
<td>121</td>
</tr>
<tr>
<td>7.1</td>
<td>Primes benchmark results</td>
<td>125</td>
</tr>
<tr>
<td>7.2</td>
<td>Primes speed-up curves</td>
<td>125</td>
</tr>
<tr>
<td>7.3</td>
<td>Fan-out benchmark results</td>
<td>126</td>
</tr>
<tr>
<td>7.4</td>
<td>Fan-out speed-up curves</td>
<td>127</td>
</tr>
<tr>
<td>7.5</td>
<td>Fan-in benchmark results</td>
<td>128</td>
</tr>
<tr>
<td>7.6</td>
<td>Fan-in speed-up curves</td>
<td>128</td>
</tr>
<tr>
<td>7.7</td>
<td>Monitor benchmark results</td>
<td>130</td>
</tr>
<tr>
<td>7.8</td>
<td>Monitor speed-up curves</td>
<td>131</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Professor John H. Reppy, for his support and guidance throughout my graduate research at the University of Chicago and for his help with the proof reading of this dissertation. Without his insight and guidance, I would never be able to accomplish this dissertation. I would also like to thank other members of my committee, Dave MacQueen and Xinyu Feng, for reading my dissertation and providing feedback on it.

I would like to give my heartfelt thanks to my parents, who raised me with their endless love and support. I would also like to give my heartfelt thanks to my wife, who has accompanied me with her love, understanding and encouragement. I dedicate this dissertation to my parents and my wife.

Finally, I would like to thank the computer science department, the University of Chicago for providing me the opportunity to study and research here.

The research presented in this dissertation was supported, in part, by the NSF under Awards CCF-0811389 and CCF-0937561.
ABSTRACT

Program analysis plays an important role in modern software. With the end of free lunch for single-threaded programs and the advent of cheap parallel processing on the desktop (and laptop), concurrency is increasingly used in software. The consequence is that designing and implementing static analyses for concurrent programs is more and more important. Many concurrent languages provide powerful concurrency primitives, such as dynamic creation of processes, dynamic creation of first-class channels, and communication over those channels. These primitives provide more flexibility to programers, but they also make programs more difficult to analyze. The consequence is that designing and implementing static analyses for such programs is more challenging.

One fundamental challenge for static analysis of concurrent programs is that deciding if a given program point is reachable is undecidable. The presence of dynamic concurrency makes it more difficult to analyze concurrent programs. To solve this problem, we have to provide a safe approximation of run-time behaviors so that we can compute properties of programs without losing too much precision. The goal of this dissertation is to investigate techniques for static analysis of dynamic communication for concurrent programs and present a new approach to analyze concurrent programs.

In this dissertation, we present a new approach for computing a precise approximation of all possible control paths that reach each program points. We call the approach abstract trace analysis. The approximation is precise because it only considers valid control paths where function calls return to their corresponding call sites. We demonstrate how to do powerful analysis based on abstract trace analysis. We show how to analyze communication topology based on abstract trace analysis. The analysis can extract channel
usage information, distinguish different abstract dynamic instances of same channels, and generate more precise output than our previous work and other existing work. In this dissertation, we also show how to extract concurrency information from an abstract trace and use it to do optimization by detecting and transforming monitor threads into more efficient monitor functions. We also present performance data for our Manticore implementation of specialized communication primitives and monitor optimization.
CHAPTER 1
INTRODUCTION

Program analysis plays an important role in modern software. Various program-analysis tools have been developed to enable program optimization and verify program correctness. One of the approaches, dynamic analysis, is to analyze program behaviors by executing them. The other approach is to use static-analysis techniques to predict run-time behaviors without actually executing programs. In this work, we present a new static-analysis technique for message passing programs.

For decades, many single-threaded programs have enjoyed the free ride provided by major processor manufacturers and architectures; many single-threaded programs have free performance improvement when the clock speed increases. Now, major processor makers seem to reach the limit of making processors faster in terms of the clock speed, and tend to make multi-core chips to boost performance. Herb said that the free lunch for single-threaded programs is over [Sut05]. With the advent of cheap parallel processing on the desktop (and laptop), concurrency is increasingly used in software. The consequence is that designing and implementing static analyses for concurrent programs is more and more important.

Many concurrent languages provide powerful concurrency primitives, such as dynamic creation of processes, dynamic creation of first-class channels, and communication over those channels. These primitives provide more flexibility to programmers, but they also make programs more difficult to analyze. The consequence is that designing and implementing static analyses for such programs is more challenging.
There is interesting similarity between functions in higher-order functional language such as ML and channels in higher-order concurrent language such as CML. In languages like ML, functions are first-class values, thus the control flow of these languages is highly dynamic. It is difficult to determine the call graph (i.e., which functions are called at a given call site). In order to determine the call graph, data flow information is needed, while computing data flow information requires the call graph. Thus data flow information and control flow information must be computed at the same time. Similar to functions in higher-order functional languages, channels are first-class values in higher-order concurrent languages, and synchronization and communication are highly dynamic. It is difficult to determine the communication graph (which channel is used to synchronized or which two communication primitives are synchronized). In order to determine the communication graph, data flow information is needed, while data flow information can be computed only if communication graph is available. As in the case of higher-order functions, it is more challenging to analyze properties in the presence of dynamic communication.

1.1 The Challenges

In this section, we use two examples to illustrate the challenges of analysis of concurrent programs in the presence of dynamic communication. Before that, we introduce the basis of our analysis, our simple concurrent language (SCL). In this dissertation, we present our analysis in the context of this simple language.

1.1.1 A Simple Concurrent Language

This language is a subset of CML[Rep91], without the support for CML events. Figure 1.1 gives the abstract syntax for SCL.
A program $p$ is an expression. The sequential expression forms include let-bindings, nested function bindings, function application, condition branches, pair construction and projection. In addition, there are four concurrent expression forms: channel definition, process spawning, message sending, and message receiving. We assume that variables and function identifiers are globally unique. Types include abstract types ($T$), function types, pair types, and channel types. Abstract types are predefined types (e.g., unit, int, bool, etc.).

For a given program $p$, we assume that each expression in $p$ is labeled with a unique program point $a \in \text{PROGPT}$. We write $a : e$ to denote that $e$ is the expression at program point $a$. 
Figure 1.2: A simple example in SCL

1.1.2 A Simple Example

We use the simple example in Figure 1.2 to show dynamic communication in SCL and the challenge of analyzing it. We show its flow graph in Figure 1.3. For simplicity and clarity, we use line numbers as program points to label expressions. In this example, a channel `ch` is created at line 1 and three processes `P_1, P_2, P_3` are spawned. Both `P_1` and `P_2` attempt to receive a message on `ch` at lines 3 and 8 respectively, expecting the message received to be some channel value. Then they try to send the corresponding value on the channel value just received at line 5 and 10 respectively. Process `P_3` creates a local channel `replCh`. After sending it on `ch`, it waits for a message back over `replCh` at line 13. The channel `ch` is only used at line 3, 8 and 15. The communication on `replCh`, however, is complicated, because it is transmitted as a value on `ch` at line 15. To determine communication on
Figure 1.3: The flow graph of the simple example
replCh, we first need to determine the flow of replCh as data on ch. We observe that replCh may flow to line 3 or 8 through channel ch, which is non-determined. This is the challenge of static analysis in the presence of dynamic communication, that we need to compute communication graph information and data flow information at the same time. By further observation, we can find that the simple example program has the following properties:

- replCh₁ and replCh₂ can only be bound to replCh created at line 13.
- Although the message sent on replCh is non-determined (1 or 2) and the actual send-site on replCh is non-determined (line 5 or 10), it is guaranteed that only one message will be sent on the channel at one of the send-sites, 5 or 10. Thus, replCh is a one-shot channel, i.e., a channel that is used at most once.

1.1.3 A Pipeline Example

The communication graph becomes more complicated in the presence of recursive procedures or functions and dynamic process creation. We use a simple pipeline example in Figure 1.4 to illustrate these new challenges. We show its flow graph in Figure 1.5. Function pipe creates channel outCh at line 11 and spawns a process to keep receiving messages on inCh and sending them on outCh. In this example, function pipe is called three times at line 28, 29 and 30 respectively. There are different instances of the function body for each call. Three instances of channel outCh at line 11 are created. We differentiate them by outCh₁, outCh₂ and outCh₃. And three processes are spawned at line 18. We call them P₁, P₂, and P₃. Note that P₁, P₂, and P₃ have different instances of communication statements at lines 13 and 15. In total, five processes are spawned to implement the pipeline. Function starter spawns a process, called starter, to produce a stream
fun pipeline () =
  chan pipeInCh in
fun starter () =
  fun loop i = (send(pipeInCh, i);
               loop(i+1))
in
  spawn loop 0

fun pipe (inCh) =
  chan outCh in
fun loop () =
  let i = recv inCh
  in
    (send (outCh, i)
     loop ())
in
    (spawn (loop ());
     outCh)

fun ender (endCh) =
  fun loop () = (
    recv(endCh, i);
    loop())
in
    spawn (loop ())
let
  pipeCh1 = pipe (pipeInCh)
  pipeCh2 = pipe (pipeCh1)
  pipeCh3 = pipe (pipeCh2)
in
  (starter ();
   ender (pipeCh3))
in
  pipeline ()

Figure 1.4: Simple pipeline
Figure 1.5: The flow graph of the simple pipeline example
of numbers on channel `pipeInCh`, which is created at line 2. Function `ender` spawns a process to receive messages from the end of pipeline. We call this process `ender`.

In this example, we are interested in possible communication on channel `outCh` created at line 11. By tracking the flow of channel value `outCh`, we have that `outCh` is used at lines 15, 13 and 23. The operations on `outCh` at lines 15, 13, 12 are `send`, `recv` and `recv` respectively. Thus, it is safe to say that communication on `outCh` can only happen between two processes of `P_1`, `P_2`, `P_3` and `ender` at lines 15, 13 and 12 as shown in Figure 1.5. With more careful examination, however, we observe that in each process, communication at line 15 and communication at line 13 are not over the same dynamic channel instance. In other words, for any dynamic instance `outCh_i`, `P_i` cannot both send a message on `outCh_i` at line 15 and receive a message on `outCh_i` at line 13, and `inCh` must be different instances of `outCh` in each process. With further examination, we can observe the following facts about the pipeline.

- `pipeInCh` is only used to communicate between `starter` and `P_1`
- `outCh_1` is only used to communicate between `P_1` and `P_2`
- `outCh_2` is only used to communicate between `P_2` and `P_3`.

A possible history of five processes is shown in Figure 1.6. It is challenging for a static analysis to discover that all instances of `outCh` have a point-to-point communication pattern.

### 1.2 This Dissertation

One fundamental challenge of static analysis for concurrent programs is that deciding if a given program point is reachable is undecidable. As shown from the above two examples,
the presence of dynamic concurrency introduces nondeterminism and makes programs dif-
cult to analyze statically. To solve this problem, we have to provide a safe approximation
of the set of possible run-time behaviors so that we can compute properties of programs
without losing too much precision.

In our previous work, we presented a modular technique to solve the problem by track-
ing values of abstract type that escape from the module “into the wild” and a data-flow
analysis that uses an extended control-flow graph to compute a safe approximation of the
set of possible dynamic communication over channels [Xia05, RX07]. The extended CFG
has extra edges to represent process creation, values communicated by message-passing,
and values communicated via the outside world (a.k.a. the wild). The data-flow analysis
computes an approximation of the number of processes that send or receive messages on
the channel, as well as an approximation of the number of messages sent on the channel.
Although the previous work is a significant step toward optimization of CML, it has the following limitations or shortcomings.

- It does not distinguish different dynamic instances of a channel. For instance, our previous work cannot determine that `outCh` and `inCh` must be different instances of `outCh` at line 11 for each process when analyzing the program in Figure 1.4.

- It is not precise enough. In our previous work, the analysis is over-conservative so that special communication patterns detected in programs may not be precise enough. For example, our previous analysis would identify `replCh` in the simple example as a fan-in channel instead of a one-shot channel.

- It is not context sensitive. In our previous work, the extended control flow graph has control flow edges from function exit points to all successors of possible function call sites. When analyzing the function exit node, the analysis computes the union of the analysis for all outgoing edges. Thus the analysis is over-conservative.

The goal of this dissertation is to investigate techniques for static analysis of dynamic communication in concurrent programs and present a new approach to analyze concurrent programs as follows.

- In this dissertation, we present a new approach to compute a precise approximation of all possible control paths that reach each program points. We call this approach *abstract trace analysis*. The approximation is precise because it only considers valid control paths where function calls return to their corresponding call sites.

- In this dissertation, we also demonstrate two applications of abstract trace analysis. We show how to analyze the communication topology of a program based on abstract trace analysis. The analysis can extract channel usage information from the abstract
trace, distinguish different abstract dynamic instances of same channels and generate more precise output than our previous work. We also show how to extract concurrency information from abstract traces and use it to do optimization by detecting and transforming monitor threads into more efficient monitor functions.

- In this dissertation, we also present a lock-free implementation of specialized communication primitives for fan-in and fan-out channels.

- In this dissertation, we also present performance data for our Manticore implementation of specialized communication primitives and the monitor optimization.

### 1.3 Organization of This Dissertation

The remainder of this dissertation is organized as follows. In the next chapter, we give an introduction of main techniques for static analysis of concurrent programs and discuss the related work. We then present a dynamic semantics for SCL and describe our new approach of static analysis for SCL. The new approach consists of two major components. The first is a twist on type-sensitive CFA [Rep05] to obtain whole program control-flow and data-flow information. The second component is a abstract trace analysis that uses an extended control-flow graph constructed from the result of the CFA to compute a precise approximation of all possible control paths that reach each program points. In Chapter 4, we use two non-trivial applications of abstract trace analysis to show how to use the new technique to further analyze and determine useful properties of concurrent programs. As the first application, we present an analysis to extract channel usage information from abstract trace analysis. This information can be used to distinguish different abstract dynamic instances of same channels. As the second application, we present an analysis to extract concurrency information from abstract trace information and compute a partial ordering between state-
ments in programs. This information allows us to detect monitor-communication patterns in programs. These special patterns can then be exploited by transforming monitor threads into more efficient monitor functions. The correctness of our analyses is proved in Chapter 5. In Chapter 6, we describe important details for real implementations of our analyses and also describe a reference implementation of fan-in and fan-out channels. We present performance data for our Manticore implementation of specialized communication primitives and monitor optimization in Chapter 7. Finally we conclude in Chapter 8.
CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, I introduce the basics of static analysis for concurrent programs and the related work in this field.

2.1 Data-Flow Analysis

Data-flow analysis (DFA) is one approach to static program analysis that computes facts about definitions and uses of data in programs. Usually, data-flow analysis is flow-sensitive and based on a control-flow graph (CFG) that represents the possible flow of control in a program. A CFG is a directed multigraph, where nodes represent program statements and edges represent flow of control.

The classic way to perform data-flow analysis is to give a transfer function $f$ for each node in the CFG. The transfer function describes how facts about data are transferred by the statements at the node. The domain is required to be a bounded lattice $(L, \sqcap, \sqsubseteq, 0, 1)$ where $\sqcap$ is a meet operation, $\sqsubseteq$ is a partial order relation, 0 is a zero element and 1 is a unit element. The transfer function can be extended to finite paths. A valid path in a flow graph is a sequence of nodes $(n_1, n_2, ..., n_i)$ such that $n_j$ is a predecessor of $n_{j+1}$ for $1 \leq j < i$. For each path $p = (n_1, n_2, ..., n_i)$, the transfer function of $p$ is defined as $F(p) = f_{n_i} \circ ... \circ f_{n_2} \circ f_{n_1}$ where $f_{n_i}$ is the transfer function associated with node $n_i$. The fact that we want to compute for each node is that the fact meets over all paths (MOP) that reach the node. And the solution for node $n$ is generated by $F(n) = \sqcap \{F(p)(0) \mid p \text{ is a valid path from program entry to } n \}$. This solution is called
MOP solution. As the computation for all possible paths that reach each node is usually not effective, we computes an approximate solution to the MOP solution. The alternative method defines two classes of data flow equations for each node in the graph. One class of equations represent the relation between the entry state of the node and the exit states of its predecessors in the graph. The other class of equations represent the relation between the exit state and entry state of the same node, which corresponds to the transfer function in the MOP solution. These two classes of equations form a system of equations. The solution for node $n$ is generated by solving maximal fixed point (MFP) of the equation system. This solution is called MFP solution. Kildall proved that if transfer functions associated for each node are distributive then the MFP solution equals to the MOP solution [Kil73]. Kam and Ullman prove that if transfer functions associated for each node are monotone then the MFP solution is a safe approximation of the MOP solution and show that MFP solutions might not be MOP solutions [KU77].

The classic data-flow analyses include reaching definition analysis and liveness analysis. The reaching definition analysis computes for each program point a set of assignments that may reach the program point. The liveness analysis computes for each program point a set of variables that may be used afterwards before they are re-defined. The liveness analysis can be used as the basis for dead code elimination.

### 2.1.1 An Example: Reaching-Definition Analysis for Channels

The reaching definitions analysis for channels in SCL follows the standard MFP solution of data-flow analysis. The function $\text{kill}_{RD}$ produces the set of pairs of channel variables and program points that are killed by the statement. The function $\text{gen}_{RD}$ produces the set of pairs of channel variables and program points generated by the statement. Only assignments to channel variables kill or generate channel definitions. And we say that
channel \( ch \) is defined at program point \( a \), if there is assignment to \( ch \) at \( a \). The two functions are defined as follows for channel-definition statements.

\[
\begin{align*}
\text{kill}_{RD}(a : \text{chan } ch) & = \bigcup \{ (ch, a') \mid \text{ch is defined at } a' \} \\
\text{kill}_{RD}(a : ch = e) & = \bigcup \{ (ch, a') \mid \text{ch is defined at } a' \} \\
\text{gen}_{RD}(a : \text{chan } ch) & = \{(ch, a)\} \\
\text{gen}_{RD}(a : ch = e) & = \{(ch, a)\}
\end{align*}
\]

Any other statement \( a : e \) is mapped to the empty set by \( \text{kill}_{RD} \) and \( \text{gen}_{RD} \).

\[
\begin{align*}
\text{kill}_{RD}(a : e) & = \emptyset \\
\text{gen}_{RD}(a : e) & = \emptyset
\end{align*}
\]

For simplicity, we regard program point \( a \) as equivalent to the expression labeled with \( a \). The data-flow equations for reaching-definitions analysis are defined as follows.

\[
\begin{align*}
\text{RD}_{in}(a) & = \begin{cases} 
\{ \epsilon \} & \text{if } a \in \text{Program Entry} \\
\bigcup \{ \text{RD}_{out}(a') \mid a' \in \text{pred}(a) \} & \text{otherwise}
\end{cases} \\
\text{RD}_{out}(a) & = (\text{RD}_{in}(a) \setminus \text{kill}_{RD}(a)) \cup \text{gen}_{RD}(a)
\end{align*}
\]

where \( a' \in \text{pred}(a) \) if there exists a edge from \( a' \) to \( a \).

### 2.2 Control-Flow Analysis

Control-flow analysis is another approach to static program analysis. Control-flow analysis computes facts about flow of control in a program. Most, if not all, static program analyses require control-flow information about programs.

The term control-flow analysis originates from Allen’s paper [All70]. Allen uses a
control-flow graph to codify the control-flow relationships in a program and presents an interval construct analysis and uses it to compute dominance relationships in a given control flow graph. In a language without higher-order functions, the flow of control information is apparent from program syntax. In a language with higher-order functions, however, the flow of control information depends on data-flow information [Shi88]. There is interdependency between control flow information and data flow information in languages with higher-order functions. The control-flow graph cannot be determined at compile time from the program text. For example, consider the following expression in SML:

```sml
let
  fun addx x = let
    fun g y = x + y
  in
    g
  end
val add12 = addx 12
in
  add12 1
end
```

Here, the control flow of applying `add12` to `1` depends on the value bound to variable `add12`. In a language with higher-order functions, the actual function that is invoked at a given call site may not be available until run time.

```sml
let
  fun applyh h = h 1
  fun f x = x + 1
  fun g x = x + 2
  in
    (applyh f) + (applyh g)
end
```

Here, the control flow of applying `h` to `1` in the body of `applyh` depends on the runtime argument passed to the function.

Shivers presents a context-insensitive analysis, called $0\text{CFA}$, as a solution for continuation-passing style (CPS) based languages and generalizes it to a class of context-sensitive analyses $k\text{CFA}$ for $k \geq 1$ [Shi91]. His analysis computes control-flow information and data flow
\[
P ::= \alpha.P \mid P \mid P \mid (\nu x)P \mid !P \mid 0
\]
\[
\alpha ::= x(\tilde{y}) \mid \bar{x}\langle\tilde{y}\rangle
\]

Figure 2.1: Syntax of \(\pi\)-calculus

information at the same time. Serrano adapts Shivers’ 0CFA to deal with the full Scheme language in Bigloo [Ser95]. Reppy later presents a type-sensitive control-flow analysis that exploits type abstraction in ML [Rep05].

2.3 \(\pi\)-calculus

The \(\pi\)-calculus is a simple process algebra presented by Milner, Parrow, and Walker to describe and analyze process mobility of concurrent programs [MPW92]. In \(\pi\)-calculus, the most basic notion is a name. Communication channels and variables are all names. Communication channels are allowed to be communicated along other channels as data values. Thus, the \(\pi\)-calculus can be easily used to model low-level communication behaviors for concurrent programs. By using type systems, useful information can be automatically inferred. We will discuss it in details in the following section.

The syntax in Figure 2.1 defines processes of a basic version of polyadic \(\pi\)-calculus. For simplicity, we omit definitional devices of abstraction and concretion and the operators of choice and matching.

Here \(\alpha.P\) is a process waiting for action \(\alpha\) before proceeding as \(P\). \(P\mid P\) are two concurrent processes. \((\nu x)P\) is a process creating a new name \(x\) in process \(P\). Replication of process \(P\) denoted by \(!P\), stands for any finite number of copies of \(P\) executed concurrently. \(0\) is an inactive process, whose execution is complete. Prefix \(\alpha\) is either an input or output action. \(x\) is a name and \(\bar{x}\) is the co-name of \(x\). \(\tilde{y}\) represents a sequence of names \(y_1, y_2, \ldots, y_n\) (\(n\) may be 0). \(x(\tilde{y})\) stands for the action of receiving values along \(x\) and storing in \(\tilde{y}\), while \(\bar{x}\langle\tilde{y}\rangle\) stands for the action of sending values \(\tilde{y}\) along \(x\).
• $P \equiv Q$, if $P$ and $Q$ are equivalent by $\alpha$ conversion.

• $P|Q \equiv Q|P$, $P|(Q|R) \equiv (P|Q)|R$, $P|0 \equiv 0$

• $(\nu x)0 \equiv 0$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$

• $((\nu x)P)|Q \equiv (\nu x)(P|Q)$, if $x$ is not free in $Q$.

• $!P \equiv P|!P$

Figure 2.2: Structural congruence of the $\pi$-calculus

$$x(\bar{y}).P \mid \bar{x}(\bar{z}).Q \rightarrow P[\bar{z}/\bar{y}] \mid Q$$

R-COMM

$$P \rightarrow P'$$

R-PAR

$$P|Q \rightarrow P'|Q$$

R-PAR

$$(\nu x)P \rightarrow (\nu x)P'$$

R-RES

$$P \equiv P' \rightarrow Q' \equiv Q$$

R-STRUCT

Figure 2.3: Reduction rules of the $\pi$-calculus

Milner presents the operational semantics of the $\pi$-calculus by using a reduction relation. And a structure congruence is used to make the reduction system simple. The structure congruence relation in Figure 2.2 identifies processes without computational significance. A single computational step is written as $P \rightarrow Q$. The reduction relation $\rightarrow$ is the least relation closed under the rules in Figure 2.3. Note that in R-COMM $x$ and $\bar{x}$ must have equal arity. Therefore, a type system is presented to enforce name-use disciplines, where arities are assigned to channels by sortings [MPW92].
2.4 Related Work

There are a number of papers that describe various program analyses for concurrent programs. These analyses can be organized by the techniques used.

A number of researchers have used type systems to analyze the communication behavior of concurrent programs. For example, Pierce and Sangiorgi [PS93] developed a type system to restrict a channel’s usage as input/output only. Kobayashi, Pierce and Turner proposed a refined π-calculus type system to detect linear-type channels [KPT96]. Nielson and Nielson [NN94] developed an effect-based analysis to detect communication behavior during the execution of the CML program. The analysis gives an approximate number of processes and channels that will be created for a CML program.

There have also been a number of abstract interpretation-style analyses of concurrent languages that are closer in style to our analysis. Mercouroff designed and implemented an abstract-interpretation style analysis for CSP programs based on an approximation of the number of messages sent between processes [Mer91]. While this analysis is one of the earliest for message-passing programs, it is of limited utility for our purposes, since it is limited to a very static language. Jagannathan and Weeks proposed an analysis for parallel scheme programs that distinguishes memory accesses/updates by thread [JW94]. Unfortunately, their analysis is not fine-grained enough to distinguish different dynamic instances as our analysis since it collapses multiple threads that have the same spawn point to a single approximate thread. Marinescu and Goldberg have developed a partial evaluation technique for CSP [MG97]. Their algorithm can eliminate redundant synchronization, like Mercouroff’s work, it is limited to programs with static structure. The closest work to ours is Colby’s abstract-interpretation for a subset of cml [Col95], which analyses the communication topology of programs.

In the following sections, I will cover some of the above work in more detail.
2.4.1 Input and Output Only Channels

Milner’s π-calculus has no notion of restricted access to channels. In some cases, however, we would like to restrict processes’ access to channels. Let us consider a client-server example. A server and clients share the service channel \( ch \). But clients are only allowed to send requests over the service channel \( ch \), while the server is only allowed to receive requests over the service channel \( ch \). If a bad implementation of a client outputs garbage over \( ch \), then the protocol will be disrupted.

Pierce and Sangiorgi [PS93] refined the π-calculus by restricting the use of a channel in a given context and distinguishing between the ability to input and output. In their presentation, bound names are explicitly typed with tagged sorts. Channel sorts (\( S \)) are tagged with input/output tags standing for input/output only ability. Figure 2.4 gives the language syntax. The operational semantics only differs from Milner’s in a few rules. First, structure congruence rules for \(\text{wrong} \) are added in an obvious way.

\[
P|\text{wrong} \equiv \text{wrong}, \quad !\text{wrong} \equiv \text{wrong}, \quad \text{and} \quad (\nu x : S)\text{wrong} \equiv \text{wrong}
\]

Then a partial-order relation among tags is presented to enforce channel-use disciplines. The relation is called sub-tag relation and contains \( \pm \leq \) and \( \pm \leq + \). The reduction rule R-COMM is modified as follows.

\[
\begin{align*}
m = n, & \quad I \leq -, & \quad O \leq +, & \quad O_i \leq \hat{E}(S_i) \\
x^I(y_1 : S_1, \ldots, y_m : S_m).P | x^O\langle z_1^{O_1}, \ldots, z_n^{O_n} \rangle.Q \rightarrow P[\tilde{z}/\tilde{y}]|Q
\end{align*}
\]

\[
\begin{align*}
m \neq n \quad \text{or} \quad I \not\leq - \quad \text{or} \quad O \not\leq + \quad \text{or} \quad O_i \not\leq \hat{E}(S_i) \\
x^I(y_1 : S_1, \ldots, y_m : S_m).P | x^O\langle z_1^{O_1}, \ldots, z_n^{O_n} \rangle.Q \rightarrow \text{wrong}
\end{align*}
\]

Here \( \hat{E}(S_i) \) denotes the outmost tag in \( S_i \).
\begin{align*}
S & := \ (S_1 \ldots S_n)^T | \mu A. S | A \\
T & := \ - | + | \pm \\
P & := \ 0 | P | P | (va : S)P | x^I(y_1 : S_1, \ldots, y_n : S_n).P \\
& \quad | \ x^I(z_1^{I_1}, \ldots, z_n^{I_n}).P | !P | \text{wrong}
\end{align*}

Figure 2.4: Syntax of tagged processes

Pierce and Sangiorgi extend Milner's type system by using subtyping relation. They also present an algorithm to generate the subtyping relation from the sub-tag relation. The typing judgement has form $\Gamma \vdash P : ok$ where $\Gamma$ is a type environment binding names to tagged sorts. Figure 2.5 gives the typing rules. Note that $\vdash \Gamma(x) \leq (S_1, \ldots, S_n)^-$ in rule T-IN and $\vdash \Gamma(x) \leq (\Gamma(z_1), \ldots, \Gamma(z_n))^+$ in rule T-OUT ensure that channel $x$ can be used for input or output. Thus bad implementations of clients that output over channels without output ability are caught by type checking.

Pierce and Sangiorgi’s type system can only be used to restrict a channel’s usage as input/out only. It is not fine-grained enough to determine the number of messages sent/received on a given channel.

### 2.4.2 Linear Types for Concurrent Languages

Kobayashi, Pierce and Turner propose a refined $\pi$-calculus type system to detect linear-type channels [KPT96]. A linear-type channel is a channel that is used at most once during execution. The type system gives a safe approximation of how many times a channel is used to input and output respectively in a given context.

Figure 2.6 gives the language syntax. For simplicity, we omit booleans and conditional expressions. New channels are explicitly typed. Here $p$ is a polarity metavariable ranging over subsets of I/O ability set \{i, o\}, and $m$ is a multiplicity metavariable ranging over usage set \{1, $\omega$\}, where $\omega$ stands for any number of times. Note that $x^?(*y).P$ is a replicated input
\[\Gamma \vdash \mathbf{0} : \text{ok}\]  
\[\Gamma \vdash P : \text{ok} \quad \Gamma \vdash Q : \text{ok} \]
\[\Gamma \vdash P \mid Q : \text{ok}\]  
T-NIL

T-PAR

\[\Gamma \vdash P : \text{ok}\]
\[\Gamma \vdash \! P : \text{ok}\]

T-REP

\[\Gamma, x : S \vdash P : \text{ok}\]
\[\Gamma \vdash (\nu x : S)P : \text{ok}\]

T-RES

\[\vdash \Gamma(x) \leq (S_1, \ldots, S_n)^- \quad \Gamma, y_1 : S_1, \ldots, y_n : S_n \vdash P : \text{ok}\]
\[\Gamma \vdash x(y_1 : S_1, \ldots, y_n : S_n).P : \text{ok}\]

T-IN

\[\vdash \Gamma(x) \leq (\Gamma(z_1), \ldots, \Gamma(z_n))^+ \quad \Gamma \vdash P : \text{ok}\]
\[\Gamma \vdash \bar{x}(z_1, \ldots, z_n).P : \text{ok}\]

T-OUT

\[P ::= P \mid Q \mid x!(\bar{y}) \mid x?!(\bar{y}).P \mid x?*(\bar{y}).P \mid (\nu x : T)P\]

\[T ::= p^m[\bar{T}]\]

Figure 2.5: Typing rules

expression which guards replication with input. The intuitive meanings is that a process waits for input over channel \(x\) and it forks a copy of \(P\) and restarts itself as long as it receives any message over channel \(x\).

Following Pierce and Sangiorgi’s work, Kobayashi, Pierce and Turner present the operational semantics by a structural congruence relation and a reduction relation. The structural congruence relation has no difference from the \(\pi\)-calculus’ relation in Figure 2.2, while the reduction rules are more refined. The single-step relation is annotated with channel usage information \(\alpha\), written as \(P \xrightarrow{\alpha} Q\). The channel usage information \(\alpha\) is either a channel name \(x\) or a channel’s multiplicity \(m\). Figure 2.7 gives the refined reduction rules. Since new names are explicitly typed, rules for new names are interesting. Note that there
\[
\begin{align*}
\frac{x!\langle \tilde{z} \rangle | x?\langle \tilde{y} \rangle}. P \xrightarrow{x} P[\tilde{z}/\tilde{y}] & \quad \text{R-Comm} \\
\frac{x!\langle \tilde{z} \rangle | x?\langle \tilde{y} \rangle}. P \xrightarrow{x} P[\tilde{z}/\tilde{y}]. x?^*(\tilde{y}). P & \quad \text{R-Rep} \\
\frac{P \xrightarrow{\alpha} Q} {P|R \xrightarrow{\alpha} Q|R} & \quad \text{R-Par} \\
\frac{P \xrightarrow{x} Q \quad p = \{i, o\}} {P \xrightarrow{x} Q} & \quad \text{R-Res1} \\
\frac{(\nu x : p^\omega[\tilde{T}]). P \xrightarrow{\omega} (\nu x : p^\omega[\tilde{T}]). Q} {P \xrightarrow{x} Q \quad p = \{i, o\} \quad q = \emptyset} & \quad \text{R-Res2} \\
\frac{(\nu x : p^1[\tilde{T}]). P \xrightarrow{1} (\nu x : q^1[\tilde{T}]). Q} {P \xrightarrow{\alpha} Q \quad x \neq \alpha} & \quad \text{R-Res3} \\
\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q} {P \xrightarrow{\alpha} Q} & \quad \text{R-Cong}
\end{align*}
\]

Figure 2.7: Reduction rules of linear types channels

are three rules to track a channel’s usage for each case of channel creation.

The typing judgement has a form of \( \Gamma \vdash P \) [KPT96], where \( \Gamma \) is a type environment assigning types to distinct variables. Since types contain channel usage information, operations on types are presented to migrate the usage information from component types. The operations are defined as follows.

\[
\begin{align*}
p^\omega[\tilde{T}] + q^\omega[\tilde{T}] &= (p \cup q)^\omega[\tilde{T}] \\
p^1[\tilde{T}] + q^1[\tilde{T}] &= (p \cup q)^1[\tilde{T}] \text{ if } p \cup q = \emptyset
\end{align*}
\]

Otherwise, it is undefined. And the operations on type environment are extended pointwise
\[
\begin{align*}
\Gamma \vdash P & \quad \Gamma \vdash Q \\
\Gamma_1 + \Gamma_2 \vdash P|Q & \\
\text{T-PAR} \\
\Gamma \text{ unlimited} & \quad p = \{o\} \\
\Gamma + x : p^m[T] + \tilde{z} : T \vdash x!(\tilde{z}) & \\
\text{T-OUT} \\
\Gamma, \tilde{y} : T \vdash P & \quad p = \{i\} \\
\text{T-IN} \\
\Gamma + x : p^m[T] \vdash x?(\tilde{y}).P & \\
\text{T-REP} \\
\Gamma, \tilde{y} : T \vdash P \quad \Gamma \text{ unlimited} & \quad p = \{i\} \\
\Gamma + x : p^m[T] \vdash x^*(\tilde{y}).P & \\
\text{T-NEW} \\
\Gamma, x : p^m[T] \vdash P & \quad p = \emptyset \text{ or } \{i, o\} \\
\Gamma \vdash (\nu x : p^m[T])P & \end{align*}
\]

Figure 2.8: Typing rules

to:

\[(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \\
\Gamma_1(x) + \Gamma_2(x) & \text{if } x \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2) \\
\Gamma_1(x) & \text{if } x \in \text{dom}(\Gamma_1) \text{ and } x \notin \text{dom}(\Gamma_2) \\
\Gamma_2(x) & \text{if } x \in \text{dom}(\Gamma_2) \text{ and } x \notin \text{dom}(\Gamma_1) \\
\end{cases}\]

Typing rules are given in Figure 2.8. Note that the type environment is split into two type environments in rule T-PAR. The rule ensures that \(P\) and \(Q\) use channels correctly and their composition also uses channels correctly. A type environment \(\Gamma\) is unlimited if all its bindings contain no linear types. T-OUT and T-REP both have the condition, because the condition ensures that type environment contains no linear bindings. Another interesting rule is T-NEW. The condition \(p = \emptyset \text{ or } \{i, o\}\) rejects expressions with non-usable local channels.

Kobayashi, Pierce and Turner’s refined type system can only determine if a given channel is used at most once during execution. It is not obvious how to extend the type system to provide more communication topology information such as the number of processes that send/receive on a given channel.
2.4.3 Nielson and Nielson’s Type and Effect System

Nielson and Nielson [NN94] presents an effect-based analysis to determine communication behavior during the execution of the CML program. The analysis gives an approximate number of processes and channels that are needed to create for a CML program.

The language is a subset of CML defined by

\[
e ::= c \mid x \mid \text{fn } x \Rightarrow e \mid e_1 e_2 \\
| \quad \text{let } x = e_1 \text{ in } e_2 \mid \text{rec } f x \Rightarrow e \\
| \quad \text{if } e \text{ then } e_1 \text{ else } e_2
\]

Here \(c\) is the meta-variable for constants including base type values and primitive operations. Types are extended to record concurrency behaviour and defined by

\[
t ::= \text{unit} \mid \text{bool} \mid \text{int} \mid \alpha \\
| \quad t_1 \rightarrow^b t_2 \mid t_1 \times t_2 \mid t \text{ list} \\
| \quad t \text{ chan } r \mid t \text{ com } b
\]

\[
b ::= \epsilon \mid r!t \mid r?t \mid t \text{ CHAN } r \mid \beta \\
| \quad \text{FORK } b \mid b_1; b_2 \mid b_1 + b_2 \mid \text{REC } \beta . b
\]

\[
r ::= r_0 \mid r_1 \mid \ldots \mid \rho
\]

Here \(b\) is a concurrency behavior and \(r\) is a region where a channel is created. The channel created in region \(r\) has type \(t \text{ chan } r\), where \(t\) is the type of values allowed to be communicated over the channel. A function type has form \(t_1 \rightarrow^b t_2\) where \(b\) is the latent behavior of the function body.

Nielson and Nielson present a typing system for the language to collect behavior in-
For simplicity, we make a small modification to the typing rules by omitting
the notion of type schemes. Figure 2.9 gives the modified typing rules where \( \square \) is an or-
dering on behaviors. The typing judgement has form \( \Gamma \vdash e : t \& b \) saying that under the
type environment \( \Gamma \) expression \( e \) has type \( t \) and its concurrency behavior is \( b \). The behavior
information is collected in a straightforward way for each rule.

Nielson and Nielson also present an operational semantics that has a sequential part
and concurrent part [NN94]. The sequential part follows a standard small-step presentation
based on evaluation contexts. (We omit its details just for now). The more interesting part is
the concurrent part. The transition relation has form \( CI \& PP \rightarrow_{ev}^{ps} CI' \& PP' \) where \( CI \)
is the set of channel identifiers, \( PP \) is a process identifiers assignment, \( ev \) is the event and
\( ps \) are the process identifiers involved in the event. An event is either sequential evaluation
denoted as \( \epsilon \), creation of channel \( ci \) denoted as \( CHAN ci \), creation of process \( pi \) denoted
as \( FORK pi \) or a communication over channel identifier \( ci \) denoted as \( (ci!, ci?) \). For each
event, there is an evaluation rule as given in Figure 2.10.
If one of $e$’s evaluation sequence is

$$\emptyset & \circ [pi_e \mapsto e] \longrightarrow^{ev_1} \ldots \longrightarrow^{ev_k} CI & PP$$

then by counting the number of process and channel identifiers in the event sequence $ev_1, \ldots, ev_k$, we will know the exact number of processes and channels created in that particular execution. If, for all executions, we have finite numbers of processes and channels, then $e$ has a finite communication topology. Nielson and Nielson also present a predicate on the process’s behaviors to determine whether it has finite communication topology, where its behavior is obtained by the above typing system.

Nielson and Nielson’s effect-based type system can be used to determine if a finite number of processes and channels are needed to create for a given CML program. It, however, is not obvious how to refine their type system to handle non-finite communication topologies. And they did not propose any applications of their analysis.
2.4.4 Session Types

Takeuchi, Honda and Kubo first introduce a notion of session type [THK94], where an interaction-based language and a typing system are presented. Session types are used to categorize the interactions between communication processes. A session is an establishment of a local channel between two communication processes. A session channel can be established by matched request and accept action between two processes. Once a session is established, values of any type can be communicated over it. It is different from $\pi$-calculus, where only values of certain type can be communicated over a channel.

Session channels capture communication structures between processes, while channels in the $\pi$-calculus capture data structures communicated between them. Honda, Vasconcelos and Kubo extend the session types with recursion and delegation [HVK98], which enriches session type’s expressibility. Recursion allows unbound interactions. In $\pi$-calculus, channels are first-class values which can be communicated over other channels. Therefore, delegation of session type is introduced to match up the mobility of processes in $\pi$-calculus. By delegation, a session channel can be shared by multiple communication entities. But at any time, a session can be only shared by two communication entities.

Gay and Hole present types and sub-types for client-server interactions, and use the type system to specify POP3 protocol [GH99]. Instead of using special syntax to establish session channels, Gay and Hole use the standard $\pi$-calculus new construct with usage annotations on session types. Gay, Vasconcelos and Ravara conduct the first study of session types in an ideal $\lambda$-calculus language without concurrency [GVR03]. Vasconcelos, Gay and Ravara extend the language with concurrency, and present a type-checking algorithm [VGR04]. Neubauer and Thiemann encode session types into Haskell and implement the Simple Mail Transfer Protocol (SMTP) in it [NT04]. Ciancaglini, Yoshida, Ahern and Drossopoulou propose a simple distributed object-oriented language with session types
[CYAD05]. However, one may notice that one big limitation of session type is that it can only capture “one to one” communication structure. Honda, Yoshida and Carbone [HYM08] have extended the binary session type to multiparty session types.

Session types are more suitable to specify and verify properties of protocols, rather than analyze programs. Session types, however, may be a useful way to represent behaviors in an analysis.

2.4.5 Colby’s Abstract Interpretation Approach

Colby [Col95] presents a static analysis to determine the communication topology of programs in a context of a subset of CML [Rep92]. The analysis uses abstract interpretation to match the communication operations. Colby calls it a non-uniform analysis; a analysis can distinguish between different iterations of a recursive communication pattern.

A central notion in Colby’s approach is a control path. For a given program $p$, each expression in $p$ is labeled with a unique program address $a$. A control path is a sequence of such program addresses, written as $a_1, \ldots, a_n$, which represents a sequential evaluation sequence from program address $a_1$ to $a_2$. Note that control paths from $a_1$ to $a_2$ may differ for different executions, but each execution has a unique control path. Thus process and channel identifiers can be replaced by a control path from program starting address to the process and channel’s creation address respectively.

The dynamic semantics for the language is split into a sequential part and a concurrent part. Colby adopts a standard evaluation-context-based small-step relation to define sequential transitions between states. A state has the form $a : \pi$ where $a$ is the current program address of the process and $\pi$ is a frame containing state’s information. For the concurrent evaluation, a computation of a program is represented as a tree. Each control path from branch to leaf in a tree corresponds to a sequential evaluation sequence of that
process, and each leaf corresponds to the current state of that process. Transitions between trees are defined by semantic rules.

Colby’s analysis computes an approximate set of states that a program can evolve into. The approximate set is computed by the least fixpoint of \( \hat{T} \) where \( \hat{T} \) is the abstract transition function mapping abstract states to abstract states. An abstract state is a relation between a branch decision made at \( a_s \) during execution, the current program address, and abstract frames. An abstract frame contains abstract information about variable bindings. The most interesting case is a channel-variable binding, which binds a channel variable to an abstract value that extracts information from the control path starting from \( a_s \) to the creation site of the channel. Only communications that are operated on the same abstract values can be matched. Thus an approximate communication topology is determined.

Colby’s analysis is the closest work to ours. His analysis is based on a semantics that uses control paths to identify threads. Unlike using spawn points to identify threads, control paths distinguish multiple threads created at the same spawn point, which is a necessary condition to understand the topology of a program. The method used to abstract control-paths is left as a “tunable” parameter in his presentation, so it is not immediately obvious how to use his approach to provide the information that is needed to determine the special communication pattern in the pipeline example in Section 1.1.3.

2.4.6 Our Previous Work

In our previous work, we present a program analysis for CML. In that work, we present a technique that tracks values of abstract type that escape from the module “into the wild” and a data-flow analysis that uses an extended control-flow graph to compute an approximation of the number of processes that send or receive messages on the channel (whether there are
more than one senders or receivers), as well as an approximation of the number of messages sent on the channel (whether there are more than one messages transmitted).

Given the extended control-flow graph and a channel, the analysis starts at the ECFG node that corresponds to the site where the channel is created. The analysis computes a finite map $\hat{P}$ that maps program points to an approximation of the abstract control paths that one follows to get to the program point.

$$\hat{P} \in \text{PATHTo} = \text{PROGPT} \xrightarrow{\text{fin}} 2^{\text{CTLPATH}}$$

where the abstract control paths $\text{CTLPATH}$ is defined by the syntax

$$\hat{\pi} ::= *:\pi \mid \pi_1:\pi_2$$

A partial ordering $\sqsubseteq$ on abstract control paths is defined as follows: $\pi_1:\pi'_1 \sqsubseteq \pi_2:\pi'_2$ if $\pi_1 = \pi_2$ and $\pi'_1 \preceq \pi'_2$. In other words, $\hat{\pi}_1 \sqsubseteq \hat{\pi}_2$ if they are in the same process and $\hat{\pi}_1$ is a prefix of $\hat{\pi}_2$.

The analysis of the ECFG is defined by a pair of mutually recursive functions:

$$\mathcal{N}_G^c : \text{NODE} \rightarrow \text{CTLPATH} \rightarrow \text{PATHTo} \rightarrow \text{PATHTo}$$

$$\mathcal{E}_G^c : \text{EDGE} \rightarrow \text{CTLPATH} \rightarrow \text{PATHTo} \rightarrow \text{PATHTo}$$

The definition of these functions can be found in Figure 2.11, where $\hat{P}_{\text{empty}} = \{ a \mapsto \emptyset \mid a \in \text{PROGPT} \}$ is the finite map that assigns the empty path set to every program point. If a known channel $c$ is defined at $a : \text{chan} c$, then we compute $\hat{P}_c = \mathcal{N}_G^c[a] \cdot c \cdot \hat{P}_{\text{empty}}$.

The $\mathcal{N}_G^c$ function is defined based on the kind of graph node. For a function entry it
\[ N_G^c[(f, \text{entry})] \hat{\pi} \hat{P} = N_G^c[a'] \hat{\pi} \hat{P} \quad \text{where} \quad \text{Succ}_G(f, \text{entry}) = \{ a' \} \]

\[ N_G^c[(f, \text{exit})] \hat{\pi} \hat{P} = \hat{P} \cup \left( \bigcup_{e \in \text{Edge}_G(a)} \mathcal{E}_G^c[e] \hat{\pi} \hat{P} \right) \]

\[ N_G^c[a] \hat{\pi} \hat{P} = \hat{P} \quad \text{if} \quad \exists \text{pid} : \pi_1 a \pi_2 \in \hat{P}(a) \quad \text{such that} \quad \text{pid} : \pi_1 a \pi_2 \subseteq \hat{\pi} \quad \text{and} \quad \text{pid} : \pi_1 \in \hat{P}(a). \]

\[ = \hat{P} \quad \text{if} \quad \text{NumProcs}(\hat{P}(a)) \geq 3 \]

\[ = \hat{P}' \cup \left( \bigcup_{e \in \text{Edge}_G(a)} \mathcal{E}_G^c[e] \hat{\pi} \hat{P}' \right) \]

\[ \text{where} \quad \hat{P}' = \hat{P} \cup \{ a \mapsto \hat{P}(a) \cup \{ \hat{\pi} \} \} \]

\[ \mathcal{E}_G^c[(a, \text{ctl}, n)] \hat{\pi} \hat{P} = N_G^c[n] \hat{\pi} a \hat{P} \]

\[ \mathcal{E}_G^c[(a, \text{spawn}, n)] \hat{\pi} \hat{P} = N_G^c[n] * : e \hat{P} \quad \text{if} \quad \hat{\pi} = * : \pi \]

\[ N_G^c[n] \pi_1 \pi_2 * : e \hat{P} \quad \text{if} \quad \hat{\pi} = \pi_1 : \pi_2 \]

\[ \mathcal{E}_G^c[(a, \text{msg}, n)] \hat{\pi} \hat{P} = N_G^c[n] * : e \hat{P}_{\text{empty}} \quad \text{if} \quad \hat{\pi} = * : \pi \]

\[ N_G^c[n] \pi_1 \pi_2 * : e \hat{P}_{\text{empty}} \quad \text{otherwise and} \quad \hat{\pi} = \pi_1 : \pi_2 \]

\[ \mathcal{E}_G^c[(a, \text{wild}, n)] \hat{\pi} \hat{P} = N_G^c[n] * : e \hat{P}_{\text{empty}} \]

Figure 2.11: Analyzing the CFG $G$ for channel $c$

follows the unique control edge to the first program point of the function, for a function exit it computes the union of the analysis for all outgoing edges. These edges will either be control edges to $f$’s call sites, when $f$ is a known function, or wild edges, when $f$ is an escaping function. For other nodes, we have three subcases. If the approximation $\hat{P}$ already contains a path $\text{pid} : \pi_1 a \pi_2$ that precedes $\hat{\pi}$ and $\text{pid} : \pi_1 \in \hat{P}(a)$, then we have looped (the loop is $a \rightarrow \pi_2 \rightarrow a$) and can stop. If the number of processes that can reach the program point $a$ is greater than two, then we stop, as we are interested in channels that have *single* senders or receivers. Otherwise, we record the visit to $a$ in $\hat{P}'$ and compute the union over the outgoing edges.
The $E_G^C$ function is defined by cases on the edge kind. When the edge is a control edge, we analyze the destination node passing the extended path $\widehat{\pi}a$. When the edge is a spawn edge, we analyze the destination node passing a new process ID paired with the empty path. For message edges, we analyze the receive site using the extended control path to send-site program point as a new process ID. This choice of process ID distinguishes the send from other sends that target the same receive sites, but in conflates multiple receive sites that are targets of the same send, which is safe since only one receive site can actually receive the message. For wild edges, we analyze the destination node using ‘*’ as the process ID. This value represents the fact than any number of threads might call the target of the wild edge with the same dynamic instance of the channel $c$.

Although the previous work is a significant step toward optimization of CML, it has the following limitations or shortcomings.

- It does not distinguish different dynamic instances of a channel. For instance, our previous work can not determine that $\text{outCh}$ and $\text{inCh}$ must be different instances of $\text{outCh}$ at line 11 for each process when analyzing the program in Figure 1.4.

- It is not precise enough. In our previous work, the analysis is over-conservative so that special communication patterns detected in programs may not be precise enough. For example, our previous analysis would identify $\text{replCh}$ in the simple example as a fan-in channel instead of a one-shot channel.

- It is not context sensitive. In our previous work, the extended control flow graph has control flow edges from function exit points to all successors of possible function call sites. When analyzing the function exit node, the analysis computes the union of the analysis for all outgoing edges. Thus the analysis is over-conservative.
CHAPTER 3
ABSTRACT TRACE ANALYSIS

In this chapter, I present an abstract trace analysis for concurrent programs. First, I give a formal definition of the dynamic semantics of SCL, which is suitable to model concurrency and specify interesting program properties. A type-sensitive control flow analysis will then be presented to compute approximate values for variables. I also describe how to construct a extended control-flow graph based on the information from control-flow analysis. At last, I present the abstract trace analysis on the extended control-flow graph.

3.1 Dynamic Semantics

In order to specify and analyze properties of programs in SCL, we need a formal description of program behaviors. There are many techniques to describe how sequential computations are evaluated: typically, operational semantics, denotational semantics and axiomatic semantics. Traditional approaches to these techniques, however, are not suitable to model concurrency, especially when we are interested in communication topologies of concurrent programs. In this section, we give a dynamic semantics for our simple concurrent language by following Colby [Col95].

For a given program $p$, we assume that each expression in $p$ is labeled with a unique program point $a \in \text{PROGPT}$. We write $a : e$ to denote that $e$ is the expression at program point $a$. Furthermore, we assume that for each $a \in \text{PROGPT}$, there is a $\bar{a} \in \text{PROGPT}$. The $\bar{a}$ labels are not used to label expressions, but serve to distinguish between parent and child threads in control paths. A control path is a finite sequence of program points:
$$\text{CTLPATH} = \text{PROGPT}^*.$$ We use $\pi$ to denote an arbitrary control path and juxtaposition to denote concatenation. We say that $\pi \leq \pi'$ if $\pi$ is a prefix of $\pi'$.

**Definition 3.1.1** A control path is a finite sequence of program points. We use $\pi$ to denote an arbitrary control path and juxtaposition to denote concatenation. A trace to program point $a$ is a control path from the program entry to $a$.

Evaluation of the sequential features of the language follows a standard small-step presentation based on evaluation contexts [FF86]. We modify the syntax of expression terms to distinguish values as follows:

$$v ::= \bullet$$
$$| \; (\text{fun} \; f \; (x) = e)$$
$$| \; k$$
$$| \; \langle v_1, v_2 \rangle$$

$$e ::= v$$
$$| \; \ldots$$

Here $k$ represents the dynamic channel values.

The unit value ($\bullet$) was already part of the syntax, but we add function values, dynamic channel values, and pairs of values. With these definitions, we can define the sequential evaluation relation $e \rightsquigarrow e'$ by the rules in Figure 3.1. An evaluation context is a expression with one hole where the next evaluation must take place. Evaluation contexts are defined...
let $x = v$ in $e \leadsto e[x \mapsto v]$

if $true$ then $e_1$ else $e_2 \leadsto e_1$

if $false$ then $e_1$ else $e_2 \leadsto e_2$

$\text{fun } f(x) = e_1 \text{ in } e_2 \leadsto e_2[f \mapsto (\text{fun } f(x) = e_1)]$

$(\text{fun } f(x) = e) \; v \leadsto e[f \mapsto (\text{fun } f(x) = e), x \mapsto v]$

$\#i \langle v_1, v_2 \rangle \leadsto v_i$

Figure 3.1: Sequential evaluation

in the standard call-by-value way:

$$E ::= []$$

$$| \text{let } x = E \text{ in } e$$

$$| \text{if } E \text{ then } e \text{ else } e$$

$$| E \; e \mid v \; E$$

$$| \text{send}(E, e) \mid \text{send}(v, E) \mid \text{recv} \; E$$

$$| \langle E, e \rangle \mid \langle v, E \rangle \mid \#i \; E$$

We use these below in the definition of concurrent evaluation.

Following Colby [Col95], the dynamic semantics for our language tracks execution history on a per-process basis. The state of a computation is represented as a tree, where the nodes of the tree are labeled with expressions representing process states and edges are labeled with the program point corresponding to the evaluation step taken from the parent to the child. Branches in the tree represent process creations. The execution history of a process is represented as a path from its creation node to the leaf representing the current process state, while the path from the root to the process creation node is used to denote process identity. By this tree representation, control paths can also be used to uniquely label dynamic instances of channels, which we write $c@\pi$, where $c \in \text{CHANID}$. We also
use \( k \) to denote dynamic channel values, and \( K \) to denote all the dynamic channel values.

Because this tree representation captures the history of the computation as well as its current state, we call it a *trace tree*. Nodes in a trace tree are uniquely named by control paths that describe the path from the root to the node. In defining trace trees, it is useful to view them as prefix-closed finite functions from control paths to expressions. If \( t \) is a trace tree, then we write \( t.\pi \) to denote the node one reaches by following \( \pi \) from the root, and if \( t.\pi \) is a leaf of \( t \), \( a \) is a program point of a redex of \( t.\pi \), and \( e \) an expression, then \( t \cup \{ \pi a \mapsto e \} \) is the new trace tree with a child \( e \) added to node \( t.\pi \) with the new edge labeled by \( a \). For a program \( p \), the initial trace tree will be the map \( \{ \epsilon \mapsto p \} \), where \( \epsilon \) is the empty control path.

To record the communication history between the dynamic send sites and receive sites, we define the communication history set \( H \) as follows:

\[
H \subset \{(\pi_1, k, \pi_2) \mid \pi_1, \pi_2 \in \text{CTLPath}, k \in K\}
\]

where \( (\pi_1, k, \pi_2) \in H \) if there is communication between the dynamic receive site \( \pi_1 \) and send site \( \pi_2 \) on channel instance \( k \).

We define concurrent evaluation as the smallest relation \( (\Rightarrow) \) satisfying the following four rules. The first rule lifts sequential evaluation to traces.

\[
\frac{t.\pi = E[a : e] \text{ is a leaf} \quad e \leadsto e'}{(t, H) \Rightarrow (t \cup \{ \pi a \mapsto E[e'] \}, H)}
\]

The second rule addresses channel creation.

\[
\frac{t.\pi = E[a : \text{chan}ce] \text{ is a leaf} \quad \pi_1, \pi_2 \in \text{CTLPath}, k \in K}{(t, H) \Rightarrow (t \cup \{ \pi a \mapsto E[e[c \mapsto c \circ \pi a]] \}, H)}
\]
The third rule addresses process creation.

\[
\frac{t.\pi = E[a : \text{spawn } e]}{(t, H) \Rightarrow (t \cup \{\pi a \mapsto E[\bullet], \pi \bar{a} \mapsto e\}, H)}
\]

The last rule addresses communication.

\[
\frac{t.\pi_1 = E_1[a_1 : \text{recv } k]}{t.\pi_2 = E_2[a_2 : \text{send}(k, v)]}
\]

\[
\frac{(t, H) \Rightarrow (t \cup \{\pi_1 a_1 \mapsto E_1[v], \pi_2 a_2 \mapsto E_2[\bullet]\}, H \cup \{(\pi_1, k, \pi_2)\})}{(t, H) \Rightarrow (t \cup \{\pi_1 a_1 \mapsto E_1[v], \pi_2 a_2 \mapsto E_2[\bullet]\}, H \cup \{(\pi_1, k, \pi_2)\})}
\]

The set of traces of a program represents all possible executions of the program. It is defined as

\[
\text{Trace}(p) = \{t \mid (\{\epsilon \mapsto p\}, \epsilon) \Rightarrow^* (t, H)\}
\]

### 3.1.1 Properties of Traces

Let \( p \) be a program and let \( c \) be a channel identifier in \( p \). For any trace \( t \in \text{Trace}(p) \) and \( k = c \circ \pi \) occurring in \( t \), we define the dynamic send and receive sites of \( k \) as follows:

\[
\begin{align*}
\text{Sends}_t(k) &= \{\pi \mid t.\pi = E[\text{send}(k, v)]\} \\
\text{Recvs}_t(k) &= \{\pi \mid t.\pi = E[\text{recv } k]\}
\end{align*}
\]

We say that \( c \) has the single-sender property if for any \( t \in \text{Trace}(p), k = c \circ \pi \) occurring in \( t \), and \( \pi_1, \pi_2 \in \text{Sends}_t(k) \), either \( \pi_1 \preceq \pi_2 \) or \( \pi_2 \preceq \pi_1 \). The intuition here is that if \( \pi_1 \preceq \pi_2 \) then \( \pi_1 \) is before \( \pi_2 \) and the sends can not be concurrent. On the other hand, if \( \pi_1 \) and \( \pi_2 \) are not related by \( \preceq \), then they may be concurrent.\(^1\) Note that the single-sender

\(^1\) There may be other causal dependencies, such as synchronizations, that would order \( \pi_1 \) and \( \pi_2 \), but our model does not take these into account.
property allows multiple processes to send messages on a given channel, they are just not
allowed to do it concurrently. Likewise, we say that \( c \) has the \textit{single-receiver} property if
for any \( t \in \text{Trace}(p) \), \( k = c \circ \pi \) occurring in \( t \), and \( \pi_1, \pi_2 \in \text{Recvs}_t(k) \), either \( \pi_1 \preceq \pi_2 \) or \( \pi_2 \preceq \pi_1 \).

We can now state the special channel topologies from Section 4.1.1 as properties of the
set of traces of a program. For a channel identifier \( c \) in a program \( p \), we can classify its
topology as follows:

- The channel \( c \) is a \textit{one-shot} channel if for any \( t \in \text{Trace}(p) \) and \( k = c \circ \pi \) occurring
  in \( t \), \( |\text{Sends}_t(k)| \leq 1 \).
- The channel \( c \) is \textit{point-to-point} if it has both the single-sender and single-receiver
  properties.
- The channel \( c \) is a \textit{fan-out} channel if it has the single-sender property, but not the
  single-receiver.
- The channel \( c \) is a \textit{fan-in} channel if it has the single-receiver property, but not the
  single-sender.

3.2 Abstract Trace Analysis Overview

The first step in our analysis is a type-sensitive control-flow analysis. This analysis is
based on Serrano’s 0-CFA algorithm. Our analysis computes a mapping from variables to
approximate values. We are particularly interested in the mapping for channel variables.

The second step in our analysis is to construct an extended control-flow graph with the
information from the first step. There is a node in the graph for each program point; in
addition, there is an entry and exit node for each function definition. A node with a label \( a \)
corresponds to the point in the program’s execution where the next redux is labeled with \( a \).
The graph has three kinds of edges. The first two of these represent control flow, while the other one is used to trace the flow of channel values.

1. **Control edges** represent normal sequential control-flow.

2. **Spawn edges** represent process creation. If there is an expression \( a_1 : \text{spawn } f \) and \( a_2 \) is the label of the entry of \( f \), then there will be a spawn edge from \( a_1 \) to \( a_2 \).

3. **Return edges** represent function return. If there is an expression \( a : e_1 \ e_2 \), then there will be return node \( a_r \) for \( e_1 \), and for any possible known function could be called at \( a \), there will be a return edge from the function exit point to \( a_r \).

4. **Message edges** are added from send sites to receiver sites for known channels.

The next step in our analysis is to compute abstract traces for each program points.

### 3.3 Type-Sensitive Control-Flow Analysis for CML

The first step in our analysis is a twist on *type-sensitive* control-flow analysis (CFA) [Rep05], which is based on Serrano’s 0-CFA algorithm [Ser95], but has the additional property that it exploits first-class message passing. The full details of our algorithm can be found in Section 6.1; here we cover main ideas of 0-CFA and those aspects of the analysis that are unique to our situation.

#### 3.3.1 Introduction of 0-CFA

Traditional compiler optimization techniques require a knowledge of the control flow of programs. Control-flow analysis serves to construct such control-flow graph for programs. Higher-order programming languages (HOL) such as Scheme and ML, allow programs to
take functions as first class values, where functions can be passed as arguments to other functions and returned as results from functions calls. Therefore, for HOL, control-flow and data-flow interdepend on each other; the control-flow can not be be determined from the program text at compile time.

This fact makes optimization for higher-order languages harder than for first-order languages. Shivers presents a control-flow analysis for Scheme Shi88, called 0-CFA. Shivers’s algorithm uses continuation-passing style (CPS) as intermediate language. By CPS representation, all control transfers are represented by tail-recursive function calls. Thus control-flow graph construction reduces to determing the set of all functions that could be called from each call site. Note that in Shivers’s analysis, a function is represented as a lambda/contour. His analysis computes a approximation of that set by using a abstract interpretation. The analysis is called zeroth-order control-flow analysis, because the approximation identifies all functions that have the same lambda expressions. Although this approximation may introduce more control-flow edges than exist at runtime, it is safe; that is, any control-flow edge at runtime is included in the control-flow graph. Serrano’s 0-CFA adapts Shivers’ analysis to deal with the full Scheme language, which is direct style than CPS. The analysis also statically computes an approximation of the set of functions that could be called from each call site. Our analysis described below is based on Serrano’s algorithm.
3.3.2 Abstract Values

Our analysis computes a mapping from variables to approximate values, which are given by the following grammar:

\[
v ::= \perp \mid \bullet \mid \langle v_1, v_2 \rangle \mid F \mid C \mid \text{chan} \tau \rangle \mid \tau_1 \rightarrow \tau_2 \mid \top
\]

where \( F \in 2^{\text{FUNID}} \) and \( C \in 2^{\text{CHANID}} \). We use \( \perp \) to denote undefined or not yet computed values, \( \langle v_1, v_2 \rangle \) for an approximate pair, \( F \) for a set of known functions, and \( C \) for a set of known channels. Our analysis will only compute sets of functions \( F \) and sets of channels where all the members have the same type (see [Rep05] for a proof of this property) and so we extend our type-annotation syntax to include such sets. In addition to the single top value found in most presentations of CFA, we have a family of top values \( \hat{\tau} \) indexed by type. The value \( \hat{\tau} \) represents an unknown value of type \( \tau \) (where \( \tau \) is either a function or abstract type). The auxiliary function \( U : \text{TYPE} \rightarrow \text{VALUE} \) maps types to their corresponding top value:

\[
U(\text{unit}) = \bullet \\
U(\tau_1 \rightarrow \tau_2) = \tau_1 \rightarrow \tau_2 \\
U(\tau_1 \times \tau_2) = \langle U(\tau_1), U(\tau_2) \rangle
\]

Lastly, the \( \top \) value is used to curtail expansion of recursive types as described below.

We define the join of two approximate values as in Figure 3.2 Note that this operation is not total, but it is defined for any two approximate values of the same type and we show in [Rep05] that it preserves types. One technical complication is that we need to keep our
\[ \bot \lor v = v \\
v \lor \bot = v \\
\bullet \lor \bullet = \bullet \\
\langle v_1, v_2 \rangle \lor \langle v'_1, v'_2 \rangle = \langle v_1 \lor v'_1, v_2 \lor v'_2 \rangle \\
F \lor F' = F \cup F' \\
C \lor C' = C \cup C' \\
\top \lor v = \top \\
v \lor \top = \top \\
\hat{\tau} \lor v = \hat{\tau} \\
v \lor \hat{\tau} = \hat{\tau} \]

Figure 3.2: The join of two abstract values

approximate values finite; we discuss this issue in the appendix.

3.3.3 Type-Sensitive CFA

Our analysis algorithm computes a 3-tuple of approximations: \( A = (V, C, R) \), where

\[ V \in \text{VAR} \rightarrow \text{VALUE} \quad \text{variable approximation} \]
\[ C \in \text{CHANID} \rightarrow \text{VALUE} \quad \text{channel message approximation} \]
\[ R \in \text{FUNID} \rightarrow \text{VALUE} \quad \text{function-result approximation} \]

Our \( V \) approximation corresponds to Serrano’s \( A \). The \( C \) approximation is an approximation of the messages sent on a given known channel. The \( R \) approximation records an approximation of function results for each known function; this approximation is used in lieu of analyzing a function’s body when the function is already being analyzed and is needed to guarantee termination.

Our algorithm follows the same basic structure as that of Serrano[Ser95]. The major difference between ours and Serrano’s algorithm is that our language has channels. Send operations on channels are treated much the same way as function calls. If the approximation of the first argument to a send is \( C \) and the second argument is \( v \), then we add \( v \).
SendSites(c) = \{ a | a : \texttt{send}(e_1, e_2) \in p \wedge c \in A(e_1) \} \\
RecvSites(c) = \{ a | a : \texttt{recv} e \in p \wedge c \in A(e) \}

Figure 3.3: Approximation of channel send and receive sites

to the approximation of message values sent on each channel \( c \in C \). We use \( C \) to track this information. The message-receive operation is treated much like a function entry, the possible values are taken from the \( C \) approximation.

3.3.4 Properties

The analysis presented in the previous section allows one to compute certain static approximations of the dynamic properties described in Section 3.1.1. Figure 3.3 gives the approximation of the send and receive sites for a given channel.

3.4 The Extended CFG

With the information from the CFA in hand, the next step of our analysis is to construct an extended control-flow graph (CFG) for the program that we are analyzing. We then use this extended CFG to compute approximate trace fragments that can be used to analyze the properties of the program.

There is a node in the graph for each program point; in addition, there is an entry and exit node for each function definition and there is a return node for each function call. We use RETNODE to denote the set of all return nodes for function calls. A node with a label \( a \) corresponds to the point in the program’s execution where the next redux is labeled with \( a \). The graph has four kinds of edges. The first three of these represent control flow, while the last one is used to trace the flow of channel values.
1. **Control edges** represent normal sequential control-flow.

2. **Spawn edges** represent process creation. If there is an expression $a_1 : \text{spawn } e$ and $a_2$ is the label of the first redux in $e$, then there will be a spawn edge from $a_1$ to $a_2$.

3. **Return edges** represent function return. If there is an expression $a : e_1 e_2$, then there will be return node $a_r$ for $e_1$, and for any possible known function could be called at $a$, there will be a return edge from the function exit point to $a_r$.

4. **Message edges** are added from send sites to known receiver sites.

The graph is constructed such that following a control edge from $a_1$ to $a_2$ corresponds to an edge labeled with $a_1$ in a trace that leads to the trace node labeled by $a_2$. Similarly, following a spawn edge from $a_1$ to $a_2$ corresponds to $\bar{a}_1$ in a trace. More formally, the sets of nodes and edges are defined to be

$$
\begin{align*}
    n \in \text{NODE} & = \text{PROGPT} \cup (\text{FUNID} \times \{\text{entry}, \text{exit}\}) \cup \text{RETNODE} \\
    \text{EGLABEL} & = \{\text{ctl}, \text{spawn}, \text{ret}, \text{msg}\} \\
    \text{EDGE} & = \text{NODE} \times \text{EGLABEL} \times \text{NODE} \\
    G \in \text{GRAPH} & = 2^{\text{NODE}} \times 2^{\text{EDGE}}
\end{align*}
$$

Constructing the CFG is done in three steps. First we create the basic graph with control and spawn edges in the standard way. One important point is that we use the results of the CFA to determine the edges from call sites to known functions and the edges from function exit points to return nodes. Note that because we are only interested in tracking known channels, we can ignore calls to unknown functions when constructing the graph. Message edges are added in much the same way as control edges for known function calls. Let $a : \text{send}(e_1, e_2)$ be a send in the program and assume that the CFA computed $C$ as the
approximation of $e_1$. Then for each channel $c \in C$ and $a' \in \text{RecvSites}(c)$, we add a send edge from $a$ to $a'$ to the graph.

3.5 Abstract Trace Analysis

We wish to statically compute properties of all possible behaviors of programs. In other words, we wish to compute properties that hold in any possible trace tree $t \in \text{Trace}(p)$.

As described in Chapter 2, there are many approaches to static analysis of concurrent programs. Here, we present a new approach of abstract interpretation to statically compute properties of concurrent programs. We call this analysis abstract trace analysis.

The first step is to define collecting semantics, which collects the set of all possible traces that might reach any given program point. The trace set of a given program point $a$ is denoted as $\text{Coll}_{\text{out}}(a)$. $\text{Coll}_{\text{out}}(a)$ can be trivially collected as follows:

$$
\text{Coll}_{\text{in}}(a) = \begin{cases}
\{\epsilon\} & \text{if } a \in \text{Program Entry} \\
\cup\{\text{Coll}_{\text{out}}(a') \mid a' \in \text{pred}(a)\} & \text{otherwise}
\end{cases}
$$

$$
\text{Coll}_{\text{out}}(a) = \{\pi \cdot a \mid \pi \in \text{Coll}_{\text{in}}(a)\}
$$

These equations are simple. But the information collected is not precise in the presence of procedures or functions. There is no guarantee that when information flows into functions, it will flow back to the corresponding return node. Some invalid traces to $a$ are collected into $\text{Coll}_{\text{out}}(a)$. For example, for a pair of call node $a_c$ and return node $a_r$, the equations compute traces flowing from $a_c$ to the entry of all possible functions called at $a_c$, but traces might flow to other return nodes instead of $a_r$ when exiting functions.

**Definition 3.5.1** We say that $\pi$ is a valid trace w.r.t. return node $a$, if $\pi \in \text{Coll}_{\text{out}}(a')$ where $a' \in \text{pred}(a)$ and $\exists \pi_1, \pi_2$ s.t. $\pi = \pi_1 \cdot \pi_2$ where $\pi_1 \in \text{Coll}_{\text{out}}(a'')$, $a''$ is the
corresponding call node of \( a \) and \( \pi_2 \) contains the same number of call nodes and return nodes.

In order to have more precise results, we exclude the invalid traces as follows:

\[
Coll_{in}(a) = \begin{cases} 
\{\epsilon\} & \text{if } a \in \text{Program Entry} \\
\text{valid}_a(\cup\{Coll_{out}(a') \mid a' \in \text{pred}(a)\}) & \text{if } a \in \text{RETNode} \\
\cup\{Coll_{out}(a') \mid a' \in \text{pred}(a)\} & \text{otherwise}
\end{cases}
\]

\[
Coll_{out}(a) = \{\pi \cdot a \mid \pi \in Coll_{in}(a)\}
\]

Here \( \text{valid}_a \) is used to exclude invalid traces w.r.t. return node \( a \). Now \( Coll_{out}(a) \) is precise. The least fixed point, however, is not decidable for these equations. Thus, we need to design an abstraction function to compute a safe approximation of these sets. When designing this abstraction function, we do not want to lose too much precision. Here, we give a precise abstraction function.

**Definition 3.5.2** We say that \( \pi \) is a redundant trace w.r.t. function entry node \( a \), if \( \pi \in A_{out}(a') \) where \( a' \in \text{pred}(a) \) and \( \exists \pi_1, \pi_2, \pi_3, \pi_4 \) s.t. \( \pi = \pi_1 \cdot a \cdot \pi_2 \cdot a \cdot \pi_3 \cdot a \cdot \pi_4 \) where \( \pi_1 \in A_{in}a \), \( a' \notin \pi_2 \), \( a' \notin \pi_3 \), \( a' \notin \pi_4 \) and \( a' \) is the corresponding function exit point of \( a \).

The abstract trace analysis collects the abstract trace set as follows:

\[
A_{in}(a) = \begin{cases} 
\{\epsilon\} & \text{if } a \in \text{Program Entry} \\
\text{valid}_a(\cup\{A_{out}(a') \mid a' \in \text{pred}(a)\}) & \text{if } a \in \text{RETNode} \\
\cup\{A_{out}(a') \mid a' \in \text{pred}(a)\} & \text{otherwise}
\end{cases}
\]

\[
A_{out}(a) = \{\pi \cdot a \mid \pi \in A_{in}(a)\}
\]
\( a_1: \) let chan ch
\( a_2: \) fun server () = ( 
\( a_3: \) let replCh’ = recv ch in
\( a_4: \) send (replCh’, 1);
\( a_5: \) server ()
\( a_6: \) fun client () = ( 
\( a_7: \) let chan replCh in
\( a_8: \) send (ch, replCh);
\( a_9: \) recv replCh)
in
\( a_{10}: \) (spawn (a_{11}: server );
\( a_{12}: \) spawn (a_{13}: server );
\( a_{14}: \) spawn (a_{15}: client ))

Figure 3.4: Simple service

where \( \bigcup_a \) excludes redundant traces w.r.t. \( a \).

\[
S_1 \uplus_a S_2 = (S_1 \cup S_2) \setminus \{\pi \mid \pi \text{ is redundant w.r.t. } a\}
\]

We use a simple service example in Figure 3.4 to illustrate how the abstract trace flow analysis works. The service is created by allocating a new channel and spawning two server threads to handle requests on the channel. The representation of the service is the request channel. The \texttt{client} function creates a fresh reply channel and sends it along the request to the service. One of the two server threads handles the request and send the result back to the client on the reply channel.
Our analysis produces the following information for communication sites:

\[
A_{out}(a_3) = \{ \pi_1 a_2 a_3, \, \pi_1 a_2 \pi_2 a_2 a_3, \, \pi_1 a_2 \pi_2 a_2 a_3 \pi_2 a_2 a_3 \\
\pi_1' a_2 a_3, \, \pi_1' a_2 \pi_2 a_2 a_3, \, \pi_1' a_2 \pi_2 a_2 a_3 \pi_2 a_2 a_3 \} \\
A_{out}(a_4) = \{ \pi_1 a_2 a_3 a_4, \, \pi_1 a_2 \pi_2 a_2 a_3 a_4, \, \pi_1 a_2 \pi_2 a_2 a_3 a_4 \pi_2 a_2 a_3 a_4 \\
\pi_1' a_2 a_3 a_4, \, \pi_1' a_2 \pi_2 a_2 a_3 a_4, \, \pi_1' a_2 \pi_2 a_2 a_3 a_4 \pi_2 a_2 a_3 a_4 \} \\
A_{out}(a_8) = \{ \pi a_8 \} \\
A_{out}(a_9) = \{ \pi a_8 a_9 \}
\]

where \( \pi = a_1 a_{10} a_{12} a_{14} a_{15} a_{6} a_{7}, \ \pi_1 = a_1 a_{10} a_{11}, \ \pi_1' = a_1 a_{10} a_{12} a_{13} \) and \( \pi_2 = a_3 a_4 a_5 \).

The only dynamic instance of \( replCh \) is represented by \( \pi \), while the only dynamic instance of \( ch \) is represented by \( a_1 \).

With abstract trace information at hand, we have a foundation to compute other program properties. We will discuss two applications of abstract trace analyses in the following chapter.
CHAPTER 4
APPLICATIONS OF ANALYSIS

In this chapter, we present two applications of abstract trace analysis. In one application, we compute approximate communication topology of concurrent programs with abstract trace information available. By taking advantage of replacing general communication primitives with specialized communication primitives in certain communication pattern, we can optimize programs while ensuring safety of transformations. In the other application, we compute partial ordering among statements of concurrent programs. Partial ordering information give us opportunity to smartly schedule threads to execute on multiprocessors to achieve performance improvements. In this chapter, we also describe the monitor optimization, which automatically recognizes the monitor communication pattern, and transforms the monitor threads into monitor functions and communications with the monitor threads into function calls.

4.1 Detecting Special Communication Pattern

The underlying protocols used to implement CML’s communication and synchronization primitives (e.g.channels) are necessarily general, since they must function correctly and fairly in arbitrary contexts. In practice, most uses of these primitives fall into one of a number of common patterns that may be amenable to more efficient implementation. As is often the case, the difficulty of this optimization technique is developing an effective, but efficient, analysis that identifies when it is safe to specialize.
In this section, we explain how specific communication topologies can lead to more efficient implementation and discuss the problem of determining such topologies via abstract trace analysis.

### 4.1.1 Specialized Channel Operations

In general, a CML channel must support communication involving multiple sending and receiving processes transmitting multiple messages in arbitrary contexts. This generality requires a complicated protocol to implement with commiserate overhead.\(^1\) Because of this generality, the protocol used to implement channel communication involves locking overhead. In practice, however, many (if not most) channels are used in restricted ways, such as for point-to-point and single-message communication. Assuming that the basic communication primitive is a buffered channel, then we consider the following possible communication topologies:

<table>
<thead>
<tr>
<th>number of</th>
<th>topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>senders</td>
<td>receivers</td>
</tr>
<tr>
<td>≤1</td>
<td>≤1</td>
</tr>
<tr>
<td>≤1</td>
<td>≤1</td>
</tr>
<tr>
<td>≤1</td>
<td>&gt;1</td>
</tr>
<tr>
<td>&gt;1</td>
<td>≤1</td>
</tr>
<tr>
<td>&gt;1</td>
<td>&gt;1</td>
</tr>
</tbody>
</table>

In this table, the notation \(>1\) denotes the possibility that more than one thread or message may be involved and the notation \(≤1\) denotes that at most one thread or message is involved. For example, a point-to-point topology involves arbitrary numbers of messages, but at most

---

1. Chapter 10 of *Concurrent Programming in ML* describes CML’s implementation, while Knabe has described a similar protocol in a distributed setting [Kna92].
one sender and receiver. An analysis is safe if whenever it approximates the number of messages of threads as $\leq 1$, then that property holds for all possible executions. It is always safe to return an approximation of $>1$.

We believe that specialized implementations of channel operations (and possibly channel representations) can have a significant impact on communication overhead. For example, CML provides I-variables, which are a form of synchronous memory that supports write-once semantics [ANP89]. Using I-variables in place of channels for one-shot communications can reduce synchronization and communication costs by 35% [Rep99]. Demaine [Dem98] proposes a dead-lock-free protocol for the efficient implementation of a generalized alternative construct, where fan-out and fan-in channel operations can be implemented with fewer message cycles per user-level communication than many-to-many channel operations. In Section 6.3.3, we also present an efficient lock-free implementation of communication primitives for fan-out and fan-in channels.

While programmers could apply these optimizations by hand, doing so would complicate the programming model and lead to less reliable software. Furthermore, correctness of the protocol depends on the properties of the chosen primitives. Changes to the protocol may require changes in the choice of primitives, which makes the protocol harder to maintain. For these reasons, we believe that an automatic optimization technique based on program analysis and compiler transformations is necessary.

### 4.1.2 An Example

To illustrate how the analysis and optimization might proceed, consider the simple service implemented in Figure 4.1.\footnote{To keep the example concise, we use direct operations on channels instead of CML’s event operations, but the analysis handles event values without difficulty.}
signature SIMPLE_SERV =
  sig
    type serv
    val new : unit -> serv
    val call : (serv * int) -> int
  end

structure SimpleServ :> SIMPLE_SERV =
  struct
    datatype serv = S of (int * int chan) chan

    fun new () = let
      val ch = channel()
    fun server v = let
      val (req, replCh) = recv ch
      in
        send(replCh, v);
        server req
      end
      in
        spawn (server 0);
        S ch
    end

    fun call (S ch, v) = let
      val replCh = channel()
      in
        send (ch, (v, replCh));
        recv replCh
      end
    end

Figure 4.1: A simple service with an abstract client-server protocol
The `new` function creates a new instance of the service by allocating a new channel and spawning a new server thread to handle requests on the channel. The representation of the service is the request channel, but it is presented as an abstract type. The `call` function sends a request to a given instance of the service. The request message consists of the request and a fresh channel for the reply. Because the connection to the service is represented as an abstract type, we know that even though it escapes out of the `SimpleServ` module, it cannot be directly accessed by unknown code. Figure 4.2 illustrates the data-flow of the service’s request channel. Specifically, we observe the following facts:

- For a given instance of the service, the request channel has a many-to-one (or fan-in) communication pattern.

- For a given client request, the reply channel is used at most once and has a one-to-one (or one-shot) communication pattern.

We can exploit these facts to specialize the communication operations, which result in the
structure SimpleServ :> SIMPLE_SERV =
struct
  datatype serv
  = S of (int * int OneShot.chan) FanIn.chan

  fun new () = let
    val ch = FanIn.channel()
  fun server v = let
    val (req, replCh) = FanIn.recv ch
    in
      OneShot.send(replCh, v);
      server req
    end
  in
    spawn (server 0);
    S ch
  end

  fun call (S ch, v) = let
    val replCh = OneShot.channel()
    in
      FanIn.send (ch, (v, replCh));
      OneShot.recv replCh
    end
end

Figure 4.3: A version of Figure 4.1 with specialized communication operations

optimized version of the service shown in Figure 4.3. We have highlighted the specialized code and have assumed the existence of a module FanIn that implements channels specialized for the many-to-one pattern and a module OneShot that is specialized for one-shot channels.

4.1.3 Communication Topology Analysis

In this section, we discuss how to use abstract trace information to compute approximate communication topology of concurrent programs.
**Definition 4.1.1** We say program point $a$ is a syntactic use site of channel variable $c$, if and only if $c$ is passed as arguments, sent as values or used to communicate at program point $a$. Set $U(c)$ is used to represent all use sites of $c$.

**Definition 4.1.2** There are three types of use for channel variables: argument, send, communication. Function $\text{UseType}: Ch \times ppt \rightarrow \{\text{arg, send, comm}\}$ maps a channel and its use site to its corresponding use types.

**Definition 4.1.3** We say program point $a$ is a definition site of channel variable $c$, if and only if channel $c$ is created, received as a value or declared as a parameter at program point $a$. Set $D(c)$ is used to represent all definition sites of $c$. We say a trace $\pi \cdot a$ defines $c$, if $a \in D(c)$ and $\pi \cdot a$ is a trace to program point $a$.

**Definition 4.1.4** There are three types of definition for channel variables: new, recv, para. Function $\text{DefType}: Ch \times ppt \rightarrow \{\text{new, recv, para}\}$ maps a channel and its definition site to its corresponding definition types.

We turn to our running example in Figure 3.4 to illustrate the above definitions. Recall that we have three channels: $ch$, which is created at $a_1$; $replCh$, which is created at $a_7$; and $replCh'$ which is bounded at $a_3$. By definition, we have

$$D(ch) = \{a_1\}$$
$$U(ch) = \{a_3, a_8\}$$
$$D(replCh) = \{a_7\}$$
$$U(replCh) = \{a_8, a_9\}$$
\[ D(\text{replCh}') = \{a_3\} \]
\[ U(\text{replCh}') = \{a_4\} \]
\[ \text{DefType}(ch, a_1) = \text{new} \]
\[ \text{UseType}(ch, a_3) = \text{comm} \]
\[ \text{UseType}(ch, a_8) = \text{comm} \]
\[ \text{DefType}(\text{replCh}, a_7) = \text{new} \]
\[ \text{UseType}(\text{replCh}, a_8) = \text{send} \]
\[ \text{UseType}(\text{replCh}, a_9) = \text{comm} \]
\[ \text{DefType}(\text{replCh}', a_3) = \text{recv} \]
\[ \text{UseType}(\text{replCh}', a_4) = \text{comm} \]

**Definition 4.1.5** For a channel variable \( c \) and a trace \( \pi = \pi_1 \cdot d \cdot \pi_2 \cdot a \) such that \( a \in U(c) \), \( d \in D(c) \), and \( \forall b \in D(c), b \notin \pi_2 \), we define \( \text{def}(c, \pi) = \pi_1 \cdot d \) and \( \text{use}(c, \pi) = \pi_2 \cdot a \). Otherwise, \( \text{def}(c, \pi) \) and \( \text{use}(c, \pi) \) are undefined.

Informally, \( \text{def}(c, \pi) \) produces the longest prefix of \( \pi \) that defines \( c \) and \( \text{use}(c, \pi) \) produces the rest of \( \pi \). We turn to our running example to illustrate how \( \text{def} \) and \( \text{use} \) work. Recall that we have three channels: \( ch, \text{replCh} \) and \( \text{replCh}' \). Their use sites are \( a_3, a_8, a_8 \) and \( a_4 \) respectively. The definition traces for channel \( ch \) and each trace reaching its use site \( a_3 \) are as follows:

\[ \text{def}(ch, \pi_1 a_2 a_3) = a_1 \]
\[ \text{use}(ch, \pi_1 a_2 a_3) = a_10 a_11 a_2 a_3 \]
\[
\begin{align*}
def(ch, \pi_1a_2\pi_2a_3) &= a_1 \\
use(ch, \pi_1a_2\pi_2a_3) &= a_{10}a_{11}a_2\pi_2a_3 \\
def(ch, \pi_1'a_2a_3) &= a_1 \\
use(ch, \pi_1'a_2a_3) &= a_{10}a_{12}a_{13}a_2a_3 \\
def(ch, \pi_1'a_2\pi_2a_3a_4) &= a_1 \\
use(ch, \pi_1'a_2\pi_2a_3a_4) &= a_{10}a_{12}a_{13}\pi_2a_2a_3 \\
\end{align*}
\]

The definition traces for channel \(\text{replCh}'\) and each trace that reaches its use site \(a_4\) are as follows:

\[
\begin{align*}
def(\text{replCh}', \pi_1a_2a_3a_4) &= \pi_1a_2a_3 \\
use(\text{replCh}', \pi_1a_2a_3a_4) &= a_4 \\
def(\text{replCh}', \pi_1a_2\pi_2a_2a_3a_4) &= \pi_1a_2\pi_2a_2a_3 \\
use(\text{replCh}', \pi_1a_2\pi_2a_2a_3a_4) &= a_4 \\
def(\text{replCh}', \pi_1'a_2a_3a_4) &= \pi_1'a_2a_3 \\
use(\text{replCh}', \pi_1'a_2a_3a_4) &= a_4 \\
def(\text{replCh}', \pi_1'a_2\pi_2a_2a_3a_4) &= \pi_1'a_2\pi_2a_2a_3 \\
use(\text{replCh}', \pi_1'a_2\pi_2a_2a_3a_4) &= a_4 \\
\end{align*}
\]

Once we have definition trace along \(\pi a\) for channel variable \(c\), we can use it to represent the dynamic value of \(c\) that is used at \(a\) along \(\pi\). If the dynamic value is assigned to \(c\) at \(d\) by receiving on some other channel \(c'\) or by creating a new instance along \(\pi\), we just use the definition trace \(\text{def}(c, \pi a)\) to represent the dynamic value of \(c\). If the dynamic value is
passed from $c'$ to $c$ as an argument bound to parameter along $\pi$, then we use the dynamic value of $c'$ along $\pi$ to represent the dynamic value of $c$ along $\pi$. The formal definition is as follows:

**Definition 4.1.6** For a channel variable $c$ and a trace $\pi$, $\text{Dyn}(c, \pi)$ is a partial function that maps the channel $c$ and a trace $\pi$ to the corresponding dynamic instance of channel $c$ along trace $\pi$. Let $\text{def}(c, \pi) = \pi_1 \cdot a \cdot d$,

$$
\text{Dyn}(c, \pi) = \begin{cases} 
\pi_1 \cdot a \cdot d & \text{if } \text{DefType}(c, d) = \text{new/recv} \\
\text{Dyn}(c', \pi_1 \cdot a) & \text{otherwise, } c' \text{ is bound to } c \text{ at } a
\end{cases}
$$

Again, we turn to our running example to illustrate how $\text{Dyn}$ works. Recall that we have three channels: $ch$, $\text{replCh}$ and $\text{replCh}'$. Their definition sites are $a_1$, $a_7$ and $a_3$ respectively. The definition types are either new or recv. So dynamic values of all three channels along all traces are represented by their definition traces as follows:

$$
\text{Dyn}(ch, \pi_1 a_2 a_3) = a_1 \\
\text{Dyn}(ch, \pi_1 a_2 \pi_2 a_2 a_3) = a_1 \\
\text{Dyn}(ch, \pi'_1 a_2 a_3) = a_1 \\
\text{Dyn}(ch, \pi'_1 a_2 \pi_2 a_2 a_3) = a_1 \\
\text{Dyn}(ch, \pi a_8) = a_1 \\
\text{Dyn}(\text{replCh}', \pi_1 a_2 a_3 a_4) = \pi_1 a_2 a_3 \\
\text{Dyn}(\text{replCh}', \pi_1 a_2 \pi_2 a_2 a_3 a_4) = \pi_1 a_2 \pi_2 a_2 a_3 \\
\text{Dyn}(\text{replCh}', \pi'_1 a_2 a_3 a_4) = \pi'_1 a_2 a_3 \\
\text{Dyn}(\text{replCh}', \pi'_1 a_2 \pi_2 a_2 a_3 a_4) = \pi'_1 a_2 \pi_2 a_2 a_3 \\
\text{Dyn}(\text{replCh}, \pi a_8 a_9) = \pi a_8
$$
Note that there are four traces reaching \(a_4\), the only sending site of channel \(replCh'\). However, these four dynamic instance of channel \(replCh'\) are represented by four different traces. Their real dynamic values are dependent on the dynamic values received at \(a_3\) along their traces. If all dynamic values received at \(a_3\) are not same, then messages are sent over different channel instances at \(a_4\).

**Definition 4.1.7** Predicate \(CSend(c \circ \pi, a, ch)\) returns true if the dynamic channel instance \(c \circ \pi\) may be sent on channel \(ch\) at program point \(a\). Otherwise, false is returned.

Predicate \(MultiSend(c \circ \pi, a, ch)\) returns true if the dynamic channel instance \(c \circ \pi\) may be sent twice on channel \(ch\) at program point \(a\). Otherwise, false is returned.

**Definition 4.1.8** Predicate \(CRecv(c \circ \pi, a, ch)\) returns true if the dynamic channel instance \(c \circ \pi\) may be received on channel \(ch\) at program point \(a\). Otherwise, false is returned.

Predicate \(MultiRecv(c \circ \pi, a, ch)\) returns true if the dynamic channel instance \(c \circ \pi\) may be received twice on channel \(ch\) at program point \(a\). Otherwise, false is returned.

**Theorem 4.1.1** \(CSend(c \circ \pi, a, ch) = true\) if and only if \(a \in \hat{\text{SendSites}(ch)}\) and \(\exists \pi_1 \in A_{out}(a)\) with \(Dyn(c, \pi_1) = \pi'_1 \cdot d_1\) s.t. one of the following conditions is satisfied:

- \(\text{DefType}(c, d_1) = \text{new}\) and \(\pi'_1 \cdot d_1 = \pi\)
- \(\text{DefType}(c, d_1) = \text{recv}\) and \(CRecv(c \circ \pi, a, ch) = true\)

where \(CRecv(c \circ \pi, a, ch) = true\) if and only if \(\exists c', d'_1\) such that \((d'_1, d_1) \in \text{Syn}(G)\) and \(CSend(c \circ \pi, c', d'_1, c') = true\)

**Theorem 4.1.2** \(MultiSend(c \circ \pi, a, ch) = true\) if and only if \(a \in \hat{\text{SendSites}(ch)}\) and \(\exists \pi_1, \pi_2 \in A_{out}(a),\) with \(Dyn(c, \pi_1) = \pi'_1 \cdot d_1,\) \(Dyn(c, \pi_2) = \pi'_2 \cdot d_2\) such that any one of the following conditions is satisfied:
• \( \text{DefType}(c, d_1) = \text{DefType}(c, d_2) = \text{new} \) and \( \pi'_1 \cdot d_1 = \pi'_2 \cdot d_2 = \pi \)

• \( \text{DefType}(c, d_1) = \text{DefType}(c, d_2) = \text{recv} \) and \( d_1 \neq d_2 \), \( \text{CRecv}(c \oplus \pi, d_1, c') = \text{true} \) and \( \text{CRecv}(c \oplus \pi, d_2, c'') = \text{true} \)

• \( \text{DefType}(c, d_1) = \text{DefType}(c, d_2) = \text{recv} \) and \( d_1 = d_2 \) and \( \text{MultiRecv}(c \oplus \pi, d_1, c') = \text{true} \)

where \( \text{MultiRecv}(c \oplus \pi, d_1, c') = \text{true} \) only if \( \exists c', d'_1 \) such that \( (d'_1, d_1) \in \text{Syn}(G) \) and \( \text{MultiSend}(c \oplus \pi, d'_1, c') = \text{true} \)

To compute communication topology for channel \( ch \), we compute all possible traces that reach communication sites of all dynamic instances of \( ch \). The trivial solution is to compute \( \bigcup \{ A_{\text{out}}(a) \mid a \in U(ch) \) and \( \text{UseType}(a) = \text{comm} \} \). Rather, we can improve the analysis precision by distinguish dynamic instances and using information gained by \( \text{MultiSend} \). If we know none of dynamic instances of channel \( c \) may be sent twice on channel \( ch \) at program point \( a \), then any dynamic instance of \( c \) can be sent to only one of the possible receivers over \( ch \). In other words, for any dynamic instance of \( c \), only one receiver can actually receive it on \( ch \) by communicating with \( a \).

**Definition 4.1.9** We say program point \( d_1 \) is a synchronization ancestor w.r.t. channel \( c \) of program point \( d_2 \), denoted as \( d_1 \prec_c d_2 \), if \( \exists c \) s.t. \( c \) is sent on \( ch \), \( \exists d'_2 \in D(ch) \) and \( \exists \pi \) from \( d'_2 \) to \( d_2 \) s.t. \( (d_1, d'_2) \in \text{Syn}(G) \) or \( \exists (d'_1, d'_2) \in \text{Syn}(G) \) and \( d_1 \prec_c d'_1 \). And we have \( d \prec_c d \).

In our running example, we have \( a_8 \prec_{\text{replCh}} a_3 \).

**Definition 4.1.10** We say program point \( d' \) is a synchronization dominator of program point \( d \) of channel \( c \), if \( d' \neq d \) and \( \forall d'' \prec_c d \), either \( d'' \prec_c d' \) or \( d' \prec_c d'' \).
In our running example, $a_8$ is the only synchronization ancestor of $a_3$ other than $a_3$ itself. Therefore, $a_8$ is a synchronization dominator of $a_3$.

**Definition 4.1.11** Predicate $\text{MutualEx}(d_1, d_2, c \circ \pi)$ returns true if $d_1$ and $d_2$ have a least common synchronization dominator $a$ w.r.t. channel $c$, where $c$ is sent over $ch$ at $a$ and $\text{MultiSend}(c \circ \pi, a, ch) = \text{false}$. Otherwise it returns false.

**Definition 4.1.12** Predicate $\text{ME}(\pi_1, \pi_2, c \circ \pi)$ returns false if $\text{Dyn}(c, \pi_1) = \pi_1' \cdot d_1$, $\text{Dyn}(c, \pi_2) = \pi_2' \cdot d_2$ and either

- $\pi_1' \cdot d_1 = \pi_2' \cdot d_2$, or
- $\text{MutualEx}(d_1, d_2, c \circ \pi) = \text{false}$

Otherwise it returns true.

**Definition 4.1.13** Given a dynamic channel instance $c \circ \pi$, we say $c \circ \pi$ flows to $\pi' \cdot a$ if and only if $\pi' \cdot a \in A_{out}(a)$, $a \in \hat{\text{SendSites}}(c) \cup \hat{\text{RecvSites}}(c)$, $\text{Dyn}(c, \pi' \cdot a) = \pi'' \cdot d$, and any one of the following conditions is satisfied:

- $\text{DefType}(c, d) = \text{new}$ and $\pi'' \cdot d = \pi$
- $\text{DefType}(c, d) = \text{recv}$ and $\exists c' \text{ s.t. } C_{\text{Recv}}(c \circ \pi, d, c') = \text{true}$

**Definition 4.1.14** Given a program $p$ and a dynamic channel instance $c \circ \pi$, we define

\[
\hat{S}_{c \circ \pi} = \bigcup_{a \in \text{SendSites}(c)} \{ \pi' \mid \pi' \in A_{out}(a) \text{ and } c \circ \pi \text{ flows to } \pi' \}
\]

\[
\hat{R}_{c \circ \pi} = \bigcup_{a \in \text{RecvSites}(c)} \{ \pi' \mid \pi' \in A_{out}(a) \text{ and } c \circ \pi \text{ flows to } \pi' \}
\]

**Definition 4.1.15** Given a program $p$ and a dynamic channel instance $c \circ \pi$, we say there are three types of communication topology of send/receive over $c \circ \pi$: The first being is that there is only one send/receive over the channel, represented as 1. The second is that one
thread sends/receives multiple times over the channel, represented as 2. And the third is that multiple threads send/receive over the channel, represented as m.

**Definition 4.1.16** Function $\text{Topology}_{c@\pi}$ maps $\hat{S}_{c@\pi}$ and $\hat{R}_{c@\pi}$ to their communication topology types. Given $S = \hat{S}_{c@\pi}$ or $S = \hat{R}_{c@\pi}$, $\text{Topology}_{c@\pi}$ is defined as follows:

$$
\text{Topology}_{c@\pi} S = \begin{cases} 
  m & \text{if } \exists \pi_1, \pi_2 \in S \text{ that } \text{Proc}(\pi_1) \neq \text{Proc}(\pi_2) \\
  2 & \text{if } \exists \pi_1, \pi_2 \in S \text{ that } \text{ME}(\pi_1, \pi_2, c@\pi) = \text{false} \\
  1 & \text{otherwise}
\end{cases}
$$

**Definition 4.1.17** Given a program $p$ and a channel $c$, we say there are three types of communication topology of send/receive over $c$: One is that there is only one send/receive over the channel, represented as 1. Another is that one thread sends/receives multiple times over the channel, represented as 2. And the third is that multiple threads send/receive over the channel, represented as m.

A channel $c$ in a program $p$ has communication patterns only if all possible dynamic instances of $c$ have the same communication patterns during execution. The problem is that it is not possible to compute all possible dynamic instance and all possible runtime behaviors of the program. The abstract trace analysis, however, gives a safe approximation of all possible dynamic instances of channel $c$ and all possible runtime communications over $c$. By extracting channel use information from traces, we can safely compute communication patterns for channels as the following definition.

**Definition 4.1.18** Function $\text{Topology}$ maps channels to their communication topology types of send/receive. Given a channel $c$ created at $a$, its communication topology type of send/rece-
ceive is defined as follows:

\[
\text{Topology}_c = \begin{cases} 
  m & \text{if } \exists \pi \in A_{out}(a) \text{ s.t. } \text{Topology}_{c@\pi}S = m \\
  1 & \text{if } \forall \pi \in A_{out}(a) \text{ s.t. } \text{Topology}_{c@\pi}S = 1 \\
  2 & \text{otherwise}
\end{cases}
\]

Once again, we turn to our running example to illustrate how trace refinement works and how to compute approximate communication topology for \textit{replCh}. The only abstract dynamic instance of \textit{replCh} is \(\pi = a_1a_{10}a_{12}a_{14}a_{15}a_6a_7\). The only place where channel \textit{replCh} is sent as value is at \(a_8\). As \(A(a_8)\) only contains trace \(\pi a_8\), \textit{MultiSend} \((\textit{replCh}@\pi, a_8, \textit{ch}) = false\). Because \(a_8\) is the only synchronization dominator of \(a_3\), by definition, we have \(\text{MutualEx}(a_3, a_3, \textit{replCh}@\pi) = true\).

Recall that program point \(a_4\) is the only sending site of channel \textit{replCh}. Also, we have

\[
A_{out}(a_4) = \{\pi_1a_2a_3a_4, \pi_1a_2\pi_2a_2a_3a_4, \pi_1a_2\pi_2a_2a_3a_4\pi_2a_2a_3a_4, \\
\pi_1'a_2a_3a_4, \pi_1'a_2\pi_2a_2a_3a_4, \pi_1'a_2\pi_2a_2a_3a_4\pi_2a_2a_3a_4\}
\]

By definition, \textit{Dyn} returns six different dynamic channel instances for traces from \(A_{out}(a_4)\), \(\pi_1a_2a_3, \pi_1a_2\pi_2a_2a_3, \pi_1a_2\pi_2a_2a_3a_4\pi_2a_2a_3, \pi_1'a_2a_3, \pi_1'a_2\pi_2a_2a_3, \pi_1'a_2\pi_2a_2a_3a_4\pi_2a_2a_3\) respectively. Thus, by definition, these six dynamic communications are mutually exclusive. So there is only one of these six communications that can successfully reach the sending site of \textit{replCh}. Thus \textit{replCh} is a one-shot channel.

### 4.2 Concurrency Analysis

Another useful application of abstract trace analysis is to statically determine potential pairs of statements in a concurrent program that might execute in parallel. In other words, abstract trace analysis can be used to statically determine potential pairs of statements that can
not execute in parallel. A trivial solution would be the set of all pairs of statements in the program. Although this is a correct solution, it gives no useful information. A perfect set of such pairs will be the set that contains all possible concurrent pairs of statements and does not contain ordered pairs of statements. As computation of the perfect solution is known to be a NP-complete problem [Tay83], most approaches are to compute a conservative and safe approximation to the perfect set at the cost of precision. In this section, we show how to use abstract trace analysis to compute a precise set of such pairs.

4.2.1 Overview

In this section, we will provide an overview of our concurrency analysis. We say that two instances of statements are ordered if one can start to execute only after the other completed. We say that two statements are ordered if any instances of the two statements are ordered. A pair of statements might execute in parallel if the two statements are not ordered. Thus the core of the analysis is the computation of a partial order among statements. Our analysis is performed on an extended control-flow graph (ECFG) which is constructed as in abstract trace analysis. For the sake of brevity, node $n$ and the statements represented by node $n$ are interchangeable terms. Thus the analysis is to compute the partial order among nodes in $G$.

Our analysis associate five sets with each node $n$ of the ECFG:

- The set $B_{ctl}(n)$ is the current approximation of the set of nodes that must execute before $n$, which is enforced only by control flow;

- $A_{ctl}(n)$ is the current approximation of the set of nodes that are ordered to execute after $n$, which is enforced only by control flow;

- $B_{syn}(n)$ is the current approximation of the set of nodes that are ordered to execute before $n$, which are enforced by all edges;
• \( A_{\text{syn}}(n) \) is the current approximation of the set of nodes that are ordered to execute after \( n \), which are enforced by all edges;

• \( \text{Order}(n) \) is the current approximation of the set of nodes that are ordered w.r.t. node \( n \).

Intuitively one may think \( \text{Order}(n) = B_{\text{ctl}}(n) \cup A_{\text{ctl}}(n) \cup B_{\text{syn}}(n) \cup A_{\text{syn}}(n) \). Actually \( \text{Order}(n) \) may contain nodes that are not in these four sets. For example, for two statements \( r \) and \( s \) in the same loop, any instance of \( r \) and \( s \) are ordered, but it is not guaranteed that all instances of \( r \) are executed before or after all instances of \( s \).

The concurrency analysis consists of two steps. The first step is to compute the partial order among statements enforced by control flows. The next step is to compute the ordering enforced by synchronizations and then propagate the first step results across synchronization edges throughout the ECFG.

In the following sections, we will discuss each steps of the analysis in more details.

### 4.2.2 Ordering Enforced by Control Flow

For each node \( n \), we use \( \text{pred}_{\text{ctl}}(n) \) and \( \text{succ}_{\text{ctl}}(n) \) to represent the sets of immediate control predecessors and successors of \( n \), respectively. That is \( n' \in \text{pred}_{\text{ctl}}(n) \) and \( n \in \text{succ}_{\text{ctl}}(n') \), if and only if there exists a control/spawn edge from \( n' \) to \( n \).

A standard data-flow-based approach to compute the partial ordering is to use data equations to compute \( A_{\text{ctl}}(n) \), \( B_{\text{ctl}}(n) \) and \( \text{Order}(n) \) as in Figure 4.4. However, the ordering computed may be invalid due to the presence of dynamic concurrency. Consider Figure 4.5 and the statements in the body of \( p \) function. In any instance of \( p \) function, \( a_3 \) always executes before \( a_4 \) as its ECFG is shown in Figure 4.6. But two instances of \( p \) function may execute concurrently, and one instance of \( a_2 \) may execute after the other instance
\[
\begin{align*}
mayB(n) &= \bigcap_{n' \in \text{pred}_{\text{ctl}}(n)} (mayB(n') \cup \{n\}) \\
mayA(n) &= \bigcap_{n' \in \text{succ}_{\text{ctl}}(n)} (mayA(n') \cup \{n'\}) \\
B(n) &= mayB(n) - mayA(n) \\
A(n) &= mayA(n) - mayB(n) \\
Order(n) &= B(n) \cup A(n)
\end{align*}
\]

Figure 4.4: Standard data-flow equations

\begin{verbatim}
a1 : let
a2 : fun p () = (a3 : let t = recv ch in
a4 : send (ch, t)
in
a5 : (spawn (a6 : p ));
a7 : spawn (a8 : p ))
\end{verbatim}

Figure 4.5: A self-concurrent example

of \(a_3\). Thus part of the ordering computed by Figure 4.4 may be invalid. It is not easy to
detect this false ordering.

By previous abstract trace analysis, we shall have abstract trace information available
for each node \(n\) in \(G\) as \(A_{in}(n)\). Before stating analysis, we clarify some notations first.

**Definition 4.2.1** Given an ECFG \(G\), \(\text{Self}_G = \{n \mid n \in G \text{ and } \exists \pi_1, \pi_2 \in A_{in}(n) \text{ s.t. } \text{Proc}(\pi_1) \neq \text{Proc}(\pi_2)\}\).

**Definition 4.2.2** Given an ECFG \(G\), \(\text{Twice}_G = \{n \mid n \in G \text{ and } \exists \pi_1, \pi_2 \in A_{in}(n) \text{ s.t. } \pi_1 \prec \pi_2\}\).

A more efficient approach to compute control-flow ordering among the nodes is to take
advantage of trace flow information as follows:

\[
B_{\text{ctl}}(n) = \begin{cases} 
\text{Common}(A_{in}(n)) \setminus (\text{Loop}_{n,G} \cup \text{Self}_G) & n \notin \text{Self}_G \\
\text{Common}(A_{in}(n)) \setminus (\text{Loop}_{n,G} \cup \text{Twice}_G) & n \in \text{Self}_G
\end{cases}
\]

68
Here, function $Common(S)$ maps a set of traces $S$ to a set of nodes that appear in all traces of $S$. $Loop_{n, G}$ be the set of all nodes in any loops containing $n$ in $G$; $Nodes(S)$ maps a set of traces $S$ to the set of all nodes in $S$.

To illustrate how the computation works, we revisit the self-concurrent-execution example. By abstract trace analysis, we have:

$$A_{in}(a_3) = \{a_1 a_5 a_6 a_2, a_1 a_5 a_7 a_8 a_2\}$$

$$A_{in}(a_4) = \{a_1 a_5 a_6 a_2 a_3, a_1 a_5 a_7 a_8 a_2 a_3\}$$
By definition, we have

\[
\text{Common}(A_{in}(a_3)) = \{a_1, a_5, a_2\} \\
\text{Common}(A_{in}(a_4)) = \{a_1, a_5, a_2, a_3\}
\]

By further investigation, we can observe that traces reaching body of function \( p \) have different thread id parts, which means that function \( p \) may have different instances execute concurrently. By definition, we have

\[
\text{SelfP}(G) = \{a_2, a_3, a_4\} \\
B_{ctl}(a_3) = \text{Common}(A_{in}(a_3)) - \text{SelfP}(G) = \{a_1, a_5\} \\
B_{ctl}(a_4) = \text{Common}(A_{in}(a_4)) - \text{SelfP}(G) = \{a_1, a_5\}
\]

It is more tricky to compute \( A_{ctl}(n) \). Here, function \( \text{Succ}_n(S) \) maps a set of traces to the set of suffix of traces in \( S \), which start from the first appearing \( n \) in the trace. \( \text{Exit}_n \) is the subset of \( A_{in}(\text{Exit}) \), which has \( n \) appearing. We can compute \( A_{ctl}(n) \) as follows:

\[
A_{ctl}(n) = \begin{cases} 
\text{Common}(\text{Succ}_n(\text{Exit}_n)) \setminus \{(n) \cup \text{Loop}_{n,G} \cup \text{SelfP}_G \} & n \notin \text{SelfP}_G \\
\text{Common}(\text{Succ}_n(\text{Exit}_n)) \setminus \{(n) \cup \text{Loop}_{n,G} \cup \text{Twice}_G \} & n \in \text{SelfP}_G 
\end{cases}
\]

And we can compute \( \text{Order} \) as follows:

\[
\text{Order}(n) = \begin{cases} 
B_{ctl}(n) \cup A_{ctl}(n) & n \in \text{SelfP}_G \\
B' \cup A' & n \notin \text{SelfP}_G
\end{cases}
\]

\( B' = \text{Common}(A_{in}(n)) \setminus \text{SelfP}_G \) and \( A' = \text{Common}(\text{Succ}_n(\text{Exit}_n)) \setminus \text{SelfP}_G \).

Again, we use the simple self-concurrent execution example to illustrate how to com-
pute \( A_{ctl}(n) \). By trace flow analysis, we have

\[
A_{in}(Exit) = \{a_1a_5a_7, a_1a_5a_6a_2a_3a_4, a_1a_5a_7a_8a_2a_3a_4\}
\]

Consider \( a_3 \), by definition, we have

\[
\text{Exit}(a_3) = \{a_1a_5a_6a_2a_3a_4, a_1a_5a_7a_8a_2a_3a_4\}
\]
\[
\text{Succ}_{a_3}(\text{Exit}(a_3)) = \{a_3a_4\}
\]
\[
A_{ctl}(a_3) = \emptyset
\]

### 4.2.3 Ordering Enforced by Synchronization

For each node \( n \), we use \( \text{syn}(n) \) to represent the set of nodes that are matching to communicate with \( n \). We have \( n' \in \text{syn}(n) \) and \( n \in \text{syn}(n') \) if and only if there exists a synchronization edge between \( n' \) and \( n \).

After the control-flow ordering has been computed for every node, the new ordering enforced by synchronization is computed. The new before and after ordering are updated by the intersection of current before set and after set over all synchronization edges respectively. Once the propagation across synchronization edges has been done, new orderings are further propagated throughout the control-flow edges. The data-flow equations for this step are shown in Figure 4.7
To illustrate how to compute the partial ordering enforced by synchronization, we use a simple example in Figure 4.8. Figure 4.9 shows its representation in ECFG. By computation in step 1, we have

\[
B_{ctl}(a_3) = \{a_1, a_8, a_9, a_2\} \\
A_{ctl}(a_3) = \{a_4\} \\
B_{ctl}(a_6) = \{a_1, a_8, a_10, a_5\} \\
A_{ctl}(a_6) = \{a_7\}
\]

Consider the ordering enforced by synchronization edge between \(a_3\) and \(a_6\). The nodes execute before \(a_3\) should also execute before \(a_6\) as it is the only matching node. The nodes execute before \(a_6\) should also execute before \(a_3\). The computation of after ordering is the same. The ordering enforced by this synchronization edges is as follows:

\[
B_{syn}(a_3) = B_{ctl}(a_6) \\
A_{syn}(a_3) = A_{ctl}(a_6) \\
B_{ctl}(a_3) = A_{ctl}(a_3) \cup B_{syn}(a_3) \\
B_{syn}(a_6) = B_{ctl}(a_3) \\
A_{syn}(a_6) = A_{ctl}(a_3) \\
B_{ctl}(a_6) = B_{ctl}(a_6) \cup B_{syn}(a_6)
\]

The computation of the ordering enforced by one more synchronization edge between \(a_4\) and \(a_7\) is more complicated. Now we need to merge the propagation from control-flow predecessor \(a_3\) and the propagation from synchronization predecessor \(a_7\). The computation
\[ a_1 : \text{let chan ch} \]
\[ a_2 : \text{fun p1 () = (} \]
\[ a_3 : \text{let t = recv ch in} \]
\[ a_4 : \text{send (ch, t)} \]
\[ a_5 : \text{fun p2 () = (} \]
\[ a_6 : \text{let t = send (ch, 1) in} \]
\[ a_7 : \text{recv ch} \]
\[ a_8 : \text{(spawn (a_9 : p));} \]
\[ a_{10} : \text{spawn (a_{11} : p))} \]

Figure 4.8: A simple example with synchronization edge

results are as follows:

\[
B_{\text{syn}}(a_4) = B_{\text{syn}}(a_3) \cup B_{\text{ctl}}(a_7)
\]
\[
B_{\text{syn}}(a_7) = B_{\text{syn}}(a_6) \cup B_{\text{ctl}}(a_4)
\]
\[
B_{\text{ctl}}(a_4) = B_{\text{ctl}}(a_4) \cup B_{\text{syn}}(a_4)
\]
\[
B_{\text{ctl}}(a_7) = B_{\text{ctl}}(a_7) \cup B_{\text{syn}}(a_7)
\]
\[
\text{Order}(a_4) = \{a_1, a_2, a_3, a_5, a_6, a_8, a_9, a_{10}, a_{11}\}
\]
\[
\text{Order}(a_7) = \{a_1, a_2, a_3, a_5, a_6, a_8, a_9, a_{10}, a_{11}\}
\]

4.2.4 Refinement

There is one way to increase the precision by applying [MR93]’s widening theorem.

\[
\text{Order}(n) = \bigcap_{n' \in N - \text{Order}(n) - \{n\}} \text{Order}(n')
\]

4.2.5 Automatic Monitor Recognition and Transformation

Monitor tasks are a special kind of Ada task in which all computation is carried out within a rendezvous[SN94]. The rendezvous ensures mutual exclusion in the task body. A typi-
cal example of a monitor task is the buffer implementation shown in Figure 4.10. In this implementation, the task body consists of an unconditional loop that contains a select statement waiting to execute one of the accept statements. Access to the underlying buffer is protected by rendezvous with two monitor task entries, Read and Write.

For tasks like Buffer, it is possible to have a much simpler implementation than normal Ada tasks. Upon task creation, a monitor task is not created as a full-blown process. Instead, a Monitor Entries Table and a semaphore become associated with the monitor task.
task Buffer is
  entry Read (item : out Integer);
  entry Write (item : in Integer);
end Buffer;

task body Buffer is
  buf : array (1..N) of Integer;
  ...  
  begin  
  loop  
  select  
  accept Read (item : out Integer) do  
  ...  
  end Read;
  or
  accept Write (item : in Integer) do  
  ...  
  end Write;
  end select;
  end loop;
end Buffer;

Figure 4.10: An example of monitor tasks

All accept statements in the monitor task are transformed into procedures and registered in the Monitor Entries Table. An entry call from a calling task is regarded as a call to the runtime system. The runtime system then secures the semaphore and acts by calling the corresponding procedure registered in the Monitor Entries Table. Thus when interacting with other tasks, a monitor task does not have to be a real task. A monitor task never appears on scheduler queues, and therefore is also called passive task. The task-control block of a monitor task is reduced to contain only local data information and no thread-control information. This light-weight implementation can avoid context switching overhead and is a powerful optimization.

As a special kind of “process”, monitor tasks interact with other tasks in a special kind of communication pattern. By identifying this communication pattern in a message-passing
program, we can replace a monitor process with a more efficient passive implementation and the expensive message passing with function calls.

To illustrate how this analysis and optimization proceed, consider the simple service implemented in Figure 4.11.

The program creates an instance of the service by allocating a new channel and spawning a new monitor thread to handle requests on the channel. The \texttt{client} function sends a request to the service. The request message consists of the request and a fresh channel for the reply. We assume that the monitor thread does pure computation based on the request and the service state, which is abstracted as $f(v, w)$. The result is then sent back on the reply channel. We observe the following facts:

- The \texttt{monitor} function recursively calls itself, forming an unconditional loop.

- The handling of the request is enclosed by \texttt{recv} at $a_3$ and \texttt{send} at $a_5$.

- The monitor state is protected by the above two synchronizations.
\begin{verbatim}
a₁: chan ch in
a₂: MVar state = 0 in
a₃: fun monitor w = (  
a₄:   let v = take (state) 
( v', w' ) = f ( v, w ) 
in
a₅:   ( put (state, v');  
( w' )) 
end
a₆: fun client w = (  
monitor (w) 
in
a₇:   spawn ( a₈: client 1 ))  
\end{verbatim}

Figure 4.12: The optimized version of simple monitor

- No other concurrency statements are executed between \( a₃ \) and \( a₅ \); Function \( f \) does pure computation by assumption.

- There are no threads that break the protocol, such as sending the request and not receiving the result back.

We can exploit these facts to transform the service into the optimized version show in Figure 4.12.

This optimization consists of two major components. The first component is to automatically recognize the monitor-communication pattern in ECFG. The second is to transform the monitor processes into M-variable-protected functions, while transforming the communications with the monitor processes into function calls. We discuss this optimization in detail in the following sections.

4.2.6 Monitor Communication Pattern Recognition

In this section, we discuss how to analyze ECFG and recognize the monitor thread and monitor communication pattern.
Definition 4.2.3  For an ECFG $G(V, E)$ and $n, n' \in V$, we have $n \xrightarrow{\pi} n'$, iff there exists a control path $\pi$ from $n$ to $n'$ in $G$.

We say $n$ reaches $n'$ by control edges, denoted by $n \rightarrow n'$, in an ECFG $G$ if $\exists \pi, n \xrightarrow{\pi} n'$.

We also extend this definition to edges.

Definition 4.2.4  For an ECFG $G(V, E)$, $e = (n', n'') \in E$ and $n \in V$, we have $n \xrightarrow{\pi} e$, iff $n \xrightarrow{\pi} n'$.

We say $n$ reaches $e$, denoted by $n \rightarrow e$, in an ECFG $G$ if $\exists \pi, n \xrightarrow{\pi} e$.

Definition 4.2.5  For an ECFG $G(V, E)$, we say a subgraph $G'(V', E', t)$ of $G$ is a Thread Graph, iff $\exists e = (n, t) \in E$ such that $e$ is a spawn edge, $V' = \{n' | t \rightarrow n'\}$ and $E' = \{e' | t \rightarrow e'\}$.

The above definition defines threads in ECFG representation.

To automatically recognize monitor threads in programs, we only need to analyze Thread Graphs in programs’ ECFG representation to check whether they have the monitor communication pattern.

Definition 4.2.6  For a Thread Graph $G(V, E, t)$, we say a subgraph $G'(V', E', h, B)$ of $G$ is a Loop Graph, iff the following conditions hold:

- $h \in V'$, $h$ is a function entry node, $h$ dominates $V'$ and $h$ postdominates $t$. We call $h$ a loop header of $G'$. And we call the function the loop function of $G'$.

- $\forall n' \in B$, we have $n' \neq h$, $(n', h) \in E$ and $\nexists n'' \in V$ such that $n'' \neq h$ and $(n', n'') \in E$. We call $n'$ a loop trailer of $G'$.

- $V' = \{n'' | h \rightarrow n''\}$, $E' = \{e | h \rightarrow e\}$
Loop Graph defines a subgraph that consists of an infinite loop. Note that for a Loop Graph there may exist more than one loop trailer, but all loop trailers point to the same loop header.

**Definition 4.2.7** For a Loop Graph \(G(V, E, h, B)\), a subgraph \(G'(V', E', r, S)\) of \(G\) is a recv&send Graph, iff the following conditions hold.

- \(r \in V', r\) is a recv statement on a fan-in channel, \(r\) dominates \(V'\) and \(r\) postdominates \(h\). We call \(r\) the receive node of \(G'\).
- \(\forall n' \in S, \text{we have that} n' \in V', h \rightarrow n'\) and \(n''\) is a send statement. We call \(n'\) a send node of \(G'\).
- \(\forall n' \in B, \text{we have that} \exists n'' \in S, \pi_1, \pi_2\) such that \(h \xrightarrow{\pi_1} n'', n'' \xrightarrow{\pi_2} n'\) and \(n \notin \pi_1, \pi_2\).
- \(V' = \{m | r \xrightarrow{\pi_1'} m, m \xrightarrow{\pi_2'} n'', n \notin \pi_1', \pi_2', n'' \in S\}\) and \(E' = \{e | r \xrightarrow{\pi_1' e} e, e \xrightarrow{\pi_2'} n'', n \notin \pi_1', \pi_2', n'' \in S\}\).
- \(\forall n' \in V\) such that \(n'\) is a concurrency node, we have that \(n' \in \{r\} \cup S\).

**Definition 4.2.8** We say \(n\) and \(n'\) are matching nodes if \(n \notin \text{Order}(n')\) and there is a message edge between them.

**Definition 4.2.9** For an ECFG \(G\) and a recv&send Graph \(G'(V, E, r, S)\) of \(G\), we say \(a : e\) is an isolated statement that communicates with \(G'\), if one of the following conditions holds.

- \(a\) is a matching node of \(r\) but \(\forall n \in S\) its intermediate successor is not a matching node of \(n\).
• \( \exists n \in S \) such that \( a \) is a matching node of \( n \) but its intermediate predecessor is not a matching node of \( r \).

The isolated statement that communicates with \( G' \) is the one that breaks the monitor communication protocol and blocks the monitor thread.

**Definition 4.2.10** For a Thread Graph \( G \) and a Loop Graph \( G' \) of \( G \), we call \( G \setminus G' \) an Init Graph.

The Init Graph captures the initialization part of the thread before getting the monitor part. For a statement like \( a_0 : \text{spawn}(a_1 : f (a_2 : e)) \), the initialization part consists of the computation of \( a_2 : e \) before entering \( f \).

**Definition 4.2.11** For an ECFG \( G(V, E) \), a Thread Graph \( G'(V', E', t) \) of \( G \) and a \( G''(V'', E'', r, S) \), we say \( G' \) has the monitor communication pattern, iff the following conditions hold:

- \( G'' \) is a recv\&send Graph of \( G' \).
- There exist no other recv\&send Graphs of \( G' \) than \( G'' \).
- There exist no concurrency nodes in Init Graph of \( G'' \).
- There exist no isolated statements in \( G \) that communicate with \( G'' \).
- \( \exists n \in V \) such that \( \forall n'' \in S \) we have \( \forall \pi \in A_{out}(n'') \), \( \exists \pi' \) such that Dyn\((c, \pi) = \pi' \cdot n \), where \( c \) is the channel sent on at \( n'' \).

And we say that \( G' \) is a monitor thread.

The last condition ensures that all sends in a recv\&send Graph are operated on the same dynamic channel.
4.2.7 Monitor Thread Transformation

By the above definitions, the monitor communication pattern in ECFG can be recognized automatically. The next step is to transform monitor threads and communications. This transformation consists of two components. The first component is to transform the monitor thread into a monitor function. The second component is to transform communications with the monitor thread into the corresponding function calls.

The transformation for monitor thread spawned by $a_0 : \text{spawn}(a_1 : f(a_2 : e))$ is done based on its Thread Graph $G(V,E,r,S)$ in the following steps. Let the loop function have type $\tau_1 \rightarrow \tau_2$.

1. An initialization function $\text{initf}$ is created. $\text{initf}$ is an identity function with type $\tau_1 \rightarrow \tau_1$.

2. The statement at $a_0$ is transformed from spawning to an M-variable initialization and put right before the loop function’s declaration. An M-variable $\text{mvar}$ with type $\tau_1$ is created and initialized by the result of calling $\text{initf}$ with $a_2 : e$ as the argument.

3. A monitor function $\text{monitor}$ is created after M-variable initialization. The monitor function has type $\tau'_1 \rightarrow \tau'_2$ where $\tau'_1$ is the channel type at the receive node of $G$ and $\tau'_2$ is the channel type at the send node of the $G$.

4. The monitor function’s parameter is the variable that is bound at $r$.

5. Statements in the Loop Graph are moved into the body of the monitor function.

6. The statement $a : \text{fun } f(x) = e$ at the loop header is transformed into a binding of the parameter $x$ by a take operation on $\text{mvar}$, $a : \text{let } x = \text{MVar.take(mvar)}$ in e.
7. The statement \( a : \text{send}(c, v) \) at the send node of \texttt{recv\&send Graph} is transformed into a binding of variable \( \text{res} \), \( a : \text{let res} = v \).

8. The recursive call statement \( a : fe \) at the loop trailer is transformed into two sequential statements. The first statement is a put operation on \texttt{mvar} with \( e \). The second statement is a return of \( \text{res} \).

Communications with monitor threads are transformed by pairs. Because there are no isolated statements that communicate with monitor threads, communications appear in pairs. For each consecutive communication pair \( (a : \text{send}(c v), a' : \text{recv} c') \) that communicates with the monitor thread \( G \), we transform the pair into a monitor function call. Let \( a : \text{send}(c v) \) be the matching node of the receive node of the \texttt{recv\&send Graph} of \( G \); Let \( a' : \text{recv} c' \) be the consecutive communication node; and let the monitor thread \( G \) be transformed into the monitor function \( f \). The consecutive communication pair is merged and transformed into a function call as \( a : f v \).

Let’s revisit the example in Figure 4.11 to illustrate how transformation works. Figure 4.13 shows the program after the monitor transformation.
Figure 4.13: The simple monitor after transformation
CHAPTER 5
ANALYSIS CORRECTNESS

In this chapter, we show that our static analysis computes safe approximation of program runtime behaviors. Because we are only concerned with possible execution of programs, when we say a control path, we expect that it is a valid control path. For our notation, we use $\pi(i)$ to denote the $i_{th}$ program point in $\pi$ from left, and $\pi(-i)$ to denote the $i_{th}$ program point in $\pi$ from right. For instance, $(a \cdot \pi)(1) = a$ and $(\pi \cdot a)(-1) = a$. We use $(\pi)^i$ to denote $(\pi) \cdot \ldots \cdot (\pi)$ in which $\pi$ repeats $i$ times. For instance, $(\pi)^2 = \pi \cdot \pi$.

5.1 Communication Topology

In this section, we show that our static analysis of communication topology is safe. Our work is in the context of the simple concurrent language (SCL) defined in [section1.1]. A nice property about SCL is that only function entry point can be the loop header

**Definition 5.1.1** Given a program $p$, an ECFG $G(V, E)$ of $p$ and a node $h$ in $G$, we say that $h$ is a loop header if $h$ is a function entry point of $f$ and $\exists n' \in V, \exists \pi \text{ s.t. } h \xrightarrow{\pi} n'$, $(n', h) \in E$ and $\pi$ does not contain $h'$, the function exit point of $f$. We call $h'$ the loop exit point.

**Definition 5.1.2** Given a program $p$, an ECFG $G(V, E)$ of $p$ and a loop header $h$, we say that $n$ is a loop trailer of $h$ if $\exists \pi$ s.t. $h \xrightarrow{\pi} n$, $(n, h) \in E$ and $\pi$ does not contain the function exit point of $f$. We use $\text{Trailer}(h)$ to denote the set of all trailers of $h$. 
**Definition 5.1.3** Given a control path $\pi$, we say that $\pi$ is a balanced control path, if $\pi$ has the same number of call nodes and return nodes. We say that $\pi$ is a balanced control path w.r.t. function entry point $a$, if $\pi$ has the same number of $a$ and the corresponding function exit point of $a$.

**Definition 5.1.4** Given a program $p$, an ECFG $G(V, E)$ of $p$ and a control path $\pi$ in $G$, we say that $\pi$ is a level 1 iteration of a loop, if $\pi = h \cdot \pi' \cdot n$, $h$ is a loop header, $n \in \text{Trailer}(h)$ and $\pi'$ is balanced. And we use header $h$ of a loop to represent it. We also use $\tilde{h}$ to denote the set of all level 1 iterations of loop $h$.

**Definition 5.1.5** Given a program $p$, an ECFG $G(V, E)$ of $p$ and a control path $\pi$ in $G$, we say that $\pi$ is a complete iteration of a loop $h$, if $\pi = h \cdot \pi' \cdot n$, $h$ is a loop header, $n$ is the loop exit point of $h$ and $\pi'$ is balanced. We use $\dot{h}$ to denote the set of all complete iterations of loop $h$.

**Lemma 5.1.1** Given a program $p$ and a complete iteration $\pi = h \cdot \pi' \cdot n$ of a loop $h$, $\exists i, h_j \in \tilde{h}, h_i \in \dot{h}, h_i' \in h'$ s.t. $\pi = h_1 \cdot h_2 \cdot \ldots \cdot h_i \cdot \ldots \cdot h_i' \cdot h_i$ where $(h_i')_{(-1)}$ is loop exit node of $h$, and $(h_i')_{(1)}$ is the corresponding return node of $(h_i)_{(-1)}$.

**Proof:** This is obvious by Definition 5.1.5 and valid control path.

For a complete iteration of $h$, we call $h_i$ as the inner complete loop iteration of $h$.

**Definition 5.1.6** Given a program $p$, an ECFG $G(V, E)$ of $p$ and a control path $\pi$ in $G$, we say that $n$ is in a incomplete iteration of a loop $h$ of $\pi$, if $\pi = \pi_1 \cdot h \cdot \pi' \cdot n$, $h$ is a loop header, $n$ is not the loop exit point of $h$ and $\pi'$ is balanced w.r.t. $h$. We say that $\pi' \cdot n$ is the inner incomplete loop iteration of $h$ of $\pi$. 
By standard analysis, we can obtain information about loops in SCL programs. So we assume that we have information about loop headers and loop trailers for all loops at hand.

**Definition 5.1.7** We say that a control path $\pi$ has a level $i$ iteration of loop $h$, represented by $h \in^i \pi$, if $\exists \pi_1, \pi_2 \neq \epsilon$ and for $1 \leq j \leq i-1$, $\exists h_j \in \tilde{h}$, such that $\pi = \pi_1 \cdot h_1 \cdot \ldots \cdot h_i \cdot \pi_2$. We say that a control path $\pi$ has a high level iteration of loop $h$, more than level 3, represented by $h \in^{>3} \pi$, if $\exists i > 3$ s.t. $h \in^i \pi$.

**Definition 5.1.8** Given a program $p$ and a control path $\pi$ in an ECFG $G(V, E)$ of $p$, we say that a control path $\pi$ is an abstract control path, if $\nexists h \in V$ such that $h \in^{>3} \pi$. We use $\text{ABSTRACTPATH}$ to denote the set of all abstract control paths.

**Definition 5.1.9** Given a control path that is a balanced control path w.r.t. $h$, we define $\text{Collapse}_h$ function as follows.

$$\text{Collapse}_h(\pi) = \begin{cases} \text{Collapse}_h(\pi_1 \cdot \pi_2' \cdot \pi_3) & \text{if } \pi = \pi_1 \cdot \pi_2 \cdot \pi_3, \; h \in^2 \pi_2 \\
\pi_2 \text{ is a complete iteration of loop } h & \\
\pi_2' \text{ is the inner complete iteration of loop } h & \\
\pi & \text{if } h \notin^2 \pi \end{cases}$$

Intuitively, $\text{Collapse}_h$ collapse each complete iterations of loop $h$ occurring in $\pi$ into the corresponding inner complete iteration.

**Definition 5.1.10** Given a program $p$ and a control path $\pi$ in an ECFG $G(V, E)$ of $p$, we say that $\pi_{21} \cdot \ldots \cdot \pi_{2i}$ is the top loop in $\pi$, if $\pi = \pi_1 \cdot (\pi_{21} \cdot \ldots \cdot \pi_{2i}) \cdot \pi_3$, $i \geq 3$, $\pi_{2j} \in \tilde{h}$, $1 \leq j \leq i$, and $\exists \pi_1', \pi_{1j}, \pi_3', l$ s.t. $\pi = \pi_1' \cdot (\pi_{11} \cdot \ldots \cdot \pi_{1k}) \cdot \pi_3', k \geq 3$, $\pi_1' \cdot (\pi_{11} \cdot \ldots \cdot \pi_{1k}) \prec \pi_1 \cdot (\pi_{21} \cdot \ldots \cdot \pi_{2i})$, $\pi_{1j} \in \tilde{h}'$, $1 \leq j \leq k$ and $\pi_{1l} \neq \pi_{2l}$.
\[ \text{abs}(\pi') \begin{cases} \text{abs}(\pi') & \text{if } \pi = \pi_1 \cdot \pi_3, \pi_2 = \pi_2' \cdot \pi_2'' \\ \pi_2 \text{ is a complete iteration of loop } h \\ \pi_2' = \pi_21 \cdot \ldots \cdot \pi_{2i} \text{ is the top loop } h \text{ in } \pi \\ \pi_2'' = \pi_2''' \cdot \pi_2' \cdot \ldots \cdot \pi_2' \\ \pi_{2j} \in \widetilde{h}, 1 \leq j \leq i, \pi_3 \neq \epsilon, \\ \pi' = \pi_1 \cdot \pi_21 \cdot \pi_22 \cdot \pi_23, i \geq 3 \\ \pi_{21} \cdot \ldots \cdot \pi_{2i} \text{ is the top loop } h \text{ in } \pi \\ \pi_3' \text{ is the incomplete iteration of loop } h \text{ in } \pi \\ \pi_2j \in \widetilde{h}, 1 \leq j \leq i, \pi_3'' = \text{Collapse}_h(\pi_3') \\ \pi' = \pi_1 \cdot \pi_21 \cdot \pi_3'' \\ \pi \text{ otherwise} \end{cases} \]

Figure 5.1: The definition of \text{abs}

**Lemma 5.1.2** Given a program \( p \) and a control path \( \pi \) in an ECFG \( G(V, E) \) of \( p \), if \( \pi \) has a top loop, then it is unique.

**Proof:** This is obvious by definition.

**Definition 5.1.11** A control path abstract function \( \text{abs} \) is a function that maps from \text{CTLPATH} to \text{ABSCTLPATH}. \( \text{abs} \) is defined recursively as in Figure 5.1.

Function \( \text{abs} \) maps control paths to abstract control paths. Intuitively, \( \text{abs} \) preserves the first, second and last iteration of the loop and discard other iterations. In other words, all other iterations than the first and second iteration is considered as a collapsed iteration. Note that if the last iteration is not completed, the collapsed iteration is considered incomplete.
Lemma 5.1.3 For any control path $\pi \in \text{CTLPath}$, $\exists! \hat{\pi} \in \text{ABSCTLPath}$ such that $\hat{\pi} = \text{abs}(\pi)$.

Proof: Prove by induction of the length of $\pi$.

Basis: $|\pi| = 1$. This is obvious by definition. $\text{abs}(\pi) = \pi$ as $\pi$ has no loops.

Induction Step: Assume for any control path $\pi$ such that $|\pi| = n - 1$, $\exists \hat{\pi} \in \text{ABSCTLPath}$ such that $\hat{\pi} = \text{abs}(\pi)$ and $\forall \pi' \in \text{ABSCTLPath}$ such that $\pi' = \text{abs}(\pi)$, $\pi' = \hat{\pi}$.

Now consider any control path $\pi$ such that $|\pi| = n$ in the following two cases.

i) $\nexists h$ s.t. $h \in >^3 \pi$

By definition, $\pi \in \text{ABSCTLPath}$. And by definition of $\text{abs}$, we have that $\text{abs}(\pi) = \pi$.

ii) $\exists h$ s.t. $h \in >^3 \pi$

Let $\pi_21 \cdot \ldots \cdot \pi_2i$ be the top loop in $\pi$. We have two cases to consider.

a) $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3, \pi_2 = \pi_{21} \cdot \ldots \cdot \pi_{2i}, \pi_2$ is a complete iteration of loop $h$, $\pi''_2 = \pi''_2 \cdot \pi'_{2i} \cdot \ldots \cdot \pi'_{21}, \pi_{2j} \in \tilde{h}, 1 \leq j \leq i, \pi_3 \neq \epsilon$

By definition of $\text{abs}$, we have that $\text{abs}(\pi) = \text{abs}(\pi_1 \cdot \pi_{21} \cdot \pi_{22} \cdot \pi''_2 \cdot \pi'_{22} \cdot \pi'_{21} \cdot \pi_3)$.

Because $|\pi_1 \cdot \pi_{21} \cdot \pi_{22} \cdot \pi''_2 \cdot \pi'_{22} \cdot \pi'_{21} \cdot \pi_3| < |\pi|$, we have that $\exists! \hat{\pi} \in \text{ABSCTLPath}$ such that $\hat{\pi} = \text{abs}(\pi)$.

b) $\pi(-1)$ is in a incomplete iteration of top loop

Then we have $\pi = \pi_1 \cdot (\pi_{21} \cdot \ldots \cdot \pi_{2i}) \cdot \pi_3, \pi'_3$ is the incomplete iteration of loop $h$ in $\pi$, $\pi''_3 = \text{Collapse}_h(\pi'_3)$. By definition of $\text{Collapse}_h$, we have that $\pi''_3$ is unique. By definition of $\text{abs}$, we have that $\text{abs}(\pi) = \text{abs}(\pi_1 \cdot \pi_{21} \cdot \pi''_3)$. Because $|\pi_1 \cdot \pi_{21} \cdot \pi''_3| < |\pi|$, we have that $\exists! \hat{\pi} \in \text{ABSCTLPath}$ such that $\hat{\pi} = \text{abs}(\pi)$.

Lemma 5.1.4 For any control path $\pi \in \text{CTLPath}$ and abstract control path $\hat{\pi}$ of $\pi$, we have $\pi(-1) = \hat{\pi}(-1)$.

Proof: This is obvious by the definition of the control path abstract function $\text{abs}$. 88
Lemma 5.1.5 For any control path $\pi \in \text{CTLPath}$ and abstract control path $\hat{\pi}$ of $\pi$, we have $\hat{\pi} \in A_{\text{out}}(\hat{\pi}_{(-1)})$.

Proof: Prove by induction of the length of $\pi$.

Basis: $|\pi| = 1$. This is obvious by abstract control path definition and abstract trace flow definition.

Induction Step: Assume for any control path $\pi$ such that $|\pi| = n - 1$, that $\text{abs}(\pi) \in A_{\text{out}}((\text{abs}(\pi))_{(-1)})$.

Now consider any control path $\pi$ such that $|\pi| = n$. Suppose $(\pi)_{(-1)} = a$ and $\pi = \pi' \cdot a$.

There are two cases. One is that $\nexists h$ s.t. $h \notin^{>3} \pi$. So $\text{abs}(\pi) = \pi' \cdot a$. By the induction hypothesis, we have $\pi' \in A_{\text{out}}((\pi')_{(-1)})$. And we have $(\pi')_{(-1)} = (\pi_{(-2)})$. Because there is a control edge from $\pi_{(-2)}$ to $\pi_{(-1)} = a$ and $a$ is either not a loop header or $\pi'$ is not redundant, by definition of $A_{\text{out}}$, we have $\text{abs}(\pi') \cdot a \in A_{\text{out}}(a)$. If $\exists h$ s.t. $h \notin^{>3} \pi$. Let $\pi_{21} \cdot \ldots \cdot \pi_{2i}$ be the top loop $h$ in $\pi$. We have the following two cases to consider.

a) $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3, \pi_2 = \pi_{21} \cdot \ldots \cdot \pi_{2i} \cdot \pi''_2, \pi_2$ is a complete iteration of loop $h, \pi''_2 = \pi''_2 \cdot \pi''_{21} \cdot \ldots \cdot \pi''_{2j}, \pi_{2j} \in \tilde{h}, 1 \leq j \leq i, \pi_3 \neq \epsilon$

Let $\pi' = \pi_1 \cdot \pi_{21} \cdot \pi_{22} \cdot \pi''_{21} \cdot \pi_{22} \cdot \pi''_{21} \cdot \pi_3$. By definition of $\text{abs}$, we have that $\text{abs}(\pi) = \text{abs}(\pi')$. Because $|\pi'| < |\pi|$ and $(\pi')_{(-1)} = (\pi_{(-2)})$, we have that $\hat{\pi} \in A_{\text{out}}(\hat{\pi}_{(-1)})$ by Lemma 5.1.4.

b) $\pi_{(-1)}$ is in a incomplete iteration of top loop

Then we have $\pi = \pi_1 \cdot (\pi_{21} \cdot \ldots \cdot \pi_{2i}) \cdot \pi_3, \pi'_3$ is the incomplete iteration of loop $h$ in $\pi, \pi''_3 = \text{Collapse}_h(\pi'_3)$, Let $\pi' = \pi_1 \cdot \pi_{21} \cdot \pi''_3$. By definition of $\text{abs}$, we have that $\text{abs}(\pi) = \text{abs}(\pi')$. Because $|\pi'| < |\pi|$ and $(\pi')_{(-1)} = (\pi_{(-2)})$, we have that $\hat{\pi} \in A_{\text{out}}(\hat{\pi}_{(-1)})$ by Lemma 5.1.4.  

Corollary 5.1.6 For any channel instance $k$ in trace $t \in \text{Trace}(p)$, let $k = c @ \pi, \exists! \hat{\pi} \in \text{ABSCTLPath}$ such that $\hat{\pi} \in A_{\text{out}}(\hat{\pi}_{(-1)})$.  

89
**Definition 5.1.12** For a channel variable \( c \) and a dynamic value \( \text{Dyn}(c, \pi) \) along \( \pi \), there are two types of the dynamic value, \( \text{new} \) and \( \text{recv} \). Function \( \text{DynType} : \text{CTLPATH} \rightarrow \{ \text{new}, \text{recv} \} \) maps a dynamic value to its corresponding definition type and is defined as follows.

\[
\text{DynType}(\pi) = \begin{cases} 
\text{new} & \text{if it is a channel creation statement at } \pi(-1) \\
\text{recv} & \text{otherwise}
\end{cases}
\]

**Definition 5.1.13** For any trace \( t \in \text{Trace}(p) \), channel instance \( k = c \otimes \pi' \) in \( t \), and control path \( \pi \in \text{Sends}_t(k) \cup \text{Recvs}_t(k) \), let \( ch \) be the variable that is used to communicate at \( \pi(-1) \) and \( \pi = \pi'' \pi''' \) where \( \pi''' = \text{Dyn}(ch, \pi) \). We define

\[
\text{PathH}_{t,k}(\pi) = \begin{cases} 
< (\pi, \pi''') > & \text{if } \text{DynType}(\pi''') = \text{new} \\
< (\pi_1, \pi'_1), (\pi_2, \pi'_2), \ldots, (\pi_m, \pi'_m) > & \text{otherwise}
\end{cases}
\]

where \( \pi_1, \pi'_1, \ldots, \pi_m, \pi'_m \) satisfies that \( \pi_1 = \pi, \pi'_1 = \pi''' \), \( (\pi'_i, k_{i+1}, \pi_{i+1}) \in H \), and let \( c_{i+1} \) be the variable sent at \( (\pi_{i+1})(-1) \), then \( \pi'_{i+1} = \text{Dyn}(c_{i+1}, \pi_{i+1}) \) and \( \text{DynType}(\pi'_m) = \text{new} \), for \( i = 1, \ldots, m - 1 \). Note that \( \pi'_m = \pi' \).

Given an trace \( t \) of program \( p \), a dynamic channel instance \( k \) in that trace, and any dynamic communication instance \( \pi \) over \( k \), the above definition of \( \text{PathH}_{t,k} : \text{PATH} \rightarrow \text{list of PATH} \) gives us the history in which channel instance \( k \) flows between threads. Here a list of \( \text{PATH} \) represents a list of dynamic communication instances. For instance, \( \text{PathH}_{t,k}(\pi) = < (\pi_1, \pi'_1), (\pi_2, \pi'_2) > \), this means that the dynamic channel instance \( k \) is created at \( (\pi'_2)(-1) \) along \( \pi'_2 \), then \( k \) flows to \( (\pi_2)(-1) \) and is sent from thread \( \text{Proc}(\pi_2) \) to thread \( \text{Proc}(\pi'_1) \) by communication between \( (\pi_2)(-1) \) and \( (\pi'_1)(-1) \). And \( k \) flows to \( (\pi_1)(-1) \) and is used to communicate at \( (\pi_1)(-1) \).
Definition 5.1.14 Given a trace \( t \) of program \( p \), channel instance \( k \) in \( t \), and any control path \( \pi, \pi' \in \text{Sends}_t(k) \cup \text{Recvs}_t(k) \), Let \( \text{Path}_{t,k}(\pi) = \langle \pi_1, \pi_1', \ldots, (\pi_{i-1}, (\pi_{i-1}', \ldots, (\pi_{m}, (\pi_{m}', \ldots, (\pi_{m'}') \rangle \rangle \text{ Path}_{t,k}(\pi_2) = \langle \pi_{21}, \pi_{21}', \ldots, (\pi_{2m}, (\pi_{2m}', \ldots, (\pi_{2m'}') \rangle \rangle \) and \( \text{Path}_{t,k}(\pi_1) \neq \text{Path}_{t,k}(\pi_2) \). We say \( \text{Path}_{t,k}(\pi) \oplus \text{Path}_{t,k}(\pi') \) if \( \exists i, j \) such that \( 1 < i \leq m, 1 < j \leq m', \pi_{1i} = \pi_{2j} \) and \( \pi_{1(i-1)}' \neq \pi_{2(j-1)}' \). \( \text{Path}_{t,k}(\pi) \oplus \text{Path}_{t,k}(\pi') \), otherwise.

Intuitively, we say two channel instance flow histories are conflicting( represented as \( \oplus \) ) if the two histories of the same channel instance are not same, but they share some communication instances. A communication instance, however, can only appear once in \( H \).

Lemma 5.1.7 For any trace \( t \) of program \( p \), channel instance \( k \) in \( t \), and any control path \( \pi_1, \pi_2 \in \text{Sends}_t(k) \cup \text{Recvs}_t(k) \), if \( \pi_1 \neq \pi_2 \) then \( \text{Path}_{t,k}(\pi_1) \oplus \text{Path}_{t,k}(\pi_2) \).

Proof: This is obvious by definition of dynamic semantics on Section 3.1. □

Lemma 5.1.8 Given an trace \( t \) of program \( p \), if \( \exists \pi \cdot a, \pi' \cdot a \) in trace \( t \) s.t. \( \pi \neq \pi' \), then \( \exists \pi'', \pi''' \in \text{ABSCTLPath} \) s.t. \( \pi'' \neq \pi''' \) and \( \pi'' \cdot a \in A_{out}(a) \) and \( \pi''' \cdot a \in A_{out}(a) \).

Proof: We assume that \( \hat{\pi} \cdot a = \text{abs}(\pi \cdot a) \) and \( \hat{\pi'} \cdot a = \text{abs}(\pi' \cdot a) \) by Lemma 5.1.4. We show this lemma in the following two cases.

i) \( \hat{\pi} \neq \hat{\pi'} \)

By Lemma 5.1.5, we have \( \hat{\pi} \cdot a \in A_{out}(a), \hat{\pi'} \cdot a \in A_{out}(a) \). So \( \pi'' = \hat{\pi}, \pi''' = \hat{\pi'} \).

ii) \( \hat{\pi} = \hat{\pi'} \)

We have that \( \exists h \text{ s.t. } h \in^>^3 \pi \cdot a \) or \( h \in^>^3 \pi' \cdot a \). Otherwise, we have that \( \text{abs}(\pi \cdot a) = \pi \cdot a \) and \( \text{abs}(\pi' \cdot a) = \pi' \cdot a \), which is contradictory to that \( \hat{\pi} = \hat{\pi'} \). We assume that \( \pi \cdot a \) has high level iteration of loop. By definition of \( \text{abs} \), We have the following two cases to consider.

a) \( \pi \cdot a = \text{abs}(\pi_1 \cdot \pi_2 \cdot \pi_3 \cdot a) = \pi_1 \cdot \pi_{21} \cdot \pi_{22} \cdot \pi_{2m} \cdot \pi_{2m} \cdot \pi_{2m'} \cdot \pi_{2m'} \cdot \pi_{21} \cdot \pi_3 \cdot a, \pi_2 = \pi_{21} \cdot \ldots \pi_{2i} \cdot \pi_{2i}' \),
Lemma 5.1.10

Let $\pi = \pi_2 = \pi_2'' = \pi_2''' \cdot \pi_2' \cdot \ldots \cdot \pi_2'_{21}, \pi_2_j \in \tilde{h}, 1 \leq j \leq i, i \geq 3$

Let $\pi_i = \pi_1 \cdot \pi_2 \cdot \pi_2 \cdot \pi_2'' \cdot \pi_2' \cdot \pi_3 \cdot a$. Because $\text{abs}(\pi_i) = \pi_i$, by Lemma 5.1.5, we have that $\pi_i \in A_{out}(a)$. Because $\pi_2, \pi_2 \in \tilde{h}$, we have that $(\pi_2)_1 = h, (\pi_2)_1 = h, (\pi_2)_1(1), (\pi_2)_2(-1)  \in \text{Trailer}(h)$, and $((\pi_2)_1(-1), h)$ is a valid edge. Thus we have that $\pi_1 \cdot \pi_2 \cdot \pi_2'' \cdot \pi_2' \cdot \pi_3 \cdot a$ is a valid control path. And $\text{abs}(\pi_1 \cdot \pi_2 \cdot \pi_2'' \cdot \pi_2' \cdot \pi_3 \cdot a) = \pi_1 \cdot \pi_2 \cdot \pi_2'' \cdot \pi_2' \cdot \pi_3 \cdot a$. By Lemma 5.1.5, we have that $\pi_1 \cdot \pi_2 \cdot \pi_2'' \cdot \pi_2' \cdot \pi_3 \cdot a \in A_{out}(a)$.

b) $\pi \cdot a = \text{abs}(\pi_1 \cdot (\pi_2 \cdot \ldots \cdot \pi_2) \cdot \pi_3 \cdot a) = \pi_1 \cdot \pi_2 \cdot \pi_3, \pi_2 = \tilde{h}, 1 \leq j \leq i, i \geq 3, \pi_2' \pi_3$ is the incomplete iteration of loop $h$ in $\pi_1 \cdot (\pi_2 \cdot \ldots \cdot \pi_2) \cdot \pi_3 \cdot a, \pi_2'' = \text{Collapse}_h(\pi_3')$

Let $\pi_l = \pi_1 \cdot \pi_2 \cdot \pi_3$. Because $\text{abs}(\pi_l) = \pi_l$, by Lemma 5.1.5, we have that $\pi_l \in A_{out}(a)$.

Because $\pi_2 \in \tilde{h}$, we have that $(\pi_2)_1 = h, (\pi_2)_1 \in \text{Trailer}(h)$, and $((\pi_2)_1(-1), h)$ is a valid edge. Because $\pi_3$ is the incomplete iteration of loop $h, (\pi_3')(-1) = h$. And by definition of $\text{Collapse}_h$, we have that $(\pi_3')(-1) = (\pi_3'')(-1) = h$. Thus we have that $\pi_1 \cdot \pi_3''$ is a valid control path. And $\text{abs}(\pi_1 \cdot \pi_3'') = \pi_1 \cdot \pi_3''$. By Lemma 5.1.5, we have that $\pi_1 \cdot \pi_3'' \in A_{out}(a)$.

Lemma 5.1.9

For any trace $t$ of program $p$, channel instance $k = c@\pi$ in $t$, and any control path $\pi_1, \pi_2 \in \text{Sends}_t(k) \cup \text{Receivs}_t(k)$, if $\pi_1 \neq \pi_2$ then $\text{ME}(\pi_1, \pi_2, c) = \text{false}$.

Proof: Let $\text{Path}_t,k(\pi_1) = (\ldots, (\pi_1_{11}, \pi_1_{11}'), \ldots, (\pi_1_{1m}, \pi_1_{1m}'), \ldots)$ and $\text{Path}_t,k(\pi_2) = (\ldots, (\pi_2_{11}, \pi_2_{11}'), \ldots, (\pi_2_{1n}, \pi_2_{1n}'), \ldots)$. By Lemma 5.1.7, we have $\text{Path}_t,k(\pi_1) \oplus \text{Path}_t,k(\pi_2)$. Then the proof is obvious by definition of $\text{Path}_t,k$ and $\text{ME}$.

Lemma 5.1.10

Given a program $p$, a channel $c$, and control path $\pi_1 \neq \pi_2$ s.t. $\text{Dyn}(c, \pi_1) = \text{Dyn}(c, \pi_2)$, we have that $\exists \pi_1' \neq \pi_2$ s.t. $\pi_1' \in A_{out}((\pi_1)_1(-1)), \pi_2' \in A_{out}((\pi_2)_1(-1))$ and $\text{Dyn}(c, \pi_1') = \text{Dyn}(c, \pi_2')$.

Proof: Let $\text{Dyn}(c, \pi_1) = \text{Dyn}(c, \pi_2) = \pi'$. Then we have $\exists \pi_3, \pi_4$ s.t. $\pi_1 = \pi' \cdot \pi_3$ and $\pi_2 = \pi' \cdot \pi_4$. Let $\pi'$ be the control path from root to $(\pi')_1(-1)$ that does not contain any loop.
trailers. By Lemma 5.1.5, we have that $\hat{\pi}' \in A_{out}(\pi'_{(-1)})$. We show this lemma in the following two cases.

i) $(\pi_1)_{(-1)} \neq (\pi_2)_{(-1)}$

Then $\exists \pi'_3$ from $(\pi')_{(-1)}$ to $(\pi_1)_{(-1)}$ and $\exists \pi'_4$ from $(\pi')_{(-1)}$ to $(\pi_2)_{(-1)}$ s.t. $\pi'_3$ and $\pi'_4$ do not contain new instance creation of $c$ and if any loop trailer occurs in $\pi'_3$ and $\pi'_4$, it only occurs once. By definition of $\text{abs}$, we have that $\text{abs}(\hat{\pi}' \cdot \pi'_3) = \hat{\pi}' \cdot \pi'_3$ and $\text{abs}(\hat{\pi}' \cdot \pi'_4) = \hat{\pi}' \cdot \pi'_4$. By Lemma 5.1.5, we have that $\hat{\pi}' \cdot \pi'_3 \in A_{out}(a)$ and $\hat{\pi}' \cdot \pi'_4 \in A_{out}(a)$. And by definition of $\text{Dyn}$, we have that $\text{Dyn}(c, \hat{\pi}' \cdot \pi'_3) = \text{Dyn}(c, \hat{\pi}' \cdot \pi'_4) = \hat{\pi}'$.

ii) $(\pi_1)_{(-1)} = (\pi_2)_{(-1)}$

Let $(\pi_1)_{(-1)} = (\pi_2)_{(-1)} = a$. Then we have that $\exists \pi'_3$ from $(\pi')_{(-1)}$ to $a$ s.t. $\pi'_3$ does not contain any new instance creation of $c$ and if any loop trailer occurs in $\pi'_3$, it only occurs once. By definition of $\text{abs}$, we have that $\text{abs}(\hat{\pi}' \cdot \pi'_3) = \hat{\pi}' \cdot \pi'_3$. By Lemma 5.1.5, we have that $\hat{\pi}' \cdot \pi'_3 \in A_{out}(a)$. And $\text{Dyn}(c, \hat{\pi}' \cdot \pi'_3) = \pi''$. And we have that $\exists \pi'_4 \neq \pi'_3$ from $(\pi')_{(-1)}$ to $a$ s.t. $\pi'_4$ does not contain any new instance creation of $c$ and if any loop trailer occurs in $\pi'_4$, it occurs no more than twice. By definition of $\text{abs}$, we have that $\text{abs}(\hat{\pi}' \cdot \pi'_4) = \hat{\pi}' \cdot \pi'_4$. By Lemma 5.1.5, we have that $\hat{\pi}' \cdot \pi'_4 \in A_{out}(a)$. And by definition of $\text{Dyn}$, we have that $\text{Dyn}(c, \hat{\pi}' \cdot \pi'_3) = \text{Dyn}(c, \hat{\pi}' \cdot \pi'_4) = \hat{\pi}'$.

**Theorem 5.1.11 One-Send Soundness**

*Given a program $p$ and a channel $c$, if $\exists t \in \text{Trace}(p)$ and $\exists k = c@\pi$ in $t$ s.t. $|\text{Sends}_t(c@\pi)| \geq 2$, then $\exists c', \exists \pi'_1, \pi'_2 \in \widehat{S}_c$ s.t. $\pi'_1 \neq \pi'_2$ and $\text{Topology}_c(\widehat{S}_c) \neq 1$."

**Proof:** From $|\text{Sends}_t(c@\pi)| \geq 2$, we have $\exists \pi_1, \pi_2 \in \text{Sends}_t(c@\pi)$ s.t. $\pi_1 \neq \pi_2$. Because $\pi_1 \neq \pi_2$, by Lemma 5.1.7, we have that $\text{Path}_{H_{t,k}}(\pi_1) \neq \text{Path}_{H_{t,k}}(\pi_2)$. We show this lemma in the following two cases.

i) $\text{Path}_{H_{t,k}}(\pi_1) = \langle (\pi_{11}, \pi'_{11}) \rangle$

In this case $\pi_{11} = \pi_1$ and $\pi'_{11} = \text{Dyn}(c, \pi_1) = \pi$. If $\text{Path}_{H_{t,k}}(\pi_2) = \langle (\pi_{21}, \pi'_{21}) \rangle$, 

93
then we have \( \pi_2 = \pi_1 \) and \( \pi'_2 = \text{Dyn}(c, \pi_2) \). And \( \text{Dyn}(c, \pi_1) = \text{Dyn}(c, \pi_2) = \pi \).

By Lemma 5.1.10, we have that \( \exists \pi_1, \pi_2 \text{ s.t. } \pi_1 \in A_{out}(\pi_1(-1)), \pi_2 \in A_{out}(\pi_2(-1)) \) and \( \text{Dyn}(c, \pi_1) = \text{Dyn}(c, \pi_2) = \hat{c} \). So \( \pi_1, \pi_2 \in \widehat{S_c} \). By definition of \( ME \), we have that \( ME(\pi_1, \pi_2, c) = \text{false} \). Thus by definition, \( \text{Topology}_c(\widehat{S_c}) \neq 1 \).

ii) \( Path_{H_{t,k}}(\pi_1) = < (\pi_{11}, \pi'_{11}),..., (\pi_{1m}, \pi'_{1m}) > \)

In this case \( \pi_{11} = \pi_1 \) and \( \pi'_{1m} = \text{Dyn}(c, \pi_{1m}) = \pi \). If \( Path_{H_{t,k}}(\pi_2) = < (\pi_{21}, \pi'_{21}),..., (\pi_{2n}, \pi'_{2n}) > \), then \( \pi_{21} = \pi_2 \) and \( \pi'_{2n} = \text{Dyn}(c, \pi_{2n}) = \pi \). Let \( \hat{c} = \text{abs}(\pi) \). Let \( \widehat{\pi_1} = \text{abs}(\pi_1) \) and \( \widehat{\pi_2} = \text{abs}(\pi_2) \). If \( \widehat{\pi_1} \neq \widehat{\pi_2} \), by Lemma 5.1.5, \( \widehat{\pi_1} \in A_{out}((\pi_1(-1))) \) and \( \widehat{\pi_2} \in A_{out}((\pi_2(-1))) \). And \( \widehat{\pi_1}, \widehat{\pi_2} \in \widehat{S_c} \). By definition of \( ME \), we have that \( ME(\widehat{\pi_1}, \widehat{\pi_2}, c) = \text{false} \). Thus by definition, \( \text{Topology}_c(\widehat{S_c}) \neq 1 \).

Given a program \( p \) and a channel \( c \), if \( \exists t \in \text{Trace}(p) \) and \( \exists k = c @ \pi \) in \( t \) s.t. \( \text{Recvs}_t(c @ \pi) \geq 2 \), then \( \exists \hat{c} \), \( \exists \widehat{\pi}_1, \widehat{\pi}_2 \in \widehat{R_c} \) s.t. \( \widehat{\pi}_1 \neq \widehat{\pi}_2 \) and \( \text{Topology}_c(\widehat{R_c}) \neq 1 \).

Proof: This is similar to the proof of Theorem 5.1.11.

Lemma 5.1.13 Given a program \( p \), a channel \( c \), and control path \( \pi_1 \neq \pi_2 \) s.t. \( \text{Dyn}(c, \pi_1) = \text{Dyn}(c, \pi_2) \) and \( \text{Proc}(\pi_1) \neq \text{Proc}(\pi_2) \), we have that \( \exists \pi'_1 \neq \pi'_2 \) s.t. \( \pi'_1 \in A_{out}((\pi_1(-1))) \), \( \pi'_2 \in A_{out}((\pi_2(-1))) \), \( \text{Dyn}(c, \pi'_1) = \text{Dyn}(c, \pi'_2) \) and \( \text{Proc}(\pi'_1) \neq \text{Proc}(\pi'_2) \).
**Proof:** Let $Dyn(c, \pi_1) = Dyn(c, \pi_2) = \pi'$. Then we have $\exists \pi_3, \pi_4$ s.t. $\pi_1 = \pi' \cdot \pi_3$ and $\pi_2 = \pi' \cdot \pi_4$. Let $\hat{\pi}'$ be the control path from root to $(\pi')_{(-1)}$ that does not contain any loop trailers. By Lemma 5.1.5, we have that $\hat{\pi}' \in A_{out}(\pi')_{(-1)}$. Because $Proc(\pi_1) \neq Proc(\pi_2)$, we have that either $\pi_3$ or $\pi_4$ has a spawn point. Let $\pi_1 = pid_1 \cdot \pi''_1$ and $\pi_2 = pid_2 \cdot \pi''_2$ where $pid_1 = Proc(\pi_1)$ and $pid_2 = Proc(\pi_2)$. We show this lemma in the following two cases.

i) $pid_1 = \pi' \cdot pid'_1$ and $pid_2 = \pi' \cdot pid'_2$

a) $(pid_1)_{(-1)} \neq (pid_2)_{(-1)}$

Then $\exists \pi'_3$ from $(\pi')_{(-1)}$ to $(pid_1)_{(-1)}$ and $\exists \pi'_4$ from $(\pi')_{(-1)}$ to $(pid_2)_{(-1)}$ s.t. $\pi'_3$ and $\pi'_4$ do not contain new instance creation of $c$ and if any loop trailer occurs in $\pi'_3$ and $\pi'_4$, it only occurs once. Then $\exists \pi''_3$ from $(pid_1)_{(-1)}$ to $(\pi_1)_{(-1)}$ and $\exists \pi''_4$ from $(pid_2)_{(-1)}$ to $(\pi_2)_{(-1)}$ s.t. $\pi''_3$ and $\pi''_4$ do not contain new instance creation of $c$ and if any loop trailer occurs in $\pi''_3$ and $\pi''_4$, it only occurs once. By definition of $abs$, we have that $abs(\hat{\pi}' \cdot \pi'_3 \cdot \pi''_3) = \hat{\pi}' \cdot \pi'_3 \cdot \pi''_3$ and $abs(\hat{\pi}' \cdot \pi'_4 \cdot \pi''_4) = \hat{\pi}' \cdot \pi'_4 \cdot \pi''_4$. By Lemma 5.1.5, we have that $\hat{\pi}' \cdot \pi'_3 \cdot \pi''_3 \in A_{out}(a)$ and $\hat{\pi}' \cdot \pi'_4 \cdot \pi''_4 \in A_{out}(a)$. And by definition of $Dyn$, we have that $Dyn(c, \hat{\pi}' \cdot \pi'_3 \cdot \pi''_3) = Dyn(c, \hat{\pi}' \cdot \pi'_4 \cdot \pi''_4) = \hat{\pi}'$ and $Proc(\hat{\pi}' \cdot \pi'_3 \cdot \pi''_3) \neq Proc(\hat{\pi}' \cdot \pi'_4 \cdot \pi''_4)$.

b) $(pid_1)_{(-1)} = (pid_2)_{(-1)}$

If $\exists \pi'_3$ from $(\pi')_{(-1)}$ to $(pid_1)_{(-1)}$ s.t. $\pi'_3$ does not contain new instance creation of $c$ and does not contain any loop trailer occurs in $\pi'_3$, then it is obvious that $\exists \pi'_4 \neq \pi'_3$ from $(\pi')_{(-1)}$ to $(pid_1)_{(-1)}$ s.t. $\pi'_4$ does not contain new instance creation of $c$ and if any loop trailer occurs in $\pi'_4$, it only occurs once. Then $\exists \pi''_3$ from $(pid_1)_{(-1)}$ to $(\pi_1)_{(-1)}$ and $\exists \pi''_4$ from $(pid_2)_{(-1)}$ to $(\pi_2)_{(-1)}$ s.t. $\pi''_3$ and $\pi''_4$ do not contain new instance creation of $c$ and if any loop trailer occurs in $\pi''_3$ and $\pi''_4$, it only occurs once. If only $\exists \pi'_3$ from $(\pi')_{(-1)}$ to $(pid_1)_{(-1)}$ s.t. $\pi'_3$ does not contain new instance creation of $c$ and some loop trailers occur once in $\pi'_3$, if $\exists \pi''_3$ from $(pid_1)_{(-1)}$ to $(\pi_1)_{(-1)}$ s.t. $\pi''_3$ does not contain new instance
creation of $c$ and does not contain any loop trailer occurs in $\pi'_3$, then this is obvious. Otherwise, there must $\exists \pi''_4$ from $(\pi')(-1)$ to $(\pi_1)(-1)$ s.t. $\pi''_4$ does not contain new instance creation of $c$ and does not contain any loop trailer occurs in $\pi''_4$. By definition of abs, we have that abs$(\pi' \cdot \pi'_3 \cdot \pi''_3) = \pi' \cdot \pi'_3$ and abs$(\pi' \cdot \pi''_4) = \pi' \cdot \pi''_4$. By Lemma 5.1.5, we have that $\pi' \cdot \pi'_3 \cdot \pi''_3 \in A_{out}(a)$ and $\pi' \cdot \pi''_4 \in A_{out}(a)$. And by definition of Dyn, we have that $\text{Dyn}(c, \pi' \cdot \pi'_3 \cdot \pi''_3) = \hat{\pi}' \cdot \pi''_3$ and Proc($\pi' \cdot \pi'_3 \cdot \pi''_3$) $\neq$ Proc($\pi' \cdot \pi''_4$).

ii) $\pi'_3$ from $(\pi')(-1)$ to $(\pi_1)(-1)$ s.t. $\pi'_3$ does not contain new instance creation of $c$ and if any loop trailer occurs in $\pi'_3$, it only occurs once. And Then $\exists \pi''_3$ from $(\pi_1)(-1)$ to $(\pi_2)(-1)$ s.t. $\pi''_3$ does not contain new instance creation of $c$ and if any loop trailer occurs in $\pi''_3$, it only occurs once. By definition of abs, we have that abs$(\pi''_3 \cdot \pi'_3 \cdot \pi''_3) = \pi' \cdot \pi'_3$.

By Lemma 5.1.5, we have that $\hat{\pi}' \cdot \pi'_3 \cdot \pi''_3 \in A_{out}(a)$ and $\exists \pi''_4$ from $(\pi')(-1)$ to $(\pi_2)(-1)$ s.t. $\pi''_4$ does not contain new instance creation of $c$ and if any loop trailer occurs in $\pi''_4$, it only occurs once. By definition of abs, we have that abs$(\pi''_4 \cdot \pi''_4) = \pi' \cdot \pi''_4$. By Lemma 5.1.5, we have that $\hat{\pi}' \cdot \pi''_4 \in A_{out}(a)$. And by definition of Dyn, we have that $\text{Dyn}(c, \hat{\pi}' \cdot \pi''_4) = \hat{\pi}'$ and Proc($\hat{\pi}' \cdot \pi''_4$) $\neq$ Proc($\hat{\pi}' \cdot \pi''_4$).

\textbf{Theorem 5.1.14 ONE-SENDER SOUNDNESS}

Given a program $p$ and a channel $c$, if $\exists t \in \text{Trace}(p)$ and $\exists k = c@\pi$ in $t$ s.t. $\exists \pi_1, \pi_2 \in \text{Sends}_t(c@\pi)$ s.t. Proc($\pi_1$) $\neq$ Proc($\pi_2$), then $\exists \hat{c}$, $\exists \pi_1, \pi_2 \in \tilde{S}_c$ s.t. $\pi_1 \neq \pi_2$ and Topology$_c(\pi_1, \pi_2) = m$.

\textbf{Proof: By Lemma 5.1.13 and the definition of Topology$_c$, proof is similar to the proof of Theorem 5.1.11.}

\textbf{Theorem 5.1.15 ONE-RECEIVER SOUNDNESS}

Given a program $p$ and a channel $c$, if $\exists t \in \text{Trace}(p)$ and $\exists k = c@\pi$ in $t$ s.t. $\exists \pi_1, \pi_2 \in$
Recvs_{\ell}(c\pi) s.t. \text{Proc}(\pi_1) \neq \text{Proc}(\pi_2), then \exists \pi_1, \pi_2 \in \hat{S}_c s.t. \pi_1 \neq \pi_2 and Topology_c(\hat{R}_c) = m

Proof: This is similar to the proof of Theorem 5.1.14. ■

5.2 Concurrency Analysis

In this section, we show that our concurrency analysis is safe. Before stating the theorem and proof, we clarify the necessary fundamental notions first.

Definition 5.2.1 Given a program p and two statements s_1, s_2 of p, we say that s_1 executes before s_2, represented as s_1 \in \text{Before}(s_2), if for any instance s'_2 of s_2 and any execution of p, all instances of s_1 must execute before s'_2.

Definition 5.2.2 Given a program p and two statements s_1, s_2 of p, we say that s_1 executes after s_2, represented as s_1 \in \text{After}(s_2), if for any instance s'_2 of s_2 and any execution of p, all instances of s_1 must executes after s'_2.

Definition 5.2.3 Given a program p and two statements s_1, s_2 in p, we say that s_1 and s_2 might execute concurrently, represented as s_1 \in \text{MEC}(s_2), if there exists an execution of p and instances s'_1 and s'_2 of s_1 and s_2 s.t. s'_1 and s'_2 execute concurrently.

Lemma 5.2.1 Given a program p and two statements s_1, s_2 in p, if s_1 \in \text{MEC}(s_2), then s_2 \in \text{MEC}(s_1).

Proof: This is obvious by definition. ■

Definition 5.2.4 Given a program p and two statements s_1, s_2 of p, we say that s_1 and s_2 are ordered, represented as s_1 \in \text{Ordered}(s_2), if for any execution of p and any instances s'_1 and s'_2 of s_1 and s_2, s'_1 and s'_2 can not execute concurrently.
Lemma 5.2.2 Given a program $p$ and two statements $s_1, s_2$ in $p$, if $s_1 \in \text{Ordered}(s_2)$, then $s_2 \in \text{Ordered}(s_1)$.

Proof: This is obvious by definition.

Lemma 5.2.3 Given a program $p$ and any statement $s$ in $p$, Let $S$ be the set of all statements in $p$. We have $S = \text{MEC}(s) \cup \text{Ordered}(s)$ and $\text{MEC}(s) \cap \text{Ordered}(s) = \emptyset$.

Proof: This is obvious by definition of $\text{Ordered}$ and $\text{MEC}$.

Definition 5.2.5 Given a program $p$, a statement $s_1$ with label $a_1$, and a statement $s_2$ with label $a_2$, we say $s_1 \in B_{4_{ctl}}(s_2)$, if and only if $\forall t \in \text{Trace}(p)$, $\forall \pi_1 \cdot a_1 \in t$ and $\forall \pi_2 \cdot a_2 \in t$ s.t. $\pi_1 \cdot a_1 \prec \pi_2 \cdot a_2$.

$B_{4_{ctl}}$ is a relation between statements in a thread or between a statement in a parent thread and a statement in a child thread.

Lemma 5.2.4 Given a program $p$ and a statement $s$ in $p$, $B_{4_{ctl}}(s) \subseteq \text{Before}(s)$.

Proof: This is obvious by definition of $\text{Before}$ and $B_{4_{ctl}}$.

$\text{Before}(s)$ may contain statements not in $B_{4_{ctl}}(s)$.

Lemma 5.2.5 Given a program $p$ and a statement $s$ with label $a$ in $p$, $s \in \text{MEC}(s)$ if and only if $\exists t \in \text{Trace}(p)$ and $\exists \pi_1 \cdot a, \pi_2 \cdot a$ in $t$ s.t. $\text{Proc}(\pi_1 \cdot a) \neq \text{Proc}(\pi_2 \cdot a)$.

Proof: This is obvious by definition of $\text{MEC}$.

Definition 5.2.6 Given a program $p$, a statement $s_1$ with label $a_1$ and a statement $s_2$ with label $a_2$ in $p$, we say $s_1 \in \text{Loop}(s_2)$, if and only if $\exists \pi_1, \pi_2, \pi_3$ in $G$ s.t. $\pi_1 \cdot a_2 \cdot \pi_2 \cdot a_1 \cdot \pi_3 \cdot a_2$ is in $t$. 
Intuitively, a statement is in $\text{Loop}(s)$ if and only if it is in a loop containing $s$.

**Definition 5.2.7** Given a program $p$ and a statement $s$ with label $a$, we define $\text{TPath}(a) = \{ \pi \cdot a \mid \exists t \in \text{Trace}(p) \text{ and } \pi \cdot a \in t \}$.

**Definition 5.2.8** Given a program $p$ and a statement $s$ with label $a$, we define $\text{MightB}_4\text{ctl}(s) = \{ s' \mid s'$ is the statement with label $a'$ s.t. $\forall \pi \in \text{TPath}(a), a' \in \pi \}$

**Definition 5.2.9** Given a program $p$ and ECFG of $p$, $T(p)$ is a set of all statements in $p$ such that there are two control paths to the statement with different thread id or one is the prefix of the other.

**Lemma 5.2.6** Given a program $p$, a statement $s_1$ with label $a_1$ and a statement $s_2$ with label $a_2$, if $s_1 \in \text{MightB}_4\text{ctl}(s_2)$, $s_1 \notin \text{MEC}(s_1)$, $s_2 \notin \text{MEC}(s_2)$ and $s_1 \notin \text{Loop}(s_2)$ or if $s_1 \in \text{MightB}_4\text{ctl}(s_2)$, $s_1 \notin T(p)$, $s_2 \in \text{MEC}(s_2)$ and $s_1 \notin \text{Loop}(s_2)$, then $s_1 \in \text{B}_4\text{ctl}(s_2)$.

**Proof:** Suppose that $\exists t \in \text{Trace}(p)$, $\pi_1 \cdot a_1, \pi_2 \cdot a_2 \in t$ s.t. $\pi_1 \cdot a_1 \not\preceq \pi_2 \cdot a_2$. Because $s_1 \in \text{MightB}_4\text{ctl}(s_2)$, we have $\exists \pi_1', \pi_2'$ s.t. $\pi_2 = \pi_1' \cdot a_1 \cdot \pi_2'$, $\pi_1 \not\preceq \pi_1'$, $\pi_1 \not\preceq \pi_1'$ and $a_1 \not\in \pi_2'$. In a trace, however, a label occurs more than twice only when in the following cases.

i) Two threads execute $a_1$ concurrently.

This is contradictory to $s_1 \notin \text{MEC}(s_1)$ and $s_1 \notin T(p)$.

ii) One thread execute $a_1$ sequentially.

In this case, $s_1$ must be in the set $T(p)$. Because $\pi_1 \not\preceq \pi_1'$, we have that $\pi_1' \cdot a_1 \not\prec \pi_1 \cdot a$. By $a_1 \not\in \pi_2'$, we have that $\pi_1' \cdot a_1 \pi_2' \cdot a_2 \not\prec \pi_1$, because $a_1$ is executed sequentially by the same thread. Thus $\exists \pi_1''$ s.t. $\pi_1 = \pi_1' \cdot a_1 \pi_2' \cdot a_2 \cdot \pi_1''$. Then we have $\pi_1 = \pi_1' \cdot a_1 \cdot \pi_2' \cdot a_2 \cdot \pi_1'' \cdot a_1$ in $t$. There is a loop in this control path. So $\pi_1' \cdot a_1 \cdot \pi_2' \cdot a_2 \cdot \pi_1'' \cdot a_1 \cdot \pi_2' \cdot a_2 \cdot \pi_1'' \cdot a_1$ is a
valid control path in $t$ and is contradictory to $s_1 \notin \text{Loop}(s_2)$

So $s_1 \in B_{\text{ctl}}(s_2)$. $\blacksquare$

**Definition 5.2.10** Given a program $p$, a statement $s_1$ with label $a_1$, and a statement $s_2$ with label $a_2$, we say $s_1 \in \text{After}_c(tl)(s_2)$, if and only if $\forall t \in \text{Trace}(p)$, $\forall \pi_1 \cdot a_1 \in t$ and $\forall \pi_2 \cdot a_2 \in t$ s.t. $\pi_1 \cdot a_2 \prec \pi_2 \cdot a_1$.

**Lemma 5.2.7** Given a program $p$ and a statement $s$ in $p$, $\text{After}_c(tl)(s) \subseteq \text{After}(s)$.

**Proof:** This is obvious by definition of $\text{After}$ and $\text{After}_c(tl)$. $\blacksquare$

**Definition 5.2.11** Given a program $p$, a statement $s$ with label $a$ and an ECFG $G$ of $p$, we define $\text{TPath}'(a) = \{ \pi' \mid \exists \pi \cdot a \pi' \text{ in } G \}$.

**Definition 5.2.12** Given a program $p$ and a statement $s$ with label $a$, we define $\text{Might}_c(tl)(s) = \{ s' \mid s'$ is the statement with label $a'$ s.t. $\forall \pi \in \text{TPath}'(a)$, $a' \in \pi \}$

**Lemma 5.2.8** Given a program $p$, a statement $s_1$ with label $a_1$ and a statement $s_2$ with label $a_2$, if $s_1 \in \text{Might}_c(tl)(s_2)$, $s_1 \notin \text{MEC}(s_1)$, $s_2 \notin \text{MEC}(s_2)$ and $s_1 \notin \text{Loop}(s_2)$ or if $s_1 \in \text{A}_c(tl)(s_2)$, $s_1 \notin T(p)$, $s_2 \in \text{MEC}(s_2)$ and $s_1 \notin \text{Loop}(s_2)$, then $s_1 \in \text{After}_c(tl)(s_2)$.

**Proof:** This is similar to the proof of Lemma 5.2.6. $\blacksquare$

**Definition 5.2.13** Given a program $p$, a statement $s_1$ with label $a_1$, and a statement $s_2$ with label $a_2$, we say $s_1 \in \text{Order}_c(tl)(s_2)$, if and only if $\forall t \in \text{Trace}(p)$, $\forall \pi_1 \cdot a_1 \in t$ and $\forall \pi_2 \cdot a_2 \in t$ s.t. either $\pi_1 \cdot a_2 \prec \pi_2 \cdot a_1$ or $\pi_1 \cdot a_1 \prec \pi_2 \cdot a_2$.

**Lemma 5.2.9** Given a program $p$ and a statement $s$ in $p$, $\text{Order}_c(tl)(s) \subseteq \text{Order}(s)$.

**Proof:** This is obvious by definition of $\text{After}$ and $\text{After}_c(tl)$. $\blacksquare$
**Definition 5.2.14** Given a program $p$ and a statement $s$ with label $a$, we define $\text{MightA}_{\text{ctl}}(s) = \{s' \mid s' \text{ is the statement with label } a' \text{ s.t. } \forall \pi \in \text{TPath}'(a), a' \in \pi\}$

**Lemma 5.2.10** Given a program $p$, a statement $s_1$ with label $a_1$ and a statement $s_2$ with label $a_2$, if $s_2 \in \text{MEC}(s_2)$ and $s_1 \in B_4_{\text{ctl}}(s_2 \cup \text{After}_{\text{ctl}}(s_2))$, or if $s_2 \notin \text{MEC}(s_2)$, $s_1 \notin \text{MEC}(s_1)$, $s_1 \in \text{MightB}_{4_{\text{ctl}}}(s_2)$ or $s_1 \in \text{MightA}_{\text{ctl}}(s_2)$, then $s_1 \in \text{Order}_{\text{ctl}}(s_2)$.

**Proof:** This is similar to the proof of Lemma 5.2.6.

**Lemma 5.2.11** Given a program $p$, a statement $s_1$ with label $a_1$ and a statement $s_2$ with label $a_2$, if $a_1 \in \text{Common}(A_{\text{in}}(s_2))$, then $s_1 \in \text{MightB}_{\text{ctl}}(s_2)$.

**Proof:** This is obvious by definition.

**Lemma 5.2.12** Given a program $p$, a statement $s_1$ with label $a_1$ and a statement $s_2$ with label $a_2$, if $a_1 \in \text{Common}(\text{Succ}_n(\text{Exit}_n))$, then $s_1 \in \text{MightA}_{\text{ctl}}(s_2)$.

**Proof:** This is obvious by definition.

**Theorem 5.2.13** Given a program $p$ and a statement $s$ with label $a$ in $p$, we have that $B_{\text{ctl}}(a) \subseteq \{n \mid s' \in B_4_{\text{ctl}}(s) \text{ and } s' \text{ is with label } n\}$, $A_{\text{ctl}}(a) \subseteq \{n \mid s' \in \text{After}_{\text{ctl}}(s) \text{ and } s' \text{ is with label } n\}$, $\text{Order}(a) \subseteq \{n \mid s' \in \text{Order}_{\text{ctl}}(s) \text{ and } s' \text{ is with label } n\}$.

**Proof:** This is obvious by Lemma 5.2.11, Lemma 5.2.12, Lemma 5.2.6, Lemma 5.2.8 and Lemma 5.2.10.

**Theorem 5.2.14** Given a program $p$ and a statement $s$ with label $a$ in $p$, data flow equations in Figure 4.7 have a fixed point solution that is a meet over path solution.
Proof: The equations in Figure 4.7 fall in the standard monotone dataflow analysis framework. Thus these equations have a fixed point solution and it is a meet over path solution.

We also show that in monitor thread detection and transformation, we preserve the fact that \texttt{recv\&send Graph} acts like a lock-protected section and all communications with the section is serialized.

**Definition 5.2.15** Given an \( ECFG(V, E) \) and a monitor thread \( G'(V', E', t) \) of \( G \), Let \( G''(V'', E'', r, S) \) be the \texttt{recv\&send Graph} of \( G' \). We say that one thread enters \( G'' \), if it finishes communication with \( r \) but not finishes communication with the node in \( S \).

**Theorem 5.2.15** Monitor Transformation Correctness

Given a program \( p \), an \( ECFG(V, E) \) of \( p \) and a monitor thread \( G'(V', E', t) \) of \( G \), Let \( G''(V'', E'', r, S) \) be the \texttt{recv\&send Graph} of \( G' \). Then we have that in any execution, no threads in \( p \) can enter \( G'' \) concurrently.

Proof: By definition of monitor thread 4.2.11, we have that \( r \) is a node that represents a receive on a fan-in channel. By definition of fan-in channel, we know that no two instances of \( r \) can execute concurrently. So in any trace \( t \in \text{Trace}(p) \), \( \forall \pi_1 \cdot r, \pi_2 \cdot r \in t \), we have that \( \text{Proc}(\pi_1 \cdot r) = \text{Proc}(\pi_2 \cdot r) \). Thus \( \forall \pi_1 \cdot r, \pi_2 \cdot r \in t \), we have that \( \text{Proc}(\pi_1 \cdot r) \prec \text{Proc}(\pi_2 \cdot r) \) or \( \text{Proc}(\pi_2 \cdot r) \prec \text{Proc}(\pi_1 \cdot r) \). Again by definition of monitor thread, we have that there is no other concurrency statements in \( G'' \). So we have that for any \( s \in S \), \( \forall \pi_1 \cdot s, \pi_2 \cdot s \in t \), we have that \( \text{Proc}(\pi_1 \cdot s) \prec \text{Proc}(\pi_2 \cdot s) \) or \( \text{Proc}(\pi_2 \cdot s) \prec \text{Proc}(\pi_1 \cdot s) \). And by definition of monitor thread, for any trace \( t \in \text{Trace}(p) \), \( \exists \pi \) s.t. \( \pi = \pi' \cdot r \cdot \pi'' \cdot r \) and \( \pi'' \) does not contain any node in \( S \), and for any trace \( t \in \text{Trace}(p) \), \( \exists s, s' \in S \), \( \pi = \pi' \cdot s \cdot \pi'' \cdot s' \) and \( \pi'' \) does not contain \( r \). Thus the receive node and send node in \( G'' \) are interleaving. So no threads in \( p \) can enter \( G'' \) concurrently. 

\[102\]
CHAPTER 6
IMPLEMENTATION

In this chapter, we discuss the implementation of trace flow analysis for message passing programs first. Then we discuss the implementation of the specialized communication primitives. As we describe in Chapter 3, the traceflow analysis consists of three components. The first step is to do type-sensitive control flow analysis. The second step is to construct extended control flow graph based on the information from the first step. The third step is to do traceflow analysis based on ECFG constructed in the second step. We discuss the first step in details in the following section. We skip the implementation of ECFG construction, since it is straightforward according to description in section 3.4. Then we discuss the final step in section 6.2.

6.1 The Type-Sensitive CFA Algorithm

In this section, we present the details of our type-sensitive CFA algorithm. For our notation, we use SML syntax extended with mathematical notation such as set operations, and the ∨ operation on approximate values. We use the notation $\llbracket e \rrbracket$ to denote an object-language syntactic form $e$ and $\mathcal{V}[x \mapsto v]$ to denote the functional update of an approximation (likewise for $C$ and $R$).

Expressions are analysed by the $\text{cfaExp}$ function, whose code is given in Figure 6.1 and Figure 6.2.
fun cfaExp \((M, A as (V, C, R), [x])\) = 
  if \(x \in \text{FUNID}\) orelse \(x \in \text{CHANID}\) 
  then \((A, \{x\})\) 
  else \((A, V(x))\)
| cfaExp \((M, A, [\bullet])\) = \bullet
| cfaExp \((M, A, [\text{let } x = e_1 \text{ in } e_2])\) = let 
  val ((V, R, T), v) = cfaExp \((M, A, [e_1])\)
  val \(V = V[x \mapsto V(x) \lor v]\)
  in 
  cfaExp \((M, (V, R), [e_2])\)
end
| cfaExp \((M, A, [\text{fun } f(x) = e_1 \text{ in } e_2])\) = 
  cfaExp \((M, A, [e_2])\)
| cfaExp \((M, A, [e_1 e_2])\) = let 
  val \((A, v_1) = cfaExp \((M, A, [e_1])\)\)
  val \((A, v_2) = cfaExp \((M, A, [e_2])\)\)
  in 
  apply \((M, A, v_1, v_2)\)
end

Figure 6.1: CFA for expressions Part I

This function takes the set of active functions, an approximation triple, and an syntactic expression as arguments and returns updated approximations and a value that approximates the result of the expression.

The escape function records the fact that a value escapes into the wild. If the value is a set of known functions, then we apply them to the appropriate top value; and if it is a tuple, we record that its subcomponents are escaping. The escape function also takes the set of currently active functions as its first argument.
| cfaExp (M, A, [[e1, e2]]) = let
  val (A, v1) = cfaExp (M, A, [e1])
  val (A, v2) = cfaExp (M, A, [e2])
  in
  (A, (v1, v2))
end |
| cfaExp (M, A, [#ie]) = let
  val (A, (v1, ..., vn)) = cfaExp (M, A, [e])
  in
  (A, vi)
end |
| cfaExp (M, A, [chan c in e]) = cfaExp(M, A, [e]) |
| cfaExp (M, A, [spawn e]) = (cfaExp(M, A, [e]); •) |
| cfaExp (M, A, [send(e1, e2)]) = let
  val (A, v1) = cfaExp (M, A, [e1])
  val (A, v2) = cfaExp (M, A, [e2])
  in
  send (M, A, v1, v2)
end |
| cfaExp (M, A, [recv e]) = let
  val (A, v) = cfaExp (M, A, [e])
  in
  receive (A, v)
end |

Figure 6.2: CFA for expressions Part II
fun escape (M, A, F) = let
  fun esc (f:τ₁→τ₂, A) = let
    val (A, v) = applyFun(M, A, f, U(τ₁))
    in A end
  in fold esc A F end

| escape (M, (V, C, R), C) = let
  fun esc (e:τ, C) = C[c↦[C(c)∨τ]]
  in (V, fold esc C C, R) end

| escape (M, A, ⟨v₁,v₂⟩) = let
  val A = escape (M, A, v₁)
  val A = escape (M, A, v₂)
  in A end

| escape (_, A, v) = A

For function applications, we use the apply helper function to record the fact that an approximate function value is being applied to a approximate argument. When the approximation is a set of known functions, then we apply each function in the set to the argument compute the join of the results. When the function is unknown (i.e., a top value), then the argument is marked as escaping and the result is the top value for the function’s range.

fun apply (M, A, F, arg) = let
  fun applyf (f, (A, res)) = let
    val (A, v) = applyFun (M, A, f, arg)
    in (A, res ∨ v) end
  in fold applyf (V, T) F end

| apply (M, A, τ₁→τ₂, v) = let
  val A = escape (M, A, v)
  in (A, τ₂) end
The applyFun function analyses the application of a known function $f$ to an approximate value $v$. The first argument to applyFun is a set $M \in 2^{\text{FUNID}}$ of known functions that are currently being analysed; if $f$ is in this set, then we use the approximation $R$ instead of recursively analyzing the $f$’s body. This mechanism is necessary to guarantee termination when analyzing recursive functions. We assume the existence of the function bindingOf that maps known function names to their bindings in the source.

```ml
fun applyFun (M, A as (V, C, R), f, v) =
  if f \in M
  then (A, R(f))
  else let
    val [fun f(x) = e] = bindingOf (f)
    val V = V[x \mapsto [V(x) \lor v]]
    val ((V, C, R), r) =
      cfaExp (M \cup \{f\}, (V, C, R), [e])
    val R = R[f \mapsto [R(f) \lor r]]
  in
    ((V, C, R), r)
  end

The send function is used to analyse message-send operations.

```ml
fun send (M, (V, C, R), C, v) = let
  fun esc (c, C) = C[c \mapsto [C(c) \lor v]]
  in
    ((V, fold esc C C, R), •)
  end
| send (M, A, _, v) = (escape (M, A, v), •)
```
6.2 Abstract Trace Analysis Algorithm

In this section, we cover the challenges and solutions when implementing the abstract trace analysis. We discuss implementation of valid trace and redundant trace first.

In section 3.5, we define $A_{\text{out}}$ as follows.

$$A_{\text{in}}(a) = \begin{cases} 
\epsilon & \text{if } a \in \text{Program Entry} \\
\bigcup_{a' \in \text{pred}(a)} A_{\text{out}}(a') & \text{if } a \in \text{Function Entry} \\
\text{valid}_a\left(\bigcup_{a' \in \text{pred}(a)} A_{\text{out}}(a')\right) & \text{if } a \in \text{RETNODE} \\
\bigcup_{a' \in \text{pred}(a)} A_{\text{out}}(a') & \text{otherwise}
\end{cases}$$

$$A_{\text{out}}(a) = \{\pi \cdot a \mid \pi \in A_{\text{in}}(a)\}$$

where $\text{valid}_a$ is used to exclude invalid traces w.r.t. return node $a$ and $\bigcup_a$ is used to exclude redundant traces w.r.t. $a$.

$$S_1 \uplus_a S_2 = (S_1 \cup S_2) \setminus \{\pi \mid \pi \text{ is redundant w.r.t. } a\}$$

The straightforward implementation of $A_{\text{out}}$ is to iterate over the program points until the information stabilizes. But it is not efficient. A better approach is to recursively add valid and non-redundant traces to program points, which is show in Figure 6.3. Based on this implementation, we can further simplify the condition of a valid trace and redundant trace w.r.t. $a$, which are shown in Figures 6.4 and 6.5 respectively.
collect(currentNode, currentTrace) {
  if (currentNode is a function entry node)
    and (currenttrace is redundant)
  then return
  newTrace = append currentNode to currentTrace
  for each nextNode in Succ(currentNode) do
    if (nextNode is a return node)
      then
        if isValid(nextNode, newTrace)
          then collect(nextNode, newTrace)
        else
          collect(nextnode, newTrace)
  endfor
}

Figure 6.3: Algorithm for abstract traceflow analysis

isValid(retnode, trace) {
  callnode = the corresponding call node of retnode
  callnumber = 0
  returnnumber = 0
  for each node in trace starting from right to left do
    if (node is a call node)
      then callnumber+=1
    if (node is a return node)
      then returnnumber+=1
    if (callnumber -returnnumber == 1)
      then
        return (node == callnode)
  endfor
}

Figure 6.4: Algorithm for testing for a valid trace
isRedundant(entrynode, trace) {
    exitnode = the corresponding function exit point of entrynode
    entrylevel = 0
    for each node in trace starting from right to left do
        if (node == exitnode)
            then entrylevel -= 1
        if (node == entrynode)
            then entrylevel += 1
        if (entrylevel == 3)
            then return true
    endfor
    return false
}

Figure 6.5: Algorithm for redundant trace

6.3 Implementation of Specialized CML Primitives

Concurrent ML is a higher-order concurrent language that is embedded into Standard ML [Rep91, Rep99]. It supports a rich set of concurrency mechanisms, but for purposes of this dissertation we focus on the core mechanism of communication over synchronous channels. The interface to these operation is

    val spawn : (unit -> unit) -> unit
    type 'a chan
    val channel : unit -> 'a chan
    val recv : 'a chan -> 'a
    val send : ('a chan * 'a) -> unit

The spawn operation creates new threads, the channel function creates new channels, and the send and recv operations are used for message passing. Because channels are synchronous, both the send and recv operations are blocking.

One major limitation of CML is that its implementation is single-threaded and cannot take advantage of multicore or multiprocessor system. In our previous papers [RX08, RRX09], we describe a solution to this problem by presenting an optimistic-concurrency protocol for CML synchronization. Although it is a purpose-built optimistic concurrency
protocol designed for correctness and performance on shared-memory multiprocessors, it must be general enough to support communication involving multiple sending and receiving processes transmitting multiple messages in arbitrary contexts. This generality requires the implementation of channel communication primitives to use locks to ensure correctness. In certain cases, however, that channels are used in restricted ways, such as point-to-point, fan-in and fan-out message passing, we may reduce significant amount of overhead by using specialized implementation of communication primitives.

We present our implementation using SML syntax with a few extensions. For simplicity, we elide several aspects of the actual implementation, such as thread IDs, processor affinity and the support for events. In order to set the stage for the specialized implementation, we review some aspects of the general parallel implementation in the following sections. And we use PCML to refer to the general parallel implementation.

6.3.1 Preliminaries

PCML uses queues to track pending messages and waiting threads in channels. We omit the implementation details here, but give the interface to the queue operations that we use as follows:

```sml
type 'a queue
val queue : unit -> 'a queue
val isEmptyQ : 'a queue -> bool
val enqueue : ('a queue * 'a) -> unit
val dequeue : 'a queue -> 'a option
val atomicEnqueue : ('a queue * 'a) -> unit
val atomicDequeue : 'a queue -> 'a option
val dequeueMatch : ('a queue * ('a -> bool)) -> 'a option
```

These operations have the expected single-threaded semantics.

As in the uniprocessor implementation of CML, PCML use first-class continuations to implement threads and thread-scheduling. The continuation operations have the following specification:
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b

We represent the state of a suspended thread as a unit continuation

type thread = unit cont

The interface to the scheduling system is represented by two atomic operations:

val enqueueRdy : thread -> unit
val dispatch : unit -> 'a

The first enqueue a ready thread in the scheduling queue and the second transfers control to the next ready thread in the scheduler.

The implementation of PCML also relies on the atomic compare-and-swap instruction and assume the existence of spin locks. These low-level operations have the following interface:

val CAS : ('a ref * 'a * 'a) -> 'a
type spin_lock
val spinLock : spin_lock -> unit
val spinUnlock : spin_lock -> unit

In PCML, channels are represented by a spin lock and a pair of queues as follows:

datatype 'a chan = Ch of {
  lock : spin_lock,
  sendq : ('a * unit cont) queue,
  recvq : 'a cont queue
}

Here, sendq is for waiting senders and recvq for waiting receivers. And spin_lock is used to protect the queues to avoid race when multiple senders and receivers are attempting to modify the queues at the same time.
fun send (Ch{lock, sendq, recvq}, msg) = let
    val _ = spinLock
    in case dequeue recvq
    of SOME recvK => (spinUnlock lock;
        enqueueRdy sendK;
        throw recvK msg)
    | NONE => (callcc (fn sendK =>
        enqueue (sendq, (msg, sendK));
        spinUnlock lock);
        dispatch())
    end

Figure 6.6: The reference implementation of general send

fun recv (Ch{lock, sendq, recvq}) = let
    val _ = spinLock
    in case dequeue sendq
    of SOME (msg, sendK) => (spinUnlock lock;
        enqueueRdy sendK;
        msg)
    | NONE => (callcc (fn recvK =>
        enqueue (recvq, recvK);
        spinUnlock lock);
        dispatch())
    end

Figure 6.7: The reference implementation of general recv
6.3.2 The General Parallel Implementation

The general parallel implementation of send operation and recv operation is shown in Figures 6.6 and 6.7 respectively.

From the reference implementation of general send and recv, we can see that every operation attempted on a channel will cause contention of competing to acquire the lock associated with the channel. The overhead is significant for programs that communicate frequently. If preemption is allowed and the process that holds the lock get preempted, other processes that are attempting to communicate over the channel will be simply spinning and wasting the resource. In some special cases, however, that channels are used in restricted ways, such as point-to-point, fan-in and fan-out message passing, spin_lock may be not necessary to ensure correctness.

6.3.3 A Reference Implementation of Fan-In and Fan-Out Channels

The underlying data structure of channels in PCML is a FIFO queue. Spin locks are used to protect accesses to the data structure, but this does not scale well as more processes access the queue at the same time. When one of the process acquires the lock, all other processes will spin and wait until the process releases the lock. Fortunately, there is an alternative way to perform operations on shared queues without the need to use locks. And in some special cases that channels are used in restricted ways, such as point-to-point, fan-in and fan-out communication patterns, we may perform operations on the underlying queues without the need to use locks.

In our previous work, we presented a specialized implementation for one-to-one channel [RX07]; i.e., a channel that has at most one sender and one receiver thread at any time and is not used in a choice context. Such a channel can only be in one of three distinct states:
**EMPTY** — neither thread is waiting for communication,

**RECV** — the receiver thread is waiting for the sender, or

**SEND** — the sender thread is waiting for the receiver.

Furthermore, the possible state transitions are restricted by the channel’s usage pattern as is illustrated by the following state diagram:

Thus, the **send** operation can be implemented under the assumption that the channel’s state must be either **EMPTY** or **RECV**. A single atomic swap instruction can be used to atomically swap the state with **SEND**. If the original state is not **EMPTY**, then it must be **RECV** and the **send** operation can set the state to **EMPTY** and be completed without any further synchronization. ¹

In the general implementation, when a **send** is attempted on a channel, the **recvq** of the channel is checked to find any available receiver. If the **recvq** is empty, the sender has to be enqueued into the **sendq**. If there is a available receiver, the receiver will be dequeued from the **recvq** and complete the synchronization with the attempted sender. It is possible that there are multiple processes that attempt to send on a general channel at the same time, so any attempt to operate on a general channel should be protected by a lock. If a channel has a fan-in/fan-out communication pattern, however, things may be simplified, and the channel lock may be avoided. Take fan-in channels as an example. There is at most one receiver at any time for a fan-in channel. If there are available senders in the

¹ Of course, there is also the need to schedule the receiver thread for execution, but that cost would be required no matter how the channel protocol is implemented.
sendq, it is guaranteed that the receiver is the only one that attempts to dequeue the first available sender in the sendq, and if there are any attempts to send on the channel at the same, the senders will be appended at the end of the sendq. Thus, in the case that the sendq is not empty, an attempt to send or receive does not need to acquire the channel’s lock to operate on the channel. If the sendq is empty, the channel’s lock still needs to be acquired because there is a race condition between a sender and the receiver. With careful observation, however, we find that we can merge the sendq and the recvq into a unified waiting queue, because it is not possible that there are waiting senders and waiting receivers at the same time. And with this unified waiting queue, the fan-in channel can be further optimized. Now, the previous case that the sendq is empty becomes the case that the unified waiting queue is empty. Thus, the race between the sender and the receiver could be avoided by using an atomic instruction on the head of the waiting queue. For fan-out channels, it is similar.

In this work, we present a specialized lock-free implementation of communication primitives for fan-in and fan-out channels. In this implementation, a channel is represented as a FIFO queue. The detailed implementations of send and recv primitives for fan-in and fan-out channels are shown in Figures 6.8, 6.9, 6.10, and 6.11 in SML like syntax. A typical lock-free algorithm on a FIFO queue is that the queue is represented as a singly linked list, each process updates a local copy of head or tail of the queue and apply the changes with an atomic instruction such as compare-and-swap(CAS), test-and-set(TAS), etc [MM96]. There is an interesting problem about lock-free algorithms implemented with CAS. The problem is how and when the memory should be reclaimed when portion of a shared queue is dequeued. If the memory is reclaimed right after dequeue operation, this will cause problems since other processes might still have access to the resource. An typical scenario is that when a process try to update the head of the queue, another process dequeue
several items from the queue and then enqueue one item. If the enqueued item happens to
be the reclaimed dequeued head, then the first process would think that the queue has not
been updated by other process. This problem is known as ABA problem. Many solutions
to this problem are provided [MM96, Mic04]. The ABA problem is automatically solved
in the presence of garbage collector.

We have used stateless model checking to verify the implementation. Our approach is
based on the ideas of the CHESS model checker [MQ07], but we built our own tool tailored
to our problem [RRX09]. Our approach to model checking was to implement a virtual ma-
chine in SML that supported preemption, a scheduling infrastructure, and memory cells
with both atomic and non-atomic operations. The implementation of the virtual machine
operations are allowed to inject preemptions. We used SML/NJ’s first-class continuations
to implement a roll-back facility that allowed both the preempted and non-preempted exe-
cution paths to be explored. To keep the number of paths explored to a tractable number,
we bound the number of preemptions to 3 on any given trace. Our reference implementa-
tion was then coded as a functor over the virtual machine API. On top of this we wrote a
number of test cases that we ran through the checker. These tests required exploring over
million distinct execution traces.

Our experience with using this tool was very positive. We strongly recommend such
automated testing approaches to anyone developing concurrent language implementations.
Perhaps the best proof of its usefulness is that when we translated the reference implemen-
tation to Manticore runtime system, it worked “out of the box.”
fun send (Ch{q}, msg) = callcc (fn sendK => let
    fun trylp () = let
    val headpt = the head pointer of q
    val tail = the tail pointer of q
    val head = the head of q
    val tail = the tail of q
    val nextpt = the next pointer of tail
    val next = the next item of tail
    in case next
      of NONE => (case tail
            of a receive node with recvK => (if (head and tail is same)
                then let
                    val node = new queue item as a send node with (msg, sendK)
                    in
                    if CAS (nextpt, next, SOME node)
                        then dispatch ()
                        else trylp ()
                    end
            else (if CAS(headpt, head, tail)
                then (enqueueRdy (sendK);
                    throw recvK msg)
                else trylp ()))
            | not a receive node => let
                val node = new queue item as a send node with (msg, sendK)
                in
                if CAS(nextpt, next, SOME node)
                    then dispatch ()
                    else trylp ()
                end
            | SOME node => (CAS(tailpt, tail, node);
                trylp())
            end
        end
    in trylp() end
end)

Figure 6.8: The specialized implementation of send for fan-in channels
fun recv (Ch\{q}\}) = callcc (fn recvK => let
    fun trylp () = let
        val headpt = the head pointer of q
        val head = the head item of q
        val nextpt = the next pointer of head
        val next = the next item of head
    in case next
        of NONE => let
            val node = new queue item as a receive node with recvK
            in
                if CAS (nextpt, NONE, SOME node)
                    then dispatch ()
                    else trylp ()
            end
        | SOME node => (case node
            of a send node with (msg, sendK) => let
                val _ = update the head pointer of q to node
            in
                enqueueRdy (sendK);
                msg
            end
        )
    end
    in
    trylp()
end)

Figure 6.9: The specialized implementation of recv for fan-in channels
fun send (Ch\{q\}, msg) = callcc (fn sendK => let

fun trylp () = let
  val headpt = the head pointer of q
  val head = the head item of q
  val nextpt = the next pointer of head
  val next = the next item of head
  in case next
    of NONE => let
      val node = new queue item as a sender node with (msg, sendK)
      in
        if CAS (nextpt, NONE, SOME node)
        then dispatch ()
        else trylp ()
      end
    | SOME node => (case node
      of a receive node with recvK => let
        val _ = update the head pointer of q to node
        in
          enqueueRdy (sendK);
          throw recvK msg
        end
      )
    end
  end
in
  trylp()
end

Figure 6.10: The specialized implementation of \texttt{send} for fan-out channels
fun recv (Ch{q}) = callcc (fn recvK => let
  fun trylp () = let
    val headpt = the head pointer of q
    val tail = the tail pointer of q
    val head = the head of q
    val tail = the tail of q
    val nextpt = the next pointer of tail
    val next = the next item of tail
  in case next
    of NONE => (case tail
        of a send node with (msg, sendK) => (if (head and tail is same)
            then let
              val node = new queue item as
                a receive node with recvK
            in
              if CAS (nextpt, next, SOME node)
                then dispatch ()
                else trylp ()
            end
            else (if CAS(headpt, head, tail)
                then (enqueueRdy (sendK);
                    msg)
                else trylp ()))
        | not a send node => let
            val node = new queue item as
              a receive node with recvK
          in
            if CAS(nextpt, next, SOME node)
                then dispatch ()
                else trylp ()
          end)
        | SOME node => (CAS(tailpt, tail, node);
            trylp())
        end)
  in
    trylp()
  end
end

Figure 6.11: The specialized implementation of recv for fan-out channels
CHAPTER 7
PERFORMANCE

In Section 6.3.3, we discussed the implementation of communication primitives for general channels and a reference implementation for one-to-one channels, and presented a reference implementation for fan-in and fan-out channels. We have translated these implementations into continuation-based implementations as part of the Manticore system [FFR07]. In this chapter, we present some benchmark results for these specialized Manticore channel implementations. We compare these results with the general Manticore implementation. We also present some benchmark results for the monitor optimization.

The Manticore implementation is written in a low-level functional language that serves as one of the intermediate representations of our compiler. This language can be viewed as a stripped-down version of ML with a few extensions. Specifically, it supports first-class continuations via a continuation binder and it provides access to mutable memory objects and operations (including CAS).

The Manticore runtime system is designed to emphasize separation between processors. While this design helps with scalability, it does impose certain burdens on the implementation of the communication primitives. One aspect is that each processor has its own local scheduling queue, which other processors are not allowed to access. Thus to schedule a thread on a remote processor requires pushing it on a concurrent stack that each processor maintains (called the landing pad) and then waiting until the remote processor notices

1. Manticore’s surface language does not have mutable storage.
it and schedules it. The effect of this design is that message passing and remote thread creation have increased latency.

Our test system has four quad-core AMD Opteron 8380 processors running at 2.5GHz. Each core has a 512Kb L2 cache and each processor has a shared 6Mb L3 cache. The system has 32Gb of RAM and is running Debian Linux (kernel version 2.6.31.6-amd64). Each benchmark was run 30 times and we report the average wall-clock time and the standard deviation.

### 7.1 One-To-One Benchmark

The one-to-one benchmark measures the performance of specialized one-to-one implementation against the general implementation. The program involves two threads, where one thread sends 0.5 million messages to the other one. We measured two versions of the program: one that runs on a single processor and one that runs on two processors. In the two-processor case, each thread runs on its own processor. For each version, we report the average time, the speed ratio relative to the general implementation, the standard deviation and the number of messages that are sent per second. The results for this experiment are given in the following table.

<table>
<thead>
<tr>
<th>One-to-one benchmark on 1P</th>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Ratio</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>1.78</td>
<td>1.00</td>
<td>0.02</td>
<td>280,898</td>
<td></td>
</tr>
<tr>
<td>specialized</td>
<td>1.38</td>
<td>0.78</td>
<td>0.02</td>
<td>362,318</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One-to-one benchmark on 2P</th>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Ratio</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>general (2P)</td>
<td>11.67</td>
<td>1.00</td>
<td>0.33</td>
<td>42,844</td>
<td></td>
</tr>
<tr>
<td>specialized (2P)</td>
<td>10.33</td>
<td>0.89</td>
<td>0.18</td>
<td>48,402</td>
<td></td>
</tr>
</tbody>
</table>
As expected, this benchmark demonstrates that we will get speedups by replacing the general communication primitives with the specialized one-to-one communication primitives.

### 7.2 Primes Benchmark

The Primes benchmark computes the first 5000 prime numbers using the *Sieve of Eratosthenes* algorithm.

The benchmark measures the performance of specialized one-to-one implementation against the general implementation. The computation is structured as a pipeline of filter threads as each new prime is found, a new filter thread is added to the end of pipeline. Here, a pipeline between two filter threads has one-to-one communication topology. Thus, the pipeline can be implemented as a one-to-one channel rather than a general channel. We measured four versions of the program: one that runs on a single processor, one that runs on two processors, one that runs on four processors and one that runs on eight processors. For the multiprocessor version, the filters were assigned in a round-robin fashion. The results for this benchmark are given in Figure 7.1 and the speed-up curves are plotted in 7.2. We report the average time, the speed ratio relative to the general version and the standard deviation.

It should be understood that the single processor version takes less time than the multiprocessor version, because in the multiprocessor version, the total execution time is significantly increased by the overhead of scheduling threads on remote processors. Note that we see parallel speedup in this benchmark, and that the specialized version has slightly better scaling than the general version.
### Primes benchmark on 1P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>7.50</td>
<td>0.02</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>7.15</td>
<td>0.03</td>
<td>0.95</td>
</tr>
</tbody>
</table>

### Primes benchmark on 2P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>17.20</td>
<td>0.17</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>14.94</td>
<td>0.22</td>
<td>0.87</td>
</tr>
</tbody>
</table>

### Primes benchmark on 4P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>9.96</td>
<td>0.08</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>8.74</td>
<td>0.12</td>
<td>0.88</td>
</tr>
</tbody>
</table>

### Primes benchmark on 8P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>5.24</td>
<td>0.06</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>4.63</td>
<td>0.13</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Figure 7.1: Primes benchmark results

---

![Graph showing speedup curves for Primes benchmark](image)

Figure 7.2: Primes speed-up curves
<table>
<thead>
<tr>
<th>Fan-out benchmark on 1P</th>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>3.42</td>
<td>0.06</td>
<td>292,397</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>specialized</td>
<td>2.87</td>
<td>0.03</td>
<td>348,432</td>
<td>0.84</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fan-out benchmark on 2P</th>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>13.26</td>
<td>0.29</td>
<td>75,414</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>specialized</td>
<td>9.17</td>
<td>0.23</td>
<td>109,051</td>
<td>0.69</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fan-out benchmark on 5P</th>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>14.67</td>
<td>0.29</td>
<td>68,166</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>specialized</td>
<td>8.13</td>
<td>0.03</td>
<td>123,001</td>
<td>0.55</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fan-out benchmark on 9P</th>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>16.31</td>
<td>0.54</td>
<td>61,312</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>specialized</td>
<td>9.09</td>
<td>0.06</td>
<td>110,011</td>
<td>0.56</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3: Fan-out benchmark results

### 7.3 Fan-Out and Fan-In Benchmarks

The Fan-out benchmark measures the performance of the specialized fan-out implementation against the general implementation. The program involves one sender and eight receivers that receive 1 million messages in total. We measured four versions of the program: one that runs on a single processor, one that runs on two processors, one that runs on five processors and one that runs on nine processors. For the multiprocessor version, the sender is assigned to the first processor and the receivers are assigned evenly to the remaining processors. The results for this experiment are given in Figure 7.3 and the speed-up curves are plotted in Figure 7.4. For each version, we report the average time, the speed ratio relative to the general implementation, the standard deviation and the number of messages that are sent per second.

Similar to the fan-out benchmark, the Fan-in benchmark measures the performance of
Figure 7.4: Fan-out speed-up curves

the specialized fan-in implementation against the general implementation. The results for this experiment are given in Figure 7.5 and the speed-up curves are plotted in Figure 7.6. For each version, we report the average time, the speed ratio relative to the general implementation, the standard deviation and the number of messages that are sent per second.

As before, it should be understood that the single processor version takes less time than the multiprocessor version, because in the multiprocessor version, the total execution time is significantly increased by the overhead of scheduling threads on remote processors.

It should also be understood that we do not expect to see parallel speedup in the fan-out and fan-in speed-up curves, because the server thread acts as a serial bottleneck. The performance, however, does not degrade significantly after the two-processor version.

As we expected, the results of fan-out and fan-in benchmarks demonstrate that we get significant speedups. It should be understood that the single processor version does not show as significant speedup as the multiprocessor version, because there is less contention.
### Fan-in benchmark on 1P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>3.23</td>
<td>0.03</td>
<td>309,597</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>2.78</td>
<td>0.04</td>
<td>361,010</td>
<td>0.86</td>
</tr>
</tbody>
</table>

### Fan-in benchmark on 2P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>12.31</td>
<td>0.25</td>
<td>81,234</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>9.62</td>
<td>0.17</td>
<td>103,950</td>
<td>0.78</td>
</tr>
</tbody>
</table>

### Fan-in benchmark on 5P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>14.70</td>
<td>0.17</td>
<td>68,027</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>8.31</td>
<td>0.11</td>
<td>120,336</td>
<td>0.57</td>
</tr>
</tbody>
</table>

### Fan-in benchmark on 9P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Messages/sec.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>17.17</td>
<td>0.38</td>
<td>58,241</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>9.75</td>
<td>0.20</td>
<td>102,564</td>
<td>0.57</td>
</tr>
</tbody>
</table>

Figure 7.5: Fan-in benchmark results

Figure 7.6: Fan-in speed-up curves
for the sending threads to access the channel in the single processor version than in the multiprocessor version.

### 7.4 Monitor Benchmark

The monitor benchmark measures the performance of the optimized monitor transformation against the general monitor communication pattern and the optimized monitor communication by specialized communication primitives. The monitor-communication-pattern program involves a simple client-server protocol. The server provides a simple service on a fan-in service channel and handles the requests that are received on it. The server handles 5 million requests, where each client sends 500 requests and there are 10,000 clients in total. We measured four versions of the program: one that runs on a single processor, one that runs on two processors, one that runs on five processors and one that runs on nine processors. For the multiprocessor version, the server is assigned to the first processor and the clients are assigned evenly to the remaining processors. Note that because there is no server thread in the optimized monitor transformation, the clients are assigned evenly to the total processors in the multiprocessor version. The results for this experiment are given in Figure 7.7, and the speed-up curves are plotted in Figure 7.8. For each version, we report the average time, the speed ratio relative to the general implementation and the standard deviation.

As before, it should be understood that the single processor version takes less time than the multiprocessor version, because in the multiprocessor version, the total execution time is significantly increased by the overhead of scheduling threads on remote processors.

Our results show that the monitor version performs significantly better than the message-passing versions, although the difference decreases for clients running on multiple processors.
### Monitor benchmark on 1P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>6.06</td>
<td>0.04</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>5.53</td>
<td>0.03</td>
<td>0.91</td>
</tr>
<tr>
<td>monitor</td>
<td>1.22</td>
<td>0.59</td>
<td>0.20</td>
</tr>
</tbody>
</table>

### Monitor benchmark on 2P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>21.06</td>
<td>0.09</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>19.00</td>
<td>0.05</td>
<td>0.90</td>
</tr>
<tr>
<td>monitor</td>
<td>2.34</td>
<td>0.15</td>
<td>0.11</td>
</tr>
</tbody>
</table>

### Monitor benchmark on 5P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>23.25</td>
<td>0.28</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>19.32</td>
<td>0.25</td>
<td>0.83</td>
</tr>
<tr>
<td>monitor</td>
<td>7.74</td>
<td>0.24</td>
<td>0.33</td>
</tr>
</tbody>
</table>

### Monitor benchmark on 9P

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (sec.)</th>
<th>Std. Dev.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>23.32</td>
<td>0.26</td>
<td>1.00</td>
</tr>
<tr>
<td>specialized</td>
<td>19.42</td>
<td>0.21</td>
<td>0.83</td>
</tr>
<tr>
<td>monitor</td>
<td>8.62</td>
<td>0.55</td>
<td>0.37</td>
</tr>
</tbody>
</table>

Figure 7.7: Monitor benchmark results
Figure 7.8: Monitor speed-up curves
CHAPTER 8
CONCLUSION

8.1 Summary

Developing a static analysis for concurrent programs is a challenging problem. Developing a precise one in the presence of dynamic communication is more challenging.

In this dissertation, we present a new approach to static analysis for concurrent programs. The new approach computes a finite set of abstract control paths that reach each program point, and uses the set to approximates the set of all possible control paths during execution. With abstract trace analysis as a foundation, we can further conduct specific analysis for concurrent programs. In this dissertation, we have demonstrated how to conduct communication topology analysis and concurrency analysis based on abstract trace analysis.

Based on abstract trace analysis, abstract control paths can be used to represent dynamic channel instances and dynamic communications on different dynamic channel instances. We use the set of abstract control paths that reach the channel’s creation site to approximate the possible dynamic channel instances created in run-time and the set of abstract control paths that reach the channel’s communication sites to approximate the possible dynamic communications on the channel in run-time. We also present how to extract dynamic information from dynamic communications to approximate the underlying possible dynamic channel instances. In other words, given a dynamic communication on some channel \( c \), we can determine the possible relationship between this dynamic communication and
any dynamic instance of channel \( c \). This approach enables to distinguish different dynamic communications on different dynamic channel instances, which appears to be the first analysis that can do so to determine communication topology for concurrent programs. And it is this capability, which our previous work lacks, that makes this communication topology analysis produce more precise results than our previous work.

We have also demonstrated how to use abstract trace analysis to do concurrency analysis and presented an analysis to detect monitor communication pattern and do optimization transformation by replacing monitor threads with monitor functions and replacing communications with monitor threads with monitor function calls.

In this dissertation, we also present a lock-free implementation of communication primitives for fan-in and fan-out channels. By benchmark results based on Manticore, we show that we can get significant speedups by replacing the general communication primitives with the specialized communication primitives. Thus, our analysis and the transformation is necessary and beneficial.

### 8.2 Future Work

We have prototyped the abstract trace analysis and communication topology analysis for the simple concurrent language. The next step is to extend the analyses and incorporate them into Manticore system. Eventually, we plan to implement the analysis and optimization as a source-to-source tool for optimizing Manticore programs.

The communication topology analysis presented in this dissertation does not include a number of important CML features, such as non-deterministic choice and event combinator. By adding these features to the analysis enables us to further refine our characterization of known channels to distinguish between those that appear in a choice context versus those that do not.
We have also considered the question of modeling asynchronous (or buffered) message passing. Our dynamic semantics would have to be extended with a mechanism to trace "in-flight" messages, but this extension is not difficult. We believe that our analysis is already correct for buffered channels, since it is not sensitive to the order of messages.

One possible optimization to make abstract trace analysis more efficient is to ignore those non-concurrency parts of the graph, because the analysis is only concerned with the concurrent parts of the graph, such as process spawning, channel creation and channel communication. By removing these parts of the graph, nodes in the graph may have smaller sets of abstract control paths and the abstract control paths may be shorter.

In the future, we also plan to use communication topology information to further optimize Manticore programs by optimizing thread scheduling. One aspect of Manticore is that each processor has its own local scheduling queue, which other processors are not allowed to access. Thus, to schedule a thread on a remote processor requires pushing it on a concurrent stack and then waiting until the remote processor notices it and schedules it. The effect of this design is that message passing and remote thread creation or continuation have increased latency. By taking advantage of statically computing communication topology information, we may schedule threads that would heavily communicate with each other in the future in the same processor to avoid inter-processor overhead.
REFERENCES


[CYAD05] Ciancaglini, M.D., N. Yoshida, A. Ahern and S. Drossopoulou, A Distributed Object-Oriented Language with Session Types, LNCS, TGC 2005


[MLt] http://mlton.org/ConcurrentML.


[Sut05] Sutter, H. The free lunch is over. In Dr. Dobb’s Journal, 30(3), March 2005


