박 사 학 위 논 문

프로그램 실행 중 처리되지 않는 예외상황을 예측하는 방법

류 석 영 (柳 石 英)

전자전산학과 전산학전공

한국과학기술원

2001

프로그램 실행 중 처리되지 않는 예외상황을 예측하는 방법

Exception Analysis for Languages with Exceptions

Exception Analysis for Languages with Exceptions

Advisor : Professor Kwangkeun Yi

by

Sukyoung Ryu

Department of Electrical Engineering & Computer Science

Division of Computer Science

Korea Advanced Institute of Science and Technology

A thesis submitted to the faculty of the Korea Advanced Institute of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical Engineering & Computer Science Division of Computer Science

Daejeon, Korea

2001. 5. 31

Approved by

Professor Kwangkeun Yi

Major Advisor

프로그램 실행 중 처리되지 않는

예외상황을 예측하는 방법

류 석 영

위 논문은 한국과학기술원 박사 학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2001년 5월 31일

심사위원장 이 광 근 (인)

심사위원 신 성 용 (인)

심사위원 최 광 무 (인)

심사위원 차 성 덕 (인)

심사위원 최 창 선 (인)

사랑하는 부모님께 이 논문을 드립니다.

DCS 965808 류석 영. Sukyoung Ryu. Exception Analysis for Languages with Exceptions. 프로그램 실행 중 처리되지 않는 예외상황을 예측하는 방법. Department of Electrical Engineering & Computer Science Division of Computer Science. 2001. 160p. Advisor: Prof. Kwangkeun Yi.Text in English.

Abstract

Exception handling facilities are useful but dangerous. Many modern programming languages such as C++, Java, Ada, Modula-3, and ML provide the programmer with exception mechanisms. However, the exception facilities can provide a hole for program safety. Even for type-safe programming languages like ML, exceptions provide a hole for program safety. ML programs can abruptly halt when an exception is raised and never handled. This is the only one "safety hole" in well-typed ML programs.

Multithreaded exception handling is not well supported. Even though many languages offer both exception handling and multithreading mechanisms, they do not provide a good method to use these two features together. Although Java provides a simple and tightly integrated language support for multithreaded programs, throwing exceptions across threads is deprecated because of the safety problem.

In this thesis, we present two static analyses that detect potential runtime exceptions that are raised and never handled. We first design such an analysis for single-threaded ML programs and then for multithtreaded Java programs. We found that even though the exception flow and control flow are in general intertwined in ML programs, the two analyses could be safely and cost-effectively decoupled. For cases where exceptions carry functions(i.e., where control flow analysis needs exception

analysis) our control flow analysis uses a crude approximation to assure its safety against the decoupling. In Java, throwing exceptions across threads is deprecated because of the safety problem. Instead of restricting programmers' freedom, we extend the Java language to support multithreaded exception handling and propose a tool to detect uncaught exceptions. Our analysis firstly estimates concurrently evaluated expressions among threads, and then predicts uncaught exceptions by using the pre-analyzed concurrency information.

Our method can be applied to other languages with exception handling facilities. One contribution of this thesis is an analysis in divide and conquer style. By dividing one complex analysis to several simpler analyses, the analysis result is achieved systematically. Another merit of our method is designing a sparse constraint system and proving its consistency with smaller-grained constraint systems. When the interesting properties are sparse in programs, it is reasonable to analyze the properties at a larger granularity than at every expression.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem

Many modern programming languages such as C++, Java, Ada, Modula-3, and ML provide the programmer with exception mechanisms. They provide not only primitive exceptions like `Overflow` but also user-defined exceptions. Exceptional conditions are brought (by a raise expression) to the attention of another expression where the raised exceptions may be handled. Use of the exception facilities is not limited to deal with errors. The programmer can use exceptions to escape any control structure and can tailor an operation's results to particular purposes in a wide variety of cases.

The exception facilities, however, can be problematic in assuring software quality. Even for type-safe programming languages like ML, exceptions provide a hole for program safety. ML programs can abruptly halt when an exception is raised and never handled. This is the only one "safety hole" in well-typed ML programs. Uncaught exceptions are sometimes disastrous.

Even though many languages offer both exception handling and multithreading mechanisms, they do not provide a good method to use these two features together. Although Java provides simple and expressive language constructs for multithreaded programs, throwing exceptions across threads is deprecated because of the safety problem.

1.2 Our Approach

In this thesis, we present two static analyses that detect potential runtime exceptions that are raised and never handled inside programs. We first design such an analysis for ML programs and then for multithtreaded Java programs.

Even though ML program's control flow and exception flow are in general mutually dependent, analyzing the two flows are safely and cost-effectively decoupled. Program's control-flow is firstly estimated by simple case analysis of call expressions and type information. Using this call-graph estimation, program's exception flow is derived as set-constraints, whose least model is our analysis result.

We show the safety of our exception analysis (constraint system $\triangleright_3$) in two steps, using two intermediate systems ($\triangleright_1$ and $\triangleright_2$). This safety proofs are done by showing the consistencies between the three constraint systems. We use fixpoint induction for continuous functions that are derived from the constraint rules.

Our implementation of this analysis has been applied to realistic SML programs and shows a promising cost-accuracy performance. For the ML-Lex program, for example, the analysis takes 1.50 seconds and it reports 4 possibly-uncaught exceptions, among which 3 exceptions can really escape.

Instead of restricting programmers' freedom like in Java, we extend Java language to support multithreaded exception handling and propose a tool to detect uncaught exceptions in the input programs. Our analysis consists of two steps. The analysis firstly estimates concurrently evaluated expressions of the multithreads in Java programs by the synchronization relation among the threads. Using this concurrency information, program's exception flow is derived as set-constraints, whose least model is our analysis result.

1.3 Thesis Overview

This thesis is organized as follow:

- Chapter 2 presents exception mechanism. We introduce how to declare, raise and handle exceptional conditions in some modern programming languages. And we show that exceptions are useful but dangerous.

- Chapter 3 describes an exception analysis for Standard ML (SML) programs. We define a source language and design its control flow analysis and exception analysis. We explain the implementation of the analyses and show the experimental results. And then we review the related work.

- Chapter 4 presents multithreaded exception handling mechanism. We introduce how to create, terminate, and synchronize threads in some modern programming languages. We investigate a problem of multithreaded exception handling in Java programs and show several existing ad-hoc solutions for this problem.

- Chapter 5 describes an exception analysis for multithreaded Java programs. We extend the Java language to support throwing exceptions across threads and present a static analysis that estimates uncaught exceptions in multithreaded Java programs. The analysis consists of two steps: analyzing Java program's concurrency information and then exception flow. We describe the analysis complexity and then review the related work.

- Chapter 6 concludes this thesis. We summarize contributions and discuss future work.

Chapter 2

Exception Mechanism

In this chapter, we present exception mechanism. We introduce how to declare, raise and handle exceptional conditions in some modern programming languages. And we show that exceptions are useful but dangerous.

2.1 Exception Mechanisms in Modern Languages

This section introduces exception mechanisms in some modern programming languages. Many modern languages such as C++[63], Java[30, 31], Ada[35, 33], Modula-3[14, 15], and ML[53, 52, 46] provide the programmer with exception mechanisms. They provide not only primitive exceptions like `Overflow` but also user-defined exceptions. Exceptional conditions are brought (by a raise expression) to the attention of another expression where the raised exceptions may be handled.

We will consider ML's exception mechanism for our exception analysis in chapter 3. As one can notice by following sections, ML provides the most expressive exception mechanism among modern programming languages. Exception mechanisms in other languages can be expressed in ML's exception mechanism.

2.1.1 In C++

The notion of an exception in C++ is provided to help deal with error reporting. C programmers can think of exception handling as a mechanism replacing `setjmp/longjmp`. Since C++ is an object oriented language, an exception is represented as an object of some class representing an exceptional occurrence. Code that detects an error *throw*s an exception object and unwinds the stack until a suitable *catch* is found in a function that invoked the function that threw the exception.

The exception `Range_error`, for example, is declared as follows:

```
struct  Range_error {
        int i;
        Range_error(int ii) {i = ii;} // constructor
};
```

and raised as follows:

```
throw Range_error(3);.
```

In C++, the syntax for an exception handler is:

$$\texttt{try}\ block_0\ \ \texttt{catch}\ (p_1)\ block_1\ \ \ldots\ \ \texttt{catch}\ (p_n)\ block_n.$$

When any code in a $block_0$ - or called from it - raises an exception, the handlers will be examined. If the raised exception is of a type p_i specified for a handler "`catch` (p_i) $block_i$", $block_i$ is executed.

2.1.2 In Java

The possible exceptions in a Java program are organized in a hierarchy of classes, rooted at class `Throwable`, a direct subclass of `Object`. The classes `Exception` and `Error` are direct subclasses of `Throwable`. The class `RuntimeException` is a direct subclass of `Exception`. Java programs can raise the pre-existing exception classes, or define additional exception classes, as subclasses of `Throwable` or of any of its subclasses, as appropriate.

In Java, an exception is raised by a `throw` statement:

$$\texttt{throw}\ e$$

where the expression e must produce a reference to an object that is an instance of the `Throwable` class or one of its subclasses, and a raised exception is handled by a `try` statement:

$$\begin{array}{ll} \texttt{try}\ \mathit{block}_0 & \texttt{catch}\ (p_1)\ \mathit{block}_1 \\ & \quad\quad \cdots \\ & \texttt{catch}\ (p_n)\ \mathit{block}_n. \end{array}$$

A `try` statement executes a block_0. If a value is thrown and the `try` statement has one or more `catch` clauses that can catch it, then control will be transferred to the first such `catch` clause. If the `try` statement has a `finally` clause:

$$\begin{array}{ll} \texttt{try}\ \mathit{block}_0 & \texttt{catch}\ (p_1)\ \mathit{block}_1 \\ & \quad\quad \cdots \\ & \texttt{catch}\ (p_n)\ \mathit{block}_n \\ & \texttt{finally}\ \mathit{block}_{n+1}. \end{array}$$

then another $block_{n+1}$ of code is executed, no matter whether the `try` block completes normally or abruptly, and no matter whether a `catch` clause is first given control.

If a method can raise any exceptions, the method declaration must declare that fact in a `throws` clause. If a method contains a `throw` statement in its body or calls another method declared with a `throws` clause, it is possible that an exception will be raised within the method. If the exception is not handled inside the method, it will be raised out of the method. Any exception that can be raised out of a method in this way must be listed in a `throws` clause in the method declaration:

$$m(x) \texttt{ = } e \texttt{ throws } c$$

Java compiler checks that the input program will have no uncaught exceptions at run-time by relying on `throws` clauses.

However, Java compiler's check for uncaught exceptions is not good enough. First, if the raised exception is an instance of `Error`, `RuntimeException`, or a subclass of one of those classes, Java compiler does not check these exceptions. Second, the accuracy of Java compiler's check relies on the programmer's specifications (`throws` clauses). Thus, when a programmer excessively declares that some exceptions can be uncaught at every method, Java compiler cannot avoid unnecessary handlers. Another problematic case is when a programmer declares that a method throws exceptions of the most general class `Exception` even if the actual uncaught exceptions are of much lower, specific classes. When the method is called, its handler cannot deal with the raised exception specific to the exact classes of the exception.

2.1.3 In Ada

Ada provides an exception handling mechanism that makes it possible to separate two classes of program parts: those parts which implement the normal algorithm, and those parts which handle errors. In Ada, the error handling code is always the last part of a program block and conspicuously separated from the normal part.

In Ada, an exception declaration consists of a name and the keyword `exception`:

```
Error:  exception;.
```

A `raise` statement raises an exception with the exception name:

```
raise Error;
```

and a particular exception may be raised in many different places within the scope of that exception.

At the end of the scope of the exception declaration, or at the end of every inner block of that scope, one can write an exception handler. The syntax of handlers is:

$$\begin{array}{lll} \texttt{exception} & \texttt{when } p_1 & \Rightarrow \ statement_1 \\ & \quad \cdots & \\ & \texttt{when } p_n & \Rightarrow \ statement_n \\ \texttt{end;} & & \end{array}$$

and these handlers are the last statements before the end of a scope of that exception. When an exception is raised in a raise statement, program execution is transferred to the exception handler with a matching name that is the highest in the runtime subprogram callstack.

2.1.4 In Modula-3

In Modula-3, if *id* is an identifier and t is a type, then:

$$\texttt{EXCEPTION}\ \mathit{id(t)}$$

declares *id* as an exception with argument type t. If "(t)" is omitted, the exception takes no argument. Exception declarations are allowed only at the top level of interfaces and modules. All declared exceptions are distinct.

A `RAISE` statement without an argument has the form:

```
RAISE Error
```

where `Error` is an exception that takes no argument, and a `RAISE` statement with an argument has the form:

$$\texttt{RAISE Error}(x)$$

where `Error` is an exception that takes an argument and x is an expression assignable to `Error`'s argument type.

A `TRY-EXCEPT` statement has the form:

$$\begin{array}{rl}\texttt{TRY}\ \mathit{Body}\ \texttt{EXCEPT} & id_1(v_1) \Rightarrow \mathit{Handler}_1 \\ | & \quad\cdots \\ | & id_n(v_n) \Rightarrow \mathit{Handler}_n \\ & \texttt{ELSE}\ \mathit{Handler}_{n+1} \\ \texttt{END} & \end{array}$$

where *Body* and each $Handler_i$ are statements, each *id* names an exception, and each v_i is an identifier. The "`ELSE` $Handler_{n+1}$" and each "(v_i)" are optional. It is a static error for an exception to be named more than once in the list of *id*'s.

2.1.5 In ML

We present the exception mechanism in ML by using the case of SML[53]. In SML, exceptions are treated just like other values (until they are raised). They can be passed as function arguments, returned as the results of function applications, bound to identifiers, stored in locations, etc.

An exception consists of an exception name possibly paired with some argument values. For example,

```
Fail("at line 10")
```

constructs the `Fail` exception with the string argument. (In what follows, an exception name such as `Fail` is called an "exception constructor.") The exception constructor `Fail` must be declared earlier:

```
exception Fail of string
```

An exception is raised by

`raise` e

where the expression e must evaluate to an exception. For example, `raise !x`, where `x` is dereferenced for an exception value. A raised exception is called an exception packet.

Once an exception is raised, a handler is located by dynamic means: by going up the current evaluation chain to find potential handlers. During this process, one or more levels of the currently active call chain are aborted, up to the function containing the handler.

In SML, the syntax for an exception handler is:

$$e\ \texttt{handle}\ p_1\ \texttt{=>}\ e_1\ |\cdots|\ p_n\ \texttt{=>}\ e_n$$

Patterns p_i's are compared with a raised exception from the computation of e. When the exception's name (constructor) matches with pattern p_k, the corresponding expression e_k is evaluated. If the match fails, the raised exception continues to propagate back along the evaluation chain until it meets another handler, and so on.

2.2 Exceptions Are Useful

Using exceptions to manage errors has some advantages over traditional error management techniques. First, it separates error handling codes from regular codes. Exception mechanism enable the programmers to write the main code and deal with the exceptional cases elsewhere. Second advantage is the ability to propagate error reporting up the call stack. Finally, programmers can create groups of exceptions and handle exceptions in a general fashion, or they can use the specific exception to differentiate exceptions and handle exceptions in an exact fashion.

Use of the exception facilities is not necessarily limited to deal with errors. The programmer can use exceptions as a "control diverter" to escape any control structure to a point where the corresponding exception is handled. Also, using the exceptions, the programmer can tailor an operation's results to particular purposes

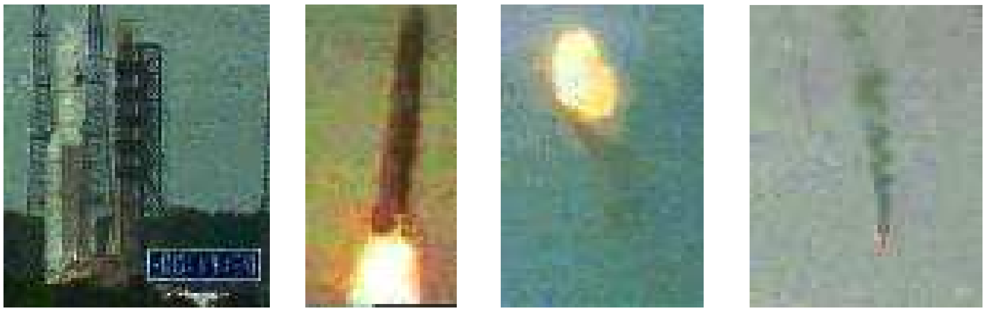

Figure 2.1: Ariane 5 Flight 501 Failure

in a wider variety of contexts than would otherwise be the case.

2.3 But Exceptions Are Dangerous

The exception facilities, however, can be problematic in assuring the software quality. Even for type-safe programming languages like ML, exceptions provide a hole for program safety. ML programs can abruptly halt when an exception is raised and never handled. This is the only one "safety hole" in well-typed ML programs.

Uncaught exceptions are sometimes disastrous [3][1].

> "On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Engineers from the Ariane 5 project teams of CNES and Industry immediately started to investigate the failure. ... The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value."

[1]The flight software lacked an exception handler for a data conversion code(in Ada). (`http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`)

Chapter 3

Exception Analysis for Standard ML Programs

This chapter presents a static analysis that detects potential runtime exceptions that are raised and never handled inside SML programs. This analysis will predict abrupt termination of SML programs, which is SML's only one "safety hole."

Even though SML program's control flow and exception flow are in general mutually dependent, analyzing the two flows are safely decoupled. Program's control-flow is firstly estimated by simple case analysis of call expressions. Using this call-graph information, program's exception flow is derived as set-constraints, whose least model is our analysis result. Both of these two analyses are proven safe and the reasons behind each design decision are discussed.

Section 3.1 and 3.2 present the analysis problem and our approach. And Section 3.3 shows the syntax and semantics of source language. Section 3.4 presents our exception analysis and Section 3.5 describes the implementation of the analysis. Section 3.6 shows experimental results and Section 3.7 presents related work.

3.1 Analysis Problem

- SML exceptions are first-class objects. Consider:

  ```
  fun f(x) = ···raise !x···
  ```

  Function `f` raises an exception `!x` in a location `x` passed to `f`.

- Precise exception analysis needs a precise call-graph estimation. Consider:

  ```
  fun f(g) = ···g(x) handle E =>···
  ```

  In order to estimate the uncaught exceptions from `g(x)`, we must analyze which functions are bound to `g` when `f` is called.

- Conversely, precise call-graph estimation needs a precise exception analysis. Consider:

  fun f(x) = $\cdots e$ handle E(g) => g(x)$\cdots$ (*)

  In order to decide which functions are called at `g(x)`, we must decide whether the e's uncaught exceptions include `E` and, if so, which functions are carried by it.

3.2 Our Approach

In earlier work [71], all the above problems are tackled by a monolithic abstract interpreter. Functions, exceptions, and other data values are parts of the abstract values. The analysis is a collecting analysis that compute stable program states at

program	exns[a]	w/ args[b]	arg types[c]	ftn arg[d]
Knuth-Bendix.sml	1	1	string	0
ml-lex.sml	8	1	int list * string	0
SML/NJ 109	339	34	string, string*int, int list, intmap, System.Unsafe.object list, symbol list, exn, unit→unit	1 (the `Found` exception in `debug/run.sml`)
HOL	60	18	string, int, record of string/int	0

[a]number of exception declarations (static count)
[b]number of exceptions with arguments (static count)
[c]argument types: basic building blocks after chasing type abbreviations and datatype arguments
[d]number of exceptions whose argument has a function (static count)

Figure 3.1: Exception Use Statistics in SML Programs

each expression point of the input program. This monolithic approach is appealing because the analysis design and its correctness proof is done at once by a sound abstraction of SML's concrete semantics. The collecting analyzer is, however, too expensive. It takes about one hour to analyze the ML-Lex program, for example.

For a better cost-effective analysis, we surveyed SML codes and found that such a full-fledged analysis may be an overkill in almost all cases. In particular, we found that such case as (*) almost never happened(Figure 3.1)[1]. This suggests that, in most cases, the call-graph estimation can be done independent of the exception analysis. Preparing for the rare case that exceptions carry functions would not pay-off in practice.

This does *not* mean that we don't guarantee the safety of our call-graph estimation. For such cases when functions to call are brought by uncaught exceptions,

[1]At least for hand-written codes. Situation may be different in automatically generated programs.

we choose to do a crude approximation, believing that this "large" approximation would be rarely detrimental to the call-graph accuracy. Note that we cannot use standard techniques for closure analysis [61, 37, 55, 41, 54] because their correctness does not consider languages with function-carrying exceptions.

Program's call-graph is estimated by a set of call-graph rules. For example, "`x(0)`" calls functions that are bound to `x` when λ`x`$.e$ is called, "`(f 0) 1`" calls functions `(f 0)` that the `f`'s body(a function expression) represents. The crude approximation happens when an exception's argument is the function to call. In this case we collect functions whose types unify with the call expression's function type. This simple call-graph estimation, which enables us to separate the control flow analysis from the exception flow, substantially reduces the total analysis cost and the consequent loss in accuracy of our exception analysis is not high because exceptions (or datatypes) rarely carry functions. The exact definition and its correctness proof are in Proposition 3 at page 34, after we fix the source language.

This call graph information is then used in exception analysis. For each function `f`, we express its exception flow as two classes of set constraints:

- One class is for set P_f of `f`'s uncaught exceptions[2]. For example, P_f of the following function

  `fun f(x) =` e`(0)+1`

  includes the union $\bigcup_g P_g$ of P_g's for g that may be called at "e`(0)`." In some cases, P_f is also composed of the set of exceptions that are available in f. For

[2] In SML, uncaught exceptions are called exception *p*ackets. Hence "P_f."

example, P_f of the following function

```
fun f(x) = raise x
```

includes the set of exceptions passed to `f`.

- The other class of constraints is for set X_f of exception values that are available during f's application. In the previous example function `f`, X_f includes the sum $\bigcup_g X_g$ of X_g's for g that may call `f`, because the caller g may pass its available exceptions to `f` through `x`.

Our exception analysis is to build the set of constraints(for P_f and X_f) and to compute its least solution(model). After the analysis, two things are reported to the programmer:

1. The solution of P_f for each top-level function f. The existence of such exceptions implies that the program can terminate abnormally.

2. Uncaught exceptions from each handle expression. From this information the programmer can check the completeness of the handler patterns.

Caveat

One subtlety of the SML's exception declaration is that it is generative. (This is also true for the datatype declarations.) Each evaluation of an exception declaration binds a new, unique name to the exception constructor. An exception handler looks up this internal name to determine a match. For example, in the following *incorrect* definition of the factorial function, each recursive call to `fact` generates a new instance of exception `ZERO` (line `(1)`). Thus, the handler in line `(3)`, which can only handle exceptions declared in its lexical scope, cannot handle another instance of ZERO that is declared and raised inside the recursive call fact(n-1). Hence this fact function always stops with an uncaught exception ZERO.

```
fun fact(n) =
    let exception ZERO                          (1)
    in if n <= 0 then raise ZERO                (2)
       else n * fact(n-1) handle ZERO => 1      (3)
    end
```

Our analysis cannot correctly analyze programs that utilize such generative nature of the exception (and the datatype) declarations. This limitation is not severe; exceptions (and datatypes) are largely declared at the global scope or at a module level, or we can move existing local declarations out to the global level without affecting the "observational" behavior of the programs. Programs where this hoisting is impossible cannot be analyzed correctly by our analysis.

We consider only exceptions that appear in the program's text (including library sources). This limitation can easily be lifted if our analysis starts with a table of primitive operators and their exceptions.

3.3 Language

For presentation brevity, we present our analysis for an imaginary language L. The language is a monomorphically typed, call-by-value, high-order language. The language L's abstract syntax and the usual monomorphic type rules are shown in Figure 3.2 and Figure 3.3, respectively. We use the usual notation that for a finite function $f \in A \stackrel{\text{fin}}{\to} B$, $f[a \mapsto b]$ denotes the new function which maps a into b and all other $a' \in dom(f)$ into $f(a')$.

For brevity, we have omitted datatype values, numbers, strings, primitive operators, and memory operations (assignment, reference, and dereference). In reality,

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
& | & \lambda x_\tau . e & \text{function} \\
& | & e_1\ e_2 & \text{application} \\
& | & \texttt{exn}\ \kappa\ e & \text{exception construction} \\
& | & \texttt{decon}\ e & \text{deconstruction} \\
& | & \texttt{case}\ e_1\ \kappa\ e_2\ e_3 & \text{switch} \\
& | & \texttt{fix}\ f\ \lambda x_\tau . e_1\ \texttt{in}\ e_2 & \text{recursive function binding} \\
& | & \texttt{raise}\ e & \text{exception raise} \\
& | & \texttt{+raise}\ e\ \kappa & \text{exception raise-only} \\
& | & \texttt{-raise}\ e\ \kappa_1 \cdots \kappa_n & \text{exception raise-except} \\
& | & \texttt{handle}\ e_1\ \lambda x_\tau . e_2 & \text{exception handler} \\
& | & \texttt{1} & \text{constant}
\end{array}
$$

Figure 3.2: L's Abstract Syntax

we work on the SML source level[3].

Values in L are either exceptions or functions. An exception value is constructed by "`exn` κ e" where κ is an exception name and expression e is for its argument value. The argument of an exception is recovered by "`decon` e". "`fix` f $\lambda x_\tau . e_1$ `in` e_2" binds recursive function $f = \lambda x_\tau . e_1$ in e_2. The case expression "`case` e_1 κ e_2 e_3" branches to e_2 if the value of e_1 is constructed with κ, otherwise, to e_3. "`raise` e" raises exception e. The `+raise` and `-raise` expressions are used in limited contexts, which we will discuss in the next section. The handle expression "`handle` e_1 $\lambda x_\tau . e_2$", where e_2 is typically a `case` expression on x, evaluates e_1 first. If e_1's result is a raised exception whose type is τ, the exception is bound to x inside e_2. If e_1's result is a normal value, then the value is returned. Note that this handle expression can handle only one type of exceptions.

[3]The `absyn` level of SML/NJ, which is after the input program is type-inferenced.

Type

τ	$::=$	$\tau\ exn$	exception type with argument type τ
	$\mid$	$\tau \to \tau$	function type
	$\mid$	ι	constant type

$\Gamma \in Var \overset{\text{fin}}{\to} Type$ $\qquad\qquad Arg(\kappa) = \kappa$'s argument type

[ABS] $$\frac{\Gamma[x \mapsto \tau] \vdash e \colon \tau'}{\Gamma \vdash \lambda x_\tau . e \colon \tau \to \tau'}$$

[VAR] $$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \colon \tau}$$

[FIX] $$\frac{\Gamma[f \mapsto \tau'] \vdash \lambda x_\tau . e \colon \tau'}{\Gamma \vdash \mathtt{fix}\, f\ \lambda x_\tau . e \colon \tau'}$$

[EXN] $$\frac{\Gamma \vdash e \colon \tau \qquad Arg(\kappa) = \tau}{\Gamma \vdash \mathtt{exn}\, \kappa\, e \colon\ \tau\ exn}$$

[DCON] $$\frac{\Gamma \vdash e \colon \tau\ exn}{\Gamma \vdash \mathtt{decon}\, e \colon \tau}$$

[APP] $$\frac{\Gamma \vdash e_1 \colon \tau \to \tau' \qquad \Gamma \vdash e_2 \colon \tau}{\Gamma \vdash e_1\ e_2 \colon \tau'}$$

[CASE] $$\frac{\Gamma \vdash e_1 \colon \tau'\ exn \qquad \Gamma \vdash e_2 \colon \tau \qquad \Gamma \vdash e_3 \colon \tau}{\Gamma \vdash \mathtt{case}\, e_1\ \kappa\ e_2\ e_3 \colon \tau}$$

[RS] $$\frac{\Gamma \vdash e \colon\ \tau\ exn}{\Gamma \vdash \mathtt{raise}\, e \colon \tau'}$$

[-RS] $$\frac{\Gamma \vdash e \colon\ \tau\ exn}{\Gamma \vdash \mathtt{-raise}\, e\ \kappa^+ \colon \tau'}$$

[+RS] $$\frac{\Gamma \vdash e \colon\ \tau\ exn}{\Gamma \vdash \mathtt{+raise}\, e\ \kappa \colon \tau'}$$

[HNDL] $$\frac{\Gamma \vdash e_1 \colon \tau' \qquad \Gamma[x \mapsto \tau] \vdash e_2 \colon \tau' \qquad \tau = \tau''\ exn}{\Gamma \vdash \mathtt{handle}\, e_1\ \lambda x_\tau . e_2 \colon \tau'}$$

[C] $$\Gamma \vdash \mathtt{1} \colon \iota$$

Figure 3.3: L's Type Rules

The function "$\lambda x_\tau.e_2$" in a handle expression "`handle` e_1 $\lambda x_\tau.e_2$" is called a *handler function*, the expression "e_1" a *handlee expression*, and the argument "x" of the handler function a *handle variable*.

Definition 1 (*type*$_\wp(e)$) *For an L program $\wp$ (a closed expression) of type τ_0, we write type$_\wp(e)$ for the type τ of its sub-expression e iff $\Gamma \vdash e \colon \tau$ is a sub-deduction of $\vdash \wp \colon \tau_0$. We simply write type(e) when it is clear from the context which program $\wp$ the expression e belongs to.*

Definition 2 (Typeful program) *An L program $\wp$ is typeful iff during the execution of $\wp$ (1) its every sub-expression e evaluates into a value of type(e) and (2) for each handler function $\lambda x_\tau.e$ in $\wp$ only the exceptions of type τ are bound to x.*

The second condition requires the raised exceptions to be also typeful with respect to their, if any, handler functions.

Proposition 1 *Every typeful SML program can be written in a typeful L program.*

Proof. Note that the typefulness of L requires the uncaught exceptions, as well as expression's values, to be typeful.

It is well known that any polymorphically type-checked SML program can be translated into a monomorphic program by the let-inlining. Making raised exceptions to be typeful in L is also straightforward, if L's handle expression could have multiple handler functions. Let $\{\tau_1, \cdots, \tau_n\}$ be the set of exception types in an SML program. Every SML handle expression is translated into an L handler:

$$\texttt{handle}\ e\ \ \lambda x_{\tau_1} e_1 \mid \cdots \mid \lambda x_{\tau_n} e_n$$

The semantics is that if an uncaught exception from e is of type τ_i then it is bound to x_{τ_i} inside e_i. The SML's handling expressions for exception patterns of type τ_i are translated into e_i. If the SML handler patterns do not completely cover an exception type τ_i, then the corresponding e_i is made just to re-raise the x. Then, clearly, such L program is typeful. □

Throughout this paper, we assume, for presentation brevity, that L's handle expression has only one handler function, and consider only typeful L programs whose variables are uniquely named (alpha-converted).

SML Programs in L

We assume that SML programs in L satisfy the following noteworthy things. It is straightforward to find such L program that corresponds to a given SML program. (Note that, in this section, some examples in L are not supported by the abstract syntax of Figure 3.2. For convenience we use numbers and multiple branches with the wild pattern, for example.)

- `raise` Every raise expression (`raise` e)'s argument expression e does not raise an exception. This is achieved as:

$$\texttt{raise}\, e \quad \stackrel{\text{is}}{\Longrightarrow} \quad (\lambda x.\texttt{raise}\, x)\; e.$$

- `-raise` The handler patterns are always augmented with an extra raise(`-raise`) expression, in order to re-raise exceptions that are not caught:

$$\begin{array}{lcl} & & \texttt{handle}\ e\ \lambda\, x_{exn}. \\ e\ \texttt{handle} & & \quad \texttt{case}\ x \\ \quad \texttt{ERROR => 1} & \stackrel{\text{is}}{\Longrightarrow} & \quad\quad \texttt{ERROR 1} \\ \texttt{| FAIL => 2} & & \quad\quad \texttt{FAIL\ \ 2} \\ & & \quad\quad \texttt{\_ -raise}\ x\ \texttt{ERROR FAIL} \end{array}$$

"`-raise` x `ERROR FAIL`" indicates that the re-raised exceptions are those bound to x excluding `ERROR` and `FAIL`.

- `+raise` If a handler's pattern for exception's argument part is not complete, the exception is explicitly re-raised by `+raise`:

`exception E of int list`
`...`
e `handle E nil => 1`
	$\stackrel{\text{is}}{\Longrightarrow}$	`handle` e $\lambda\ x_{(\iota list)exn}.$
		  `case` x
		    `E` $(\lambda\ y_{\iota list}.$
		        `case` y
		          `NIL 1`
		          `_ +raise` x `E) (decon` x`)`
		    `_ -raise` x `E`

Above program's handler can handle exception `E` only with `nil` list. Program in L makes this situation explicit by re-raising the `E` exceptions if their arguments are non-empty list. "`+raise` x `E`" indicates the re-raised exception shall only be the `E` exceptions.

- Exception constructors that need arguments are translated into a function, which is β-reduced whenever appropriate. For example,

`exception E of int` `··· E, ···`	$\stackrel{\text{is}}{\Longrightarrow}$	`···` $(\lambda\ x_{\iota}.$`(exn E` x`)), ···`

- Functor applications are in-lined. That is, functor definitions and applications disappear and are replaced by in-lined structures.[4]

[4]In SML, parameterized modules are called functors. A functor is a function that, given an argument structure, returns a new structure. A structure is a named collection of declarations.

3.4 Set-Constraint Systems

Our exception analysis is presented in the set-constraint framework[36, 5]. We use this formalism not because we will use its computation method (transforming set constraints into a regular tree grammar) but because the rule-based constraint formalism makes our presentation convenient. Our exception analysis is computed by the conventional iterative fixpoint method because our solution space is finite: exception names in the program. Correctness proofs are done by the fixpoint induction [62] over the continuous functions that are derived [22] from our constraint systems.

We present three set-constraint systems: $\triangleright_1$, $\triangleright_2$, and $\triangleright_3$. Our analysis is the last one $\triangleright_3$. The other two constraint systems are stepping stones to prove $\triangleright_3$'s safety. Note that 1) our analysis decouples control-flow analysis from exception analysis and 2) our interest is in uncaught exceptions from functions. These two things are done by $\triangleright_2$ and $\triangleright_3$ in order. $\triangleright_2$(Section 3.4.3) decouples control-flow analysis(Section 3.4.2) from exception analysis. $\triangleright_3$(Section 3.4.4) increases the constraint granularity to the function-level. Because exception-related expressions are sparse in programs, it is wasteful to generate constraints for every expression of the input program as in $\triangleright_2$. $\triangleright_3$ is proven consistent with $\triangleright_2$, $\triangleright_2$ with $\triangleright_1$, and $\triangleright_1$(Section 3.4.1) is assumed correct with respect to the standard semantics of L.

To review some notions of set constraint formalism, an interpretation $\mathcal{I}$ (a map from set expressions to sets) is a *model* (a solution) of a conjunction $\mathcal{C}$ of constraints if, for each constraint $\mathcal{X} \supseteq se$ (set variable $\mathcal{X}$ and set expression se) in $\mathcal{C}$, $\mathcal{I}(se)$ is defined and $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$. We write $lm(\mathcal{C})$ for the least model of $\mathcal{C}$. All our constraint systems($\triangleright_1, \triangleright_2, \triangleright_3$) guarantee the existence of the least model because every operator is monotonic(in terms of set-inclusion) and each constraint's left-hand-side is a single variable [36].

3.4.1 Concrete Constraint Construction $\triangleright_1$

Figure 3.4 and Figure 3.5 present domains, an interpretation, and analysis rules of $\triangleright_1$. Every expression e of the input program has two set constraints: $V_e \supseteq se$ and $P_e \supseteq se$. The set variable (the unknown) V_e is for e's values, P_e is for the uncaught exceptions (packets) during e's evaluation. A constraint $V_e \supseteq se$ ($P_e \supseteq se$) may be read as "expression e evaluates into a set of values (has uncaught exceptions) including those of se."

We index V and P sometimes with expressions, sometimes with numbers. For example, in

$$[\mathrm{RS}_{\triangleright_1}] \quad \frac{\triangleright_1 e_1 :\ \mathcal{C}_1}{\triangleright_1 \texttt{raise}\, e_1 :\ \{P_e \supseteq V_1\} \cup \mathcal{C}_1}$$

the $\mathcal{C}_1$ has constraints, among others, for V_{e_1}. This set variables is simply written as "V_1." The subscript "e" of set variables "V_e" and "P_e" denotes the current expression ($\texttt{raise}\, e_1$) to which the above rule applies. Note that, in L programs, raise expression's argument expression does not raise exceptions (Section 3.3), hence P_e does not include P_1.

Note that $var(x)$ indicates the values bound to variable x when the function with argument x is called:

$$\mathcal{I}(var(x)) = \{v \mid \text{``}e_1\, e_2\text{''} \in e, \lambda x_\tau.\bullet \in \mathcal{I}(V_1), v \in \mathcal{I}(V_2)\},$$

and $app_V(V_1)$ the values returned from functions V_1:

$$\mathcal{I}(app_V(V_1)) = \{v \mid \lambda x_\tau.e \in \mathcal{I}(V_1), v \in \mathcal{I}(V_e)\}$$

$$
\begin{array}{rclll}
v & \in & \mathit{Val} & = & \mathit{Closure} + \mathit{Exn} + \{\mathtt{1}\} & \text{values} \\
\lambda x_\tau.e & \in & \mathit{Closure} & = & \mathit{Expr} & \text{lambda exprs in program } \wp \\
\kappa\, v & \in & \mathit{Exn} & = & \mathit{Con} \times \mathit{Val} & \text{exceptions} \\
\kappa & \in & \mathit{Con} & = & \{\kappa_1, \cdots, \kappa_N\} & \text{exception names in program } \wp \\
\kappa\, v & \in & \mathit{Packet} & = & \mathit{Exn} & \text{raised exceptions}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{I}(V_e) & \subseteq & \mathit{Val} \\
\mathcal{I}(P_e) & \subseteq & \mathit{Packet} \\
\mathcal{I}(\lambda x_\tau.e) & = & \{\lambda x_\tau.e\} \\
\mathcal{I}(\mathtt{1}) & = & \{\mathtt{1}\} \\
\mathcal{I}(se \cup se') & = & \mathcal{I}(se) \cup \mathcal{I}(se') \\
\mathcal{I}(exn(\kappa, V_1)) & = & \{\kappa\, v \mid v \in \mathcal{I}(V_1)\} \\
\mathcal{I}(decon(V_1)) & = & \{v \mid \kappa\, v \in \mathcal{I}(V_1)\} \\
\mathcal{I}(var(x)) & = & \{v \mid e_1\; e_2 \in \wp, \lambda x_\tau.\bullet \in \mathcal{I}(V_1), v \in \mathcal{I}(V_2)\} \\
\mathcal{I}(app_V(V_1)) & = & \{v \mid \lambda x_\tau.e \in \mathcal{I}(V_1), v \in \mathcal{I}(V_e)\} \\
\mathcal{I}(app_P(V_1)) & = & \{\kappa\, v \mid \lambda x_\tau.e \in \mathcal{I}(V_1), \kappa\, v \in \mathcal{I}(P_e)\} \\
\mathcal{I}(case(V_1, \kappa, V_2, V_3)) & = & \{v \mid v \in \mathcal{I}(V_2), \kappa\, v' \in \mathcal{I}(V_1)\} \\
 & & \cup\{v \mid v \in \mathcal{I}(V_3), \kappa' v' \in \mathcal{I}(V_1), \kappa' \neq \kappa\} \\
\mathcal{I}(\mathtt{-}raise(V_1, \kappa_1, \cdots, \kappa_n)) & = & \{\kappa' v \mid \kappa' v \in \mathcal{I}(V_1), \forall i.\kappa_i \neq \kappa'\} \\
\mathcal{I}(\mathtt{+}raise(V_1, \kappa)) & = & \{\kappa\, v \mid \kappa\, v \in \mathcal{I}(V_1)\}
\end{array}
$$

Figure 3.4: Constructing Concrete Constraints: $\rhd_1$ (I)

[VAR$_{\rhd_1}$] $$\rhd_1 x: \{V_x \supseteq var(x)\}$$

[C$_{\rhd_1}$] $$\rhd_1 \mathtt{1}: \{V_e \supseteq \mathtt{1}\}$$

[ABS$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1}{\rhd_1 \lambda x_\tau.e_1: \{V_e \supseteq \lambda x_\tau.e_1\} \cup \mathcal{C}_1}$$

[FIX$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1 \qquad \rhd_1 e_2: \mathcal{C}_2}{\rhd_1 \mathtt{fix}\, f\, \lambda x_\tau.e_1\, \mathtt{in}\, e_2: \{V_e \supseteq V_2, P_e \supseteq V_2, V_f \supseteq \lambda x_\tau.e_1\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[EXN$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1}{\rhd_1 \mathtt{exn}\, \kappa\, e_1: \{V_e \supseteq exn(\kappa, V_1), P_e \supseteq P_1\} \cup \mathcal{C}_1}$$

[DCON$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1}{\rhd_1 \mathtt{decon}\, e_1: \{V_e \supseteq decon(V_1), P_e \supseteq P_1\} \cup \mathcal{C}_1}$$

[APP$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1 \qquad \rhd_1 e_2: \mathcal{C}_2}{\rhd_1 e_1\, e_2: \{V_e \supseteq app_V(V_1), P_e \supseteq app_P(V_1) \cup P_1 \cup P_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[CASE$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1 \qquad \rhd_1 e_2: \mathcal{C}_2 \qquad \rhd_1 e_3: \mathcal{C}_3}{\begin{array}{ll} \rhd_1 \mathtt{case}\, e_1\, \kappa\, e_2\, e_3: & \{ \quad V_e \supseteq case(V_1, \kappa, V_2, V_3), P_e \supseteq P_1 \cup P_2 \cup P_3 \\ & \} \quad \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \end{array}}$$

[RS$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1}{\rhd_1 \mathtt{raise}\, e_1: \{P_e \supseteq V_1\} \cup \mathcal{C}_1}$$

[-RS$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1}{\rhd_1 \mathtt{-raise}\, e_1\, \kappa_1 \cdots \kappa_n: \{P_e \supseteq \mathtt{-}raise(V_1, \kappa_1, \cdots, \kappa_n)\} \cup \mathcal{C}_1}$$

[+RS$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1}{\rhd_1 \mathtt{+raise}\, e_1\, \kappa: \{P_e \supseteq \mathtt{+}raise(V_1, \kappa)\} \cup \mathcal{C}_1}$$

[HNDL$_{\rhd_1}$] $$\frac{\rhd_1 e_1: \mathcal{C}_1 \qquad \rhd_1 e_2: \mathcal{C}_2}{\begin{array}{lll} \rhd_1 \mathtt{handle}\, e_1\, \lambda x_\tau.e_2: & \{ & V_e \supseteq app_V(\lambda x_\tau.e_2) \cup V_1, \\ & & P_e \supseteq app_P(\lambda x_\tau.e_2), V_x \supseteq P_1 \\ & \} & \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Figure 3.5: Constructing Concrete Constraints: $\rhd_1$ (II)

Similarly, $app_P(V_1)$ (with subscript P) indicates the uncaught exceptions from function calls.

Consider the rule for the `handle` expression:

$$[\mathrm{HNDL}_{\triangleright_1}] \qquad \frac{\triangleright_1 e_1 : \mathcal{C}_1 \qquad \triangleright_1 e_2 : \mathcal{C}_2}{\begin{array}{ll} \triangleright_1 \texttt{handle}\, e_1\, \lambda x_\tau.e_2 : & \{ \;\; V_e \supseteq app_V(\lambda x_\tau.e_2) \cup V_1, \\ & \quad\;\; P_e \supseteq app_P(\lambda x_\tau.e_2), V_x \supseteq P_1 \\ & \} \;\; \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

The first constraint

$$V_e \supseteq app_V(\lambda x_\tau.e_2) \cup V_1$$

indicates that the handle expression's values V_e are either the values V_1 of the handlee expression e_1 or the values returned from the handler function. Note that the $V_x \supseteq P_1$ indicates that the argument to the handler function "$\lambda x_\tau.e_2$" is the uncaught exceptions P_1 from the handlee expression e_1. The second constraint

$$P_e \supseteq app_P(\lambda x_\tau.e_2)$$

indicates that uncaught exceptions of the handle expression include those from the handler function. Recall that in L uncaught exceptions from a handle expression are explicitly re-raised from the handler function.

We could have used Heintze's method[37, 38] to compute the constraint solution. However, it is wasteful to consider all expressions and their values because exception-related expressions are sparse in programs and function values (for control-flow analysis) can be separately estimated. These two observations are reflected in the following constraint systems, $\rhd_2$ and $\rhd_3$.

The correctness of $\rhd_1$ (i.e., the least model of $\rhd_1$-constraints for a program includes the actual values and uncaught exceptions) is assumed. The proof must not be much different from those outlined in [37], using a "set-based operational semantics" as an intermediate form.

Not only is $\rhd_1$ correct but typeful. The following typefulness is important for the consistency of the following constraint system $\rhd_2$.

Proposition 2 (Typefulness of $\rhd_1$) *For a program (a closed expression) $\wp$, let $\rhd_1 \wp \colon \mathcal{C}_1$. Then its least model $lm(\mathcal{C}_1)$ preserves types: $\forall e \in \wp . lm(\mathcal{C}_1)(V_e) \subseteq Val_{type(e)}$ (the set of values of $type(e)$).*

Proof. The least model $lm(\mathcal{C}_1)$ is equivalent to the $\subseteq$-least fixpoint $\mathrm{fix}\mathcal{F}_1$ of the following continuous function $\mathcal{F}_1$ derived from $\mathcal{C}_1$ as follows[22]:

$$
\begin{array}{lcl}
\mathcal{V} & = & \{V_e \mid e \in \wp\} \cup \{P_e \mid e \in \wp\} \quad \text{the set of constraint variables for program } \wp \\
2^{Val} & = & \text{the powerset of } Val\text{, ordered by } \subseteq \\
\multicolumn{3}{l}{\mathcal{F}_1 \colon (\mathcal{V} \to 2^{Val}) \to (\mathcal{V} \to 2^{Val})}
\end{array}
$$

$$
\mathcal{F}_1(\rho)(V_e) \quad = \quad \begin{cases}
\{\mathtt{1}\} & \text{if } e = \mathtt{1} \\
\{\lambda x_\tau.e'\} & \text{if } e = f(\text{function var}) \text{ where } \mathtt{fix}\ f\ \lambda x_\tau.e' \in \wp \\
\rho(P_1) & \text{if } e = x(\text{handle var}) \text{ where } \mathtt{handle}\ e_1\ \lambda x_\tau.e_2 \in \wp \\
\{v \mid e_1\ e_2 \in \wp, \lambda x_\tau.e' \in \rho(V_1), v \in \rho(V_2)\} & \text{if } e = x(\text{normal var}) \\
\{\lambda x_\tau.e'\} & \text{if } e = \lambda x_\tau.e' \\
\rho(V_2) & \text{if } e = \mathtt{fix}\ f\ \lambda x_\tau.e_1\ \mathtt{in}\ e_2 \\
\{\kappa\, v \mid v \in \rho(V_1)\} & \text{if } e = \mathtt{exn}\ \kappa\ e_1 \\
\{v \mid \kappa\, v \in \rho(V_1)\} & \text{if } e = \mathtt{decon}\ e_1 \\
\{v \mid \lambda x_\tau.e' \in \rho(V_1),\ v \in \rho(V_{e'})\} & \text{if } e = e_1\ e_2 \\
\{v \mid v \in \rho(V_2),\ \kappa\, v' \in \rho(V_1)\} \cup \{v \mid v \in \rho(V_3),\ \kappa'\, v' \in \rho(V_1),\ \kappa' \neq \kappa\} & \\
 & \text{if } e = \mathtt{case}\ e_1\ \kappa\ e_2\ e_3 \\
\rho(V_1) \cup \rho(V_2) & \text{if } e = \mathtt{handle}\ e_1\ \lambda x_\tau.e_2 \\
\emptyset & \text{otherwise}
\end{cases}
$$

$$
\mathcal{F}_1(\rho)(P_e) \quad = \quad \begin{cases}
\rho(P_1) & \text{if } e = \mathtt{exn}\ \kappa\ e_1 \\
\rho(P_1) & \text{if } e = \mathtt{decon}\ e_1 \\
\{\kappa\, v \mid \lambda x_\tau.e' \in \rho(V_1),\ \kappa\, v \in \rho(P_{e'})\} \cup \rho(P_1) \cup \rho(P_2) & \\
 & \text{if } e = e_1\ e_2 \\
\rho(P_1) \cup \rho(P_2) \cup \rho(P_3) & \text{if } e = \mathtt{case}\ e_1\ \kappa\ e_2\ e_3 \\
\rho(P_2) & \text{if } e = \mathtt{fix}\ f\ \lambda x_\tau.e_1\ \mathtt{in}\ e_2 \\
\rho(V_1) & \text{if } e = \mathtt{raise}\ e_1 \\
\{\kappa' v \mid \kappa' v \in \rho(V_1), \forall i.\kappa_i \neq \kappa'\} & \text{if } e = \mathtt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n \\
\{\kappa\, v \mid \kappa\, v \in \rho(V_1)\} & \text{if } e = \mathtt{+raise}\ e_1\ \kappa \\
\rho(P_2) & \text{if } e = \mathtt{handle}\ e_1\ \lambda x_\tau.e_2 \\
\emptyset & \text{otherwise}
\end{cases}
$$

It is straightforward to derive this function because the $\triangleright_1$ generates at most one constraint per V_e and P_e. That is, every $\supseteq$ is $=$.

We prove *typeful*(fix$\mathcal{F}_1$) by fixpoint induction, where the assertion *typeful*(ρ) for a program $\wp$ is:

$$\mathit{typeful}(\rho) = \\ \forall e \in \wp. \begin{cases} & \rho(V_e) \subseteq \mathit{Val}_{\mathit{type}(e)} \\ \wedge & e\text{'s exn value is raised and bound to a handle var } x_\tau \Rightarrow \rho(V_e) \subseteq \mathit{Val}_\tau \\ \wedge & e\text{'s uncaught exn is bound to a handle var } x_\tau \Rightarrow \rho(P_e) \subseteq \mathit{Val}_\tau \end{cases}$$

Base *typeful*$(\emptyset)$ is trivially true. We will show that *typeful*$(\mathcal{F}_1(\rho))$ holds given the induction hypothesis (I.H.) *typeful*(ρ).

First, the cases for $\mathcal{F}_1(\rho)(V_e)$.

[C] $e = \mathtt{1}$. $\mathcal{F}_1(\rho)(V_e) = \{\mathtt{1}\} \subseteq \mathit{Val}_\iota$.

[VAR] $e = f$ (function variable) where $\mathtt{fix}\ f\ \lambda x_\tau.e' \in \wp$.

> $\mathcal{F}_1(\rho)(V_f) = \{\lambda x_\tau.e'\}$ by definition. Because the program $\wp$ is typeful, the function $\lambda x_\tau.e'$ is in $\mathit{Val}_{\mathit{type}(\lambda x_\tau.e')}$, which is equal to $\mathit{Val}_{\mathit{type}(f)}$ because of L's type rules.

[VAR] $e = x$ (handle variable) where $\mathtt{handle}\ e_1\ \lambda x_\tau.e_2 \in \wp$.

> $\mathcal{F}_1(\rho)(V_x) = \rho(P_1)$ by definition. Because the program $\wp$ is typeful, the e_1's uncaught exn is bound to x_τ. Thus by I.H. $\rho(P_1) \subseteq \mathit{Val}_\tau$.

[VAR] $e = x$ (normal var).

> $\mathcal{F}_1(\rho)(V_x) = \{v \mid e_1\ e_2 \in \wp, \lambda x_\tau.e' \in \rho(V_1), v \in \rho(V_2)\}$ by definition. If $\lambda x_\tau.e' \in \rho(V_1)$ then by I.H. $\lambda x_\tau.e' \in \mathit{Val}_{\mathit{type}(e_1)}$. Thus, because the program $\wp$ is typeful, $\mathit{type}(e_1) = \tau \to \_$ and $\mathit{type}(e_2) = \tau$. Therefore, by I.H., value v in $\rho(V_2)$ is in Val_τ.

Other cases are similarly proved.

Now, the cases for $\mathcal{F}_1(\rho)(P_e)$: assuming that e's uncaught exception is bound to a handle var x_τ, we prove $\mathcal{F}_1(\rho)(P_e) \subset Val_\tau$.

[EXN] $e = \texttt{exn}\, \kappa\, e_1$

> $\mathcal{F}_1(\rho)(P_e) = \rho(P_1)$ by definition. The assumption implies that e_1's uncaught exception is bound to x_τ. Thus by I.H. $\rho(P_1) \subseteq Val_\tau$.

[RS] $e = \texttt{raise}\, e_1$

> $\mathcal{F}_1(\rho)(P_e) = \rho(V_1)$ by definition. The assumption implies that e_1's value is bound to the x_τ. Thus by I.H. $\rho(V_1) \subseteq Val_\tau$.

Other cases are similarly proved. $\square$

3.4.2 Separate Call-Graph Estimation

Call-graph estimation methods[61, 66, 55, 41, 39, 54] in the literature cannot be directly used in our exception analysis, because their high-order source languages do not have exceptions, not to mention function-carrying exceptions.

Figure 3.6 shows our rules to estimate the call-graph of a program $\wp$. An edge $e \to \lambda x.e'$ indicates that during the execution of $\wp$ the e may evaluate into a closure of the lambda $\lambda x.e'$.

One interesting rule is the `decon` case where an exception's argument is a function:

$$\frac{\mathit{type}(e_1) = \mathit{type}(\texttt{decon}\, e), \quad e_1 \to \lambda x.e'}{\texttt{decon}\, e \to \lambda x.e'}\ .$$

We estimate that an exception's argument functions are those whose types are equal

$$\lambda x.e \rightarrow \lambda x.e$$

$$\frac{\texttt{fix}\, f\, \lambda x.e_1 \in \wp}{f \rightarrow \lambda x.e_1} \qquad \frac{e_2 \rightarrow \lambda y.e}{\texttt{fix}\, f\, \lambda x.e_1\, \texttt{in}\, e_2 \rightarrow \lambda y.e}$$

$$\frac{e_1\, e_2 \in \wp, \quad e_1 \rightarrow \lambda x.e_1', \quad e_2 \rightarrow \lambda y.e}{x \rightarrow \lambda y.e}$$

$$\frac{e_1 \rightarrow \lambda x.e_1', \quad e_1' \rightarrow \lambda y.e}{e_1\, e_2 \rightarrow \lambda y.e}$$

$$\frac{e_2 \rightarrow \lambda x.e}{\texttt{case}\, e_1\, \kappa\, e_2\, e_3 \rightarrow \lambda x.e} \qquad \frac{e_3 \rightarrow \lambda x.e}{\texttt{case}\, e_1\, \kappa\, e_2\, e_3 \rightarrow \lambda x.e}$$

$$\frac{e_1 \rightarrow \lambda y.e}{\texttt{handle}\, e_1\, \lambda x.e_2 \rightarrow \lambda y.e} \qquad \frac{e_2 \rightarrow \lambda y.e}{\texttt{handle}\, e_1\, \lambda x.e_2 \rightarrow \lambda y.e}$$

$$\frac{\mathit{type}(e_1) = \mathit{type}(\texttt{decon}\, e), \quad e_1 \rightarrow \lambda x.e'}{\texttt{decon}\, e \rightarrow \lambda x.e'}$$

Figure 3.6: Call-graph Estimation Rules

to the type of the current `decon` expression.[5]

This crude approximation is inevitable in order to separate the control flow analysis from the exception flow analysis. This simple call-graph analysis substantially reduces the total analysis cost and the consequent loss in accuracy of our exception analysis is not high because exceptions (or datatypes) rarely carry functions.

Proposition 3 (A safe call table *Lam***)** *Given a program $\wp$, let FtnExpr and 2^{Lambda} respectively be the set of function-typed expressions and the powerset of lambda ex-*

[5]Note that the L language is monomorphic. In SML, the “is equal to” must be “unifies with.”

pressions in $\wp$. Define Lam: $FtnExpr \to 2^{Lambda}$ *to be*

$$Lam(e) = \{\lambda x_\tau.e' \mid e \to \lambda x_\tau.e' \textit{ is deducible by the rules in Figure 3.6}\}.$$

Then Lam is safe: $\triangleright_1\wp\colon \mathcal{C} \to \forall e \in FtnExpr.Lam(e) \supseteq lm(\mathcal{C})(V_e)$.

Proof. Note that the *Lam* is equivalent to the $\subseteq$-least fixpoint of the following continuous function $\mathcal{L}$ [22]:

$$\mathcal{L}\colon (FtnExpr \to 2^{Lambda}) \to (FtnExpr \to 2^{Lambda})$$
$$\mathcal{L}(\ell)(e) = \begin{cases} \{\lambda x_\tau.e_1\} & \text{if } e = \lambda x_\tau.e_1 \\ \{\lambda x_\tau.e_1 \mid \texttt{fix}\, f\, \lambda x_\tau.e_1 \in \wp\} & \text{if } e = f \\ \cup\{\ell(e_2) \mid e_1\ e_2 \in \wp, \lambda x_\tau.e \in \ell(e_1)\} & \text{if } e = x \\ \cup\{\ell(e') \mid \lambda x.e' \in \ell(e_1)\} & \text{if } e = e_1\ e_2 \\ \ell(e_2) & \text{if } e = \texttt{fix}\, f\, \lambda x_\tau e_1\, \texttt{in}\, e_2 \\ \ell(e_2) \cup \ell(e_3) & \text{if } e = \texttt{case}\, e_1\, \kappa\, e_2\, e_3 \\ \ell(e_1) \cup \ell(e_2) & \text{if } e = \texttt{handle}\, e_1\, \lambda_\tau.e_2 \\ \cup\{\ell(e') \mid type(e') = type(\texttt{decon}\, e)\} & \text{if } e = \texttt{decon}\, e \end{cases}$$

We use the fixpoint induction. The assertion $Q(\ell, \rho)$ that we will prove is

$$\forall e \in FtnExpr.\ell(e) \supseteq \rho(V_e) \quad \wedge \quad typeful(\rho).$$

Note that we include the *typeful*(ρ) assertion that we used in the proof of Proposition 2. This typefulness of ρ is necessary in proving the `decon` case.

Base case $Q(\emptyset, \emptyset)$ trivially holds. We now prove that $Q(\ell, \rho)$ implies $Q(\mathcal{L}(\ell), \mathcal{F}_1(\rho))$.

Case e of a normal variable x:

$$\begin{array}{rcll}
\mathcal{L}(\ell)(x) & = & \cup\{\ell(e_2) \mid e_1\, e_2 \in \wp, \lambda x.e \in \ell(e_1)\} & \text{by definition} \\
& \supseteq & \cup\{\ell(e_1) \mid e_1\, e_2 \in \wp, \lambda x.e \in \rho(V_{e_1})\} & \text{by I.H.} \\
& \supseteq & \cup\{\rho(V_{e_1}) \mid e_1\, e_2 \in \wp, \lambda x.e \in \rho(V_{e_1})\} & \text{by I.H.} \\
& = & \mathcal{F}_1(V_x) & \text{by definition.}
\end{array}$$

Other cases are done similarly, except for the case e of $\texttt{decon}\, e_1$:

$$\begin{array}{rcll}
\mathcal{L}(\ell)(e) & = & \cup\{\lambda x.e' \mid \lambda x.e' \in \wp, \mathit{type}(e) = \mathit{type}(\lambda x.e')\} & \text{by definition} \\
& = & \mathrm{Val}_{\mathit{type}(e)}, \text{the set of lambdas of } \mathit{type}(e) \text{ in } \wp & \text{by definition} \\
& \supseteq & \mathcal{F}_1(\rho)(V_e) & \text{by that } \mathit{typeful}(\rho) \Rightarrow \mathit{typeful}(\mathcal{F}_1(\rho)), \\
& & & \text{which is proven in Proposition 2.}
\end{array}$$

□

3.4.3 Exception Constraint Construction $\triangleright_2$

We now consider a new system $\triangleright_2$ (Figure 3.7 and Figure 3.8) where only exceptions are considered. Constraints for function(non-exception) values are removed and instead, a pre-computed, safe call-graph table *Lam* (Proposition 3) is used.

Every expression e has two set constraints: $X_e \supseteq se$ and $P_e \supseteq se$. X_e is for exceptions and P_e for uncaught exceptions. For solutions of X_e and P_e we will consider only exception names. That is, X_e is for the set $|V_e|$ of exception names in e's values V_e:

$$\mathcal{I}(X_e) \supseteq |V_e|$$

Definition 3 $|V| = \{\kappa \mid \kappa\, v \in V\} \cup |\{v \mid \kappa\, v \in V\}|$.

$$
\begin{array}{lllll}
\kappa & \in & \mathit{Exn} & = & \{\kappa_1, \cdots, \kappa_N\} \quad \text{exception names in program } \wp \\
\kappa & \in & \mathit{Packet} & = & \mathit{Exn} \qquad\qquad\quad\; \text{raised exceptions}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{I}(X_e) & \subseteq & \mathit{Exn} \\
\mathcal{I}(P_e) & \subseteq & \mathit{Packet} \\
\mathcal{I}(\mathit{var}(x)) & = & \{\kappa \mid e_1\, e_2 \in \wp, \lambda x_\tau.\bullet \in \mathit{Lam}(e_1), \kappa \in \mathcal{I}(X_2)\} \\
\mathcal{I}(\mathit{app}_X(e_1)) & = & \{\kappa \mid \lambda x_\tau.e \in \mathit{Lam}(e_1), \kappa \in \mathcal{I}(X_e)\} \\
\mathcal{I}(\mathit{app}_P(e_1)) & = & \{\kappa \mid \lambda x_\tau.e \in \mathit{Lam}(e_1), \kappa \in \mathcal{I}(P_e)\} \\
\mathit{isExn}(\tau) & = & \text{true iff } \tau = \tau'\ \mathit{exn} \\
\mathcal{I}(X_1 \setminus_e \{\kappa_1, \cdots, \kappa_n\}) & = & \begin{cases} \mathcal{I}(X_1) & \text{if } \mathit{type}(e) = \tau'\mathit{exn} \wedge \mathit{isExn}(\tau') \\ \mathcal{I}(X_1) \setminus \{\kappa_1, \cdots, \kappa_n\} & \text{otherwise} \end{cases} \\
\mathcal{I}(X_1 \cap_e \{\kappa\}) & = & \begin{cases} \mathcal{I}(X_1) & \text{if } \mathit{type}(e) = \tau'\mathit{exn} \wedge \mathit{isExn}(\tau') \\ \mathcal{I}(X_1) \cap \{\kappa\} & \text{otherwise} \end{cases} \\
\mathcal{I}(\kappa) & = & \{\kappa\}
\end{array}
$$

$\mathcal{I}(\lambda x_\tau.e)$ and $\mathcal{I}(se \cup se)$ are the same as in $\rhd_1$ (Figure 3.4, page 27).

Figure 3.7: Constructing Exception Constraints: $\rhd_2$ (I)

$$[\mathrm{VAR}_{\rhd_2}] \qquad \rhd_2\, x:\ \{X_x \supseteq \mathit{var}(x)\}$$

$$[\mathrm{C}_{\rhd_2}] \qquad \rhd_2\, \mathtt{1}:\ \emptyset$$

$$[\mathrm{ABS}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 : \mathcal{C}_1}{\rhd_2\, \lambda x_\tau.e_1 :\ \mathcal{C}_1}$$

$$[\mathrm{FIX}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1 \qquad \rhd_2\, e_2 :\ \mathcal{C}_2}{\rhd_2\, \mathtt{fix}\, f\ \lambda x_\tau.e_1\ \mathtt{in}\, e_2 :\ \{X_e \supseteq X_2, P_e \supseteq P_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{DCON}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1}{\rhd_2\, \mathtt{decon}\, e_1 :\ \{X_e \supseteq X_1, P_e \supseteq P_1\} \cup \mathcal{C}_1}$$

$$[\mathrm{EXN}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1}{\rhd_2\, \mathtt{exn}\, \kappa\, e_1 :\ \{X_e \supseteq \kappa \cup X_1, P_e \supseteq P_1\} \cup \mathcal{C}_1}$$

$$[\mathrm{APP}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1 \qquad \rhd_2\, e_2 :\ \mathcal{C}_2}{\rhd_2\, e_1\ e_2 :\ \{X_e \supseteq \mathit{app}_X(e_1), P_e \supseteq \mathit{app}_P(e_1) \cup P_1 \cup P_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\mathrm{CASE}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1 \qquad \rhd_2\, e_2 :\ \mathcal{C}_2 \qquad \rhd_2\, e_3 :\ \mathcal{C}_3}{\rhd_2\, \mathtt{case}\, e_1\ \kappa\ e_2\ e_3 :\ \{X_e \supseteq X_2 \cup X_3, P_e \supseteq P_1 \cup P_2 \cup P_3\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\mathrm{RS}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1}{\rhd_2\, \mathtt{raise}\, e_1 :\ \{P_e \supseteq X_1\} \cup \mathcal{C}_1}$$

$$[\text{-}\mathrm{RS}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1}{\rhd_2\, \mathtt{-raise}\, e_1\ \kappa_1 \cdots \kappa_n :\ \{P_e \supseteq (X_1 \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\})\} \cup \mathcal{C}_1}$$

$$[\text{+}\mathrm{RS}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1}{\rhd_2\, \mathtt{+raise}\, e_1\ \kappa :\ \{P_e \supseteq (X_1 \cap_{e_1} \{\kappa\})\} \cup \mathcal{C}_1}$$

$$[\mathrm{HNDL}_{\rhd_2}] \qquad \frac{\rhd_2\, e_1 :\ \mathcal{C}_1 \qquad \rhd_2\, e_2 :\ \mathcal{C}_2}{\begin{array}{ll} \rhd_2\, \mathtt{handle}\, e_1\ \lambda x_\tau.e_2 : & \{\ \ X_e \supseteq \mathit{app}_X(\lambda x_\tau.e_2) \cup X_1, \\ & \quad\ \ P_e \supseteq \mathit{app}_P(\lambda x_\tau.e_2), X_x \supseteq P_1 \\ & \}\ \ \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Figure 3.8: Constructing Exception Constraints: $\rhd_2$ (II)

The use of set expression $app(e_1)$ for function calls is similar to that in $\triangleright_1$, except that we use the call-graph table $Lam: FtnExpr \to Lambdas$ (Proposition 3).

Set variable's indexing convention is the same as in the previous section ($\triangleright_1$).

Consider the rule for `-raise` expression:

$$[\text{-RS}_{\triangleright_2}] \qquad \frac{\triangleright_2\, e_1 :\ \mathcal{C}_1}{\triangleright_2\, \texttt{-raise}\, e_1\, \kappa_1 \cdots \kappa_n :\ \{P_e \supseteq (X_1 \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\})\} \cup \mathcal{C}_1}$$

The constraint $P_e \supseteq X_1 \setminus_e \{\kappa_1, \cdots, \kappa_n\}$ collects raised exceptions excluding κ_i's. Note the meaning of $\setminus_e$:

$$\mathcal{I}(X_1 \setminus_e \{\kappa_1, \cdots, \kappa_n\}) = \begin{cases} \mathcal{I}(X_1) & \text{if } type(e) = \tau'\,exn \ \wedge \ isExn(\tau') \\ \mathcal{I}(X_1) \setminus \{\kappa_1, \cdots, \kappa_n\} & \text{otherwise} \end{cases}$$

If an exception can have other exceptions as its arguments then the exclusion $\setminus_e$ has no effect. If blindly excluded, exceptions that are hidden as arguments of the escaping exception are considered caught. This makes the analysis unsafe. Consider a `-raise` expression whose argument e is an exception that hides another exception in its argument:

$$\texttt{-raise}\ \underbrace{\kappa_1(\kappa_2(\texttt{1}))}_{e}\ \kappa_2$$

The expression raises the exception $\kappa_1(\kappa_2(\texttt{1}))$ unless its constructor κ_1 is equal to κ_2 (which is false). Hence, the exception $\kappa_1(\kappa_2)$ is raised. If we removed κ_2 from the set $X_e = \{\kappa_1, \kappa_2\}$ then the exception κ_2 that can be available when the exception packet is later caught and deconstructed is considered missing thereafter. Therefore, the set-minus operator $\setminus_e$ is effective only when the exception values of e cannot have other exceptions hidden in its argument. (The same reason is for the definition of

$\cap_e$.)

The constraint-construction rule $\triangleright_2$ is a safe approximation of $\triangleright_1$:

Proposition 4 (Correctness of $\triangleright_2$) *For a program $\wp$, let $\triangleright_1\wp\colon \mathcal{C}_1$ and $\triangleright_2\wp\colon \mathcal{C}_2$ with their least models, $\mathcal{I}_1 = lm(\mathcal{C}_1)$ and $\mathcal{I}_2 = lm(\mathcal{C}_2)$. If Lam is safe with respect to $\triangleright_1$, then for every sub-expression $e \in \wp$:*

$$\mathcal{I}_2(X_e) \supseteq |\mathcal{I}_1(V_e)| \quad \mathit{and} \quad \mathcal{I}_2(P_e) \supseteq |\mathcal{I}_1(P_e)|.$$

Proof. The least models $\mathcal{I}_1$ and $\mathcal{I}_2$ are equivalent to the $\subseteq$-least fixpoints fix$\mathcal{F}_1$ and fix$\mathcal{F}_2$, respectively[22]. The $\mathcal{F}_1$ is defined in the proof of Proposition 2. The continuous function $\mathcal{F}_2$ is derived from $\mathcal{C}_2$ as follows:

$$\begin{array}{lll}
\Psi & = & \{X_e \mid e \in \wp\} \cup \{P_e \mid e \in \wp\} \quad \text{the set of constraint variables for a program } \wp \\
2^{Exn} & = & \text{the powerset of } Exn\text{, ordered by } \subseteq \\
\multicolumn{3}{l}{\mathcal{F}_2\colon (\Psi \to 2^{Exn}) \to (\Psi \to 2^{Exn})}
\end{array}$$

$$
\mathcal{F}_2(\varphi)(X_e) = \begin{cases}
\varphi(P_1) & \text{if } e = x (\text{handle var}) \text{ where } \texttt{handle}\ e_1\ \lambda x_\tau.e_2 \in \wp \\
\{\kappa \mid e_1\ e_2 \in \wp, \lambda x_\tau.e' \in Lam(e_1), \kappa \in \varphi(X_2)\} & \text{if } e = x (\text{normal var}) \\
\{\kappa\} \cup \varphi(X_1) & \text{if } e = \texttt{exn}\ \kappa\ e_1 \\
\varphi(X_1) & \text{if } e = \texttt{decon}\ e_1 \\
\{\kappa \mid \lambda x_\tau.e' \in Lam(e_1),\ \kappa \in \varphi(X_{e'})\} & \text{if } e = e_1\ e_2 \\
\varphi(X_2) \cup \varphi(X_3) & \text{if } e = \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \\
\varphi(X_1) \cup \varphi(X_2) & \text{if } e = \texttt{handle}\ e_1\ \lambda x_\tau.e_2 \\
\varphi(X_2) & \text{if } e = \texttt{fix}\ f\ \lambda x_\tau.e_1\ \texttt{in}\ e_2 \\
\emptyset & \text{otherwise}
\end{cases}
$$

$$
\mathcal{F}_2(\varphi)(P_e) = \begin{cases}
\varphi(P_1) & \text{if } e = \texttt{exn}\ \kappa\ e_1 \\
\varphi(P_1) & \text{if } e = \texttt{decon}\ e_1 \\
\{\kappa \mid \lambda x_\tau.e' \in Lam(e_1),\ \kappa \in \varphi(P_{e'})\} \cup \varphi(P_1) \cup \varphi(P_2) & \text{if } e = e_1\ e_2 \\
\varphi(P_1) \cup \varphi(P_2) \cup \varphi(P_3) & \text{if } e = \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \\
\varphi(X_1) & \text{if } e = \texttt{raise}\ e_1 \\
\varphi(X_1) \quad \text{if } e = \texttt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n \text{ and } type(e_1) = \tau'\ exn\ \wedge\ isExn(\tau') & \\
(\varphi(X_1) \setminus \{\kappa_1, \cdots, \kappa_n\}) & \\
\qquad \text{if } e = \texttt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n \text{ and } type(e_1) = \tau'\ exn\ \wedge\ \neg isExn(\tau') & \\
\varphi(X_1) \quad \text{if } e = \texttt{+raise}\ e_1\ \kappa \text{ and } type(e_1) = \tau'\ exn\ \wedge\ isExn(\tau') & \\
(\varphi(X_1) \cap \{\kappa\}) & \\
\qquad \text{if } e = \texttt{+raise}\ e_1\ \kappa \text{ and } type(e_1) = \tau'\ exn\ \wedge\ \neg isExn(\tau') & \\
\varphi(P_2) & \text{if } e = \texttt{handle}\ e_1\ \lambda x_\tau.e_2 \\
\varphi(P_2) & \text{if } e = \texttt{fix}\ f\ \lambda x_\tau.e_1\ \texttt{in}\ e_2 \\
\emptyset & \text{otherwise.}
\end{cases}
$$

It is straightforward to derive this function because the $\triangleright_2$ generates at most one constraint per X_i and P_i. That is, each $\supseteq$ is $=$.

We prove $Q(\text{fix}\mathcal{F}_2, \text{fix}\mathcal{F}_1)$ by fixpoint induction, where the assertion $Q(\varphi, \rho)$ for a program $\wp$ is:

$$\forall e \in \wp. \varphi(X_e) \supseteq |\rho(V_e)| \ \wedge\ \varphi(P_e) \supseteq |\rho(P_e)| \ \wedge\ \mathit{typeful}(\rho).$$

Note that we include the $\mathit{typeful}(\rho)$ assertion that we used in the proof of Proposition 2. This typefulness of ρ is necessary in proofs for the `-raise` and `+raise` cases.

Base case $Q(\emptyset, \emptyset)$ is trivially true. We prove that $Q(\mathcal{F}_2(\varphi), \mathcal{F}_1(\rho))$ holds given the induction hypothesis $Q(\varphi, \rho)$. That is, we need to show $\mathcal{F}_2(\varphi)(X_e) \supseteq |\mathcal{F}_1(\rho)(V_e)|$ and $\mathcal{F}_2(\varphi)(P_e) \supseteq |\mathcal{F}_1(\rho)(P_e)|$.

[VAR] $e = x$(handle variable) where $\texttt{handle}\ e_1\ \lambda x_\tau.e_2 \in \wp$.

$$\begin{array}{rcll} \mathcal{F}_2(\varphi)(X_x) & = & \varphi(P_1) & \text{(by definition)} \\ & \supseteq & |\rho(P_1)| & \text{(by I.H.)} \\ & = & |\mathcal{F}_1(\rho)(V_x)|. & \text{(by definition)} \end{array}$$

[VAR] $e = x$(normal variable).

$$\begin{array}{rcll} \mathcal{F}_2(\varphi)(X_x) & = & \{\kappa \mid e_1\ e_2 \in \wp, \lambda x_\tau.e' \in \mathit{Lam}(e_1), \kappa \in \varphi(X_2)\} & \text{(by definition))} \\ & \supseteq & \{v \mid e_1\ e_2 \in \wp, \lambda x_\tau.e' \in \rho(V_1), v \in |\rho(V_2)|\} & \text{(by Proposition 2} \\ & & & \quad\text{and I.H.)} \\ & = & |\mathcal{F}_1(\rho)(V_x)|. & \end{array}$$

[EXN] $e = \texttt{exn}\,\kappa\, e_1$.

$$\begin{array}{rcll}
\mathcal{F}_2(\varphi)(X_e) & = & \{\kappa\} \cup \varphi(X_1) & \text{(by definition)} \\
& \supseteq & \{\kappa\} \cup |\rho(V_1)| & \text{(by I.H.)}
\end{array}$$

By definition, $|\mathcal{F}_1(\rho)(V_e)| = |\{\kappa\, v \mid v \in \rho(V_1)\}| = \{\kappa\} \cup |\rho(V_1)|$.

Therefore, $\mathcal{F}_2(\varphi)(X_e) \supseteq |\mathcal{F}_1(\rho)(V_e)|$.

$$\begin{array}{rcll}
\mathcal{F}_2(\varphi)(P_e) & = & \varphi(P_1) & \text{(by definition)} \\
& \supseteq & |\rho(P_1)| & \text{(by I.H.)} \\
& = & |\mathcal{F}_1(\rho)(P_e)|. & \text{(by definition)}
\end{array}$$

[DCON] $e = \texttt{decon}\, e_1$.

$$\begin{array}{rcll}
\mathcal{F}_2(\varphi)(X_e) & = & \varphi(X_1) & \text{(by definition)} \\
& \supseteq & |\rho(V_1)| & \text{(by I.H.)} \\
& \supseteq & |\mathcal{F}_1(\rho)(V_e)|. & \text{(by definition)}
\end{array}$$

$$\begin{array}{rcll}
\mathcal{F}_2(\varphi)(P_e) & = & \varphi(P_1) & \text{(by definition)} \\
& \supseteq & |\rho(P_1)| & \text{(by I.H.)} \\
& = & |\mathcal{F}_1(\rho)(P_e)|. & \text{(by definition)}
\end{array}$$

[APP] $e = e_1\, e_2$.

$$\begin{array}{rcll}
\mathcal{F}_2(\varphi)(X_e) & = & \{\kappa \mid \lambda x_\tau.e' \in Lam(e_1), \kappa \in \varphi(X_{e'})\} & \text{(by definition)} \\
& \supseteq & \{v \mid \lambda x_\tau.e' \in \rho(V_1), v \in |\rho(V_{e'})|\} & \text{(by Proposition 2} \\
& & & \text{ and I.H.)} \\
& = & |\mathcal{F}_1(\rho)(V_e)|. & \text{(by definition)}
\end{array}$$

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(P_e) &= \{\kappa \mid \lambda x_\tau.e' \in Lam(e_1), \kappa \in \varphi(P_{e'})\} \cup \varphi(P_1) \cup \varphi(P_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition)} \\
&\supseteq \{v \mid \lambda x_\tau.e' \in \rho(V_1), v \in |\rho(P_{e'})|\} \cup |\rho(P_1)| \cup |\rho(P_2)| \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Proposition 2 and I.H.)} \\
&= |\mathcal{F}_1(\rho)(P_e)|. \qquad\qquad\qquad\qquad\qquad \text{(by definition)}
\end{aligned}
$$

[CASE] $e = \texttt{case}\, e_1\, \kappa\, e_2\, e_3$.

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(X_e) &= \varphi(X_2) \cup \varphi(X_3) && \text{(by definition)} \\
&\supseteq \{v \mid v \in |\rho(V_2)|,\ \kappa\, v' \in \rho(V_1)\} \cup \{v \mid v \in |\rho(V_3)|,\ \kappa'\, v' \in \rho(V_1),\ \kappa' \neq \kappa\} \\
& && \text{(by I.H.)} \\
&= |\mathcal{F}_1(\rho)(V_e)|. && \text{(by definition)} \\
\mathcal{F}_2(\varphi)(P_e) &= \varphi(P_1) \cup \varphi(P_2) \cup \varphi(P_3) && \text{(by definition)} \\
&\supseteq |\rho(P_1)| \cup |\rho(P_2)| \cup |\rho(P_3)| && \text{(by I.H.)} \\
&= |\mathcal{F}_1(\rho)(P_e)|. && \text{(by definition)}
\end{aligned}
$$

[RS] $e = \texttt{raise}\, e_1$.

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(P_e) &= \varphi(X_1) && \text{(by definition)} \\
&\supseteq |\rho(V_1)| && \text{(by I.H.)} \\
&= |\mathcal{F}_1(\rho)(P_e)|. && \text{(by definition)}
\end{aligned}
$$

[-RS] $e = \texttt{-raise}\ e_1\, \kappa_1 \cdots \kappa_n$ where $type(e_1) = \tau'\, exn \wedge isExn(\tau')$.

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(P_e) &= \varphi(X_1) && \text{(by definition)} \\
&\supseteq |\rho(V_1)| && \text{(by I.H.)} \\
&\supseteq |\mathcal{F}_1(\rho)(P_e)|. && \text{(by definition)}
\end{aligned}
$$

[-RS] $\quad e = \texttt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n$ where $type(e_1) = \tau'\ exn \wedge \neg isExn(\tau')$.

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(P_e) &= \varphi(X_1) \setminus \{\kappa_1, \cdots, \kappa_n\} \quad \text{(by definition)} \\
&\supseteq |\rho(V_1)| \setminus \{\kappa_1, \cdots, \kappa_n\} \quad \text{(by I.H.)}
\end{aligned}
$$

$$
\begin{aligned}
|\mathcal{F}_1(\rho)(P_e)| &= |\{\kappa' v \mid \kappa' v \in \rho(V_1), \forall i.\kappa_i \neq \kappa'\}| \quad \text{(by definition)} \\
&= \{\kappa' \mid \kappa' v \in \rho(V_1), \forall i.\kappa_i \neq \kappa'\} \\
&\quad \cup |\{v \mid \kappa' v \in \rho(V_1), \forall i.\kappa_i \neq \kappa'\}| \quad \text{(by definition of } |\bullet|\text{)} \\
&\quad \text{Note that the set } |\{v \mid \kappa' v \in \rho(V_1), \forall i.\kappa_i \neq \kappa'\}| \text{ is empty} \\
&\quad \text{because } \neg isExn(\tau') \text{ and the I.H. } typeful(\rho)(V_1)\text{. Thus,} \\
&= \{\kappa' \mid \kappa' v \in \rho(V_1), \forall i.\kappa_i \neq \kappa'\} \\
&\subseteq |\rho(V_1)| \setminus \{\kappa_1, \cdots, \kappa_n\} \quad \text{(by definition)}
\end{aligned}
$$

Therefore, $\mathcal{F}_2(\varphi)(P_e) \supseteq |\mathcal{F}_1(\rho)(P_e)|$.

[+RS] $\quad e = \texttt{+raise}\ e_1\ \kappa$ where $type(e_1) = \tau'\ exn \wedge isExn(\tau')$.

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(P_e) &= \varphi(X_1) \quad \text{(by definition)} \\
&\supseteq |\rho(V_1)| \quad \text{(by I.H.)} \\
&\supseteq |\mathcal{F}_1(\rho)(P_e)|. \quad \text{(by definition)}
\end{aligned}
$$

[+RS] $\quad e = \texttt{+raise}\ e_1\ \kappa$ where $type(e_1) = \tau'\ exn \wedge \neg isExn(\tau')$.

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(P_e) &= \varphi(X_1) \cap \{\kappa\} \quad \text{(by definition)} \\
&\supseteq |\rho(V_1)| \cap \{\kappa\} \quad \text{(by I.H.)}
\end{aligned}
$$

$$
\begin{aligned}
|\mathcal{F}_1(\rho)(P_e)| &= |\{\kappa\, v \mid \kappa\, v \in \rho(V_1)\}| && \text{(by definition)} \\
&= \{\kappa \mid \kappa\, v \in \rho(V_1)\} \,\cup\, |\{v \mid \kappa v \in \rho(V_1)\}| && \text{(by definition)} \\
&\quad \text{Note that the set } |\{v \mid \kappa v \in \rho(V_1)\}| \text{ is empty} \\
&\quad \text{because } \neg isExn(\tau') \text{ and I.H. } \textit{typeful}(\rho)(V_1)\text{. Thus,} \\
&= \{\kappa \mid \kappa\, v \in \rho(V_1)\} \\
&\subseteq |\rho(V_1)| \cap \{\kappa\} && \text{(by definition)}
\end{aligned}
$$

Therefore, $\mathcal{F}_2(\varphi)(P_e) \supseteq |\mathcal{F}_1(\rho)(P_e)|$.

[HNDL] $e = \texttt{handle}\, e_1 \lambda x_\tau . e_2$.

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(X_e) &= \varphi(X_2) \cup \varphi(X_1) && \text{(by definition)} \\
&\supseteq |\rho(V_2)| \cup |\rho(V_1)| && \text{(by I.H.)} \\
&= |\mathcal{F}_1(\rho)(V_e)|. && \text{(by definition)}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}_2(\varphi)(P_e) &= \varphi(P_2) && \text{(by definition)} \\
&\supseteq |\rho(P_2)| && \text{(by I.H.)} \\
&= |\mathcal{F}_1(\rho)(P_e)|. && \text{(by definition)}
\end{aligned}
$$

□

3.4.4 Function's Exception Constraint Construction $\triangleright_3$

It is wasteful to compute uncaught exceptions from every expression because exception-related expressions are sparse in a program. We need to sparsely generate constraints. Using the $\triangleright_2$ as our stepping stone, we arrive at our constraint system $\triangleright_3$ that generates constraints only for functions. The number of unknowns thus becomes proportional to the number of functions, not to the number of expressions. The least model of $\triangleright_3$-constraints for an input program is our analysis result: uncaught exceptions from each function.

$$
\begin{array}{lllll}
\kappa & \in & Exn & = & \{\kappa_1, \cdots, \kappa_N\} \quad \text{exception names in program } \wp \\
\kappa & \in & Packet & = & Exn \quad \text{raised exceptions}
\end{array}
$$

$$
\mathcal{I}(X_f) \subseteq Exn \qquad \mathcal{I}(P_f) \subseteq Packet \qquad \mathcal{X} ::= X_f \mid P_f
$$

$$
\begin{array}{lll}
Owner(x) & = & f \quad \text{where} \quad \lambda_f x_\tau.e \quad (f\text{'s parameter is } x) \\
\mathcal{I}(var(x)) & = & \mathcal{I}(X_{Owner(x)}) \\
\mathcal{I}(app_X(e_1, \mathcal{X})) & = & \{\kappa \mid \lambda_g x_\tau.e \in Lam(e_1),\ \kappa \in \mathcal{I}(X_g),\ \mathcal{I}(X_g) \supseteq \mathcal{I}(\mathcal{X})\} \\
\mathcal{I}(app_P(e_1, \mathcal{X})) & = & \{\kappa \mid \lambda_g x_\tau.e \in Lam(e_1),\ \kappa \in \mathcal{I}(P_g),\ \mathcal{I}(X_g) \supseteq \mathcal{I}(\mathcal{X})\}
\end{array}
$$

$\mathcal{I}(X_f \setminus_e \{\kappa_1, \cdots, \kappa_n\})$, $\mathcal{I}(X_f \cap_e \{\kappa\})$, $\mathcal{I}(\kappa)$, and $\mathcal{I}(\lambda x_\tau.e)$ are the same as in $\rhd_2$ (Figure 3.7, page 37).

Figure 3.9: Constructing Function's Exception Constraints: $\rhd_3$ (I)

In $\rhd_3$, set variables are indexed by the lambdas and handlee expressions of the input program. We assume that all lambdas and handlee expressions are uniquely named as f, g, h, etc. We subscript the lambda with its name: "$\lambda_f x_\tau.e$". Similarly for handlee expression such as "e_g" in "$\texttt{handle}\ e_g\ \lambda_h x_\tau.e_2$".

Every function (or handlee expression) f of the input program has two set constraints: $X_f \supseteq se$ and $P_f \supseteq se$. The set variable X_f is for exceptions that are "available" at f, and P_f is for uncaught exceptions during the call to f.

Consider the rule for application expression:

$$
[\text{APP}_{\rhd_3}] \quad \frac{f \ \rhd_3\ e_1 : \mathcal{C}_1 \qquad f\ \rhd_3\ e_2 : \mathcal{C}_2}{f\ \rhd_3\ e_1\, e_2 : \{X_f \supseteq app_X(e_1, X_f), P_f \supseteq app_P(e_1, X_f)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}
$$

The left-hand-side f of "$f \rhd_3 e$" indicates that the expression e appears in f. Thus, if f has a call $e_1\, e_2$, available exceptions X_f in f must include the exceptions $app_X(e_1, X_f)$ returned from the call. The uncaught exceptions P_f in f must in-

[VAR$_{\rhd_3}$] $$f \ \rhd_3 \ x\colon \ \{X_f \supseteq var(x)\}$$

[C$_{\rhd_3}$] $$f \ \rhd_3 \ \mathtt{1}\colon \emptyset$$

[ABS$_{\rhd_3}$] $$\frac{g \ \rhd_3 \ e_1\colon \ \mathcal{C}_1}{f \ \rhd_3 \ \lambda_g x_\tau.e_1\colon \ \mathcal{C}_1}$$

[FIX$_{\rhd_3}$] $$\frac{g \ \rhd_3 \ e_1\colon \ \mathcal{C}_1 \qquad f \ \rhd_3 \ e_2\colon \ \mathcal{C}_2}{f \ \rhd_3 \ \mathtt{fix}\, g\, \lambda_g x_\tau.e_1 \, \mathtt{in}\, e_2\colon \ \mathcal{C}_1 \cup \mathcal{C}_2}$$

[EXN$_{\rhd_3}$] $$\frac{f \ \rhd_3 \ e_1\colon \ \mathcal{C}_1}{f \ \rhd_3 \ \mathtt{exn}\, \kappa\, e_1\colon \ \{X_f \supseteq \kappa\} \cup \mathcal{C}_1}$$

[DCON$_{\rhd_3}$] $$\frac{f \ \rhd_3 \ e_1\colon \ \mathcal{C}_1}{f \ \rhd_3 \ \mathtt{decon}\, e_1\colon \ \mathcal{C}_1}$$

[APP$_{\rhd_3}$] $$\frac{f \ \rhd_3 \ e_1\colon \ \mathcal{C}_1 \qquad f \ \rhd_3 \ e_2\colon \ \mathcal{C}_2}{f \ \rhd_3 \ e_1\, e_2\colon \ \{X_f \supseteq app_X(e_1, X_f), P_f \supseteq app_P(e_1, X_f)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[CASE$_{\rhd_3}$] $$\frac{f \ \rhd_3 \ e_1\colon \ \mathcal{C}_1 \qquad f \ \rhd_3 \ e_2\colon \ \mathcal{C}_2 \qquad f \ \rhd_3 \ e_3\colon \ \mathcal{C}_3}{f \ \rhd_3 \ \mathtt{case}\, e_1\, \kappa\, e_2\, e_3\colon \ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

[RS$_{\rhd_3}$] $$\frac{f \ \rhd_3 \ e_1\colon \ \mathcal{C}_1}{f \ \rhd_3 \ \mathtt{raise}\, e_1\colon \ \{P_f \supseteq X_f\} \cup \mathcal{C}_1}$$

[-RS$_{\rhd_3}$] $$\frac{f \ \rhd_3 \ e_1\colon \ \mathcal{C}_1}{f \ \rhd_3 \ \mathtt{-raise}\, e_1\, \kappa_1 \cdots \kappa_n\colon \ \{P_f \supseteq X_f \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\}\} \cup \mathcal{C}_1}$$

[+RS$_{\rhd_3}$] $$\frac{f \ \rhd_3 \ e_1\colon \ \mathcal{C}_1}{f \ \rhd_3 \ \mathtt{+raise}\, e_1\, \kappa\colon \ \{P_f \supseteq X_f \cap_{e_1} \{\kappa\}\} \cup \mathcal{C}_1}$$

[HNDL$_{\rhd_3}$] $$\frac{g \ \rhd_3 \ e_g\colon \ \mathcal{C}_1 \qquad h \ \rhd_3 \ e_2\colon \ \mathcal{C}_2}{\begin{array}{ll} f \ \rhd_3 \ \mathtt{handle}\, e_g\, \lambda_h x_\tau.e_2\colon & \{ \ \ X_f \supseteq app_X(\lambda_h x_\tau.e_2, P_g) \cup X_g, \\ & \phantom{\{} \ \ P_f \supseteq app_P(\lambda_h x_\tau.e_2, P_g) \\ & \} \ \ \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Figure 3.10: Constructing Function's Exception Constraints: $\rhd_3$ (II)

clude the exceptions $app_P(e_1, X_f)$ uncaught during the call.

One interesting rule is [$\mathrm{VAR}_{\triangleright 3}$]. Because the constraint granularity is a function, constraint for a variable x must be expressed in terms of two functions: function f that variable x provides with exceptions and another function g that provides x with exceptions. The f is the function that appears in the left-hand-side of "$\triangleright_3\, x$" and the g is the function ($Owner(x)$) that has x as its argument. Therefore,

$$[\mathrm{VAR}_{\triangleright 3}] \quad f \;\triangleright_3\; x\colon\; \{X_f \supseteq X_{Owner(x)}\}.$$

One missing constraint is for the effect of passing exceptions through x when its owner $Owner(x)(\stackrel{\text{let}}{=} g)$ is called. This is expressed as $app_X(e_1, X_f)$'s third condition $\mathcal{I}(X_g) \supseteq \mathcal{I}(X_f)$ (in terms of caller f and callee g):

$$\mathcal{I}(app_X(e_1, X_f)) = \{\kappa \mid \lambda_g x_\tau.e \in Lam(e_1),\; \kappa \in \mathcal{I}(X_g),\; \mathcal{I}(X_g) \supseteq \mathcal{I}(X_f)\}.$$

The function-level exception constraint rule $\triangleright_3$ is a safe approximation of $\triangleright_2$:

Proposition 5 (Correctness of $\triangleright_3$) *For a closed term e, let $\triangleright_2 e\colon \mathcal{C}_2$ and* `main` $\triangleright_3\, e\colon \mathcal{C}_3$ *with their least models, $\mathcal{I}_2 = lm(\mathcal{C}_2)$ and $\mathcal{I}_3 = lm(\mathcal{C}_3)$. Then, if $g \triangleright_3 e'\colon \mathcal{C}'$ occurs during* `main` $\triangleright_3\, e\colon \mathcal{C}_3$ *then*

$$\mathcal{I}_3(X_g) \supseteq \mathcal{I}_2(X_{e'}) \quad \textit{and} \quad \mathcal{I}_3(P_g) \supseteq \mathcal{I}_2(P_{e'}).$$

Proof. We will prove that for any model $\mathcal{I}$ of $\mathcal{C}_3$, the above holds.

Note that the least model $\mathcal{I}_2$ is equivalent to the $\subseteq$-least fixpoints $\mathrm{fix}\mathcal{F}_2$. $\mathcal{F}_2$ is defined in the proof of Proposition 4.

We prove $Q(\mathrm{fix}\mathcal{F}_2)$ by the fixpoint induction, where the assertion $Q(\varphi)$ for a program $\wp$ is:

$$\forall \mathit{model}\ \mathcal{I}\ \mathit{of}\ \mathcal{C}_3.\text{“}f \rhd_3 e : \mathcal{C}\text{” } \mathit{occurs\ during}\ (\texttt{main} \rhd_3 \wp : \mathcal{C}_3) \ \Rightarrow\ \mathcal{I}(X_f) \supseteq \varphi(X_e) \wedge \mathcal{I}(P_f) \supseteq \varphi(P_e).$$

Base case $Q(\emptyset)$ trivially holds. We prove that $Q(\mathcal{F}_2(\varphi))$ holds given the induction hypothesis $Q(\varphi)$.

In the following proof, for each case $f \rhd_3 \mathit{expr}$ we abbreviate the expr by e.

[VAR] $f \rhd_3 x$(handle variable) where $\texttt{handle}\ e_g\ \lambda_h x_\tau.e_2 \in \wp$.

$$\begin{array}{rcll} \mathcal{I}(X_f) & \supseteq & \mathcal{I}(X_{Owner(x)}) = \mathcal{I}(X_h) & (\text{by [VAR}_{\rhd 3}]) \\ & \supseteq & \mathcal{I}(P_g) & (\text{by [HNDL}_{\rhd 3}]) \\ & \supseteq & \varphi(P_g) & (\text{because “}g \rhd_3 e_g\text{” occurs and by I.H.}) \\ & = & \mathcal{F}_2(\varphi)(X_x) & (\text{by definition}) \end{array}$$

[VAR] $f \rhd_3 x$(normal variable).

By [VAR$_{\rhd 3}$], $\mathcal{I}(X_f) \supseteq \mathcal{I}(X_{Owner(x)})$. Let a function g be the owner of x: $\lambda_g x_\tau.e'$.

For each “$k \rhd_3\ e_1\ e_2$” that occurs at the $Owner(x)$'s call site, “$k \rhd_3\ e_2$” occurs. Thus by I.H.,

$$\begin{array}{rcll} \varphi(X_2) & \subseteq & \mathcal{I}(X_k) & \\ & \subseteq & \mathcal{I}(X_g) & (\text{by [APP}_{\rhd 3}]) \end{array}$$

Therefore, $\mathcal{I}(X_f) \supseteq \{\kappa \mid e_1\ e_2 \in \wp, \lambda x_\tau.e' \in \mathit{Lam}(e_1), \kappa \in \varphi(X_2)\} = \mathcal{F}_2(\varphi)(X_x)$.

[EXN] $f \rhd_3 \texttt{exn}\ \kappa\ e_1$.

$$
\begin{array}{rcll}
\mathcal{I}(X_f) & \supseteq & \{\kappa\} & \text{(by [EXN}_{\rhd 3}\text{])} \\
\mathcal{I}(X_f) & \supseteq & \varphi(X_1) & \text{(because "}f \rhd_3 e_1\text{" occurs and by I.H.)} \\
\text{Therefore, } \mathcal{I}(X_f) & \supseteq & \{\kappa\} \cup \varphi(X_1) & \\
 & = & \mathcal{F}_2(\varphi)(X_e) & \text{(by definition)}
\end{array}
$$

[APP] $e = f \rhd_3 e_1\, e_2$.

$$
\begin{array}{rcl}
\mathcal{I}(X_f) & \supseteq & \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \mathcal{I}(X_g), \mathcal{I}(X_g) \supseteq \mathcal{I}(X_f)\} \quad \text{(by [APP}_{\rhd 3}\text{])} \\
 & \supseteq & \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \varphi(X_{e'})\} \quad \text{(because "}g \rhd_3 e'\text{" occurs and by I.H.)} \\
 & = & \mathcal{F}_2(\varphi)(X_e). \qquad \text{(by definition)} \\
\mathcal{I}(P_f) & \supseteq & \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \mathcal{I}(P_g), \mathcal{I}(X_g) \supseteq \mathcal{I}(X_f)\} \quad \text{(by [APP}_{\rhd 3}\text{])} \\
 & \supseteq & \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \varphi(P_{e'})\} \quad \text{(because "}g \rhd_3 e'\text{" occurs and by I.H.)} \\
\mathcal{I}(P_f) & \supseteq & \varphi(P_1) \qquad \text{(because "}f \rhd_3 e_1\text{" occurs and by I.H.)} \\
\mathcal{I}(P_f) & \supseteq & \varphi(P_2) \qquad \text{(because "}f \rhd_3 e_2\text{" occurs and by I.H.)}
\end{array}
$$

Therefore,

$$
\begin{array}{rcl}
\mathcal{I}(P_f) & \supseteq & \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \varphi(P_{e'})\} \cup \varphi(P_1) \cup \varphi(P_2) \\
 & = & \mathcal{F}_2(\varphi)(P_e). \quad \text{(by definition)}
\end{array}
$$

[RS] $e = f \rhd_3 \texttt{raise}\, e_1$.

$$
\begin{array}{rcll}
\mathcal{I}(P_f) & \supseteq & \mathcal{I}(X_f) & \text{(by [RS}_{\rhd 3}\text{])} \\
 & \supseteq & \varphi(X_1) & \text{(because "}f \rhd_3 e_1\text{" occurs and by I.H.)} \\
 & = & \mathcal{F}_2(\varphi)(P_e). & \text{(by definition)}
\end{array}
$$

[-RS] $e = \texttt{-raise}\, e_1\, \kappa_1 \cdots \kappa_n$ where $type(e_1) = \tau'\, exn \wedge isExn(\tau')$.

$$\begin{aligned}
\mathcal{I}(P_f) &\supseteq \mathcal{I}(X_f \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\}) && \text{(by [-RS}_{\rhd 3}\text{])} \\
&= \mathcal{I}(X_f) && \text{(by definition of } \setminus_{e_1}\text{)} \\
&\supseteq \varphi(X_1) && \text{(because ``} f \rhd_3 e_1 \text{'' occurs and by I.H.)} \\
&= \mathcal{F}_2(\varphi)(P_e). && \text{(by definition)}
\end{aligned}$$

[-RS] $e = \texttt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n$ where $type(e_1) = \tau'\ exn\ \wedge\ \neg isExn(\tau')$.

$$\begin{aligned}
\mathcal{I}(P_f) &\supseteq \mathcal{I}(X_f \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\}) && \text{(by [-RS}_{\rhd 3}\text{])} \\
&= \mathcal{I}(X_f) \setminus \{\kappa_1, \cdots, \kappa_n\} && \text{(by definition of } \setminus_{e_1}\text{)} \\
&\supseteq \varphi(X_1) \setminus \{\kappa_1, \cdots, \kappa_n\} && \text{(because ``} f \rhd_3 e_1 \text{'' occurs and by I.H.)} \\
&= \mathcal{F}_2(\varphi)(P_e). && \text{(by definition)}
\end{aligned}$$

[+RS] $e = \texttt{+raise}\ e_1\ \kappa$ where $type(e_1) = \tau'\ exn\ \wedge\ isExn(\tau')$.

$$\begin{aligned}
\mathcal{I}(P_f) &\supseteq \mathcal{I}(X_f \cap_{e_1} \{\kappa\}) && \text{(by [+RS}_{\rhd 3}\text{])} \\
&= \mathcal{I}(X_f) && \text{(by definition of } \cap_{e_1}\text{)} \\
&\supseteq \varphi(X_1) && \text{(because ``} f \rhd_3 e_1 \text{'' occurs and by I.H.)} \\
&= \mathcal{F}_2(\varphi)(P_e). && \text{(by definition)}
\end{aligned}$$

[+RS] $e = \texttt{+raise}\ e_1\ \kappa$ where $type(e_1) = \tau'\ exn\ \wedge\ \neg isExn(\tau')$.

$$\begin{aligned}
\mathcal{I}(P_f) &\supseteq \mathcal{I}(X_f \cap_{e_1} \{\kappa\}) && \text{(by [+RS}_{\rhd 3}\text{])} \\
&= \mathcal{I}(X_f) \cap \{\kappa\} && \text{(by definition of } \cap_{e_1}\text{)} \\
&\supseteq \varphi(X_1) \cap \{\kappa\} && \text{(because ``} f \rhd_3 e_1 \text{'' occurs and by I.H.)} \\
&= \mathcal{F}_2(\varphi)(P_e). && \text{(by definition)}
\end{aligned}$$

[HNDL] $e = f \rhd_3 \texttt{handle}\ e_g \lambda_h x_\tau.e_2$.

$$
\begin{aligned}
\mathcal{I}(X_f) &\supseteq \{\kappa \mid \kappa \in \mathcal{I}(X_h), \mathcal{I}(X_h) \supseteq \mathcal{I}(P_g)\} \cup \mathcal{I}(X_g) \quad \text{(by [HNDL}_{\triangleright 3}\text{])} \\
&= \mathcal{I}(X_h) \cup \mathcal{I}(X_g) \quad \text{(because any model } \mathcal{I} \text{ of } \mathcal{C}_3 \text{ satisfies } \mathcal{I}(X_h) \supseteq \mathcal{I}(P_g)) \\
&\supseteq \varphi(X_2) \cup \mathcal{I}(X_g) \quad \text{(because “}h \triangleright_3 e_2\text{” occurs and by I.H.)} \\
&\supseteq \varphi(X_2) \cup \varphi(X_g) \quad \text{(because “}g \triangleright_3 e_g\text{” occurs and by I.H.)} \\
&= \mathcal{F}_2(\varphi)(X_e) \quad \text{(by definition)} \\
\mathcal{I}(P_f) &\supseteq \{\kappa \mid \kappa \in \mathcal{I}(P_h), \mathcal{I}(X_h) \supseteq \mathcal{I}(P_g)\} \quad \text{(by [HNDL}_{\triangleright 3}\text{])} \\
&= \mathcal{I}(P_h) \quad \text{(because any model } \mathcal{I} \text{ of } \mathcal{C}_3 \text{ satisfies } \mathcal{I}(X_h) \supseteq \mathcal{I}(P_g)) \\
&\supseteq \varphi(P_2) \quad \text{(because “}h \triangleright_3 e_2\text{” occurs and by I.H.)} \\
&= \mathcal{F}_2(\varphi)(P_e) \quad \text{(by definition)}
\end{aligned}
$$

□

Example 1 As an analysis example, consider the following program:

(1) `fun m() = f(exn` κ `1)`

(2) `fun f(x) = handle g(x)` $\lambda_h\ y.\ 1$

(3) `fun g(x) = raise x`

In SML, (2) is `g(x) handle _ => 1.`

From line (1),

$$
\begin{aligned}
&X_m \supseteq \kappa \\
&X_f \supseteq X_m,\ X_m \supseteq X_f \quad (\text{from } X_m \supseteq app_X(f, X_m)) \\
&P_m \supseteq P_f \quad (\text{from } P_m \supseteq app_P(f, X_m))
\end{aligned}
$$

From line (2)

$$
\begin{aligned}
&X_g \supseteq X_f,\ X_f \supseteq X_g \quad (\text{from } X_f \supseteq app_X(g, X_f)) \\
&P_f \supseteq P_h,\ X_h \supseteq P_g \quad (\text{from } P_f \supseteq app_P(h, P_g))
\end{aligned}
$$

From line (3)

$$P_g \supseteq X_g$$

The least model of the above 9 constraints is the least solution of the equations:

$$\begin{array}{llll} X_m = \{\kappa\} & X_f = X_m \cup X_g & X_g = X_f & X_h = P_g \\ P_m = P_f & P_f = P_h & P_g = X_g & \end{array}$$

The least solution is:

$$\begin{array}{lll} X_m = \{\kappa\} & X_f = \{\kappa\} & X_g = \{\kappa\} \\ P_m = \emptyset & P_f = \emptyset & P_g = \{\kappa\} \end{array}$$

3.5 Implementation

This section explains the implementation of our analysis. For improved accuracy, we safely sharpen some constraint rules by type information and use extra data for exception values. Because of SML's polymorphic type system, some definitions are changed. We present the analyzer overview and describe both our control flow analyzer and the exception analyzer in detail.

3.5.1 Typeful Constraints for Improved Accuracy

Some constraint rules of $\triangleright_3$ can be safely sharpened using types. Our actual analysis implementation uses this sharpened $\triangleright_3$ rules. 1) a function f has exceptions through a variable x only when the x is of an exception type and 2) exceptions X_f in f are returned only when f's return type is an exception type:

$$
\begin{array}{lcl}
\mathcal{I}(app_X(e_1,\mathcal{X})) & = & \{\kappa \mid \lambda_f x_\tau.e \in Lam(e_1),\ \kappa \in \mathcal{I}(X_f|isExn(type(e))), \\
 & & \quad \mathcal{I}(X_f) \supseteq \mathcal{I}(\mathcal{X}|isExn(\tau))\} \\
\mathcal{I}(app_P(e_1,\mathcal{X})) & = & \{\kappa \mid \lambda_f x_\tau.e \in Lam(e_1),\ \kappa \in \mathcal{I}(P_f),\ \mathcal{I}(X_f) \supseteq \mathcal{I}(\mathcal{X}|isExn(\tau))\} \\
\mathcal{I}(var(x)) & = & \mathcal{I}(X_{Owner(x)}|isExn(type(x))) \\
X_f & \supseteq & app_X(\cdots) \cup (X_g|isExn(type(e_g))) \qquad \text{in [HNDL}_{\triangleright_3}]
\end{array}
$$

where $\mathcal{I}(\mathcal{X}|cond) = \mathcal{I}(\mathcal{X})$ if the $cond$ true, $\emptyset$ otherwise.

The correctness of this new $\triangleright_3$ can be proved with respect to a typeful version of $\triangleright_2$. The least solution of typeful $\triangleright_2$-constraints maps non-exception-typed expressions to the empty set. The typeful $\triangleright_2$ is consistent with $\triangleright_1$ because $\triangleright_1$ is already typeful. The $\triangleright_2$ becomes typeful by the new [DCON$_{\triangleright_2}$] rule:

$$
[\text{DCON}_{\triangleright_2}] \quad \frac{\triangleright_2\, e_1 :\ \mathcal{C}_1}{\triangleright_2\, \texttt{decon}\, e_1 :\ \{X_e \supseteq (X_1|isExn(type(e))), P_e \supseteq P_1\} \cup \mathcal{C}_1}
$$

Example 2 Consider the following program:

```
(1)            fun f(x) = ··· (exn κ1 1)··· g(1)···
(2)            fun g(x) = raise (exn κ2 x)
```

Note that `g` raises only κ_2. If our constraints are un-typed, we generate constraints that pass `f`'s exception κ_1 to `g` because of the call `g(1)`. This won't happen in our new rules, because `g`'s argument type is not exception. From line (1),

$$X_f \supseteq \kappa_1,\ X_f \supseteq app_X(g, X_f),\ P_f \supseteq app_P(g, X_f)$$

From line (2),

$$X_g \supseteq \kappa_2,\ P_g \supseteq \kappa_2,\ P_g \supseteq X_g$$

The $X_f \supseteq app_X(g, X_f)$ implies $X_g \supseteq \emptyset$ because g's argument type is `int`. The least solution hence maps P_g to $\{\kappa_2\}$. Meanwhile, untypeful definition of $X_f \supseteq app_X(g, X_f)$ generates $X_g \supseteq X_f$ and the least solution becomes to map X_g to $\{\kappa_1, \kappa_2\}$, concluding that P_g may raise all these exceptions.

3.5.2 Handling of Exception's Arguments

Because the analysis does not recognize the arguments of exceptions unless the arguments were exceptions, it may lead into a too conservative result for some programs.

Example 3 Consider the following program that has no uncaught exception:

```
        exception Fail of int
(1)     fun f() = g() handle Fail(1) => 1
        fun g() = raise (exn Fail 1)
```

Because the handler pattern "`Fail(1)`" is not exhaustive for the argument part, the handler is annotated with "`+raise x Fail`" expression. This `+raise` expression makes our analysis conclude that `f` has an escaping exception `Fail`.

Resolving this problem by adding constraints for non-exception values and risking the subsequent increase of the analysis cost is not appealing in two reasons. Incomplete handler patterns for exception's argument(like the above example) is rare, and the existing pattern compiler[6] already can warn of incomplete patterns for exception's arguments unless the argument type is an exception.

[6]An SML's datatype has a fixed number of ways to construct its values, and patterns are combinations of such constructors.

Our implementation makes $\rhd_3$ bring extra data by which the programmer trim overly conservative results. We pair an exception name κ with the index of the expression (`exn` κ e) where the exception is made.[7] Given a pair of exception name and its birth place information, the programmer can decide which possibly-uncaught exceptions are real, assuming that the birth place expression has the argument data explicit in the text. In the above example, the programmer can safely decide the possibly-uncaught exception `Fail` is not real because its birth place is "`(exn Fail 1)`" hence the handler pattern "`Fail(1)`" is exhaustive enough.

This can be achieved by a slight change to $\rhd_3$. The exception space becomes the set of tuples:

$$Exn = \{\kappa_1, \cdots, \kappa_N\} \times Expr$$

The rule for `exn` expression becomes:

$$[\text{EXN}_{\rhd_3}] \quad \frac{f \ \rhd_3 \ e_1 : \ \mathcal{C}_1}{f \ \rhd_3 \ \texttt{exn}\,\kappa\, e_1 : \ \{X_f \supseteq \langle\kappa, e\rangle\} \cup \mathcal{C}_1} \quad \text{where } e = \texttt{exn}\,\kappa\, e_1$$

And

$$X_1 \setminus_e \{\kappa_1, \cdots, \kappa_n\} \text{ and } X_1 \cap_e \{\kappa\}$$

removes (respectively, selects) tuples headed by κ_i's (respectively, by κ).

3.5.3 Adapting $\rhd_3$ to Standard ML

Because of SML's polymorphic types,

[7] This can be understood as abstracting the expression values by the expression index. Similar technique is used in abstracting memory locations: every malloc expression is an abstract location, representing all the locations allocated at that point during execution.

- *isExn* (Figure 3.7, page 37) needs to be conservative. The new definition is:

$$
\begin{array}{llll}
isExn(\iota \vee \tau \to \tau') & = & \text{false} & \text{constant or function type} \\
isExn(\alpha \vee \tau\ exn) & = & \text{true} & \text{generic type var or exn type} \\
isExn(\tau\ ref) & = & isExn(\tau) & \text{reference type} \\
isExn(\tau_1 \times \tau_2) & = & isExn(\tau_1) \text{ or } isExn(\tau_2) & \text{record type} \\
isExn(u) & = & \exists \kappa \in Con(u).isExn(Arg(\kappa)) & \text{user-defined datatype } u
\end{array}
$$

  The last case is for when a data type u's constructor $\kappa \in Con(u)$ receives exceptions as its argument.

- *Lam*'s (Figure 3.6, page 34) last case must test for type unifiability ($\approx$) instead of type equality: In this case, $type_{\wp}(e)$ - for an expression e of a program $\wp$ - indicates the SML type of the expression, determined by the let-polymorphic type inference system[51, 53, 68].

3.5.4 Analyzer Overview

We implemented our exception analyzer in SML/NJ 110 compiler[23]. The analyzer is completely embedded in the compiler to be interactively used inside the compiler's print-eval loop and also with the compilation manager, CM[12, 11].

The SML/NJ compilation process is shown in Figure 3.11. When the compiler gets an input SML program, it transforms the source program to an `ast` representation by *lexing* and *parsing*. Then, the compiler infers the type of the `ast` program and transforms it to an `absyn` format with type annotations. The `absyn` program is translated to a `lambda` representation by compiling pattern matching and simplifying language constructs and type information. And then, the `lambda` program is passed to a back-end of the compiler.

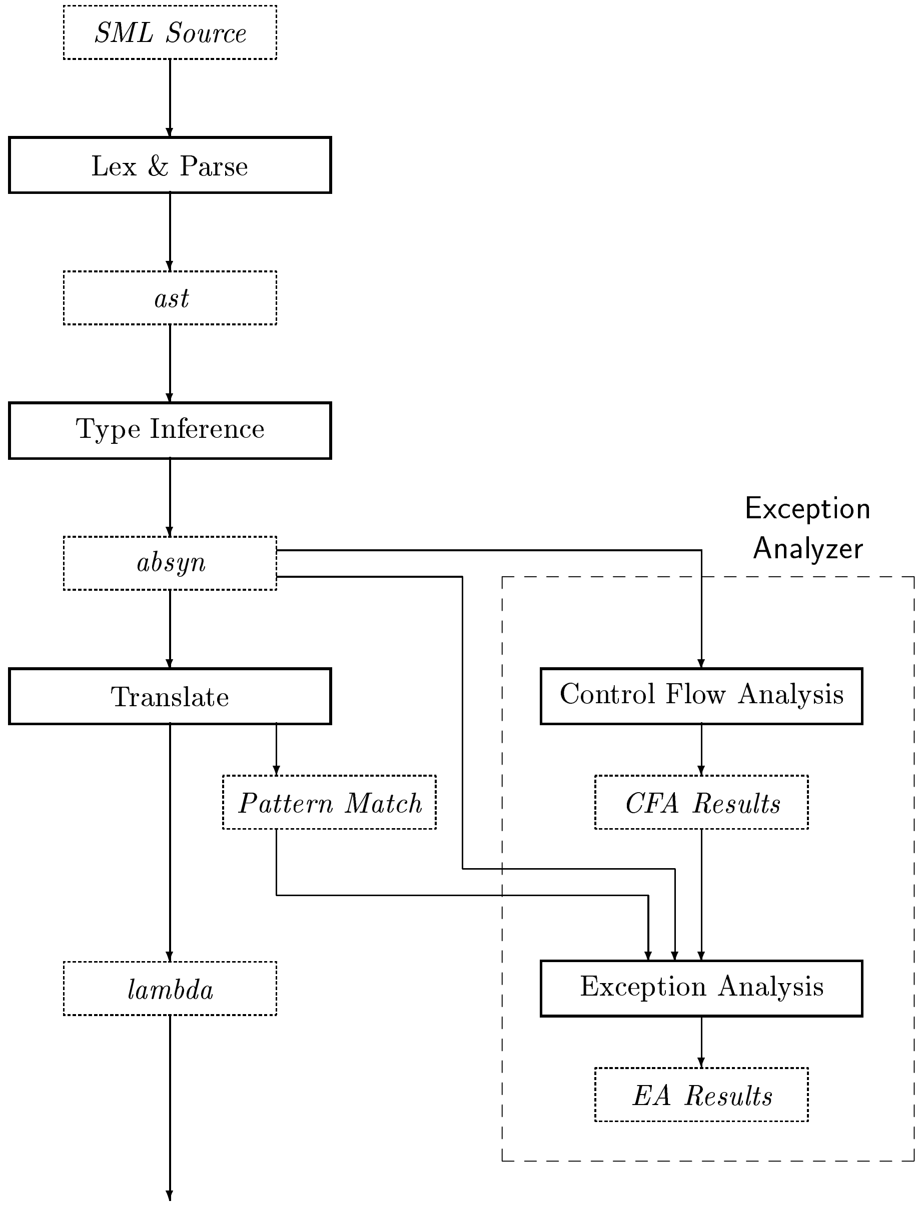

Figure 3.11: Exception Analyzer in the SML/NJ 110 Compiler

The integration of our analyzer to the SML/NJ compiler is also shown in Figure 3.11. In order to get type information for our control flow analyzer, we use the `absyn` representations as inputs for our analyzer. The `lambda` representations also have type information and have less language constructs than `absyn`. But many compiler-generated functions in `lambda` may make the performance and accuracy of the control flow analysis worse. And source program information like user defined function names and source locations are lost during the translation from `absyn` to `lambda`. Since our exception analyzer needs information for exhaustiveness of pattern matching, we get the information after the translation from `absyn` to `lambda`.

3.5.5 Control Flow Analyzer

Our control flow analyzer gets an `absyn` input program and results in a safe call table *Lam* (Section 3.4.2). For each function-typed expression in the input program, *Lam* gives its possible set of function values. When an exception's argument is a function, we estimate that an exception's argument functions are those whose types are unifiable with the type of the exception's argument.

The control flow analyzer consists of two steps: building equations and solving equations. The analyzer firstly sets up equations according to the rules in Figure 3.6 which are dependent only on source information. And then it solves the equations according to the other rules in Figure 3.6 by the fixpoint iterations.

3.5.6 Exception Analyzer

Our exception analyzer gets an `absyn` input program, pattern exhaustiveness information, and control flow analysis results. Pattern exhaustiveness information is needed for dealing with `handle` expressions. In our analysis, the handler patterns are always augmented with an extra raise (`-raise`) expression, in order to

program	lines	cost 1 (sec)[a]	cost 2 (sec)[b]	analysis result
Knuth-Bendix.sml	519	0.96[c]	0.47	1 (1x,1r,10h)[d]
ml-lex.sml	1166	1.50	0.79	4 (10x,19r,10h)
pathname.sml	426	0.12	0.06	4 (4x,6r,3h)
string-cvt.sml	455	0.17	0.09	1 (1x,10r,4h)
class compiler	3477	17.01	8.62	3 (11x,34r,4h)

[a] Control-flow analysis and exception analysis: in SML, run on Sun Ultra Sparc 1, 167 MHz, SunOS 5.5.1, 256 MB, compiled by SML/NJ 110.03

[b] Control-flow analysis and exception analysis: in SML, run on Sun Ultra Sparc 2, 296 MHz, SunOS 5.5.1, 768 MB compiled by SML/NJ 110.03

[c] SML user+system+gc time

[d] 1 may-uncaught exceptions from top-level functions among 1 exns(1x), 1 raise exprs(1r), and 10 handlers(10h).

Figure 3.12: Experimental Results

re-raise exceptions that are not caught (Section 3.3). And if a handler's pattern for exception's argument part is not complete, the exception is explicitly re-raised by `+raise` (Section 3.3). Control flow estimation is needed for analyzing application expressions (Figure 3.9).

The exception analyzer also consists of two steps: building equations and solving equations. The analyzer firstly sets up equations according to the rules in Figure 3.10, and then it solves the constraints by using a worklist algorithm[19].

3.6 Experimental Results

A prototype's preliminary performance is shown in Figure 3.12. The analysis speed ranges from 200 to 3500 SML-lines/sec. ([71] ran at 0.2 SML-lines/sec and [25] at about 10 SML-lines/sec). We still expect some improvements in the analysis speed as we better implement the control flow analysis part. In particular, a performance bottleneck is in computing the table that partitions user functions into unifiable ones.

This process' cost is proportional to the "size" of function types in the program. This is why the control flow analysis speed is not proportional to the program size.

Computing the *Lam* uses the fixpoint iteration of cubic complexity. Computing the constraints' least solution also uses the conventional fixpoint iteration[8] of cubic complexity.

The analysis accuracy is satisfying. We manually checked the above test programs and found that the reported exceptions for Knuth-Bendix.sml, pathname.sml, string-cvt.sml, and compiler.sml can actually be uncaught. For ML-Lex.sml, the analysis reports 4 may-uncaught exceptions, among which 3 exceptions can really escape.

To our knowledge, the cost-accuracy performance of Guzmán and Suárez [34]'s exception type system is not available to compare with ours.

3.7 Related Work

In this section, we present previous exception analyses for ML programs. Leroy and Pessaux [57, 47]'s exception analysis is for Objective Caml [46] programs and others are for Standard ML programs.

3.7.1 Guzmán and Suárez's Exception Analysis

Guzmán and Suárez [34] developed a simple type and effect system [48, 49, 42, 65] to collect uncaught exceptions for a simplified core ML. Their extended type system annotates the function type constructor ($\rightarrow$) with the set of possible exceptions that may be raised by applying this function. And the type of each expression is tagged with the set of exceptions that may be raised when the expression is evaluated.

[8] The iterative method is possible because the set domain is finite (the set of exception names in the input program) and all set operators(set union "$\cup$", "$X_f \cap \{\kappa\}$", "$X_f \setminus \{\kappa_1 \cdots \kappa_n\}$", etc.) are monotonic.

Their system does not treat exceptions as first-class values, nor exceptions carrying arguments. Thus every raise expression has an exception name at its exception part, like "`raise Fail`", and exceptions cannot have any arguments. Their approach may not be strong enough to deal with the realistic use of the ML exceptions.

3.7.2 Yi's Exception Analysis

Yi [71] used an abstract interpretation [20, 21] to estimate uncaught exceptions in ML programs. He used the collecting analyzer generator Z1 [70, 73] in specifying and implementing his analysis. The analysis specification is an abstract interpreter, and Z1 generates an executable, *collecting analyzer* from the specification. The collecting analysis computes stable program states at each expression point of the input program. The program state, in this case, contains the information about uncaught exceptions. The analysis design and its correctness proof is done at once by a sound abstraction of the ML's concrete semantics.

His implementation has been successfully used to analyze realistic SML programs, but it is not fast enough to be usable in practice. For example, the analysis prototype takes 6 hours for ML-Lex program. In average, 38 iterative evaluations for each expression are required to reach to the fixed-point (the analysis end point).

To improve the analysis speed, he presented a semantics-based sparse analysis technique. Before the analysis evaluates an expression, the analysis conservatively decides whether the evaluation will have the exception behavior. If the evaluation has the exception behavior, the analysis evaluates the expression, otherwise, it skips the evaluation.

3.7.3 Fähndrich and Aiken's Exception Analysis

Fähndrich and Aiken [26, 27] developed an exception analysis tool by using their BANE (Berkeley ANalysis Engine) [6] toolkit for constraint-based program analyses. As in effect systems, their system casts the type and exception inference. Every expression has a type and an effect, and the type describes the set of possible normal values of the expression, whereas the effect describes the set of exceptions that may be raised during the evaluation.

Their analysis uses the description file of the analysis results which is used for the visualization, and the file size does not scale well. For example, the description file for a points-to analysis for C written in BANE [29] uses more than 100 MB of disk space. They consider factoring repeated information in the description file.

3.7.4 Leroy and Pessaux's Exception Analysis

Leroy and Pessaux [57, 47] developed an exception analysis which relies on a unification-based type inference in a non-standard type system. In the style of effect systems, their exception analysis is presented as a type inference algorithm for a non-standard type system. The type system uses unified mechanisms both to collect the sets of uncaught exceptions of expressions and to refine the usual ML types by more precise information about the possible values of expressions. As in effect systems, the typing rules for their type system is the specifications for the exception analysis and type inference algorithm is the actual analysis.

Their analyzer has been implemented to cover the whole Objective Caml [46] language and it is efficient and precise. Their analyzer accurately handles arguments carried by exceptions. While the analyses of Yi and Ryu's and Fähndrich and Aiken's are compiled by SML/NJ, Leroy and Pessaux's analyzer is compiled by Objective Caml. So, comparing performances between them is not clear.

The results of their analysis should be displayed in a readable way. They are developing a graphical browser for exception analysis results that allows the programmer to select different levels of display.

Chapter 4

Multithreaded Exception Handling Mechanism

In this chapter, we present multithreaded exception handling mechanisms. We introduce how to create, terminate, and synchronize threads in some modern programming languages. We investigate a problem of multithreaded exception handling in Java and show several existing ad-hoc solutions for this problem.

4.1 Multithreading Mechanisms in Modern Languages

This section introduces multithreading mechanisms in some modern programming languages. Many modern languages such as C++[63], Java[30, 31], Ada[35, 33], Modula-3[14, 15], and ML[53, 52, 46] provide the programmer with multithreading mechanisms. They provide this mechanism either as a language construct or as a library. Threads provide a way for a program to do multiple tasks concurrently.

We will consider Java's multithreading mechanism for our multithreaded exception analysis in chapter 5. Java provides language support for multithreading mechanism and has the most expressive synchronization mechanism among modern programming languages.

4.1.1 In C++

Neither C nor C++ support multithreaded programming as a language construct.[50] Unix programmers rely on libraries provided with specific compilers or provided by third-party vendors. Win32 provides constructs for threading and synchronization, and MFC adds an object-oriented flavor to the constructs. We present the multi-threading mechanism in C++ by using the case of MFC.

In MFC, `AfxBeginThread()` defines threads:

`WinThread* AfxBeginThread(` *arguments* `).`

MFC distinguishes between user interface threads and worker threads: user interface threads handle user input and respond to user events, while worker threads perform background processing. Parameters of `AfxBeginThread()` are different for a worker thread and a user interface thread.

Threads are terminated in MFC when their function completes its execution. Programmers can terminate a thread by calling the `AfxEndThread()` function:

```
void AfxEndThread( UINT nExitCode );.
```

MFC provides several classes for synchronization, including `CMutex`, `Csemaphore`, `CCriticalSection`, and `CEvent`. Programmers can use mutexes, semaphores, and critical sections with a lock. To synchronize threads based on an object's state, create a synchronization object and call the method `WaitForSingleObject()` or

`WaitForMultipleObjects()` to await notification that the state has changed:

```
DWORD WaitForSingleObject arguments
DWORD WaitForSingleObject arguments
```

The method `SetEvent()` notifies an synchronization object by changing its state.

C++ exceptions are thread safe in the sense that multiple threads within a process can throw and catch exceptions concurrently.[4] However, exceptions do not propagate from one thread to another, nor can one thread catch an exception thrown by another thread.

4.1.2 In Java

Java provides language support for multithreading mechanism. Multithreads in Java independently execute code that operates on values and objects residing in a shared main memory. To synchronize threads, Java uses monitors, which are a high-level mechanism for allowing only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of locks; there is a lock associated with each object.

Java provides two ways of associating a method with a `Thread`: 1. Declare a subclass of `Thread` that defines a `run` method. 2. Pass a reference to an object that implements the `Runnable` interface to a `Thread` constructor. To define a thread by inheriting from `Thread`, only specify a run method in a new class and then call its start method. When the `start()` method of such a `Thread` object is called, the thread calls this `run` method:

```
public synchronized native void start().
```

And to define a thread by implementing the `Runnable` interface, use a form of the `Thread` constructor that takes an object of type `Runnable`. When the `start()` method of such a `Thread` object is called, the thread calls the `run()` method of the `Runnable` object.

Java threads terminated when one of the following things happens: 1. The `run` method called by the `Thread` returns. 2. An exception is thrown that causes the `run()` method to be exited. 3. The `stop()` method of the `Thread` is called:

```
public final void stop().
```

The `stop()` method throws a `ThreadDeath` object in the `run()` method of the thread.

In Java, the `wait()`, `notify()`, and `notifyAll()` methods support synchronization of threads:

```
public final native void wait ( arguments )
public final native void notify ()
public final native void notifyAll ()
```

The method `wait()` puts a thread to sleep until `notify()` or `notifyAll()` is called. While the `notifyAll()` method wakes up all the waiting threads, the `notify()` method arbitrarily awakens just one of the threads. The awakened thread resumes executing as soon as it regains this object's lock.

4.1.3 In Ada

Ada provides language support for concurrent processing. In Ada, there are two base components, the task which is a concurrent process and the protected type which is a data structure with guarded access to its data. Tasks may execute in parallel and different tasks proceed independently, except at points where they synchronize. A task is an object of a task type and all objects of a given task type have the same entries (interface), and share the same code (body). Task types are inherently limited types; assignment and equality operations are forbidden.

Task objects are created in the same ways as other objects. All tasks created by a given declarative part are activated in parallel. This means that they can logically start running in parallel with each other. The task that created these tasks waits until they have all finished elaborating their declarative parts; it then continues running in parallel with the tasks it created.

Programmer can terminate a task by using the keyword `terminate` which executes a nice orderly cleanup of the task. Every task has a master, which is the task, subprogram, block statement, or `accept` statement which contains the declaration of the task object. Before leaving the master of a task, the parent task waits for the child task. When the child task has been terminated, or is ready to terminate, the parent task proceeds.

Rendezvous are used for synchronizing tasks. A rendezvous is a meeting place where two tasks are to meet. If one task reaches it first then it waits for the other to arrive. A rendezvous can occur only if both tasks want it to happen.

4.1.4 In Modula-3

Modula-3 provides concurrent programming by standard library packages. The various systems that support threads offer quite similar facilities and we present the Digital's System Research Center (SRC) thread facilities[10]. A `Thread` is a handle on a thread. A `Mutex` is locked or unlocked by some thread. A `Condition` is a set of waiting threads.

A thread is created by calling `Fork` with a procedure and an argument record:

```
PROCEDURE Fork(proc:  Forkee; arg:  REFANY): Thread;.
```

It creates a new thread and starts that thread by executing the given procedure with the given arguments. When the procedure return, the thread terminates.

Condition variables are used for threads synchronization. A condition variable is always associated with a particular mutex, and with the data protected by that mutex. In general a monitor consists of some data, a mutex, and zero or more condition variables. The `Wait` operation atomically unlocks the mutex and blocks the thread (enqueued on the condition variable):

```
PROCEDURE Wait(m:  Mutex; c:  Condition);.
```

The `Signal` operation does nothing unless there is a thread blocked on the condition variable, in which case it awakens at least one such blocked thread (but possible more than one):

```
PROCEDURE Signal(c:  Condition);.
```

The `Broadcast` operation awakens all the threads currently blocked on the condition variable:

```
PROCEDURE Broadcast(c:  Condition);.
```

4.1.5 In ML

Objective Caml[46] provides the `threads` library[45] for concurrent programming. It is implemented by time-sharing on a single processor.

A new thread is created by the `create` function:

```
val create : ('a -> 'b) -> 'a -> t.
```

The first argument is applied to the second argument concurrently with the other threads of the program. The new thread terminates when the application returns, either normally or by raising an uncaught exception. When there is an uncaught exception during the new thread, the exception is printed on standard error, but not propagated back to the parent thread. The result of the application is discarded and not directly accessible to the parent thread. The `exit` function terminates the currently executing thread:

```
val exit : unit -> unit.
```

The `wait` and `signal` functions are used for synchronization between threads:

```
val wait: t -> Mutex.t -> unit
val signal: t -> unit
```

Condition variables should always be protected by a mutex.

4.2 Problems of Multithreaded Exception Handling in Java

Java [30] offers both exception handling facilities and multithreading mechanisms. Exceptions are first-class objects in Java. Like normal objects, they can be defined by

```java
private Thread blinker;

public void start() {
  blinker = new Thread(this);
  blinker.start();
}

public void stop() {
  blinker.stop();       // UNSAFE!
}

public void run() {
  Thread thisThread =
      Thread.currentThread();
  while (true) {
    try {
      thisThread.sleep(interval);
    } catch
      (InterruptedException e) {
    }
    repaint();
  }
}
```

```java
private volatile Thread blinker;

public void stop() {
  blinker = null;
}

public void run() {
  Thread thisThread =
      Thread.currentThread();
  while (blinker == thisThread) {
    try {
      thisThread.sleep(interval);
    } catch
      (InterruptedException e) {
    }
    repaint();
  }
}
```

Figure 4.1: Example[1] with/without `Thread.stop`

classes, instantiated, assigned to variables, passed as parameters, etc. Java provides a simple and tightly integrated support for multithreaded programs. A concurrent program consists of multiple threads that run independently at the same time.

However, Java does not provide a good method to use these two features together. The only one way for a thread to throw an asynchronous exception to another thread is invoking `Thread.stop()`. When the `stop` method of a thread c_1 is invoked by another thread c_2, the exception `ThreadDeath` is thrown to the thread c_1 and c_1 is stopped. But `Thread.stop()` is deprecated from JDK1.2 [1] because it is unsafe [2]:

[1] This example is from [2]

> "Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the `ThreadDeath` exception propagates up the stack.) ... Unlike other unchecked exceptions, `ThreadDeath` kills threads silently; thus, the user has no warning that his program may be corrupted."

Instead of throwing exceptions across threads, Java recommends a polling method. When a thread kills another thread, the former simply sets some flag to indicate that the latter should stop running. The latter thread should check this flag regularly and stop if the flag is set. Consider the example codes in Figure 4.1. The use of `blinker.stop` in the left example is replaced with checking whether the variable `blinker` is `null` in the right example.

4.3 Existing Solutions for Multithreaded Exception Handling in Java

Since Java does not provide a good method for multithreaded exception handling, several work were proposed to deal with the problem.

Lea [44] presented a method called *Completion Callbacks*. In order for a thread A to know whether the other thread B completed successfully or with an exception, A should implement a predefined interface, callback method, and B must call this interface to indicate success or failure. A callback method is just an ordinary method in the client that is designed to be called in some special circumstance by the server; here, in the case of completion or failure termination. To provide a handle for the callback, the client must send this as one of the arguments to the request setting up the service thread.

Similarly, Haggar [40] proposed a solution by using the listener paradigm. If a thread A wants to get an asynchronous signal from a thread B, A should register as a listener of B and notification is sent to objects that are registered as listeners of B by calling a predefined method. When an exception occurs in a thread B, notification could be sent to one or more other objects that are registered as listeners on the thread.

Chapter 5

Exception Analysis for Multithreaded Java Programs

This chapter presents a static analysis that estimates uncaught exceptions in multithreaded Java programs. In Java, throwing exceptions across threads is deprecated because of the safety problem. Instead of restricting programmers' freedom, we extend Java language to support multithreaded exception handling and propose a tool to detect uncaught exceptions in the input programs.

Our analysis consists of two steps. The analysis firstly estimates concurrently evaluated expressions of the multithreads in Java programs by the synchronization relation among the threads. Using this concurrency information, program's exception flow is derived as set-constraints, whose least model is our analysis result. Both of these two steps are proved safe.

Section 5.1 and 5.2 present the analysis problem and our approach. And Section 5.3 shows the syntax and semantics of source language. Section 5.4 and Section 5.5 present two analyses: concurrency analysis and exception analysis, respectively. Section 5.6 describes the analysis complexity and Section 5.7 shows related work.

5.1 Analysis Problem

Instead of throwing exceptions across threads, Java recommends a polling method. When a thread kills another thread, the former simply sets some flag to indicate that the latter should stop running. The latter thread should check this flag regularly and stop if the flag is set.

For example, let us assume that two threads A and B are concurrently evaluating. A thread A is reading a file and another thread B is doing other work while waiting for A to finish reading the file. In this case, the most natural way for A to notify B when it has finished reading the file is throwing exception to B. However, this natural solution is not possible in Java. In Java, A modifies some variable when it has finished reading the file and B checks this variable regularly.

5.2 Our Approach

We extend Java language to support throwing exceptions across threads and present a static analysis that estimates uncaught exceptions in multithreaded Java programs [60]. In contrast to Java's restriction on throwing asynchronous exceptions because of the safety problem, we give Java programmers the freedom of programming and provide them with a tool to check the safety of the programs.

Our analysis consists of two steps: analyzing Java programs' concurrency information and then exception flow. First, the analysis estimates expressions which may be evaluated concurrently in each thread. And then it analyzes the programs' exception flow with the pre-analyzed concurrency information.

```
class ThreadOne extends Thread {
    Object objectO;
    Thread main;
    public ThreadOne (Object objectO, Thread main) {
        this.objectO = objectO;
        this.main = main;
    }
    public void run() {
        try { synchronized (objectO) { objectO.wait(); }
        } catch (InterruptedException e) {}
        main.stop(new Exception());              // asynchronous exception throw
    }
}

class ThreadTwo extends Thread {
    Object objectO;
    Object objectP;
    public ThreadTwo (Object objectO, Object objectP) {
        this.objectO = objectO;
        this.objectP = objectP;
    }
    public void run() {
        try { this.sleep(1000);
        } catch (InterruptedException e) {}
        synchronized (objectP) { objectP.notify(); }
        synchronized (objectO) {
            objectO.notify();
            System.out.println("\nobjectO.notify by ThreadTwo");
        }
    }
}

class ThreadMain extends Thread {
    public static void main (String args[]) {
        Object objectO = new Object();
        Object objectP = new Object();

        Thread one = new ThreadOne(objectO, Thread.currentThread());
        one.start();
        Thread two = new ThreadTwo(objectO, objectP);
        two.start();
        try { synchronized (objectP) { objectP.wait(); }          // from HERE!!!
        } catch (Exception e) {}
        synchronized (objectO) {
            objectO.notify();
            System.out.println("\nobjectO.notify by ThreadMain");
        }
    }
}
```

Figure 5.1: A Multithreaded Java Program with An Asynchronous Exception Throw

Consider the example in Figure 5.1. When "`main.stop(new Exception())`" is called in the thread `ThreadOne`, the exception `Exception` is thrown to the thread `ThreadMain`. In order to analyze the expressions in `ThreadMain` to which the exception is thrown, our analysis estimates the expressions in `ThreadMain` which may be evaluated concurrently with `main.stop` in `ThreadOne`. Using the `wait` and `notify` relation among the threads `ThreadMain`, `ThreadOne` and `ThreadTwo`, our concurrency analysis estimates that the expressions evaluated after `objectP.wait` in `ThreadMain` may be evaluated concurrently with the `main.stop` in `ThreadOne`.

5.3 Language

For presentation brevity, we consider an imaginary core of Java with the extension of throwing exceptions across threads. Its abstract syntax is in Figure 5.2. A program is a sequence of class definitions. Class bodies consist of field variable declarations and method definitions. A method definition consists of the method name, its parameter, and its body expression.

Every expression's result is an object. Assignment expression returns the object of its right hand side expression. Sequence expression's result is the object of the last expression in the sequence and the result of a method call is the object from the method body. The `try` expression "`try` e_1 `catch` $(c\ x\ e_2)$" evaluates e_1 first. If the expression returns a normal object then this object is the result of the `try` expression. If an exception is raised from e_1 and its class is covered by c then the handler expression e_2 is evaluated with the exception object bound to x. If the raised exception is not covered by class c then the raised exception continues to propagate back along the evaluation chain until it meets another handler. Note that nested `try` expression can express multiple handlers for a single expression e_1

$$
\begin{array}{llll}
P & ::= & C^* & \text{program} \\
C & ::= & \texttt{class}\ c\ \texttt{ext}\ c\ \{\texttt{var}\ x^*\ M^*\} & \text{class definition} \\
M & ::= & m(x) = \mathrm{e}\ [\texttt{throws}\ c^*] & \text{method definition} \\
e & ::= & id & \text{variable} \\
 & | & id\ \texttt{:=}\ e & \text{assignment} \\
 & | & \texttt{new}\ c & \text{new object} \\
 & | & \texttt{this} & \text{self object} \\
 & | & e\ \texttt{;}\ e & \text{sequence} \\
 & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{branch} \\
 & | & \texttt{throw}\ e & \text{exception raise} \\
 & | & \texttt{try}\ e\ \texttt{catch}\ \texttt{(}c\ x\ e\texttt{)} & \text{exception handle} \\
 & | & e\texttt{.}m\texttt{(}e\texttt{)} & \text{method call} \\
 & | & e\texttt{.start} & \text{thread start} \\
 & | & e\texttt{.wait} & \text{object wait} \\
 & | & e\texttt{.notify} & \text{object notify} \\
 & | & \texttt{xthrow}\ e\ \texttt{to}\ e & \text{cross exception raise} \\
id & ::= & x & \text{method parameter} \\
 & | & id\texttt{.}x & \text{field variable} \\
c & & & \text{class name} \\
m & & & \text{method name} \\
x & & & \text{variable name}
\end{array}
$$

Figure 5.2: Abstract Syntax of an Extended Core of Java

: "`try (try` e_1 `catch (`$c_1\, x_1\, e_2$`)) catch (`$c_2\, x_2\, e_3$`)`." The exception object e is raised by `throw` e. The programmers have to declare in a method definition any exception class whose exceptions may escape from its body.

There are three language constructs for multithreading. A new thread e starts by "e`.start`" and the `run` method of the thread e is executed after the thread is started. For the sake of simplicity, we assume that `run` methods are used only for thread start. The `wait` expression "e`.wait`" causes a current thread to wait until the object e is notified by another thread. And `notify` expression "e`.notify`" wakes up some thread which is waiting for the object e to be notified. Note that this waiting and notification mechanism provides support for synchronizing the concurrent executions of threads.

The extended construct for throwing exceptions across threads is "$\texttt{xthrow}\ e_1\ \texttt{to}\ e_2$". It throws the exception object e_1 to the thread e_2. The exception is thrown to the expression evaluated in the thread e_2 at the same time.

We omit the formal semantics of our language. The semantics is not quite different from the existing work [16, 17] which present an event-based structural operational semantics for multi-threaded Java programs.

5.3.1 Synchronizing Expressions

Throughout this paper, we call `wait`, `notify` expressions and $hd(e_{c.\texttt{run}})$, $e_{c.\texttt{run}}$ for a thread c in the input program *synchronizing expressions* where $hd(e)$ is defined as follows:

$$
\begin{array}{rlcl}
hd(e) = & \text{case } e \text{ of} & & \\
 & id & \Rightarrow & e \\
 & id\ \texttt{:=}\ e_1 & \Rightarrow & hd(e_1) \\
 & \texttt{new}\ c & \Rightarrow & e \\
 & \texttt{this} & \Rightarrow & e \\
 & e_1\ \texttt{;}\ e_2 & \Rightarrow & hd(e_1) \\
 & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \Rightarrow & hd(e_1) \\
 & \texttt{throw}\ e_1 & \Rightarrow & hd(e_1) \\
 & \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} & \Rightarrow & hd(e_1) \\
 & e_1\texttt{.}m\texttt{(}e_2\texttt{)} & \Rightarrow & hd(e_1) \\
 & e_1\texttt{.start} & \Rightarrow & hd(e_1) \\
 & e_1\texttt{.wait} & \Rightarrow & hd(e_1) \\
 & e_1\texttt{.notify} & \Rightarrow & hd(e_1) \\
 & \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 & \Rightarrow & hd(e_1).
\end{array}
$$

The auxiliary function $hd(e)$ finds the firstly evaluated expression during the evaluation of e. Java threads synchronize with each other by `wait` and `notify` expressions. For a thread c in the input program, the first expression evaluated in c is $hd(e_{c.\mathtt{run}})$ and the last expression evaluated is $e_{c.\mathtt{run}}$.

5.4 Step One: Concurrency Analysis

In order to analyze uncaught exceptions in multithreaded Java programs, we need a concurrency analysis which estimates expressions evaluated concurrently. According to the semantics of `xthrow` expression, when "`xthrow` e_1 `to` e_2" is evaluated in a thread c_1, the exception e_1 is thrown to the thread e_2. Thus, in order to estimate the expressions in e_2 to which e_1 is thrown, our exception analysis needs to know the expressions in e_2 which are evaluated in parallel with "`xthrow` e_1 `to` e_2".

Section 5.4.1 presents our approach intuitively. We show the concurrency analysis rules and prove the soundness of the analysis in Section 5.4.2 and 5.4.3.

5.4.1 Our Approach

The examples a and b in Figure 5.3 illustrate how two different executions of the same program result in different synchronizations. The "o_1`.wait`" of Figure 5.3-(a) synchronizes with "o_1`.notify`" in the thread c_2, while the "o_1`.wait`" of Figure 5.3-(b) synchronizes with "o_1`.notify`" in the thread c_3. For the former case, since the "o_1`.wait`" is evaluated before e and simultaneously with the "o_1`.notify`" in c_2, all the expressions in c_2 which may be evaluated concurrently with e are included in "the expressions which are evaluated later than "o_1`.notify`" in c_2" ($*_1$). And for the latter case, the "o_1`.wait`" is evaluated before e and simultaneously with the "o_1`.notify`" in c_3, and the "o_2`.notify`" is evaluated before "o_1`.notify`" and simultaneously with the "o_2`.wait`" in c_2. Thus, all the expressions in c_2 which

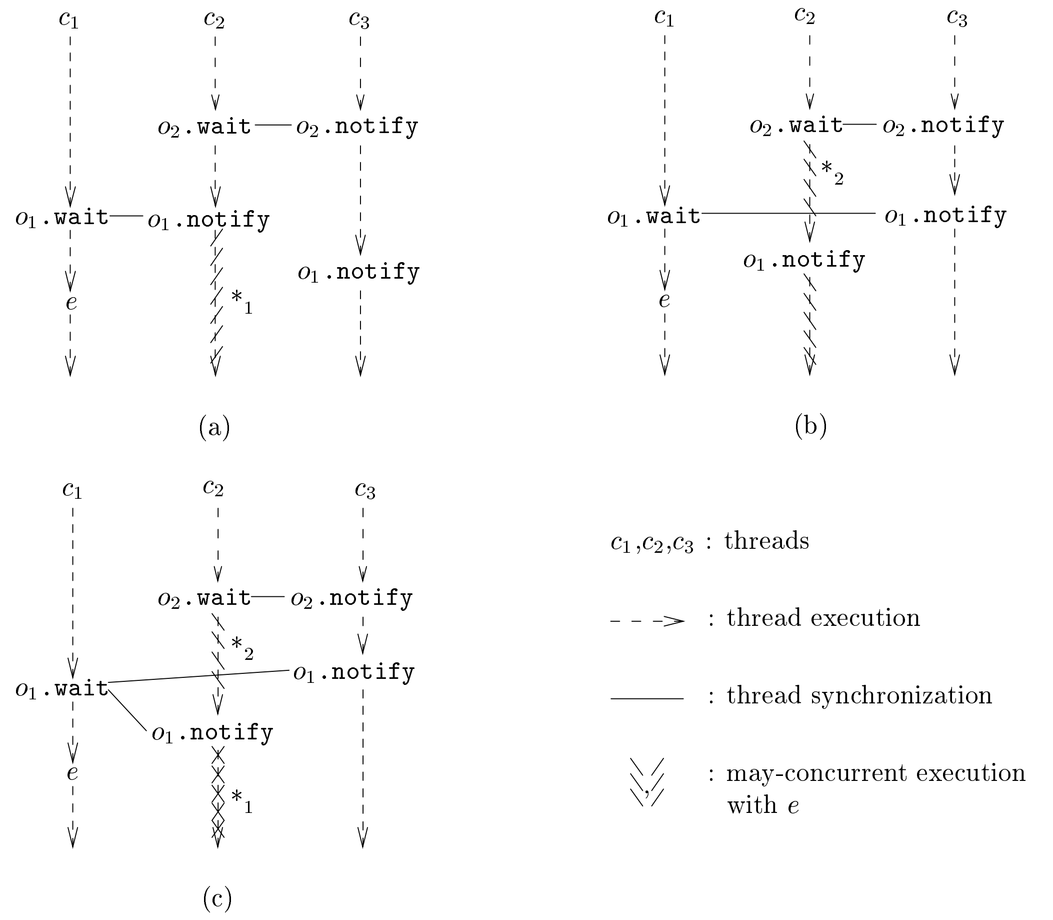

Figure 5.3: Non-Deterministic Execution Example And Our Approach

may be evaluated concurrently with e are included in "the expressions which are evaluated later than "o_2`.wait`" in c_2" ($*_2$).

In order to safely estimate the concurrent executions with e in c_2, we consider all the possible synchronizations among threads and, for each synchronization, we collect all the expressions which may be evaluated concurrently with e in c_2. As Figure 5.3-(c) represents, for each possible synchronization of "o_1`.wait`" with "o_1`.notify`" in c_2 and c_3, our approach collects all the possible concurrently executed expressions $*_1$ and $*_2$.

5.4.2 Time Analysis Rules

We first analyze timing relation among expressions in a given thread. In Section 5.4.2, $pre(e)$ ($suc(e)$) estimates expressions evaluated right before (after) e and $\hat{pre}(e)$ ($\hat{suc}(e)$) estimates the last (next) synchronizing expressions of e in Section 5.4.2.

Time Analysis among Expressions

In Figure 5.4, every expression e of the input program has two set constraints: $P_e \supseteq se$ and $S_e \supseteq se$. The set variable P_e is for the expressions which may be evaluated right before e and S_e is for the expressions which may be evaluated right after e. A set expression se is either an expression or a union of set expressions. A constraint $P_e \supseteq se$ ($S_e \supseteq se$) may be read as "expression e is evaluated right after (before) one of the expressions in se is evaluated".

Note that an interpretation $\mathcal{I}$ is defined as follows:

$$
\begin{array}{lcll}
e & \in & \mathit{Expr} & \text{expressions in program } \wp \\
\mathcal{I}(P_e) & \subseteq & \mathit{Expr} & \\
\mathcal{I}(S_e) & \subseteq & \mathit{Expr} & \\
\mathcal{I}(e) & = & \{e\} & \\
\mathcal{I}(se \cup se') & = & \mathcal{I}(se) \cup \mathcal{I}(se'). &
\end{array}
$$

Consider the rule for `throw` expression:

$$
\frac{\rhd_t e_1 : \mathcal{C}_1}{\rhd_t \texttt{throw}\ e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1} \ .
$$

$$[\text{Program}]_t \quad \frac{\rhd_t\ C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_t\ C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{ClassDef}]_t \quad \frac{\rhd_t\ M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_t\ \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{MethDef}]_t \quad \frac{\rhd_t\ e_m : \mathcal{C}}{\rhd_t\ m(x)\ \texttt{=}\ e_m : \mathcal{C}}$$

$$[\text{Variable}]_t \quad \rhd_t\ id : \emptyset$$

$$[\text{Assign}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1}{\rhd_t\ id\ \texttt{:=}\ e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{New}]_t \quad \rhd_t\ \texttt{new}\ c : \emptyset$$

$$[\text{This}]_t \quad \rhd_t\ \texttt{this} : \emptyset$$

$$[\text{Seq}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1 \quad \rhd_t\ e_2 : \mathcal{C}_2}{\rhd_t\ e_1\ \texttt{;}\ e_2 : \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_2, S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq e\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{Cond}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1 \quad \rhd_t\ e_2 : \mathcal{C}_2 \quad \rhd_t\ e_3 : \mathcal{C}_3}{\begin{array}{ll}\rhd_t\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : & \{P_{hd(e_2)} \supseteq e_1, P_{hd(e_3)} \supseteq e_1, P_e \supseteq e_2 \cup e_3, \\ & \quad S_{e_1} \supseteq hd(e_2) \cup hd(e_3), S_{e_2} \supseteq e, S_{e_3} \supseteq e\} \\ & \cup\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3\end{array}}$$

$$[\text{Throw}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1}{\rhd_t\ \texttt{throw}\ e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Try}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1 \quad \rhd_t\ e_2 : \mathcal{C}_2}{\begin{array}{ll}\rhd_t\ \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : & \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_1 \cup e_2, \\ & \quad S_{e_1} \supseteq hd(e_2) \cup e, S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

$$[\text{MethCall}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1 \quad \rhd_t\ e_2 : \mathcal{C}_2}{\begin{array}{ll}\rhd_t\ e_1\texttt{.}m\texttt{(}e_2\texttt{)} : & \{P_{hd(e_2)} \supseteq e_1, P_{hd(e_m)} \supseteq e_2, P_e \supseteq e_m, \\ & \quad S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq hd(e_m), S_{e_m} \supseteq e\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

$$[\text{ThdStart}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1}{\rhd_t\ e_1\texttt{.start} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Wait}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1}{\rhd_t\ e_1\texttt{.wait} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Notify}]_t \quad \frac{\rhd_t\ e_1 : \mathcal{C}_1}{\rhd_t\ e_1\texttt{.notify} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Xthrow}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1 \quad \rhd_t\ e_2 : \mathcal{C}_2}{\begin{array}{ll}\rhd_t\ \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : & \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_2, \\ & \quad S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

Figure 5.4: Time Analysis among Expressions

During the evaluation of e, "`throw` e_1", e_1 is firstly evaluated and then e is evaluated. So e is evaluated right after e_1 ($P_e \supseteq e_1$) and e_1 is evaluated right before e ($S_{e_1} \supseteq e$).

Consider the rule for `try` expression:

$$\frac{\rhd_t\, e_1 : \mathcal{C}_1 \quad \rhd_t\, e_2 : \mathcal{C}_2}{\begin{array}{ll}\rhd_t\, \texttt{try}\ e_1\ \texttt{catch}\ (c_1\ x_1\ e_2) : & \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_1 \cup e_2, \\ & S_{e_1} \supseteq hd(e_2) \cup e, S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}.$$

There are two possible evaluations of e, "`try` e_1 `catch` $(c_1\ x_1\ e_2)$": 1. after e_1 is evaluated, e is evaluated when e_1's value is a normal object or e_1's uncaught exceptions are not covered by c_1 2. after e_1 is evaluated, e_2 and then e are evaluated when e_1's uncaught exceptions are covered by c_1. For the former case, e is evaluated right after e_1 ($P_e \supseteq e_1$, $S_{e_1} \supseteq e$). For the latter case, $hd(e_2)$ is evaluated right after e_1 ($P_{hd(e_2)} \supseteq e_1$, $S_{e_1} \supseteq hd(e_2)$) and e is evaluated right after e_2 ($P_e \supseteq e_2$, $S_{e_2} \supseteq e$).

Definition 4 *For a program (a closed expression) $\wp$ and an evaluation of the program ε, $Time_\varepsilon(e)$ denotes the time when e is evaluated in ε. We write $Time(e)$ when it is clear from the context which evaluation the expression e belongs to.*

Our time analysis among expressions safely approximates timing relation among the expressions in a given thread:

Lemma 1 *For a program $\wp$, let $\rhd_t \wp : \mathcal{C}$ and let $lm(\mathcal{C})$ be the least model of $\mathcal{C}$. If $pre(e) = lm(\mathcal{C})(P_e)$ and $suc(e) = lm(\mathcal{C})(S_e)$, then the following hold for any evaluation of the program and for any expressions e' and e evaluated in a thread c:*

1. *If $Time(e') \leq Time(e)$ then $e' \in pre^*(e)$.*
2. *If $Time(e') \geq Time(e)$ then $e' \in suc^*(e)$.*

The proof uses Lemma 2, Lemma 3, and the auxiliary function $ExSq(e)$ which represents a set of all the possible execution sequences of e as follows:

$$
\begin{array}{lll}
\mathit{ExSq}(e) = & \text{case } e \text{ of} & \\
& \mathit{id} & \Rightarrow \{e\} \\
& \mathit{id} \texttt{ := } e_1 & \Rightarrow \mathit{ExSq}(e_1) \oplus \{e\} \\
& \texttt{new}\, c & \Rightarrow \{e\} \\
& \texttt{this} & \Rightarrow \{e\} \\
& e_1 \texttt{ ; } e_2 & \Rightarrow \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \{e\} \\
& \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \Rightarrow \mathit{ExSq}(e_1) \oplus (\mathit{ExSq}(e_2) \cup \mathit{ExSq}(e_3)) \oplus \{e\} \\
& \texttt{throw } e_1 & \Rightarrow \mathit{ExSq}(e_1) \oplus \{e\} \\
& \texttt{try } e_1 \texttt{ catch (} c_1\ x_1\ e_2 \texttt{)} & \Rightarrow (\mathit{ExSq}(e_1) \cup (\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2))) \oplus \{e\} \\
& e_1 \texttt{.} m \texttt{(} e_2 \texttt{)} & \Rightarrow \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \mathit{ExSq}(e_m) \oplus \{e\} \\
& & \quad \text{where } c \in \mathit{Class}(e_1),\ m\texttt{(}x\texttt{)} \texttt{ = } e_m \in c \\
& e_1\texttt{.start} & \Rightarrow \mathit{ExSq}(e_1) \oplus \{e\} \\
& e_1\texttt{.wait} & \Rightarrow \mathit{ExSq}(e_1) \oplus \{e\} \\
& e_1\texttt{.notify} & \Rightarrow \mathit{ExSq}(e_1) \oplus \{e\} \\
& \texttt{xthrow } e_1 \texttt{ to } e_2 & \Rightarrow \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \{e\}
\end{array}
$$

$$\{e_1e_2,\ e_3e_4\} \oplus \{e_5e_6,\ e_7e_8\} = \{e_1e_2e_5e_6,\ e_3e_4e_5e_6,\ e_1e_2e_7e_8,\ e_3e_4e_7e_8\}$$

Note that for any evaluation of e and its execution sequence $e_1 \cdots e_n \in \mathit{ExSq}(e)$, if $\mathit{Time}(e_i) \leq \mathit{Time}(e_j)$ $(1 \leq i, j \leq n)$ then $1 \leq i \leq j \leq n$.

Lemma 2 *For a program $\wp$, the following hold for a thread c in $\wp$ and any evaluation $e_1 \cdots e_n \in \mathit{ExSq}(e_{c}\texttt{.run})$ of the thread:*

1. $e_i \in \mathit{pre}(e_{i+1}) \quad (1 \leq i < n)$
2. $e_{i+1} \in \mathit{suc}(e_i) \quad (1 \leq i < n)$

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. The execution sequence "$e_i e_{i+1}$" is generated by either $ExSq(e') \oplus \{e_{i+1}\}$ ($e \cdots e_i \in ExSq(e')$) or $ExSq(e') \oplus ExSq(e'')$ ($e'_1 \cdots e_i \in ExSq(e')$, $e_{i+1} \cdots e'_2 \in ExSq(e'')$). Note that for any $e_1 \cdots e_n \in ExSq(e)$, $e_1 = hd(e)$ and $e_n = e$. Let $first(ExSq(e)) = hd(e)$, $first(\{e\}) = e$, and $last(ExSq(e)) = e$.

We prove by the case analysis of the execution sequence $e_i e_{i+1}$. For each case of $S_1 \oplus S_2$, we need to show that if $last(S_1) = e_i$ and $first(S_2) = e_{i+1}$, then $e_i \in lm(\mathcal{C})(P_{e_{i+1}})$. And it is sufficient to show that, for each $S_1 \oplus S_2$, if $last(S_1) = e_i$ and $first(S_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$.

– `Case` $ExSq(e_1) \oplus \{e\}$ where $e = id$ `:=` e_1

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by [Assign]$_t$.

– `Case` $ExSq(e_1) \oplus ExSq(e_2)$ where $e = e_1$ `;` e_2

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(ExSq(e_2)) = hd(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by [Seq]$_t$.

– `Case` $ExSq(e_2) \oplus \{e\}$ where $e = e_1$ `;` e_2

If $last(ExSq(e_2)) = e_2 = e_i$ and $first(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by [Seq]$_t$.

– `Case` $ExSq(e_1) \oplus ExSq(e_2)$ where $e =$ `if` e_1 `then` e_2 `else` e_3

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(ExSq(e_2)) = hd(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by [Cond]$_t$.

– `Case` $ExSq(e_1) \oplus ExSq(e_3)$ where $e =$ `if` e_1 `then` e_2 `else` e_3

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(ExSq(e_3)) = hd(e_3) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by [Cond]$_t$.

– `Case` $ExSq(e_2) \oplus \{e\}$ where $e =$ `if` e_1 `then` e_2 `else` e_3

If $\mathit{last}(\mathit{ExSq}(e_2)) = e_2 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Cond}]_t$.

– Case $\mathit{ExSq}(e_3) \oplus \{e\}$ where $e = \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$

If $\mathit{last}(\mathit{ExSq}(e_3)) = e_3 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Cond}]_t$.

– Case $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e = \texttt{throw } e_1$

If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Throw}]_t$.

– Case $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e = \texttt{try } e_1 \texttt{ catch (} c_1\ x_1\ e_2\texttt{)}$

If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Try}]_t$.

– Case $\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2)$ where $e = \texttt{try } e_1 \texttt{ catch (} c_1\ x_1\ e_2\texttt{)}$

If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_2)) = \mathit{hd}(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Try}]_t$.

– Case $\mathit{ExSq}(e_2) \oplus \{e\}$ where $e = \texttt{try } e_1 \texttt{ catch (} c_1\ x_1\ e_2\texttt{)}$

If $\mathit{last}(\mathit{ExSq}(e_2)) = e_2 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Try}]_t$.

– Case $\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2)$ where $e = e_1\texttt{.}m\texttt{(}e_2\texttt{)}$

If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_2)) = \mathit{hd}(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{MethCall}]_t$.

– Case $\mathit{ExSq}(e_2) \oplus \mathit{ExSq}(e_m)$ where $e = e_1\texttt{.}m\texttt{(}e_2\texttt{)}$

If $\mathit{last}(\mathit{ExSq}(e_2)) = e_2 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_m)) = \mathit{hd}(e_m) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{MethCall}]_t$.

– Case $\mathit{ExSq}(e_m) \oplus \{e\}$ where $e = e_1\texttt{.}m\texttt{(}e_2\texttt{)}$

If $\mathit{last}(\mathit{ExSq}(e_m)) = e_m = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{MethCall}]_t$.

– Case $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e = e_1\texttt{.start}$

If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{ThdStart}]_t$.

– `Case` $ExSq(e_1) \oplus \{e\}$ where $e = e_1$`.wait`

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Wait}]_t$.

– `Case` $ExSq(e_1) \oplus \{e\}$ where $e = e_1$`.notify`

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Notify}]_t$.

– `Case` $ExSq(e_1) \oplus ExSq(e_2)$ where $e =$ `xthrow` e_1 `to` e_2

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(ExSq(e_2)) = hd(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Xthrow}]_t$.

– `Case` $ExSq(e_2) \oplus \{e\}$ where $e =$ `xthrow` e_1 `to` e_2

If $last(ExSq(e_2)) = e_2 = e_i$ and $first(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Xthrow}]_t$. □

Lemma 3 *For a program* $\wp$*, the following hold for any evaluation of the program:*

1. If $e_1 \in pre(e_2)$ *then* $pre^*(e_1) \subseteq pre^*(e_2)$*.*

2. If $e_1 \in suc(e_2)$ *then* $suc^*(e_1) \subseteq suc^*(e_2)$*.*

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. First, we prove that if $e_1 \in pre(e_2)$ then $pre^n(e_1) \subseteq pre^{n+1}(e_2)$ for any n. We prove by induction on n.

– `Case` $n = 1:$ $\quad pre^2(e_2) = \bigcup_{e \in pre(e_2)} pre(e) \supseteq pre(e_1)$.

– `Case` $n = k:$
$$\begin{aligned} pre^{k+1}(e_2) &= \bigcup_{e \in pre^k(e_2)} pre(e) \\ &\supseteq \bigcup_{e \in pre^{k-1}(e_1)} pre(e) && \text{by I.H.} \\ &= pre^k(e_1). \end{aligned}$$

Since the set of expressions in the input program is finite, if $e_1 \in pre(e_2)$ then $pre^*(e_1) \subseteq pre^*(e_2)$. □

Proof of Lemma 1. Since the second property is a dual of the first one, we only prove the first property here.

1. We need to show that

$$\forall c \in \wp.\ e_1 \cdots e_n \in ExSq(e_{c}.\texttt{run}).\ 1 \leq i \leq j \leq n \Rightarrow e_i \in pre^*(e_j).$$

We prove by induction on $j - i$.

–`Case` $j - i = 0$: $e_i \in pre^*(e_i) = pre^*(e_j)$.

–`Case` $j - i = m$: We need to show that $e_i \in pre^*(e_{i+m})$. By I.H., $e_i \in pre^*(e_{i+m-1})$ and $e_{i+m-1} \in pre(e_{i+m})$ by Lemma 2. Then by Lemma 3, $pre^*(e_{i+m-1}) \subseteq pre^*(e_{i+m})$. Thus, $e_i \in pre^*(e_{i+m-1}) \subseteq pre^*(e_{i+m})$. $\square$

Time Analysis among Synchronizing Expressions

We present another timing analysis which estimates the last and next synchronizing expressions of an expression in a given thread. The auxiliary functions $p(e)$ and $s(e)$ are defined as follows:

$$p(e) = \begin{cases} \{e\} & \text{if } e \text{ is a synchronizing expression} \\ \hat{P}_e & \text{otherwise} \end{cases}$$

$$s(e) = \begin{cases} \{e\} & \text{if } e \text{ is a synchronizing expression} \\ \hat{S}_e & \text{otherwise.} \end{cases}$$

In Figure 5.5, every expression e of the input program has two set constraints: $\hat{P}_e \supseteq se$ and $\hat{S}_e \supseteq se$. The set variable $\hat{P}_e$ is for the synchronizing expressions which are to be evaluated before e and $\hat{S}_e$ is for the synchronizing expressions which are to be evaluated after e. A set expression se is an expression, a set variable, or a union

[Program]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_{\hat{t}}\ C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[ClassDef]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_{\hat{t}}\ \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[MethDef1]$_{\hat{t}}$
$$\frac{e_m : \mathcal{C} \quad m \neq \texttt{run}}{\rhd_{\hat{t}}\ m(x)\ \texttt{=}\ e_m : \mathcal{C}}$$

[MethDef2]$_{\hat{t}}$
$$\frac{e_m : \mathcal{C} \qquad m = \texttt{run}}{\rhd_{\hat{t}}\ m(x)\ \texttt{=}\ e_m : \{\hat{P}_{hd(e_m)} \supseteq hd(e_m), \hat{S}_{e_m} \supseteq e_m\} \cup \mathcal{C}}$$

[Variable]$_{\hat{t}}$
$$\rhd_{\hat{t}}\ id : \emptyset$$

[Assign]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ id\ \texttt{:=}\ e_1 : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}$$

[New]$_{\hat{t}}$
$$\rhd_{\hat{t}}\ \texttt{new}\ c : \emptyset$$

[This]$_{\hat{t}}$
$$\rhd_{\hat{t}}\ \texttt{this} : \emptyset$$

[Seq]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\rhd_{\hat{t}}\ e_1\ \texttt{;}\ e_2 : \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_2), \hat{S}_{e_1} \supseteq s(hd(e_2)), \hat{S}_{e_2} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2 \quad \rhd_{\hat{t}}\ e_3 : \mathcal{C}_3}{\begin{array}{ll}\rhd_{\hat{t}}\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : & \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_{hd(e_3)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_1), \\ & \hat{S}_{e_1} \supseteq \hat{S}_e, \hat{S}_{e_2} \supseteq \hat{S}_e, \hat{S}_{e_3} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3\end{array}}$$

[Throw]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ \texttt{throw}\ e_1 : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}$$

[Try]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\begin{array}{ll}\rhd_{\hat{t}}\texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : & \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_1), \\ & \hat{S}_{e_1} \supseteq \hat{S}_e, \hat{S}_{e_2} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

[MethCall]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\begin{array}{ll}\rhd_{\hat{t}}\ e_1\texttt{.}m\texttt{(}e_2\texttt{)} : & \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_{hd(e_m)} \supseteq p(e_2), \hat{P}_e \supseteq p(e_m), \\ & \hat{S}_{e_1} \supseteq s(hd(e_2)), \hat{S}_{e_2} \supseteq s(hd(e_m)), \hat{S}_{e_m} \supseteq \hat{S}_e\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

[ThdStart]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ e_1\texttt{.start} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}$$

[Wait]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ e_1\texttt{.wait} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

[Notify]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ e_1\texttt{.notify} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

[Xthrow]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}} e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\begin{array}{ll}\rhd_{\hat{t}}\ \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : & \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_2), \\ & \hat{S}_{e_1} \supseteq s(hd(e_2)), \hat{S}_{e_2} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

Figure 5.5: Time Analysis among Synchronizing Expressions

of set expressions. A constraint $\hat{P}_e \supseteq se$ ($\hat{S}_e \supseteq se$) may be read as "synchronizing expressions se are evaluated before (after) the expression e is evaluated".

Consider the `throw` expression:

$$\frac{\rhd_{\hat{t}}\, e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\, \texttt{throw}\ e_1 : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}\ .$$

If e_1 is either `wait` or `notify` expression, e_1 is the last synchronizing expression of e ($\hat{P}_e \supseteq e_1$). Otherwise, the last synchronizing expressions of e includes those of e_1 ($\hat{P}_e \supseteq \hat{P}_{e_1}$). Since e is not a synchronizing expression, the next synchronizing expressions of e_1 includes those of e ($\hat{S}_{e_1} \supseteq \hat{S}_e$).

Consider the `wait` and `notify` expressions:

$$\begin{array}{c} \dfrac{\rhd_{\hat{t}}\, e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\, e_1\texttt{.wait} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1} \\ \dfrac{\rhd_{\hat{t}}\, e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\, e_1\texttt{.notify} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1} \end{array}\ .$$

The last synchronizing expressions of e is either e_1 ($\hat{P}_e \supseteq e_1$) if e_1 is `wait` or `notify` expression, or those of e_1 ($\hat{P}_e \supseteq \hat{P}_{e_1}$) otherwise. Since e is a synchronizing expression, the next synchronizing expressions of e_1 includes e ($\hat{S}_{e_1} \supseteq e$).

There are two rules for method definitions:

$$\begin{array}{c} \dfrac{e_m : \mathcal{C} \quad m \neq \texttt{run}}{\rhd_{\hat{t}}\, m(x)\ \texttt{=}\ e_m : \mathcal{C}} \\ \dfrac{e_m : \mathcal{C} \quad m = \texttt{run}}{\rhd_{\hat{t}}\, m(x)\ \texttt{=}\ e_m : \{\hat{P}_{hd(e_m)} \supseteq hd(e_m), \hat{S}_{e_m} \supseteq e_m\} \cup \mathcal{C}} \end{array}\ .$$

For normal methods, the analysis does not generate any constraint. In case of `run` methods, $\hat{P}_{hd(e_m)} \supseteq hd(e_m)$ and $\hat{S}_{e_m} \supseteq e_m$ are generated since $hd(e_{\texttt{run}})$ and $e_{\texttt{run}}$ are synchronizing expressions.

Our second time analysis lower approximates timing relation among synchronizing expressions in a given thread:

Lemma 4 *For a program $\wp$, let $\triangleright_{\hat{t}}\wp : \mathcal{C}$ and let $lm(\mathcal{C})$ be the least model of $\mathcal{C}$. If $\hat{pre}(e) = lm(\mathcal{C})(\hat{P}_e)$ and $\hat{suc}(e) = lm(\mathcal{C})(\hat{S}_e)$, then the following hold for any evaluation of the program and for any expressions e evaluated in a thread c:*

1. $\forall e' \in \hat{pre}(e).\ Time(e') \leq Time(e)$.
2. $\forall e' \in \hat{suc}(e).\ Time(e') \geq Time(e)$.

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. The least model $lm(\mathcal{C})$ is equivalent to the $\subseteq$-least fixpoint fix$\mathcal{F}$ of the following continuous function $\mathcal{F}$ derived from $\mathcal{C}$ as follows[22]:

$$\mathcal{F}(\rho)(\hat{P}_e) = \begin{cases} \{e_1\} & e \in \{id, \texttt{new}\, c, \texttt{this}\},\ e \neq hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp, \\ & e = hd(e_2) \text{ and } e_1 \text{ is a synch. exp.} \\ & \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ & \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \\ \rho(\hat{P}_{e_1}) & e \in \{id, \texttt{new}\, c, \texttt{this}\},\ e \neq hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp, \\ & e = hd(e_2) \text{ and } e_1 \text{ is not a synch. exp.} \\ & \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ & \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \\ \{e_1, e''\} & e \in \{id, \texttt{new}\, c, \texttt{this}\}, \\ & e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is a synch. exp.,} \\ & e = hd(e_2) \text{ and } e_1 \text{ is a synch. exp.} \\ & \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ & \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \end{cases}$$

$$
\mathcal{F}(\rho)(\hat{P}_e) = \begin{cases}
\{e_1\} \cup \rho(\hat{P}_{e''}) & e \in \{id, \texttt{new}\,c, \texttt{this}\}, \\
 & e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is not a synch. exp.}, \\
 & e = hd(e_2) \text{ and } e_1 \text{ is a synch. exp.} \\
 & \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\
 & \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \\
\rho(\hat{P}_{e_1}) \cup \{e''\} & e \in \{id, \texttt{new}\,c, \texttt{this}\}, \\
 & e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is a synch. exp.}, \\
 & e = hd(e_2) \text{ and } e_1 \text{ is not a synch. exp.} \\
 & \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\
 & \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \\
\rho(\hat{P}_{e_1}) \cup \rho(\hat{P}_{e''}) & e \in \{id, \texttt{new}\,c, \texttt{this}\}, \\
 & e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is not a synch. exp.}, \\
 & e = hd(e_2) \text{ and } e_1 \text{ is not a synch. exp.} \\
 & \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\
 & \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \\
\{e_1\} & e \in \{id\ \texttt{:=}\ e_1, \texttt{throw}\ e_1, e_1\texttt{.start}, e_1\texttt{.wait}, e_1\texttt{.notify}\} \\
 & \text{and } e_1 \text{ is a synch. exp.} \\
\rho(\hat{P}_{e_1}) & e \in \{id\ \texttt{:=}\ e_1, \texttt{throw}\ e_1, e_1\texttt{.start}, e_1\texttt{.wait}, e_1\texttt{.notify}\} \\
 & \text{and } e_1 \text{ is not a synch. exp.} \\
\{e_2\} & e \in \{e_1\ \texttt{;}\ e_2, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2\} \text{ and } e_2 \text{ is a synch. exp.} \\
\rho(\hat{P}_{e_2}) & e \in \{e_1\ \texttt{;}\ e_2, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2\} \text{ and } e_2 \text{ is not a synch. exp.} \\
\{e_1\} & e \in \{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)}\} \\
 & \text{and } e_1 \text{ is a synch. exp.} \\
\rho(\hat{P}_{e_1}) & e \in \{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)}\} \\
 & \text{and } e_1 \text{ is not a synch. exp.} \\
\{e_m\} & e = e_1\texttt{.}m\texttt{(}e_2\texttt{)} \text{ and } e_m \text{ is a synch. exp.} \\
\rho(\hat{P}_{e_m}) & e = e_1\texttt{.}m\texttt{(}e_2\texttt{)} \text{ and } e_m \text{ is not a synch. exp.}
\end{cases}
$$

We prove $Q(\mathrm{fix}\mathcal{F})$ by the fixpoint induction, where the assertion $Q(\rho)$ for a program $\wp$ is:

$$\forall e \in \wp.\ \forall e' \in \rho(\hat{P}_e).\ \mathit{Time}(e') \leq \mathit{Time}(e) \quad \text{for any evaluation of } \wp.$$

Base case $Q(\emptyset)$ trivially holds. We prove that $Q(\mathcal{F}(\rho))$ holds given the induction hypothesis $Q(\rho)$. For brevity, we call synchronizing expressions *synch. exp.*

– **Case** $e \in \{id, \texttt{new}\, c, \texttt{this}\}$, $e \neq hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$,

$e = hd(e_2)$ and e_1 is a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $\mathit{ExSq}(e_1\texttt{;}e_2) = \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \{e\}$ and $\mathit{last}(\mathit{ExSq}(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $\mathit{Time}(e_1) \leq \mathit{Time}(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\}$, $\forall e' \in \mathcal{F}(\rho)(\hat{P}_e).\mathit{Time}(e') \leq \mathit{Time}(e)$ holds.

– **Case** $e \in \{id, \texttt{new}\, c, \texttt{this}\}$, $e \neq hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$,

$e = hd(e_2)$ and e_1 is not a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $\mathit{ExSq}(e_1\texttt{;}e_2) = \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \{e\}$ and $\mathit{last}(\mathit{ExSq}(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $\mathit{Time}(e_1) \leq \mathit{Time}(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_1})$. $\mathit{Time}(e') \leq \mathit{Time}(e_1)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1})$, the case holds.

– **Case** $e \in \{id, \texttt{new}\, c, \texttt{this}\}$,

$e = hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$ and e'' is a synch. exp.,

$e = hd(e_2)$ and e_1 is a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $\mathit{ExSq}(e_1\texttt{;}e_2) = \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \{e\}$ and $\mathit{last}(\mathit{ExSq}(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $\mathit{Time}(e_1) \leq \mathit{Time}(e)$. And because $\mathit{ExSq}(e'\texttt{.}m\texttt{(}e''\texttt{)}) = \mathit{ExSq}(e') \oplus \mathit{ExSq}(e'') \oplus \mathit{ExSq}(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $\mathit{Time}(e'') \leq \mathit{Time}(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1, e''\}$, the case holds.

– `Case` $e \in \{id, \mathtt{new}\, c, \mathtt{this}\}$,

$e = hd(e_m)$ where $e'\mathtt{.}m\mathtt{(}e''\mathtt{)} \in \wp$ and e'' is not a synch. exp.,

$e = hd(e_2)$ and e_1 is a synch. exp. where $e_1\mathtt{;}e_2 \in \wp$

Since $ExSq(e_1\mathtt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. And because $ExSq(e'\mathtt{.}m\mathtt{(}e''\mathtt{)}) = ExSq(e') \oplus ExSq(e'') \oplus ExSq(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $Time(e'') \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e''})$. $Time(e') \leq Time(e'')$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\} \cup \rho(\hat{P}_{e''})$, the case holds.

– `Case` $e \in \{id, \mathtt{new}\, c, \mathtt{this}\}$,

$e = hd(e_m)$ where $e'\mathtt{.}m\mathtt{(}e''\mathtt{)} \in \wp$ and e'' is a synch. exp.,

$e = hd(e_2)$ and e_1 is not a synch. exp. where $e_1\mathtt{;}e_2 \in \wp$

Since $ExSq(e_1\mathtt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. And because $ExSq(e'\mathtt{.}m\mathtt{(}e''\mathtt{)}) = ExSq(e') \oplus ExSq(e'') \oplus ExSq(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $Time(e'') \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_1})$. $Time(e') \leq Time(e_1)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1}) \cup \{e''\}$, the case holds.

– `Case` $e \in \{id, \mathtt{new}\, c, \mathtt{this}\}$,

$e = hd(e_m)$ where $e'\mathtt{.}m\mathtt{(}e''\mathtt{)} \in \wp$ and e'' is not a synch. exp.,

$e = hd(e_2)$ and e_1 is not a synch. exp. where $e_1\mathtt{;}e_2 \in \wp$

Since $ExSq(e_1\mathtt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. And because $ExSq(e'\mathtt{.}m\mathtt{(}e''\mathtt{)}) = ExSq(e') \oplus ExSq(e'') \oplus ExSq(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $Time(e'') \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_1})$. $Time(e') \leq Time(e_1)$ and $\forall e' \in \rho(\hat{P}_{e''})$. $Time(e') \leq Time(e'')$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1}) \cup \rho(\hat{P}_{e''})$, the case holds.

– Case $e \in \{id$ `:=` e_1, `throw` e_1, e_1`.start`, e_1`.wait`, e_1`.notify`$\}$
and e_1 is a synch. exp.

Since $ExSq(e) = ExSq(e_1) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\}$, the case holds.

– Case $e \in \{id$ `:=` e_1, `throw` e_1, e_1`.start`, e_1`.wait`, e_1`.notify`$\}$
and e_1 is not a synch. exp.

Since $ExSq(e) = ExSq(e_1) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. By I.H., $\forall e' \in \rho(\hat{P}_{e_1})$. $Time(e') \leq Time(e_1)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1})$, the case holds.

– Case $e \in \{e_1$ `;` e_2, `xthrow` e_1 `to` $e_2\}$ and e_2 is a synch. exp.

Since $ExSq(e) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_2)) = e_2$, e_2 is always evaluated earlier than e. Thus, $Time(e_2) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_2\}$, the case holds.

– Case $e \in \{e_1$ `;` e_2, `xthrow` e_1 `to` $e_2\}$ and e_2 is not a synch. exp.

Since $ExSq(e) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_2)) = e_2$, e_2 is always evaluated earlier than e. Thus, $Time(e_2) \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_2})$. $Time(e') \leq Time(e_2)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_2})$, the case holds.

– Case $e \in \{$`if` e_1 `then` e_2 `else` e_3, `try` e_1 `catch` `(`c_1 x_1 e_2`)`$\}$ and e_1 is a synch. exp.

Since $ExSq($`if` e_1 `then` e_2 `else` $e_3) = ExSq(e_1) \oplus (ExSq(e_2) \cup ExSq(e_3)) \oplus \{e\}$, $ExSq($`try` e_1 `catch` `(`c_1 x_1 e_2`)`$) = (ExSq(e_1) \cup (ExSq(e_1) \oplus ExSq(e_2))) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\}$, the case holds.

– `Case` $e \in \{$`if` e_1 `then` e_2 `else` e_3, `try` e_1 `catch` $(c_1\ x_1\ e_2)\}$

and e_1 is not a synch. exp.

Since $\mathit{ExSq}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) = \mathit{ExSq}(e_1) \oplus (\mathit{ExSq}(e_2) \cup \mathit{ExSq}(e_3)) \oplus \{e\}$, $\mathit{ExSq}(\texttt{try}\ e_1\ \texttt{catch}\ (c_1\ x_1\ e_2)) = (\mathit{ExSq}(e_1) \cup (\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2))) \oplus \{e\}$ and $\mathit{last}(\mathit{ExSq}(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $\mathit{Time}(e_1) \leq \mathit{Time}(e)$. By induction, $\forall e' \in \rho(\hat{P}_{e_1})$. $\mathit{Time}(e') \leq \mathit{Time}(e_1)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1})$, the case holds.

– `Case` $e = e_1 \texttt{.} m \texttt{(} e_2 \texttt{)}$ and e_m is a synch. exp.

Since $\mathit{ExSq}(e_1 \texttt{.} m \texttt{(} e_2 \texttt{)}) = \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \mathit{ExSq}(e_m) \oplus \{e\}$ and $\mathit{last}(\mathit{ExSq}(e_m)) = e_m$, e_m is always evaluated earlier than e. Thus, $\mathit{Time}(e_1) \leq \mathit{Time}(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_m\}$, the case holds.

– `Case` $e = e_1 \texttt{.} m \texttt{(} e_2 \texttt{)}$ and e_m is not a synch. exp.

Since $\mathit{ExSq}(e_1 \texttt{.} m \texttt{(} e_2 \texttt{)}) = \mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2) \oplus \mathit{ExSq}(e_m) \oplus \{e\}$ and $\mathit{last}(\mathit{ExSq}(e_m)) = e_m$, e_m is always evaluated earlier than e. Thus, $\mathit{Time}(e_1) \leq \mathit{Time}(e)$. By induction, $\forall e' \in \rho(\hat{P}_{e_m})$. $\mathit{Time}(e') \leq \mathit{Time}(e_m)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_m\}$, the case holds.

Other cases are similarly proved. $\square$

5.4.3 Concurrency Analysis Rules

Now, we present our concurrency analysis rules in this section. We assume that class information $\mathit{Class}(e)$ is already available for every expression e in our analysis. There are several existing work for class analysis[24, 56, 67]. The class analysis estimates for each expression e the classes (including exception classes) that the expression e's normal object belongs to. Note that exception classes are normal classes in Java.

When we say "an expression e in a thread c", it means that e may be evaluated in c. If e is the body of the `run` method of a thread c or e's enclosing method is the `run` method of c, e is evaluated only in c. Otherwise, e is evaluated in threads

$$[\delta_1]\quad \frac{e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c' \quad \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset}{e.\mathtt{wait} \xrightarrow{c\uparrow c'} e_1.\mathtt{notify} \quad e.\mathtt{wait} \xrightarrow{c\downarrow c'} e_1.\mathtt{notify}}$$

$$[\delta_2]\quad \frac{\begin{array}{c} e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \xrightarrow{c''\uparrow c'} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\mathtt{notify}) \end{array}}{e.\mathtt{wait} \xrightarrow{c\uparrow c'} e_3}$$

$$[\delta_3]\quad \frac{\begin{array}{c} e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \overset{c''\uparrow c'}{\not\longrightarrow} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\mathtt{notify}) \end{array}}{e.\mathtt{wait} \xrightarrow{c\uparrow c'} hd(e_{c'.\mathtt{run}})}$$

$$[\delta_4]\quad \frac{\begin{array}{c} e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \xrightarrow{c''\downarrow c'} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \hat{suc}(e_1.\mathtt{notify}) \end{array}}{e.\mathtt{wait} \xrightarrow{c\downarrow c'} e_3}$$

$$[\delta_5]\quad \frac{\begin{array}{c} e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \overset{c''\downarrow c'}{\not\longrightarrow} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \hat{suc}(e_1.\mathtt{notify}) \end{array}}{e.\mathtt{wait} \xrightarrow{c\downarrow c'} e_{c'.\mathtt{run}}}$$

Figure 5.6: Concurrency Analysis Rules

where e's enclosing method is called. This is formally defined as follows:

$$\mathit{owner}(e) = \begin{cases} \{c\} & \text{where } e = e_{c}.\mathtt{run} \text{ or } e \in e_{c}.\mathtt{run} \\ \bigcup \mathit{owner}(e_1 \,.\, m(e_2)) & \text{where } e = e_m \text{ or } e \in e_m, \\ & \qquad m \neq c.\mathtt{run}, \text{ and } e_1 \,.\, m(e_2) \in \wp. \end{cases}$$

We write $e \in c$ when $c \in \mathit{owner}(e)$.

Figure 5.6 presents our rules to estimate the concurrency information of a program $\wp$. An edge $e \xrightarrow{c\uparrow c'} e'$ ($e \xrightarrow{c\downarrow c'} e'$) denotes that an expression e' in a thread c' may be evaluated at the same time or before (after) the evaluation of e in c.

We show only a half of the rules; dual rules which swap e`.wait` and e`.notify` are omitted for presentation brevity. Rule $[\delta_1]$ is for the case when e in c synchronizes with e' in c' directly. In contrast, rules $[\delta_2]$ to $[\delta_5]$ are for the cases when e in c does not synchronize with e' in c' directly but gets timing relation by using other synchronizations.

Consider rule $[\delta_1]$:

$$\frac{e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c' \quad \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset}{e.\mathtt{wait} \xrightarrow{c \uparrow c'} e_1.\mathtt{notify} \quad e.\mathtt{wait} \xrightarrow{c \downarrow c'} e_1.\mathtt{notify}}\ .$$

Since e`.wait` in c may be notified by e_1`.notify` in c', these two expressions may be evaluated at the same time.

Consider rule $[\delta_2]$:

$$\frac{\begin{array}{c} e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \xrightarrow{c'' \uparrow c'} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\mathtt{notify}) \end{array}}{e.\mathtt{wait} \xrightarrow{c \uparrow c'} e_3}.$$

Since e`.wait` in c may be notified by e_1`.notify` in c'', these two expressions may be evaluated at the same time. Because e_2 is evaluated before e_1`.notify` and e_3 may be evaluated at the same time or before e_2, e_3 is evaluated before e_1`.notify`. Thus, e_3 is evaluated before e`.wait`.

Finally, consider rule $[\delta_3]$:

$$\frac{\begin{array}{c} e.\mathtt{wait} \in c \quad e_1.\mathtt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \overset{c'' \uparrow c'}{\not\longrightarrow} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\mathtt{notify}) \end{array}}{e.\mathtt{wait} \xrightarrow{c \uparrow c'} \mathit{hd}(e_{c'.\mathtt{run}})}.$$

As the same as in rules $[\delta_1]$ and $[\delta_2]$, $e.\mathtt{wait}$ in c and $e_1.\mathtt{notify}$ in c'' may be evaluated at the same time. Since the last synchronizing expressions of $e_1.\mathtt{notify}$ in c'' may not synchronize with the expressions in c', $e.\mathtt{wait}$ cannot get any timing relation here. Thus, $e.\mathtt{wait}$ safely makes an edge with $hd(e_{c'.\mathtt{run}})$ because $hd(e_{c'.\mathtt{run}})$ is the first expression evaluated in c'.

Rules in Figure 5.6 safely finds the expressions in c' which may be synchronized with the expressions in c before (after) the evaluation of e:

Lemma 5 *For a program $\wp$, let*

$$\delta_{\uparrow}(c, e, c') = \{e' \mid e \xrightarrow{c \uparrow c'} e' \textit{ is deducible by the rules in Figure 5.6}\}$$
$$\delta_{\downarrow}(c, e, c') = \{e' \mid e \xrightarrow{c \downarrow c'} e' \textit{ is deducible by the rules in Figure 5.6}\}.$$

Then the following hold for any evaluation of the program and an expression e in a thread c in $\wp$:

1. $\exists\ e' \in \delta_{\uparrow}(c, e, c')$ *such that* $\mathit{Time}(e') \leq \mathit{Time}(e)$ *or* $e' = hd(e_{c'.\mathtt{run}})$.
2. $\exists\ e' \in \delta_{\downarrow}(c, e, c')$ *such that* $\mathit{Time}(e') \geq \mathit{Time}(e)$ *or* $e' = e_{c'.\mathtt{run}}$.

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. We prove only the case of $\delta_{\uparrow}(c, e\mathtt{.wait}, c')$ and the dual case of $\delta_{\uparrow}(c, e\mathtt{.notify}, c')$ is similarly proved.

For any evaluation of $\wp$ and $e\mathtt{.wait} \in c$ in $\wp$, there exists $e_1\mathtt{.notify}$ such that $\mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset$, which notifies $e\mathtt{.wait} \in c$. Since $e_1\mathtt{.notify}$ notifies $e\mathtt{.wait}$, $\mathit{Time}(e_1\mathtt{.notify}) = \mathit{Time}(e\mathtt{.wait})$.

We prove for two cases of $e_1\mathtt{.notify}$: (1) $e_1\mathtt{.notify} \in c'$ and (2) $e_1\mathtt{.notify} \in c''$ $c' \neq c'' \in \wp$.

(1) $e_1\mathtt{.notify} \in c'$

By $[\delta_1]$, $e.\texttt{wait} \stackrel{c\uparrow c'}{\longrightarrow} e_1.\texttt{notify}$. Therefore, $e_1.\texttt{notify} \in \delta_\uparrow(c, e.\texttt{wait}, c')$ and $\mathit{Time}(e_1.\texttt{notify}) = \mathit{Time}(e.\texttt{wait})$.

(2) $e_1.\texttt{notify} \in c''$ $c' \neq c'' \in \wp$

We need to show that there exists $e' \in \delta_\uparrow(c, e.\texttt{wait}, c')$ s.t. $\mathit{Time}(e') \leq \mathit{Time}(e.\texttt{wait})$ or $e' = hd(e_{c'}.\texttt{run})$. And it is equivalent to show that $e.\texttt{wait} \stackrel{c\uparrow c'}{\longrightarrow} e'$ such that $\mathit{Time}(e') \leq \mathit{Time}(e.\texttt{wait})$ or $e' = hd(e_{c'}.\texttt{run})$ is deducible. We prove that, for any size of derivation tree, there exists $e.\texttt{wait} \stackrel{c\uparrow c'}{\longrightarrow} e'$ such that $\mathit{Time}(e') \leq \mathit{Time}(e.\texttt{wait})$ or $e' = hd(e_{c'}.\texttt{run})$.

– `Case` leaves: By $[\delta_3]$, $e = hd(e_{c'}.\texttt{run})$.

– `Case` internal nodes: By $[\delta_2]$, there exist $e_2 \in \hat{pre}(e_1.\texttt{notify})$ and $e_2 \stackrel{c''\uparrow c'}{\longrightarrow} e_3$. By Lemma 4, $\forall e_2 \in \hat{pre}(e_1.\texttt{notify})$. $\mathit{Time}(e_2) \leq \mathit{Time}(e_1.\texttt{notify})$. By induction, $\mathit{Time}(e_3) \leq \mathit{Time}(e_2)$ or $e_3 = hd(e_{c'}.\texttt{run})$. If $\mathit{Time}(e_3) \leq \mathit{Time}(e_2)$, then $\mathit{Time}(e_3) \leq \mathit{Time}(e_2) \leq \mathit{Time}(e_1.\texttt{notify}) \leq \mathit{Time}(e.\texttt{wait})$. □

Definition 5 *For some threads c, c' in a program $\wp$ and an expression e in c, the concurrency analysis is defined by:*

$$\begin{aligned} \mathit{Concur}(c, e, c') = \{e' \mid\ & e_1 \in \hat{pre}(e),\ e_2 \in \hat{suc}(e),\ e_1' \in \delta_\uparrow(c, e_1, c'),\ e_2' \in \delta_\downarrow(c, e_2, c'), \\ & e' \in suc^*(e_1') \cap pre^*(e_2')\} \end{aligned}.$$

Our concurrency analysis $\mathit{Concur}(c, e, c') \ni e'$ denotes that e' in the thread c' may be evaluated concurrently with e in c. As Definition 5 shows, the analysis is done by the following steps:

1. Since Java threads synchronize only by `wait` or `notify` expressions, the analysis first finds a pair of the last `wait` or `notify` expressions and the next `wait` or `notify` expressions of e: $e_1 \in \hat{pre}(e)$, $e_2 \in \hat{suc}(e)$.

2. And then the analysis finds expressions in c' which may be synchronized with the expressions in c before the evaluation of e – $e'_1 \in \delta_{\uparrow}(c, e_1, c')$ – and after the evaluation of e – $e'_2 \in \delta_{\downarrow}(c, e_2, c')$.

3. Finally, the analysis collects all the expressions during the pair of synchronized expressions: $e' \in suc^*(e'_1) \cap pre^*(e'_2)$.

Now, we prove the soundness of our concurrency analysis:

Theorem 1 *For a program $\wp$ and any evaluation of the program, the expression evaluated in a thread c' concurrently with an expression e in a thread c is included in $Concur(c, e, c')$.*

Proof. For a program $\wp$ and any evaluation of the program, let e' be evaluated in c' concurrently with e in c, then

$$Time(e) = Time(e'). \tag{5.1}$$

We need to show that e' is included in $Concur(c, e, c')$. By Lemma 4,

$$\forall e_1 \in \hat{pre}(e).Time(e_1) \leq Time(e) \tag{5.2}$$

and

$$\forall e_2 \in \hat{suc}(e).Time(e_2) \geq Time(e). \tag{5.3}$$

By Lemma 5, there exists $e'_1 \in \delta_{\uparrow}(c, e_1, c')$ such that

$$Time(e'_1) \leq Time(e_1) \tag{5.4}$$

or

$$e'_1 = hd(e_{c'.\mathtt{run}}) \tag{5.5}$$

and there exists $e'_2 \in \delta_{\downarrow}(c, e_2, c')$ such that

$$Time(e'_2) \geq Time(e_2) \tag{5.6}$$

or

$$e'_2 = e_{c'.\mathtt{run}}. \tag{5.7}$$

For the case of (5.4), we get

$$Time(e'_1) \leq Time(e')$$

from (5.1), (5.2) and (5.4). For the other case of (5.5), we also get

$$Time(e'_1) \leq Time(e')$$

since $hd(e_{c'.\mathtt{run}})$ is the first expression evaluated in c'. Similarly, for the case of (5.6), we get

$$Time(e'_2) \geq Time(e')$$

from (5.1), (5.3) and (5.6). And for the other case of (5.7), we also get

$$Time(e'_2) \geq Time(e')$$

since $e_{c'.\texttt{run}}$ is the last expression evaluated in c'. Thus

$$Time(e'_1) \leq Time(e') \leq Time(e'_2),$$

and then $e' \in suc^*(e'_1) \cap pre^*(e'_2)$ by Lemma 1. Hence, e' is included in $Concur(c, e, c')$. Therefore, every expression which may be evaluated in c' concurrently with e in c is included in $Concur(c, e, c')$. □

5.5 Step Two: Exception Analysis

Now, our analysis predicts uncaught exceptions in the input program by using the pre-analyzed concurrency information.

Our exception analysis is presented in the set-constraint framework [36, 5]. As we mentioned before, we assume a safe class information $Class(e)$ for every expression e and we use the safe concurrency information $Concur(c, e, c')$ (Section 5.4) for every pair of $e \in c$ and c' in the exception analysis.

We present two set-constraint systems and our analysis is the second one. Section 5.5.1 presents a constraint system that estimates uncaught exceptions from every expression of the input program. Because exception-related expressions are sparse in programs, it is wasteful to generate constraints for every expression of the program. Thus, in Section 5.5.2, we present a sparse constraint system that increases the constraint granularity to the method-level. This constraint system analyzes uncaught exceptions from every method of the input program. Similar techniques of designing a sparse constraint system that analyzes the properties at a larger granularity than at every expression have already been found to be cost-effective in ML [53]'s exception analysis [74, 76].

5.5.1 Exception Analysis at Expression-Level

Figure 5.7 shows our rules to generate set constraints for the uncaught exception classes of every expression. Every expression e of the input program has a set constraint: $X_e \supseteq se$. The set variable X_e is for the exception classes that the uncaught exceptions during e's evaluation belong to. A constraint $X_e \supseteq se$ may be read as "an expression e has uncaught exceptions including those of se" and a relation $\triangleright_1 e : \mathcal{C}$ may be read as "a set of constraints $\mathcal{C}$ is generated from an expression e."

Consider the rule for `throw` expression:

$$\frac{\triangleright_1 e_1 : \mathcal{C}_1}{\triangleright_1 \texttt{throw}\ e_1 : \{X_e \supseteq \mathit{Class}(e_1) \cup X_{e_1}\} \cup \mathcal{C}_1}.$$

It may have uncaught exceptions from inside e_1 (X_{e_1}) or throws exceptions e_1 ($\mathit{Class}(e_1)$).

Consider the rule for `try` expression:

$$\frac{\triangleright_1 e_1 : \mathcal{C}_1 \quad \triangleright_1 e_2 : \mathcal{C}_2}{\triangleright_1 \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : \{X_e \supseteq (X_{e_1} - \{c_1\}^*) \cup X_{e_2}\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2}.$$

If the classes of raised exceptions from e_1 are covered by c_1, the raised exceptions are caught by x_1 ($X_{e_1} - \{c_1\}^*$, where $\{c\}^*$ represents all the subclasses of a class c). While handling the uncaught exceptions from e_1 in e_2, this `try` expression may have uncaught exceptions from inside e_2 (X_{e_2}).

In the rule for method call:

$$\frac{\triangleright_1 e_1 : \mathcal{C}_1 \quad \triangleright_1 e_2 : \mathcal{C}_2}{\begin{array}{l}\triangleright_1 e_1\texttt{.}m\texttt{(}e_2\texttt{)} : \{X_e \supseteq X_{c'.m} \mid c' \in \mathit{Class}(e_1), m(x) = e_m \in c'\} \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \\ \qquad\qquad \cup\ \mathcal{C}_1 \cup \mathcal{C}_2\end{array}},$$

$$[\text{Program}]_1 \quad \frac{\rhd_1 C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_1 C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{ClassDef}]_1 \quad \frac{\rhd_1 M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_1 \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{MethDef}]_1 \quad \frac{\rhd_1 e_m : \mathcal{C}}{\rhd_1 m(x)\ \texttt{=}\ e_m : \{X_{c.m} \supseteq X_{e_m}\} \cup \mathcal{C}}$$

$$[\text{Variable}]_1 \quad \rhd_1 id : \emptyset$$

$$[\text{Assign}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 id\ \texttt{:=}\ e_1 : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{New}]_1 \quad \rhd_1 \texttt{new}\ c : \emptyset$$

$$[\text{This}]_1 \quad \rhd_1 \texttt{this} : \emptyset$$

$$[\text{Seq}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\rhd_1 e_1\ \texttt{;}\ e_2 : \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{Cond}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2 \quad \rhd_1 e_3 : \mathcal{C}_3}{\rhd_1 \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \{X_e \supseteq X_{e_1} \cup X_{e_2} \cup X_{e_3}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\text{Throw}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 \texttt{throw}\ e_1 : \{X_e \supseteq \mathit{Class}(e_1) \cup X_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{Try}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\rhd_1 \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : \{X_e \supseteq (X_{e_1} - \{c_1\}^*) \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{MethCall}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\begin{array}{c}\rhd_1 e_1\texttt{.}m\texttt{(}e_2\texttt{)} : \{X_e \supseteq X_{c'.m} \mid c' \in \mathit{Class}(e_1), m(x) = e_m \in c'\} \\ \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

$$[\text{ThdStart}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 e_1\texttt{.start} : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{Wait}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 e_1\texttt{.wait} : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{Notify}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 e_1\texttt{.notify} : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

$$[\text{Xthrow}]_1 \quad \frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\begin{array}{l}\rhd_1 \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{e'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2), \\ \qquad\qquad\qquad\qquad\qquad\qquad c' \in \mathit{Class}(e_2), \\ \qquad\qquad\qquad\qquad\qquad\qquad e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \qquad\qquad \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

Figure 5.7: Exception Analysis at Expression-Level

it may have uncaught exceptions from the subexpressions e_1 and e_2 ($X_{e_1} \cup X_{e_2}$), or from the body expression e_m of the method m which is defined inside the classes $c' \in \mathit{Class}(e_1)$ of e_1's objects ($X_e \supseteq X_{c'.m}$, where the subscript $c'.m$ indicates the body expression of class c''s method m).

Finally, consider the rule for `xthrow` expression:

$$\frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\begin{array}{l}\rhd_1 \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{e'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2),\ c' \in \mathit{Class}(e_2), \\ \qquad\qquad\qquad\qquad\qquad\qquad e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \qquad\qquad\qquad \cup\ \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}.$$

Uncaught exceptions from `xthrow` expression are those from the subexpressions e_1 and e_2 ($X_{e_1} \cup X_{e_2}$). Note that exceptions e_1 are thrown to the expressions e' ($X_{e'} \supseteq \mathit{Class}(e_1)$) which may be evaluated in the thread $c' \in \mathit{Class}(e_2)$ concurrently with the evaluation of `xthrow` expression in the thread c ($\mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')$).

Our exception analysis at expression-level is safe:

Theorem 2 *For a program $\wp$, let $\rhd_1 \wp : \mathcal{C}$. Then for every sub-expression e of $\wp$, $\mathit{lm}(\mathcal{C})(X_e)$ includes all the exceptions that escape from e during the execution of $\wp$.*

Proof Sketch. This correctness can be proved by the fixpoint induction over the continuous functions that are derived [22] from our constraint system [76]. Note that the correctness of our exception analysis relies on the correctness of both the class analysis and the concurrency analysis. □

5.5.2 Exception Analysis at Method-Level

Our new sparse constraint system is presented in Figure 5.8. Since this system generates constraints only for class' methods and `try`-blocks, the number of set variables becomes proportional to the number of methods and `try`-blocks instead of the num-

[Program]$_2$
$$\frac{\rhd_2\, C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_2\, C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[ClassDef]$_2$
$$\frac{\rhd_2 M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_2 \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[MethDef]$_2$
$$\frac{m \rhd_2 e_m : \mathcal{C}}{\rhd_2 m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m : \mathcal{C}}$$

[Variable]$_2$
$$m \rhd_2 id : \emptyset$$

[Assign]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ id\ \texttt{:=}\ e_1 : \mathcal{C}_1}$$

[New]$_2$
$$m \rhd_2 \texttt{new}\ c : \emptyset$$

[This]$_2$
$$m \rhd_2 \texttt{this} : \emptyset$$

[Seq]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1\quad m \rhd_2 e_2 : \mathcal{C}_2}{m \rhd_2\ e_1\ \texttt{;}\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond]$_2$
$$\frac{m \rhd_2 e_1 : \mathcal{C}_1\quad m \rhd_2 e_2 : \mathcal{C}_2\quad m \rhd_2 e_3 : \mathcal{C}_3}{m \rhd_2 \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

[Throw]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ \texttt{throw}\ e_1 : \{X_m \supseteq \mathit{Class}(e_1)\} \cup \mathcal{C}_1}$$

[Try]$_2$
$$\frac{g \rhd_2 e_g : \mathcal{C}_g\quad m \rhd_2 e_1 : \mathcal{C}_1}{m \rhd_2 \texttt{try}\ e_g\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_1\texttt{)} : \{X_m \supseteq (X_g - \{c_1\}^*)\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

[MethCall]$_2$
$$\frac{m \rhd_2 e_1 : \mathcal{C}_1\quad m \rhd_2 e_2 : \mathcal{C}_2}{\begin{array}{l} m \rhd_2\ e_1\texttt{.}m'\texttt{(}e_2\texttt{)} : \{X_m \supseteq X_{c'.m'} \mid c' \in \mathit{Class}(e_1), m'(x) = e_{m'} \in c'\} \\ \qquad \cup\, \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

[ThdStart]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ e_1\texttt{.start} : \mathcal{C}_1}$$

[Wait]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ e_1\texttt{.wait} : \mathcal{C}_1}$$

[Notify]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ e_1\texttt{.notify} : \mathcal{C}_1}$$

[Xthrow]$_2$
$$\frac{m \rhd_2 e_1 : \mathcal{C}_1\quad m \rhd_2 e_2 : \mathcal{C}_2}{\begin{array}{l} m \rhd_2\ \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{m'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2), \\ \qquad\qquad c' \in \mathit{Class}(e_2), e' \in e_{m'}, \\ \qquad\qquad m'(x) = e_{m'} \in \wp, \\ \qquad\qquad e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \qquad \cup\, \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Figure 5.8: Exception Analysis at Method-Level

ber of expressions. For each method m and `try`-block e_g in "`try` e_g `catch` $(c\ x\ e)$", the set variables X_m and X_g are for the classes that the uncaught exceptions during the evaluation of m's body and e_g, respectively, belong to. A new relation $m \rhd_2 e : \mathcal{C}$ ($g \rhd_2 e : \mathcal{C}$) is read as "a set of constraints $\mathcal{C}$ is generated from an expression e which is a subexpression of the body of a method m (`try`-block e_g)."

Consider the rule for `throw` expression:

$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ \texttt{throw}\ e_1 : \{X_m \supseteq \mathit{Class}(e_1)\} \cup \mathcal{C}_1}.$$

Uncaught exceptions from the method m, X_m, include the exception classes of the expression e_1 ($\mathit{Class}(e_1)$).

Consider the rule for `try` expression:

$$\frac{g \rhd_2 e_g : \mathcal{C}_g \quad m \rhd_2 e_1 : \mathcal{C}_1}{m \rhd_2 \texttt{try}\ e_g\ \texttt{catch}\ (c_1\ x_1\ e_1) : \{X_m \supseteq (X_g - \{c_1\}^*)\} \cup\ \mathcal{C}_g \cup \mathcal{C}_1}.$$

Among the classes of raised exceptions from e_g, those exceptions covered by c_1 are not included in the uncaught exceptions from m. Note that the left-hand side of the derivation rule for e_g is g.

Finally, consider the rule for `xthrow` expression:

$$\frac{m \rhd_2 e_1 : \mathcal{C}_1 \quad m \rhd_2 e_2 : \mathcal{C}_2}{\begin{array}{l} m\ \rhd_2\ \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{m'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2), \\ \qquad\qquad\qquad\qquad c' \in \mathit{Class}(e_2), m'(x) = e_{m'} \in \wp, e' \in e_{m'}, \\ \qquad\qquad\qquad\qquad e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \qquad\qquad \cup\, \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}.$$

Exceptions e_1 are thrown to the methods m' ($X_{m'} \supseteq \mathit{Class}(e_1)$) whose subexpressions e' may be evaluated in the thread $c' \in \mathit{Class}(e_2)$ concurrently with the evaluation of `xthrow` expression in the thread c ($Concur(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')$).

The method-level exception constraint rule $\triangleright_2$ is a safe approximation of $\triangleright_1$:

Theorem 3 *For a program $\wp$, let $\triangleright_1 \wp : \mathcal{C}_1$ and $\triangleright_2 \wp : \mathcal{C}_2$. Then, if $m \triangleright_2 e : \mathcal{C}$ occurs during $\triangleright_2 \wp : \mathcal{C}_2$ then $lm(\mathcal{C}_2)(X_m) \supseteq lm(\mathcal{C}_1)(X_e)$.*

Proof. This correctness proof is an instance of the correctness proof in [76]. $\square$

5.5.3 Separate Constraints for Improved Accuracy

We can safely improve the accuracy of our exception analysis. Since the thrown exceptions inside a thread do not propagate across the thread, it is possible to localize the effect of throwing asynchronous exceptions: for each method m, uncaught exceptions thrown from other threads are collected in X_m^T rather than in X_m. The rule for `xthrow` expression becomes:

$$\frac{m \triangleright_2 e_1 : \mathcal{C}_1 \quad m \triangleright_2 e_2 : \mathcal{C}_2}{\begin{array}{ll} m \triangleright_2 \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{m'}^T \supseteq Class(e_1) \mid & c \in owner(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2), \\ & c' \in Class(e_2), m'(x) = e_{m'} \in \wp, e' \in e_{m'}, \\ & e' \in Concur(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \quad \cup\ \mathcal{C}_1 \cup \mathcal{C}_2 & \end{array}}.$$

After constructing constraints for a given input program, only the set constraint $X_m \supseteq se$ for each method m is used during the first resolution phase. At the second phase, the set constraint $X_m^T \supseteq se$ for m is added to the partially solved system and then solved completely. The correctness of this improved $\triangleright_2$ can be proved with respect to an improved $\triangleright_1$.

5.6 Analysis Complexity

In this section, we analyze the complexity of our method. Since our analysis is divided into several analyses, the complexity of our analysis is a sum of the complexities of all sub-analyses.

The complexity of the time analysis among expressions *pre* and *suc* is linear to program size n. Constructing set constraints is linear to n and solving constraints is not need here because set expressions are just unions of expressions.

And the complexity of the time analysis among synchronizing expressions $\hat{pre}$ and $\hat{suc}$ is $O(n^3)$. Constructing constraints is linear to n and computing the constraints' least model uses the conventional fixpoint iteration of cubic complexity. The iterative method is possible because the set domain is finite (the set of expressions in the input program) and the set operator $\cup$ is monotonic.

The complexity of the concurrency analysis *Concur* is $O(n^5)$. First, the complexity of $\delta_\uparrow$ and $\delta_\downarrow$ is $O(n^5)$. Because for each c and c' in the program ($O(n^2)$ combinations), computing $\delta_\uparrow(c, e, c')$ and $\delta_\downarrow(c, e, c')$ uses the fixpoint iteration of cubic complexity. Second, the complexity of pre^* and suc^* is $O(n^3)$. Since pre^* (suc^*) applies *pre* (*suc*) at most n times and, for each application of *pre* (*suc*), there is at most n arguments of *pre* (*suc*) and n expressions may be added. At last, for each $e_1 \in \hat{pre}(e)$, $e_2 \in \hat{suc}(e)$, $e'_1 \in \delta_\uparrow(c, e_1, c')$, and $e'_2 \in \delta_\downarrow(c, e_2, c')$ ($O(n^4)$ combinations), *Concur* collects expressions in $suc^*(e'_1) \cap pre^{(}e'_2)$ ($O(n)$).

Finally, the complexity of our exception analysis at method-level is ($O(n^3)$). Since our construction of set constraints is not compositional, constructing constraints is not linear in n. Consider rule [Xthrow]$_2$. For each $c \in owner(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2)$ and $c' \in Class(e_2)$ ($O(n^2)$ combinations), it collects $e' \in Concur(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')$ ($O(n)$). And rule [MethCall]$_2$ collects $c'.m'$ for each $c' \in Class(e_1)$ ($O(n)$). Other rules generates constraints linear to program size. Thus, constructing constraints is $O(n)$ and computing the constraints' least solution uses the fixpoint iteration of cubic complexity because the set domain is finite and the set operators $\cup$ and $X_g - \{c_1\}^*$ are monotonic.

5.7 Related Work

In this section, we present previous work relating to our multithreaded exception analysis for Java. We introduce existing exception analyses for single-threaded Java programs. And we describe several class analyses and concurrency analyses.

5.7.1 Exception Analysis for Java Programs

Even though Java compiler ensures the programmer's specification for the possible uncaught exceptions in each method definition, there exist several work for helping fine-grained exception handling.

Yi and Chang [72, 18] developed an exception analyzer for single-threaded Java programs by a set-constraint framework. Their analyzer estimates the exception flows independently of the programmer's specifications. They have designed two exception analyses at expression-level and at method-level. They have proved that the expression-level analysis safely approximates uncaught exceptions and the method-level exception analysis gives the same analysis result as the expression-level analysis.

Robillard and Murphy [58] developed a tool called Jex that analyzes the flow of exceptions in Java programs by analyzing exception handling expressions. For each method, Jex extracts information about the structure of exceptions: the nested `try` block structures, including the guarded block, the `catch` clauses, and the `finally` block. But Jex does not report asynchronous exceptions. They described a set of design guidelines to simplify exception structure in [59].

5.7.2 Class Analysis

There are several existing work for class analysis. We assumed that class information *Class*(e) is already available for every expression e in our analysis. The class analysis estimates for each expression e the classes (including exception classes) that the expression e's normal object belongs to. Note that exception classes are normal classes in Java.

DeFouw, Grove, and Chambers [24] developed a general parameterized framework for interprocedural class analysis. Their system integrates both flow-based analysis [61, 32, 64, 28] and unification-based analysis [9, 7]. The system's instances include existing algorithms as well as several new algorithms whose complexity range from $O(N)$ to $O(N^3)$.

Tip and Palsberg [67] investigated algorithms between unification-based analysis and flow-based analysis. They designed four new algorithms and showed relationship among them and well-known existing algorithms. The ordering among algorithms corresponds to increased accuracy and cost. The increased accuracy is achieved by giving each expression multiple set variables.

5.7.3 Concurrency Analysis

There exist several concurrency analyses for eliminating unnecessary synchronizations. In multithreaded Java programs, useless synchronization may degrade performance of even single-threaded applications [8].

Bogda and Hölzle [13] developed an analysis which eliminates unnecessary synchronizations. Their analysis is interprocedural, flow- and context-insensitive one and based on the observation that objects reachable from only a single thread do not need any synchronization.

Ungureanu and Jagannathan [69] developed a whole-program flow analysis which uses a notion of an abstract thread of control. For each object, their method analyzes the set of threads that lock or access the object during the execution of the program. They also developed a flow-sensitive refinement of the analysis by using a per program-point abstract heap.

Chapter 6

Conclusion

6.1 Contributions

Exception handling facilities are useful but dangerous. Many modern programming languages such as C++, Java, Ada, Modula-3, and ML provide the programmer with exception mechanisms. However, the exception facilities can provide a hole for program safety. Even for type-safe programming languages like ML, exceptions provide a hole for program safety.

In this thesis, we design a static analysis that detects potential runtime exceptions that are raised and never handled inside SML programs. We found that even though the exception flow and control flow are in general intertwined in ML programs, the two analyses could be safely and cost-effectively decoupled. For cases where control flow analysis needs exception analysis, our control flow analysis uses a crude approximation to assure its safety against the decoupling.

Even though many languages offer both exception handling and multithreading mechanisms, they do not provide a good method to use these two features together. Although Java provides a simple and tightly integrated language support for multithreaded programs, multithreaded exception handling is not supported. In Java, throwing exceptions across threads is deprecated because of the safety problem.

We design another exception analysis for multithtreaded Java programs. In Java, throwing exceptions across threads is deprecated because of the safety problem. Instead of restricting programmers' freedom, we extend Java language to support multithreaded exception handling and propose a tool to detect uncaught exceptions in the input programs. Our analysis firstly estimates concurrently evaluated expressions among threads, and then predicts uncaught exceptions by using the pre-analyzed concurrency information.

One contribution of this thesis is an analysis in divide and conquer style. By dividing one complex analysis to several simpler analyses, the analysis result is achieved systematically. Another merit of our method is designing a sparse constraint system and proving its consistency with smaller-grained constraint systems. When the interesting properties are sparse in programs, it is reasonable to analyze the properties at a larger granularity than at every expression.

6.2 Future Work

Our method can be applied to other languages with exception handling facilities. As we have shown in Chapter 2, many modern programming languages provide the programmer with exception mechanisms. Since ML provides the most expressive exception mechanism among modern languages, exception mechanisms in other languages can be expressed in ML's exception mechanism. Thus, exception analyses for other languages would not be quite different from our work.

We are planning to improve and experiment our concurrency analysis. Our concurrency analysis estimates expressions evaluated concurrently in each thread at the same time. We designed this analysis to detect uncaught exceptions in multithreaded Java programs. However, the concurrency analysis gives the programmer useful information in itself. Concurrency information may be used for finding race conditions. We plan to implement our concurrency analysis and test the analyzer with existing multithreaded Java programs.

Appendix A

SML/NJ Exception Analyzer Version 0.98

A.1 Availability

SML/NJ Exception Analyzer Version 0.98 statically detects may-uncaught exceptions from Standard ML programs. This analyzer is completely embedded in the SML/NJ 110[23], to be interactively used inside the compiler's print-eval loop and also CM.

The analyzer is available at

`http://pllab.kaist.ac.kr/pub/exna/exna-README.html`.

Included files are as follows:

`exna-README` (analyzer introduction)

`exna-INSTALL` (installation guide)

`exna-bin.alpha32x.tar.gz` (executable files for DEC Alpha, OSF1 3.2) 4.5M

`exna-bin.mipseb.tar.gz` (executable files for SGI, IRIX64 R6.2) 4.2M

`exna-bin.sparc.tar.gz` (executable files for SUN Sparc, SunOS 5.5.1) 4.1M

`exna-src.tar.gz` (source files of the exception analyzer) 120K

A.2 Usage

- After installation, you can just use the compiler as usual. The exception analysis results will be printed in the print-eval loop.

- For each function of the program, it reports a set of pairs < exn, loc > of an exception exn and its birth (construction, not declaration) place loc. For example,

  - ```
    emitPar (type/e2m.sml:862.14-883.21)}
        { < NotImplemented, type/e2m.sml:58.40-58.54> }
    ```

    tells that function `emitPar` defined at `"type/e2m.sml:862.14-883.21"` may have an uncaught exception `NotImplemented`, which is constructed at `"type/e2m.sml:58.40-58.54."`
  - ```
    /unnamed ftn/ (type/e2m.sml:962.21-962.40)
        {}
    ```

    tells that the unnamed function at `"type/e2m.sml:962.21-962.40"` has no uncaught exception.
  - ```
    /emitBlock _ / (type/e2m.sml:857.18-860.58)
        { < NotSafe, type/e2m.sml:58.40-58.54> }
    ```

    tells that curried function `emitBlock` (of type `t -> t' -> t''`), given one argument, returns a function which may have an uncaught exception `NotSafe` when it is called.
  - ```
    /handler ftn/ (ppc/ppcEm2Ppc.sml:3.9-13.12)
        {}
    ```

    tells that handler function at `"ppc/ppcEm2Ppc.sml:3.9-13.12"` has no uncaught exception. Handler function is the match expression in handle expression: e handle match. If handler's match is not exhaustive, then the analysis says that the handler function (match) has uncaught exceptions.

- Both the semicolon ";" and file are the delimiters of the analysis unit. No analysis information can cross the delimiter, hence no support for separate analysis at the moment. Functor definition is analyzed assuming that no uncaught exceptions escape from the functions in the argument structure. Functor application is not analyzed.

- `- Control.CG.exn_analysis := false;`

  turns off the exception analyzer.

  `- Control.CG.exn_analysis_report := 1;`

  makes the analyzer report the analysis time.

Appendix B

Analysis Example: Knuth-Bendix Program

In this appendix, we present an analysis example with Knuth-Bendix program. The program is an implementation of the Knuth Bendix completion algorithm and processes some axioms of geometry. We show the source code of the program in nML[43] and present our control flow analysis results and exception analysis results.

B.1 Source Code

```
exception Failure of string

fun length l =
  let fun j(k,nil) = k
        | j(k, a::x) = j(k+1,x)
  in j(0,l)
  end

fun rev l =
  let fun f (nil, h) = h
        | f (a::r, h) = f(r, a::h)
  in  f(l,nil)
  end

fun failwith s = raise(Failure s)

fun fst (x,y) = x
```

```
and snd (x,y) = y

fun do_list f =
  let fun do_rec [] = ()
        | do_rec (a::l) = (f a; do_rec l)
  in do_rec
  end

fun map f =
  let fun map_rec [] = []
        | map_rec (a::l) = f a :: map_rec l
  in map_rec
  end

fun it_list f =
  let fun it_rec a [] = a
        | it_rec a (b::l) = it_rec (f a b) l
  in it_rec
  end

fun it_list2 f =
  let fun it_rec  a    []       []    = a
        | it_rec  a (a1::l1) (a2::l2) = it_rec (f a (a1,a2)) l1 l2
        | it_rec  _    _        _     = failwith "it_list2"
  in it_rec
  end

fun exists p =
  let fun exists_rec []  = false
        | exists_rec (a::l) = (p a) orelse (exists_rec l)
  in exists_rec
  end

fun for_all p =
  let fun for_all_rec []  = true
        | for_all_rec (a::l) = (p a) andalso (for_all_rec l)
  in for_all_rec
  end

fun rev_append   []    l  = l
  | rev_append (x::l1) l2 = rev_append l1 (x::l2)
```

```
fun try_find f =
  let fun try_find_rec  []    = failwith "try_find"
        | try_find_rec (a::l) =
          (f a) handle Failure _ => try_find_rec l
  in try_find_rec
  end

fun partition p =
  let fun part_rec   []   = ([],[])
        | part_rec (a::l) =
            let val (pos,neg) = part_rec l in
              if p a then  ((a::pos), neg) else (pos, (a::neg))
            end
  in part_rec
  end

fun mem a =
  let fun mem_rec [] = false
        | mem_rec (b::l) = (a=b) orelse mem_rec l
  in mem_rec
  end

fun union l1 l2 =
  let fun union_rec [] = l2
        | union_rec (a::l) =
            if mem a l2 then union_rec l else a :: union_rec l
  in union_rec l1
  end

fun mem_assoc a =
  let fun mem_rec [] = false
        | mem_rec ((b,_)::l) = (a=b) orelse mem_rec l
  in mem_rec
  end

fun assoc a =
  let fun assoc_rec [] = failwith "find"
        | assoc_rec ((b,d)::l) = if a=b then d else assoc_rec l
  in assoc_rec
  end

val print : string -> unit = print_string
```

```
val print_num : int -> unit = print_int

fun print_newline () = print "\n"

fun message (s : string) = (print s; print "\n")

fun union l1 =
  let fun union_rec [] = l1
        | union_rec (a::l) =
          if mem a l1 then union_rec l else a :: union_rec l
  in union_rec
  end

type term =
    Var of int
  | Term of string * term list

fun vars (Var n) = [n]
  | vars (Term(_,l)) = vars_of_list l

and vars_of_list [] = []
  | vars_of_list (t::r) = union (vars t) (vars_of_list r)

fun substitute subst =
  let fun subst_rec (Term(oper,sons)) =
          Term(oper, map subst_rec sons)
        | subst_rec (t as (Var n))     =
          (assoc n subst) handle Failure _ => t
  in subst_rec
  end

fun change f =
  let fun change_rec (h::t) n =
          if n=1 then f h :: t else h :: change_rec t (n-1)
        | change_rec _ _      = failwith "change"
  in change_rec
  end

fun replace m u n =
  let fun reprec (_, []) = n
        | reprec (Term(oper,sons), (n::u)) =
          Term(oper, change (fn p => reprec(p,u)) sons n)
```

```
        | reprec _ = failwith "replace"
  in reprec(m,u)
  end

fun matching term1 term2 =
  let fun match_rec subst (Var v, m) =
          if mem_assoc v subst then
            if m = assoc v subst then subst else failwith "matching"
          else
            (v,m) :: subst
        | match_rec subst (Term(op1,sons1), Term(op2,sons2)) =
          if op1 = op2 then it_list2 match_rec subst sons1 sons2
                       else failwith "matching"
        | match_rec _ _ = failwith "matching"
  in match_rec [] (term1,term2)
  end

fun compsubst subst1 subst2 =
  (map (fn (v,t) => (v, substitute subst1 t)) subst2) @ subst1

fun occurs n =
  let fun occur_rec (Var m) = (m=n)
        | occur_rec (Term(_,sons)) = exists occur_rec sons
  in occur_rec
  end

fun unify ((term1 as (Var n1)), term2) =
      if term1 = term2 then []
      else if occurs n1 term2 then failwith "unify"
      else [(n1,term2)]
  | unify (term1, Var n2) =
      if occurs n2 term1 then failwith "unify"
      else [(n2,term1)]
  | unify (Term(op1,sons1), Term(op2,sons2)) =
      if op1 = op2 then
        it_list2 (fn s => fn (t1,t2) =>
                    compsubst (unify(substitute s t1,
                                     substitute s t2)) s)
                 [] sons1 sons2
      else failwith "unify"

val infixes = ["+","*"]
```

```
fun pretty_term (Var n) =
    (print_string "v"; print_num n)
  | pretty_term (Term (oper,sons)) =
    if mem oper infixes then
      case sons
        of [s1,s2] =>
             (pretty_close s1; print_string oper; pretty_close s2)
         | _ =>
             failwith "pretty_term : infix arity <> 2"
    else
      (print_string oper;
       case sons
         of []   => ()
          | t::lt =>(print_string "(";
                     pretty_term t;
                     do_list (fn t => (print_string ",";
                                       pretty_term t)) lt;
                     print_string ")"))

and pretty_close (m as Term(oper, _)) =
    if mem oper infixes then
      (print_string "("; pretty_term m; print_string ")")
    else pretty_term m
  | pretty_close m = pretty_term m

fun mk_rule m n =
  let val all_vars = union (vars m) (vars n)
      val (k,subst) =
        it_list (fn (i,sigma) => fn v => (i+1,(v,Var(i))::sigma))
               (1,[]) all_vars
  in (k-1, (substitute subst m, substitute subst n))
  end

val check_rules =
  it_list (fn n => fn (k,_:((int * (term * term)))) =>
           if k=n+1 then k
           else failwith "Rule numbers not in sequence") 0

fun pretty_rule (k,(n:int,(m,nn))) =
 (print_num k; print_string " : ";
  pretty_term m; print_string " = "; pretty_term nn;
  print_newline())
```

```
val pretty_rules = do_list pretty_rule

fun reduce l m =
  substitute (matching l m)

fun reducible l =
  let fun redrec m =
    (matching l m; true)
    handle Failure _ =>
      case m of Term(_,sons) => exists redrec sons
              |        _     => false
  in redrec
  end

fun mreduce rules m =
  let fun redex (_,(_,(l,r))) = reduce l m r
  in try_find redex rules
  end

fun mrewrite1 rules =
  let fun rewrec m =
    (mreduce rules m) handle Failure _ =>
      let fun tryrec [] = failwith "mrewrite1"
            | tryrec (son::rest) =
                (rewrec son :: rest)
                handle Failure _ => son :: tryrec rest
      in case m of
          Term(f, sons) => Term(f, tryrec sons)
        | _ => failwith "mrewrite1"
      end
  in rewrec
  end

fun mrewrite_all rules m =
  let fun rew_loop m =
    rew_loop(mrewrite1 rules m)  handle  Failure _ => m
  in rew_loop m
  end

type ordering = Greater | Equal | NotGE

fun ge_ord order pair = case order pair of NotGE => false | _ => true
```

```
and gt_ord order pair = case order pair of Greater => true | _ => false

and eq_ord order pair = case order pair of Equal => true | _ => false

fun rem_eq equiv =
  let fun remrec x [] = failwith "rem_eq"
        | remrec x (y::l) = if equiv (x,y) then l else y :: remrec x l
  in remrec
  end

fun diff_eq equiv (x,y) =
  let fun diffrec (p as ([],_)) = p
        | diffrec ((h::t), y) =
            diffrec (t,rem_eq equiv h y) handle Failure _ =>
              let val (x',y') = diffrec (t,y) in (h::x',y') end
  in if length x > length y then diffrec(y,x) else diffrec(x,y)
  end

fun mult_ext order (Term(_,sons1), Term(_,sons2)) =
    (case diff_eq (eq_ord order) (sons1,sons2) of
         ([],[]) => Equal
       | (l1,l2) =>
         if for_all (fn n => exists (fn m => order (m,n) = Greater) l1) l2
         then Greater else NotGE)
  | mult_ext order (_, _) = failwith "mult_ext"

fun lex_ext order ((m as Term(_,sons1)), (n as Term(_,sons2))) =
      let fun lexrec ([] , []) = Equal
            | lexrec ([] , _ ) = NotGE
            | lexrec ( _ , []) = Greater
            | lexrec (x1::l1, x2::l2) =
                case order (x1,x2) of
                  Greater => if for_all (fn n' => gt_ord order (m,n')) l2
                             then Greater else NotGE
                | Equal => lexrec (l1,l2)
                | NotGE => if exists (fn m' => ge_ord order (m',n)) l1
                           then Greater else NotGE
      in lexrec (sons1, sons2)
      end
  | lex_ext order _ = failwith "lex_ext"

fun rpo op_order ext =
```

```
  let fun rporec (mm,nn) =
    if mm=nn then Equal else
      case mm of
          Var m => NotGE
        | Term(op1,sons1) =>
            case nn of
                Var n =>
                  if occurs n mm then Greater else NotGE
              | Term(op2,sons2) =>
                  case (op_order op1 op2) of
                      Greater =>
                        if for_all (fn nn' => gt_ord rporec (mm,nn')) sons2
                        then Greater else NotGE
                    | Equal =>
                        ext rporec (mm,nn)
                    | NotGE =>
                        if exists (fn mm' => ge_ord rporec (mm',nn)) sons1
                        then Greater else NotGE
  in rporec
  end

fun super mm =
  let fun suprec (nn as Term(_,sons)) =
      let fun collate (pairs,n) son =
              (pairs @ map (fn (u,sigma) => (n::u,sigma)) (suprec son), n+1)
          val insides =
              fst (it_list collate ([],1) sons)
      in ([], unify(mm,nn)) :: insides  handle Failure _ => insides
      end
    | suprec _ = []
  in suprec
  end

fun super_strict mm (Term(_,sons)) =
  let fun collate (pairs,n) son =
          (pairs @ map (fn (u,sigma) => (n::u,sigma)) (super mm son), n+1)
  in fst (it_list collate ([],1) sons)
  end
  | super_strict _ _ = []

fun critical_pairs (l1,r1) (l2,r2) =
  let fun mk_pair (u,sigma) =
```

```
        (substitute sigma (replace l2 u r1), substitute sigma r2)
  in map mk_pair (super l1 l2)
  end

fun strict_critical_pairs (l1,r1) (l2,r2) =
  let fun mk_pair (u,sigma) =
        (substitute sigma (replace l2 u r1), substitute sigma r2)
  in map mk_pair (super_strict l1 l2)
  end

fun mutual_critical_pairs eq1 eq2 =
  (strict_critical_pairs eq1 eq2) @ (critical_pairs eq2 eq1)

fun rename n (t1,t2) =
  let fun ren_rec (Var k) = Var(k+n)
        | ren_rec (Term(oper,sons)) = Term(oper, map ren_rec sons)
  in (ren_rec t1, ren_rec t2)
  end

fun deletion_message (k,_) =
  (print_string "Rule ";print_num k; message " deleted")

fun non_orientable (m,n) =
  (pretty_term m; print_string " = "; pretty_term n; print_newline())

fun kb_completion greater =
  let fun kbrec n rules =
    let val normal_form = mrewrite_all rules
        fun get_rule k = assoc k rules
        fun process failures =
          let fun processf (k,l) =
                let fun processkl [] =
                      if k<l then next_criticals (k+1,l) else
                        if l<n then next_criticals (1,l+1) else
                       (case failures
                         of [] => rules (* successful completion *)
                          | _  => (message "Non-orientable equations :";
                                  do_list non_orientable failures;
                                  failwith "kb_completion"))
                      | processkl ((mm,nn)::eqs) =
                         let val mm' = normal_form mm
                             val nn' = normal_form nn
```

```
                  fun enter_rule(left,right) =
                    let val new_rule = (n+1, mk_rule left right)
                     in
                      (pretty_rule new_rule;
                         let fun left_reducible (_,(_,(ll,_))) =
                                   reducible left ll
                             val (redl,irredl) =
                                   partition left_reducible rules
                          in (do_list deletion_message redl;
                              let fun right_reduce (m,(_,(ll,rr))) =
                                     (m,mk_rule ll
                                         (mrewrite_all
                                           (new_rule::rules) rr))
                                   val irreds =
                                        map right_reduce irredl
                                   val eqs' = map
                                    (fn (_,(_,pair)) => pair) redl
                               in kbrec (n+1) (new_rule::irreds)
                                        [] (k,l)
                                  (eqs @ eqs' @ failures)
                              end)
                         end)
                    end
              in if mm'=nn' then processkl eqs else
                 if greater(mm',nn') then enter_rule(mm',nn') else
                 if greater(nn',mm') then enter_rule(nn',mm') else
                  process ((mm',nn')::failures) (k,l) eqs
             end (* enter_rule *)
      in processkl
     end (* processf *)
and next_criticals (k,l) =
      (let val (v,el) = get_rule l in
        if k=l then
          processf (k,l) (strict_critical_pairs el (rename v el))
        else
          (let val (_,ek) = get_rule k in
                  processf (k,l) (mutual_critical_pairs
                                   el (rename v ek))
           end
          handle Failure "find" (*rule k deleted*) =>
           next_criticals (k+1,l))
        end
```

```
                     handle Failure "find" (*rule l deleted*) =>
                       next_criticals (1,l+1))
         in processf
        end (* process *)
     in process
     end (* kbrec *)
   in kbrec
   end (* kb_completion *)

fun kb_complete greater complete_rules rules =
    let val n = check_rules complete_rules
        val eqs = map (fn (_,(_,pair)) => pair) rules
        val completed_rules =
               kb_completion greater n complete_rules [] (n,n) eqs
    in (message "Canonical set found :";
        pretty_rules (rev completed_rules);
        ())
    end

val group_rules = [
  (1, (1, (Term("*", [Term("U",[]), Var 1]), Var 1))),
  (2, (1, (Term("*", [Term("I",[Var 1]), Var 1]), Term("U",[])))),
  (3, (3, (Term("*", [Term("*", [Var 1, Var 2]), Var 3]),
          Term("*", [Var 1, Term("*", [Var 2, Var 3])]))))]

val geom_rules = [
 (1,(1,(Term ("*",[(Term ("U",[])), (Var 1)]),(Var 1)))),
 (2,(1,(Term ("*",[(Term ("I",[(Var 1)])), (Var 1)]),(Term ("U",[]))))),
 (3,(3,(Term ("*",[(Term ("*",[(Var 1), (Var 2)])), (Var 3)]),
  (Term ("*",[(Var 1), (Term ("*",[(Var 2), (Var 3)]))]))))),
 (4,(0,(Term ("*",[(Term ("A",[])), (Term ("B",[]))]),
  (Term ("*",[(Term ("B",[])), (Term ("A",[]))]))))),
 (5,(0,(Term ("*",[(Term ("C",[])), (Term ("C",[]))]),(Term ("U",[]))))),
 (6,(0,
  (Term
   ("*",
    [(Term ("C",[])),
     (Term ("*",[(Term ("A",[])), (Term ("I",[(Term ("C",[]))]))]))])),
  (Term ("I",[(Term ("A",[]))])))),
 (7,(0,
  (Term
   ("*",
```

```
    [(Term ("C",[])),
     (Term ("*",[(Term ("B",[])), (Term ("I",[(Term ("C",[]))]))]))]),
  (Term ("B",[]))))
]


fun group_rank "U" = 0
  | group_rank "*" = 1
  | group_rank "I" = 2
  | group_rank "B" = 3
  | group_rank "C" = 4
  | group_rank "A" = 5

fun group_precedence op1 op2 =
  let val r1 = group_rank op1
      val r2 = group_rank op2
  in
    if r1 = r2 then Equal else
    if r1 > r2 then Greater else NotGE
  end

val group_order = rpo group_precedence lex_ext

fun greater pair =
  case group_order pair of Greater => true | _ => false

fun doit() = (kb_complete greater [] group_rules;
              kb_complete greater [] geom_rules)
```

B.2 Control Flow Analysis Results

For each function-typed expression in the input program, control flow analysis gives its possible set of function values. Because there are too many function-typed expressions in the Knuth-Bendix program, we only show control flow analysis results for e_1 expressions in the application expressions $e_1\ e_2$ in the program. Since the control flow analysis is trivial when e_1 is either an immediate function or a named function, we do not show such cases.

We present the control flow analysis results as the pairs of the location of e_1 and its possible set of functions. Because there are some functions with the same names and all the immediate functions' names are "/unnamed ftn/", we use both a function name and its location for identifying functions. But for presentation simplicity, we only show functions' names in this example.

```
loc=kb.n 23.27-28: ftns=/unnamed ftn/ pretty_rule deletion_message
                        non_orientable
loc=kb.n 29.27-28: ftns=subst_rec /unnamed ftn/ /unnamed ftn/
                        /unnamed ftn/ mk_pair mk_pair ren_rec
                        right_reduce /unnamed ftn/ /unnamed ftn/
loc=kb.n 35.36-37: ftns=/unnamed ftn/ /unnamed ftn/ collate collate
loc=kb.n 35.36-39: ftns=/unnamed ftn/ /unnamed ftn/ /collate _ /
                        /collate _ /
loc=kb.n 35.28-42: ftns=/it_rec _ /
loc=kb.n 41.48-49: ftns=match_rec /unnamed ftn/
loc=kb.n 41.48-51: ftns=/match_rec _ / /unnamed ftn/
loc=kb.n 41.40-60: ftns=/it_rec _ /
loc=kb.n 41.40-63: ftns=/it_rec _ _ /
loc=kb.n 48.31-32: ftns=occur_rec redrec /unnamed ftn/ /unnamed ftn/
                        /unnamed ftn/
loc=kb.n 54.32-33: ftns=/unnamed ftn/ /unnamed ftn/ /unnamed ftn/
loc=kb.n 59.28-41: ftns=/rev_append _ /
loc=kb.n 63.33-34: ftns=redex
loc=kb.n 71.17-18: ftns=left_reducible
loc=kb.n 85.15-20: ftns=mem_rec
loc=kb.n 112.13-18: ftns=mem_rec
loc=kb.n 124.26-40: ftns=union_rec
loc=kb.n 127.51-64: ftns=map_rec
loc=kb.n 128.42-49: ftns=assoc_rec
loc=kb.n 133.44-45: ftns=/unnamed ftn/
loc=kb.n 134.49-61: ftns=/change_rec _ /
loc=kb.n 142.24-52: ftns=change_rec
loc=kb.n 142.24-57: ftns=/change_rec _ /
loc=kb.n 149.13-24: ftns=mem_rec
loc=kb.n 150.19-26: ftns=assoc_rec
loc=kb.n 154.21-39: ftns=it_rec
loc=kb.n 154.21-45: ftns=/it_rec _ /
loc=kb.n 154.21-51: ftns=/it_rec _ _ /
loc=kb.n 157.5-17:  ftns=/match_rec _ /
loc=kb.n 161.24-41: ftns=subst_rec
```

```
loc=kb.n 161.3-45:  ftns=map_rec
loc=kb.n 165.37-53: ftns=exists_rec
loc=kb.n 171.14-23: ftns=occur_rec
loc=kb.n 174.9-18:  ftns=occur_rec
loc=kb.n 178.50-62: ftns=subst_rec
loc=kb.n 179.57-69: ftns=subst_rec
loc=kb.n 178.33-141: ftns=/compsubst _ /
loc=kb.n 178.1-144: ftns=it_rec
loc=kb.n 178.1-164: ftns=/it_rec _ /
loc=kb.n 178.1-170: ftns=/it_rec _ _ /
loc=kb.n 188.7-15:  ftns=mem_rec
loc=kb.n 200.21-72: ftns=do_rec
loc=kb.n 204.7-15:  ftns=mem_rec
loc=kb.n 210.21-35: ftns=union_rec
loc=kb.n 212.8-65:  ftns=it_rec
loc=kb.n 212.8-87:  ftns=/it_rec _ /
loc=kb.n 214.12-28: ftns=subst_rec
loc=kb.n 214.32-48: ftns=subst_rec
loc=kb.n 218.2-124: ftns=it_rec
loc=kb.n 230.14-24: ftns=/matching _ /
loc=kb.n 234.5-15:  ftns=/matching _ /
loc=kb.n 236.32-45: ftns=exists_rec
loc=kb.n 242.32-40: ftns=/reduce _ /
loc=kb.n 242.32-42: ftns=subst_rec
loc=kb.n 242.48-62: ftns=try_find_rec
loc=kb.n 246.5-18:  ftns=/mreduce _ /
loc=kb.n 259.13-28: ftns=rewrec
loc=kb.n 265.29-34: ftns=rporec
loc=kb.n 267.29-34: ftns=rporec
loc=kb.n 273.31-36: ftns=/eq_ord _ /
loc=kb.n 273.60-68: ftns=/remrec _ /
loc=kb.n 280.23-35: ftns=remrec
loc=kb.n 280.23-37: ftns=/remrec _ /
loc=kb.n 286.12-34: ftns=/diff_eq _ /
loc=kb.n 289.33-71: ftns=exists_rec
loc=kb.n 289.16-75: ftns=for_all_rec
loc=kb.n 298.21-26: ftns=rporec
loc=kb.n 299.50-62: ftns=/gt_ord _ /
loc=kb.n 299.32-70: ftns=for_all_rec
loc=kb.n 302.47-59: ftns=/ge_ord _ /
loc=kb.n 302.30-67: ftns=exists_rec
loc=kb.n 316.21-29: ftns=occur_rec
```

```
loc=kb.n 318.24-32: ftns=group_precedence
loc=kb.n 318.24-36: ftns=/group_precedence _ /
loc=kb.n 320.46-59: ftns=/gt_ord _ /
loc=kb.n 320.27-69: ftns=for_all_rec
loc=kb.n 323.24-27: ftns=lex_ext
loc=kb.n 323.24-34: ftns=/lex_ext _ /
loc=kb.n 325.45-58: ftns=/ge_ord _ /
loc=kb.n 325.27-68: ftns=exists_rec
loc=kb.n 333.25-59: ftns=map_rec
loc=kb.n 335.21-36: ftns=it_rec
loc=kb.n 335.21-43: ftns=/it_rec _ /
loc=kb.n 344.49-57: ftns=suprec
loc=kb.n 344.13-47: ftns=map_rec
loc=kb.n 345.10-25: ftns=it_rec
loc=kb.n 345.10-32: ftns=/it_rec _ /
loc=kb.n 351.24-34: ftns=/replace _ /
loc=kb.n 351.24-36: ftns=/replace _ _ /
loc=kb.n 351.6-22:  ftns=subst_rec
loc=kb.n 351.42-58: ftns=subst_rec
loc=kb.n 353.17-25: ftns=suprec
loc=kb.n 353.4-15:  ftns=map_rec
loc=kb.n 358.23-33: ftns=/replace _ /
loc=kb.n 358.23-35: ftns=/replace _ _ /
loc=kb.n 358.5-21:  ftns=subst_rec
loc=kb.n 358.41-57: ftns=subst_rec
loc=kb.n 360.17-32: ftns=/super_strict _ /
loc=kb.n 360.4-15:  ftns=map_rec
loc=kb.n 364.3-28:  ftns=/strict_critical_pairs _ /
loc=kb.n 364.37-55: ftns=/critical_pairs _ /
loc=kb.n 368.49-60: ftns=map_rec
loc=kb.n 381.25-32: ftns=assoc_rec
loc=kb.n 390.36-58: ftns=do_rec
loc=kb.n 396.57-69: ftns=/mk_rule _ /
loc=kb.n 400.47-61: ftns=redrec
loc=kb.n 402.47-71: ftns=part_rec
loc=kb.n 403.42-66: ftns=do_rec
loc=kb.n 406.54-140: ftns=/mrewrite_all _ /
loc=kb.n 405.52-62: ftns=/mk_rule _ /
loc=kb.n 409.52-68: ftns=map_rec
loc=kb.n 410.58-136: ftns=map_rec
loc=kb.n 412.46-57: ftns=/kbrec _ /
loc=kb.n 412.46-76: ftns=process
```

```
loc=kb.n 412.46-132: ftns=processf
loc=kb.n 412.46-138: ftns=processkl
loc=kb.n 419.32-39: ftns=greater
loc=kb.n 420.32-39: ftns=greater
loc=kb.n 421.30-59: ftns=processf
loc=kb.n 421.30-65: ftns=processkl
loc=kb.n 428.65-73: ftns=/rename _ /
loc=kb.n 428.39-63: ftns=/strict_critical_pairs _ /
loc=kb.n 428.23-37: ftns=processkl
loc=kb.n 432.53-61: ftns=/rename _ /
loc=kb.n 431.47-121: ftns=/mutual_critical_pairs _ /
loc=kb.n 431.31-45: ftns=processkl
loc=kb.n 448.18-47: ftns=map_rec
loc=kb.n 450.15-36: ftns=kbrec
loc=kb.n 450.15-38: ftns=/kbrec _ /
loc=kb.n 450.15-53: ftns=process
loc=kb.n 450.15-56: ftns=processf
loc=kb.n 450.15-62: ftns=processkl
loc=kb.n 499.18-38: ftns=/rpo _ /
loc=kb.n 504.14-33: ftns=/kb_complete _ /
loc=kb.n 504.14-36: ftns=/kb_complete _ _ /
loc=kb.n 505.14-33: ftns=/kb_complete _ /
loc=kb.n 505.14-36: ftns=/kb_complete _ _ /
```

B.3 Exception Analysis Results

For each function in the input program, exception analysis gives its possible uncaught exceptions during the evaluation of its body.

```
length (File "kb.n", line 3, characters 4-81)
          {}
j (File "kb.n", line 4, characters 10-47)
          {}
rev (File "kb.n", line 9, characters 4-85)
          {}
f (File "kb.n", line 10, characters 10-52)
          {}
failwith (File "kb.n", line 15, characters 4-33)
          { <Failure, File "kb.n", line 15, characters 23-30> }
fst (File "kb.n", line 17, characters 4-17)
          {}
```

```
snd (File "kb.n", line 19, characters 4-17)
          {}
do_list (File "kb.n", line 21, characters 4-100)
          {}
do_rec (File "kb.n", line 22, characters 10-66)
          { <Failure, File "kb.n", line 15, characters 23-30> }
map (File "kb.n", line 27, characters 4-100)
          {}
map_rec (File "kb.n", line 28, characters 10-69)
          { <Failure, File "kb.n", line 15, characters 23-30> }
it_list (File "kb.n", line 33, characters 4-104)
          {}
it_rec (File "kb.n", line 34, characters 10-70)
          {}
/it_rec _ / (File "kb.n", line 34, characters 10-70)
          { <Failure, File "kb.n", line 15, characters 23-30> }
it_list2 (File "kb.n", line 39, characters 4-203)
          {}
it_rec (File "kb.n", line 40, characters 10-168)
          {}
/it_rec _ / (File "kb.n", line 40, characters 10-168)
          {}
/it_rec _ _ / (File "kb.n", line 40, characters 10-168)
          { <Failure, File "kb.n", line 15, characters 23-30> }
exists (File "kb.n", line 46, characters 4-127)
          {}
exists_rec (File "kb.n", line 47, characters 10-90)
          { <Failure, File "kb.n", line 15, characters 23-30> }
for_all (File "kb.n", line 52, characters 4-132)
          {}
for_all_rec (File "kb.n", line 53, characters 10-93)
          { <Failure, File "kb.n", line 15, characters 23-30> }
rev_append (File "kb.n", line 58, characters 4-80)
          {}
/rev_append _ / (File "kb.n", line 58, characters 4-80)
          {}
try_find (File "kb.n", line 61, characters 4-165)
          {}
try_find_rec (File "kb.n", line 62, characters 10-124)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/handler ftn/ (File "kb.n", line 63, characters 32-72)
          { <Failure, File "kb.n", line 15, characters 23-30> }
```

```
partition (File "kb.n", line 67, characters 4-227)
          {}
part_rec (File "kb.n", line 68, characters 10-189)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mem (File "kb.n", line 76, characters 4-109)
          {}
mem_rec (File "kb.n", line 77, characters 10-78)
          {}
union (File "kb.n", line 82, characters 4-161)
          {}
/union _ / (File "kb.n", line 82, characters 4-161)
          {}
union_rec (File "kb.n", line 83, characters 10-119)
          {}
mem_assoc (File "kb.n", line 89, characters 4-119)
          {}
mem_rec (File "kb.n", line 90, characters 10-82)
          {}
assoc (File "kb.n", line 95, characters 4-139)
          {}
assoc_rec (File "kb.n", line 96, characters 10-104)
          { <Failure, File "kb.n", line 15, characters 23-30> }
print (File "kb.n", line 101, characters 29-41)
          {}
print_num (File "kb.n", line 103, characters 30-39)
          {}
print_newline (File "kb.n", line 105, characters 4-33)
          {}
message (File "kb.n", line 107, characters 4-48)
          {}
union (File "kb.n", line 109, characters 4-153)
          {}
union_rec (File "kb.n", line 110, characters 10-117)
          {}
vars (File "kb.n", line 120, characters 4-60)
          {}
vars_of_list (File "kb.n", line 123, characters 4-82)
          {}
substitute (File "kb.n", line 126, characters 4-193)
          {}
subst_rec (File "kb.n", line 127, characters 10-149)
          { <Failure, File "kb.n", line 15, characters 23-30> }
```

```
/handler ftn/ (File "kb.n", line 128, characters 41-78)
          {}
change (File "kb.n", line 132, characters 4-207)
          {}
change_rec (File "kb.n", line 133, characters 10-170)
          {}
/change_rec _ / (File "kb.n", line 133, characters 10-170)
          { <Failure, File "kb.n", line 15, characters 23-30> }
replace (File "kb.n", line 139, characters 4-218)
          {}
/replace _ / (File "kb.n", line 139, characters 4-218)
          {}
/replace _ _ / (File "kb.n", line 139, characters 4-218)
          { <Failure, File "kb.n", line 15, characters 23-30> }
reprec (File "kb.n", line 140, characters 10-174)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 142, characters 32-51)
          { <Failure, File "kb.n", line 15, characters 23-30> }
matching (File "kb.n", line 147, characters 4-465)
          {}
/matching _ / (File "kb.n", line 147, characters 4-465)
          { <Failure, File "kb.n", line 15, characters 23-30> }
match_rec (File "kb.n", line 148, characters 10-400)
          {}
/match_rec _ / (File "kb.n", line 148, characters 10-400)
          { <Failure, File "kb.n", line 15, characters 23-30> }
compsubst (File "kb.n", line 160, characters 4-93)
          {}
/compsubst _ / (File "kb.n", line 160, characters 4-93)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 161, characters 8-44)
          { <Failure, File "kb.n", line 15, characters 23-30> }
occurs (File "kb.n", line 163, characters 4-130)
          {}
occur_rec (File "kb.n", line 164, characters 10-94)
          { <Failure, File "kb.n", line 15, characters 23-30> }
unify (File "kb.n", line 169, characters 4-523)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 178, characters 11-143)
          {}
/unnamed ftn/ (File "kb.n", line 178, characters 19-143)
          { <Failure, File "kb.n", line 15, characters 23-30> }
```

```
pretty_term (File "kb.n", line 185, characters 4-535)
          { <Failure, File "kb.n", line 15, characters 23-30> }
pretty_close (File "kb.n", line 203, characters 4-184)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 200, characters 30-71)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mk_rule (File "kb.n", line 209, characters 4-240)
          {}
/mk_rule _ / (File "kb.n", line 209, characters 4-240)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 212, characters 17-64)
          {}
/unnamed ftn/ (File "kb.n", line 212, characters 33-64)
          {}
check_rules (File "kb.n", line 218, characters 2-126)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 218, characters 11-123)
          {}
/unnamed ftn/ (File "kb.n", line 218, characters 19-123)
          { <Failure, File "kb.n", line 15, characters 23-30> }
pretty_rule (File "kb.n", line 222, characters 4-143)
          { <Failure, File "kb.n", line 15, characters 23-30> }
pretty_rules (File "kb.n", line 227, characters 19-38)
          { <Failure, File "kb.n", line 15, characters 23-30> }
reduce (File "kb.n", line 229, characters 4-44)
          {}
/reduce _ / (File "kb.n", line 229, characters 4-44)
          { <Failure, File "kb.n", line 15, characters 23-30> }
reducible (File "kb.n", line 232, characters 4-194)
          {}
redrec (File "kb.n", line 233, characters 10-158)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/handler ftn/ (File "kb.n", line 234, characters 4-137)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mreduce (File "kb.n", line 241, characters 4-94)
          {}
/mreduce _ / (File "kb.n", line 241, characters 4-94)
          { <Failure, File "kb.n", line 15, characters 23-30> }
redex (File "kb.n", line 242, characters 10-44)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mrewrite1 (File "kb.n", line 244, characters 4-373)
          {}
```

```
rewrec (File "kb.n", line 245, characters 10-333)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/handler ftn/ (File "kb.n", line 246, characters 4-312)
          { <Failure, File "kb.n", line 15, characters 23-30> }
tryrec (File "kb.n", line 247, characters 14-157)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/handler ftn/ (File "kb.n", line 249, characters 16-75)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mrewrite_all (File "kb.n", line 257, characters 4-127)
          {}
/mrewrite_all _ / (File "kb.n", line 257, characters 4-127)
          {}
rew_loop (File "kb.n", line 258, characters 10-78)
          {}
/handler ftn/ (File "kb.n", line 259, characters 4-55)
          {}
ge_ord (File "kb.n", line 265, characters 4-69)
          {}
/ge_ord _ / (File "kb.n", line 265, characters 4-69)
          { <Failure, File "kb.n", line 15, characters 23-30> }
gt_ord (File "kb.n", line 267, characters 4-71)
          {}
/gt_ord _ / (File "kb.n", line 267, characters 4-71)
          { <Failure, File "kb.n", line 15, characters 23-30> }
eq_ord (File "kb.n", line 269, characters 4-69)
          {}
/eq_ord _ / (File "kb.n", line 269, characters 4-69)
          {}
rem_eq (File "kb.n", line 271, characters 4-149)
          {}
remrec (File "kb.n", line 272, characters 10-112)
          {}
/remrec _ / (File "kb.n", line 272, characters 10-112)
          { <Failure, File "kb.n", line 15, characters 23-30> }
diff_eq (File "kb.n", line 277, characters 4-288)
          {}
/diff_eq _ / (File "kb.n", line 277, characters 4-288)
          {}
diffrec (File "kb.n", line 278, characters 10-192)
          {}
/handler ftn/ (File "kb.n", line 280, characters 12-124)
          {}
```

```
mult_ext (File "kb.n", line 285, characters 4-318)
          {}
/mult_ext _ / (File "kb.n", line 285, characters 4-318)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 289, characters 25-74)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 289, characters 41-70)
          {}
lex_ext (File "kb.n", line 293, characters 4-636)
          {}
/lex_ext _ / (File "kb.n", line 293, characters 4-636)
          { <Failure, File "kb.n", line 15, characters 23-30> }
lexrec (File "kb.n", line 294, characters 14-489)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 299, characters 41-69)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 302, characters 38-66)
          { <Failure, File "kb.n", line 15, characters 23-30> }
rpo (File "kb.n", line 308, characters 4-738)
          {}
/rpo _ / (File "kb.n", line 308, characters 4-738)
          {}
rporec (File "kb.n", line 309, characters 10-697)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 320, characters 36-68)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 325, characters 35-67)
          { <Failure, File "kb.n", line 15, characters 23-30> }
super (File "kb.n", line 330, characters 4-361)
          {}
suprec (File "kb.n", line 331, characters 10-328)
          { <Failure, File "kb.n", line 15, characters 23-30> }
collate (File "kb.n", line 332, characters 14-116)
          {}
/collate _ / (File "kb.n", line 332, characters 14-116)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 333, characters 30-58)
          {}
/handler ftn/ (File "kb.n", line 336, characters 9-67)
          {}
super_strict (File "kb.n", line 342, characters 4-210)
          {}
```

```
/super_strict _ / (File "kb.n", line 342, characters 4-210)
          { <Failure, File "kb.n", line 15, characters 23-30> }
collate (File "kb.n", line 343, characters 10-102)
          {}
/collate _ / (File "kb.n", line 343, characters 10-102)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 344, characters 18-46)
          {}
critical_pairs (File "kb.n", line 349, characters 4-170)
          {}
/critical_pairs _ / (File "kb.n", line 349, characters 4-170)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mk_pair (File "kb.n", line 350, characters 10-92)
          { <Failure, File "kb.n", line 15, characters 23-30> }
strict_critical_pairs (File "kb.n", line 356, characters 4-183)
          {}
/strict_critical_pairs _ / (File "kb.n", line 356, characters 4-183)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mk_pair (File "kb.n", line 357, characters 10-91)
          { <Failure, File "kb.n", line 15, characters 23-30> }
mutual_critical_pairs (File "kb.n", line 363, characters 4-96)
          {}
/mutual_critical_pairs _ / (File "kb.n", line 363, characters 4-96)
          { <Failure, File "kb.n", line 15, characters 23-30> }
rename (File "kb.n", line 366, characters 4-162)
          {}
/rename _ / (File "kb.n", line 366, characters 4-162)
          { <Failure, File "kb.n", line 15, characters 23-30> }
ren_rec (File "kb.n", line 367, characters 10-103)
          { <Failure, File "kb.n", line 15, characters 23-30> }
deletion_message (File "kb.n", line 372, characters 4-85)
          {}
non_orientable (File "kb.n", line 375, characters 4-96)
          { <Failure, File "kb.n", line 15, characters 23-30> }
kb_completion (File "kb.n", line 378, characters 4-3550)
          {}
kbrec (File "kb.n", line 379, characters 10-3493)
          {}
/kbrec _ / (File "kb.n", line 379, characters 10-3493)
          {}
normal_form (File "kb.n", line 380, characters 26-44)
          {}
```

```
get_rule (File "kb.n", line 381, characters 12-38)
          { <Failure, File "kb.n", line 15, characters 23-30> }
process (File "kb.n", line 382, characters 12-3346)
          {}
processf (File "kb.n", line 383, characters 18-2563)
          {}
next_criticals (File "kb.n", line 425, characters 17-705)
          { <Failure, File "kb.n", line 15, characters 23-30> }
processkl (File "kb.n", line 384, characters 25-2459)
          { <Failure, File "kb.n", line 15, characters 23-30> }
enter_rule (File "kb.n", line 395, characters 34-1491)
          { <Failure, File "kb.n", line 15, characters 23-30> }
left_reducible (File "kb.n", line 399, characters 45-142)
          { <Failure, File "kb.n", line 15, characters 23-30> }
right_reduce (File "kb.n", line 404, characters 50-292)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 411, characters 49-72)
          {}
/handler ftn/ (File "kb.n", line 426, characters 20-664)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/handler ftn/ (File "kb.n", line 430, characters 24-334)
          { <Failure, File "kb.n", line 15, characters 23-30> }
kb_complete (File "kb.n", line 446, characters 4-345)
          {}
/kb_complete _ / (File "kb.n", line 446, characters 4-345)
          {}
/kb_complete _ _ / (File "kb.n", line 446, characters 4-345)
          { <Failure, File "kb.n", line 15, characters 23-30> }
/unnamed ftn/ (File "kb.n", line 448, characters 23-46)
          {}
group_rank (File "kb.n", line 484, characters 4-137)
          {}
group_precedence (File "kb.n", line 491, characters 4-171)
          {}
/group_precedence _ / (File "kb.n", line 491, characters 4-171)
          {}
group_order (File "kb.n", line 499, characters 18-46)
          { <Failure, File "kb.n", line 15, characters 23-30> }
greater (File "kb.n", line 501, characters 4-74)
          { <Failure, File "kb.n", line 15, characters 23-30> }
doit (File "kb.n", line 504, characters 4-98)
          { <Failure, File "kb.n", line 15, characters 23-30> }
```

요 약 문

프로그램 실행 중 처리되지 않는 예외상황을 예측하는 방법

예외상황 관리 방법 (exception handling)은 매우 유용하게 사용되고 있지만 또한 위험하기도 하다. C++, Java, Ada, Modula-3, ML과 같은 많은 언어들이 예외상황 관리 방법을 제공하여, 프로그래머가 자유롭게 예외상황을 정의하고 발생시키고 처리할 수 있도록 하고 있다. 하지만, 프로그래머가 주의를 가지고 사용하지 않으면 오히려 프로그램의 안전도를 해치는 상황으로 발전할 수 있다. 프로그램 내에서 예외 상황이 발생하면 그 예외 상황을 처리할 수 있는 곳까지 프로그램의 구조를 빠져나오게 되는데, 만일 발생된 예외 상황을 처리하는 곳이 없으면 프로그램이 갑작스럽게 그 실행을 중단해버리기 때문이다.

많은 프로그래밍 언어가 예외상황과 다중 스레드를 모두 제공하고 있긴 하지만, 이 두 가지를 같이 사용하는 경우에 대해서는 잘 다루지 못하고 있다. Java는 언어 자체가 간단하고 강력한 다중 스레드 사용 방법을 제공하고 있지만, 다중 스레드 사이에서 예외상황을 발생시키는 것은 안전성 문제로 제한하고 있다.

본 연구에서는 프로그램의 실행 중 처리되지 않는 예외상황을 프로그램 실행 전에 미리 예측하는 분석 방법을 제안한다. 먼저 다중 스레드를 고려하지 않은 ML 프로그램에 대한 예외상황 분석 방법을 고안한 후, 다중 스레드 Java 프로그램에 대한 예외상황 분석 방법을 제안한다. ML 프로그래머가 표현할 수 있는 예외 상황은 매우 광범위하지만, 일반적으로 사용되는 예외상황의 내용은 대부분이 간단한 경우들임을 알 수 있었다. 따라서 본 연구에서는 타입 정보를 이용하여 프로그램의 함수 흐름도를 간단하게 예측한 후, 그 정보를 이용하여 예외상황의 흐름을 분석하는 방법을 개발한다. 또

한 Java 언어를 확장하여 프로그래머가 자유롭게 다중 스레드 사이에 예외상황을 발생시킬 수 있도록 한 후에 그 프로그램에서 처리되지 않는 예외상황을 분석하는 도구를 제안한다. 이 분석 방법은 다중 스레드 Java 프로그램의 각 스레드에서 동시에 수행될 수 있는 부분들을 미리 예측해내고, 이 분석 정보를 이용하여 다중 스레드 사이에서 예외상황을 발생시킬 때 처리되지 않는 예외상황을 예측하는 것이다.

본 연구의 다음과 같은 장점들은 예외상황을 제공하는 다른 언어에도 사용될 수 있을 것이다. 첫째는 복잡한 하나의 분석 문제를 간단한 여러 개의 분석으로 나누어서 체계적으로 해결하는 것이다. 두번째는 분석하고자 하는 성질이 프로그램에 흩어져 있을 때 제약식을 적게 만드는 것이다. 제약식이 적으면 분석하는 데 필요한 시간이 줄어들게 되고, 이 시스템의 안전성은 더 자세한 시스템의 안전성으로부터 쉽게 이끌어낼 수 있게 된다.

References

[1] Java^{TM} 2 SDK, Standard Edition 1.2. http://java.sun.com/products/jdk/1.2.

[2] Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit deprecated? http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html.

[3] Ariane 5: Flight 501 Failure. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, July 1996.

[4] Using Compaq C++ for Tru64 UNIX and Linux Alpha. http://www.tru64unix.compaq.com/cplus/docs/ugu.htm, January 2001.

[5] Alex Aiken and Nevin Heintze. Constraint-based program analysis. Tutorial Notes of the ACM Symposium on Principles of Programming Languages, January 1995.

[6] Alex Aiken *et al.* The Berkeley ANalysis Engine. http://http.cs.berkeley.edu/Research/Aiken/bane.html.

[7] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Computer Science Division, University of California, Berkeley, December 1997. UCB/CSD-98-1017.

[8] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, Montreal, Canada, June 1998.

[9] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of The ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, San Jose, California, October 1996.

[10] Andrew D. Birrell. An introduction to programming with threads. Technical Report SRC-35, Systems Research Center, Digital Equipment Corporation, January 1989.

[11] Matthias Blume. Dependency analysis for Standard ML. *ACM Transactions on Programming Languages and Systems*, 21(4):813–846, July 1999.

[12] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):790–812, July 1999.

[13] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of The ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 35–46, Denver, Colorado, October 1999.

[14] Luca Cardelli, J. Donahue, Michael Jordan, B. Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 202–212, Austin, TX, January 1989.

[15] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report SRC-052, Systems Research Center, Digital Equipment Corporation, November 1989.

[16] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From sequential to multi-threaded Java: An event-based operational semantics. In Michael Johnson, editor, *The Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 75–90, Sydney, Australia, December 1997. Springer–Verlag.

[17] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer–Verlag, 1999.

[18] Byeong-Mo Chang, Jangwoo Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for Java. In *Proceedings of the 16th ACM Symposium on Applied Computing*, March 2001.

[19] Liling Chen, Luddy Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[21] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albequerque, New Mexico, January 1992.

[22] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Proceedings of the 7th International Conference on Computer-Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 293–308. Springer-Verlag, 1995.

[23] David MacQueen *et al.* Standard ML of New Jersey. `http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`.

[24] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, San Diego, CA, January 1998.

[25] Manuel Fähndrich and Alexander Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints*, Cambridge MA, August 1996.

[26] Manuel Fähndrich and Alxander Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 114–126, Paris, France, September 1997. Springer-Verlag.

[27] Manuel Fähndrich, Jeffrey S. Foster, Alexander Aiken, and Jason Cu. Tracking down exceptions in Standard ML programs. Technical Report UCB/CSD-98-

996, Computer Science Division, University of California, Berkeley, February 1998.

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

[29] Jeffrey S. Foster, Manuel Fähndrich, and Alxander Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report UCB/CSD-97-964, University of California at Berkeley, Computer Science Division, July 1997.

[30] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java™ Language Specification.* Addison Wesley, second edition edition, 2000.

[31] Mark Grand. *Java Language Reference.* O'Reilly & Associates, Inc., July 1997.

[32] David Grove, Greg DeFouw, Jeffrey, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of The ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, Atlanta, Georgia, October 1997. ACM Press.

[33] Laurent Guerby. Hypertext Ada 95 rationale. `http://www.adahome.com/LRM/95/Rationale/rat95html`, 1996.

[34] Juan Carlos Guzmán and Ascánder Suárez. A type system for exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 127–135, June 1994.

[35] A. Nico Habermann and Dewayne E. Perry. *Ada for Experienced Programmers.* Addison-Wesley, 1983.

[36] Nevin Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, October 1992. CMU-CS-92-201.

[37] Nevin Heintze. Set based analysis of ML programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, July 1993.

[38] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of set constraints. Technical Report CMU-CS-91-110, Carnegie-Mellon University, February 1991.

[39] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, Las Vegas, Nevada, June 1997.

[40] Joe De Russo III and Peter Haggar. Multithreaded exception handling in Java. Java Report, August 1998.

[41] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–205, Philadelphia, Pennsylvania, May 1996.

[42] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 1991.

[43] Kwangkeun Yi *et al.* nML programming language system. `http://ropas.kaist.ac.kr/n/`.

[44] Doug Lea. *Concurrent Programming in Java^{TM}, Second Edition: Design Principles and Patterns*. Addison-Wesley, 1999.

[45] Xavier Leroy. The threads library. `http://caml.inria.fr/ocaml/htmlman/manual068.html`.

[46] Xavier Leroy, Jérôme Vouillon, Damien Doligez, and et al. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/ocaml/`, 1996.

[47] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, March 2000.

[48] John M. Lucassen. *Types and Effects - Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT Laboratory for Computer Science, August 1987.

[49] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 45–57, San Diego, California, January 1988.

[50] Dianne M. Marsh. Multithreading in C++ and Java. `http://www.devx.com/upload/free/features/vcdj/2000/05may00/dm0500/dm0500.asp`, 2000.

[51] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[52] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[53] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[54] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345, New York, January 1997.

[55] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1992.

[56] Jens Palsberg and Michael I. Schwarzbach. Object-oriented type inference. In *Proceedings of The ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 141–161, Phoenix, Arizona, November 1991.

[57] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 276–290, San Antonio, TX, January 1999.

[58] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java programs. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 322–337, Toulouse, France, September 1999. Springer-Verlag.

[59] Martin P. Robillard and Gail C. Murphy. Designing robust Java programs with exceptions. In *Proceedings of the ACM SIGSOFT Eighth International*

Symposium on the Foundations of Software Engineering (FSE-8): Foundations of Software Engineering for Twenty-First Century Applications, pages 2–10, San Diego, California, November 2000. ACM Press.

[60] Sukyoung Ryu. Exception analysis for multithreaded Java programs. Technical Memorandum ROPAS-2001-11, Research On Program Analysis System, Korea Advanced Institute of Science and Technology, April 2001.

[61] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145.

[62] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[63] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, June 1997.

[64] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95, Boston, Massachusetts, January 2000. ACM Press.

[65] Jean-Piere Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[66] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *Proceedings of the Theoretical Aspect in Computer Science*, volume 789 of *Lecture Notes in Computer Science*, pages 224–243. Springer-Verlag, 1994.

[67] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of The ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, Minneapolis, Minnesota, October 2000.

[68] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, January 1990.

[69] Cristian Ungureanu and Suresh Jagannathan. Concurrency analysis for Java. In *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 413–432, Santa Barbara, CA, 2000. Springer-Verlag.

[70] Kwangkeun Yi. *Automatic Generation and Management of Program Analyses.* PhD thesis, University of Illinois at Urbana-Champaign, August 1993. Report UIUCDCS-R-93-1828.

[71] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Proceedings of the First International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 238–254, Namur, Belgium, September 1994. Springer-Verlag.

[72] Kwangkeun Yi and Byeong-Mo Chang. Exception analysis for Java. In A. Moreira and D. Demeyer, editors, *Object-Oriented Technology. ECOOP'99 Workshop Reader (Formal Techniques for Java Programs)*, volume 1743 of *Lecture Notes in Computer Science*, pages 111–112, Lisbon, Portugal, June 1999. Springer-Verlag.

[73] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation

and management of interprocedural program analyses. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, Charleston, South Carolina, January 1993.

[74] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113, Paris, France, September 1997. Springer-Verlag.

[75] Kwangkeun Yi and Sukyoung Ryu. SML/NJ Exception Analyzer 0.98. `http://compiler.kaist.ac.kr/pub/exna/exna-README.html`, December 1998.

[76] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. Extended version of [74] (to appear).

감사의 글

이 논문을 완성하기까지 많은 도움을 주신 분들께 감사의 마음을 전합니다.

언제나 변함없는 열정과 열심으로 본을 보여주신 지도교수님 이광근 교수님께 진심으로 감사드립니다. 짧지 않은 기간동안 끊임없는 관심과 격려를 아끼지 않으시고 항상 치열하게 고민하고 도전하는 연구자의 자세를 배우게 하여주셨습니다. 따뜻한 관심으로 많은 가르침을 주신 최광무 교수님께 감사드립니다. 그리고 바쁘신 중에도 논문을 심사해주신 신성용 교수님과 차성덕 교수님, 최창선 교수님께 감사드립니다.

가족보다 더 많은 시간을 같이 보낸 실험실 선배님들과 동기들, 후배들에게 감사의 마음을 전합니다. 언제나 든든하게 의지가 되어주신 준선 선배님께 감사드립니다. 저마다의 독특한 개성들이 모여서 조화롭게 지내온 실험실 생활에서 많은 도움을 얻고 배웠습니다. 활발한 토론과 함께 서로의 연구에 관심을 갖고 도움을 주고 받은 실험실 구성원들께 감사드립니다. 특히, 오랜 시간 함께 하면서 실험실 생활과 연구에서 많은 힘이 되어주었던 욱세에게 진심으로 고마움을 전합니다.

논문 연구를 하는 동안 한국고등교육재단의 정보통신 대학원 장학생과 한국학술진흥재단의 신진연구인력으로 선발되어 장학금을 지원받았습니다. 두 재단에 감사의 마음을 전합니다.

무엇과도 비교할 수 없는 사랑으로 보살펴주시는 부모님께 존경과 감사를 드립니다. 한없는 사랑과 신뢰로 모든 일에 동행해주는 남편과 모든 가족들께 고마운 마음을 전합니다. 너무나 많은 사랑을 주시는 최영순 자매님과 교회 어른들께 감사드립니다. 제 삶을 주관하시고 언제나 저와 함께 하시는 주님께 모든 영광을 돌립니다.

이 력 서

성　　　명 : 류 석 영

생 년 월 일 : 1973년 8월 28일

출　생　지 : 서울특별시

본　　　적 : 서울특별시

학 력

1991.3–1995.2　한국과학기술원 전산학과 (학사)

1995.3–1996.8　한국과학기술원 전산학과 (석사)

1996.9–2001.8　한국과학기술원 전자전산학과 전산학전공 (박사)