
ROPAS

Research On Program Analysis System
National Creative Research Initiative Center
Korea Advanced Institute of Science and Technology

ROPAS MEMO
1999-3
April 19, 2000

Automatic Test Data Generation for Exceptions in First-Order ML Programs

Sukyoung Ryu and Kwangkeun Yi
{puppy,kwang}@ropas.kaist.ac.kr

Abstract

We present a static analysis to automatically generate test data that raise exceptions in the input programs. Using the test data from our analysis, the programmer can check whether the raised exceptions are correctly handled with respect to the program's specification.

For a given program, starting from the initial constraint that a particular raise expression should be executed, our analysis derives necessary constraints for its input variable. The correctness of our analysis assures that any value that satisfies the derived constraints for the input variable will activate the designated raise expression.

In this paper, we formally present such an analysis for a first-order language with the ML-style exception handling constructs and algebraic data values, prove its correctness, and show a set of examples.

1 Introduction

Exception facilities in modern programming languages (e.g., ML[MTHM97], Modula-3[CDJ+89], Java[GJS96], and Ada[HP83]) can be problematic in assuring the software quality. First, 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. Uncaught exceptions are sometimes disastrous [Ar996]. Second, even though there is no escaping exception from the program, the dynamic nature of abnormal situations makes it difficult to assure that every raised exception will be correctly handled.

There already exist practical tools [YR97, YRon, PL99, YR98, YC99] to the first problem but, as far as we know, the second problem of assuring exception's correct handling is not yet addressed.

Our proposed solution to the second problem is a static analysis that automatically generates test data that will execute all the exception-raise expressions in the input program. By running the program with the generated test inputs, the programmer can execute all the exception-raise expressions in the program, monitor the flows of the raised exceptions and check whether the raised exceptions are to be correctly handled. The exception-raise expressions are the explicit `raise`-expressions in the input programs. Non-raise expressions (e.g. $e_1 + e_2$) that can raise some primitive exceptions (e.g. `Overflow`) are outside of our test coverage.

For example, let us consider the following program of substituting e for a variable x in an input expression, written in ML[1]:

[0]This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

[1]This example program is a core part of an instance of what Gérard Huet, in the mid-80's, called "sharing transducers."

e	::=	`0`		constant
	|	x		variable
	|	$(\lambda x.e_1)\ e_2$		immediate function application
	|	(`fix` $f\ \lambda x.e_1$) e_2	| $f\ e$	recursive function application
	|	$\kappa\ e$	| $\kappa^{-1}\ e$	data construction/deconstruction
	|	`fst` e	| `snd` e	projection
	|	$(e_1,\ e_2)$		tuple
	|	`case` $e_1\ \kappa\ e_2\ e_3$		conditional branch
	|	`raise E`		exception raise
	|	`handle` e_1 `E` e_2		exception handle

Figure 1: Abstract Syntax of L

```
fun subst (VAR x') = if x = x' then e else raise Same
  | subst (LAM (x', b')) = if x = x' then raise Same
                           else LAM (x', subst b')
  | subst (APP (e1, e2)) = APP (subst e1, subst e2)
```

Our analysis generates test data as follows: (1) `VAR` x' where $x' \neq x$ which will execute the first `raise` expression, (2) `LAM` $(x,\ e')$ with e' an arbitrary expression which will execute the second `raise` expression, or (3) `APP` $(e_1,\ e_2)$ where e_1 or e_2 has (1) or (2) as its subexpression.

This paper is organized as follows. Section 2 defines the syntax and semantics of our source language. Section 3 presents our analysis and Section 4 proves the soundness of the analysis. Section 5 provides a set of explanatory examples. Section 6 discusses some issues and concludes.

2 Language L

The source language L is a call-by-value, first-order language with algebraic data values and a subset of the ML-style exception-handling constructs:

- Data values are algebraic: values are finitely constructed with a finite number of data constructors. For example, a natural number is constructed with either a constructor `ZERO` with a constant `0` as its argument or a constructor `SUC` with a natural number as its argument.
- Expressions' values are first-order. Thus, functions can neither be passed to other functions, be stored in a location, nor be paired with exceptions. This limitation implies that for every call expression it is manifest from the program text which function to call.
- Exceptions do not have arguments. Exceptions are constructed with a 0-ary constructor, the exception name.
- Every raise expression has an exception name at its exception part, like "`raise ERROR`."

Language L's abstract syntax is shown in Figure 1. Note that every function application expression has the function immediate in the program text. For brevity, we have omitted strings, numbers, primitive operators, and memory operations (assignment, reference, and dereference).

A datum value is constructed by "$\kappa\ e$" where κ is a data constructor name and expression e is for its argument value. The argument of datum is recovered by "$\kappa^{-1}\ e$". A tuple value is constructed by "(e_1, e_2)". The first argument of a tuple is recovered by "`fst` e" and the second argument is recovered by "`snd` e". Recursive function f is defined as "`fix` $f\ \lambda x.e$". The case expression "`case` $e_1\ \kappa\ e_2\ e_3$" branches to e_2 if the value of e_1 is constructed with κ, otherwise, to e_3. Exception `E` is raised by "`raise E`". The handle expression "`handle` e_1 `E` e_2" evaluates e_1 first. If e_1's result is a normal value, the value is returned; if e_1's result is a raised exception `E`, e_2 is evaluated; otherwise, the raised exception is uncaught and propagated backward along the dynamic evaluation chain. Multiple exceptions can be handled by nested handle expressions. For example, `handle` (`handle` e_1 `E` e_2) `F` e_3 can handle two exceptions `E` and `F`.

For brevity, we have omitted the formal definition of the semantics. The semantics is defined in the natural semantics formalism [Kah88].

Throughout this paper,

- we assume that all variables are uniquely named (alpha-converted).
- we assume that no expression of a program is a dead code.
- by "L program" we mean an L expression *with* free (input) variables.

3 Analysis by Constraint Propagation

We shall now introduce a static analysis that generates test data to raise a designated exception from the input program. Our analysis is presented in the set-constraint framework [Hei92, AH95].

We construct a relation $se \rhd e : \mathcal{C}$ for each expression e of the input program. Informally, se is the set of values where e's value should be included, and $\mathcal{C}$ is the set of constraints that are necessary for e to evaluate into the values of se. Thus we can read "$se \rhd e : \mathcal{C}$" as "$\mathcal{C}$ is necessary for e to be included in se". Syntax and semantics of the set expressions and constraints are shown in Figure 2.

An *interpretation* $\mathcal{I}$ is a mapping from set expressions se (respectively set constraints $\mathcal{C}$) to sets of values or exception packets[2] (respectively boolean values). An interpretation $\mathcal{I}$ is a *model* (a solution) of a conjunction $\mathcal{C}$ of constraints if, for each constraint $\mathcal{X} \subseteq se$ and $a \subseteq \mathcal{X}$ in $\mathcal{C}$, $\mathcal{I}(se)$ and $\mathcal{I}(a)$ are defined and $\mathcal{I}(\mathcal{X}) \subseteq \mathcal{I}(se)$ and $\mathcal{I}(a) \subseteq \mathcal{I}(\mathcal{X})$. We write $gm(\mathcal{C})$ for the greatest model of $\mathcal{C}$.

$\mathcal{C}$ is a set of set constraints. The collection of set constraints means the conjunction of the constraints. The meaning of se is a set of values or exception packets. For example, $\kappa\ \mathcal{X}$ indicates the values constructed with constructor κ and its argument values in $\mathcal{X}$:

$$\mathcal{I}(\kappa\ \mathcal{X}) = \{\kappa\ v \mid v \in \mathcal{I}(\mathcal{X})\}$$

and $\kappa^{-1}\ \mathcal{X}$ indicates the values deconstructed with κ from values in $\mathcal{X}$:

$$\mathcal{I}(\kappa^{-1}\ \mathcal{X}) = \{v \mid \kappa\ v \in \mathcal{I}(\mathcal{X})\}.$$

3.1 Constructing Set Constraints

We present the analysis rules in Figure 3 and Figure 4.

[2] A raised exception is particularly called an exception packet.

$$
\begin{array}{rcll}
v & \in & \mathit{Val} = \{\mathtt{0}\} + \mathit{FtnExpr} + \mathit{Data} + \mathit{Val} \times \mathit{Val} & \text{values} \\
\lambda x.e & \in & \mathit{FtnExpr} & \text{function expressions in a program} \\
\kappa\, v & \in & \mathit{Data} = \mathit{Con} \times \mathit{Val} & \text{data} \\
\kappa,\ \kappa' & \in & \mathit{Con} & \text{data constructors in a program} \\
\mathtt{E},\ \mathtt{E}' & \in & \mathit{Exn} & \text{exception constructors in a program} \\
\hat{\mathtt{E}},\ \hat{\mathtt{E}}' & \in & \mathit{Packet} & \text{exception packets}
\end{array}
$$

$$
\begin{array}{rcll}
\mathcal{C} & ::= & \{\mathtt{true}\} & \\
 & | & \{\mathtt{false}\} & \\
 & | & \{\mathcal{X} \subseteq se\} & \\
 & | & \{a \subseteq \mathcal{X}\} & \\
 & | & \mathcal{C}_1 \cup \mathcal{C}_2 &
\end{array}
\qquad
\begin{array}{rcll}
a & ::= & \mathtt{0} & \\
 & | & \lambda x.e & \\
 & | & \kappa\ a & \\
 & | & (a_1,\ a_2) &
\end{array}
$$

$$
\begin{array}{rcll}
se & ::= & \mathcal{X} & \text{set variable} \\
 & | & \kappa\ \mathcal{X} & \text{constructed set with } \kappa \\
 & | & \kappa^{-1}\ \mathcal{X} & \text{deconstructed set with } \kappa \\
 & | & \bar{\kappa}\ \mathcal{X} & \text{constructed set with non } \kappa \\
 & | & (se, \mathcal{X}) & \text{set of tuples} \\
 & | & (\mathcal{X}, se) & \text{set of tuples} \\
 & | & (\mathcal{X}, \mathcal{Y}) & \text{set of tuples} \\
 & | & a & \text{atomic set} \\
 & | & \hat{\mathtt{E}} & \text{exception packet}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{I}(\mathcal{C}) & \in & \{\mathit{true},\ \mathit{false}\} \\
\mathcal{I}(\{\mathtt{true}\}) & = & \mathit{true} \\
\mathcal{I}(\{\mathtt{false}\}) & = & \mathit{false} \\
\mathcal{I}(\{\mathcal{X} \subseteq se\}) & = & \mathcal{I}(\mathcal{X}) \subseteq \mathcal{I}(se) \\
\mathcal{I}(\{a \subseteq \mathcal{X}\}) & = & \mathcal{I}(a) \subseteq \mathcal{I}(\mathcal{X}) \\
\mathcal{I}(\mathcal{C}_1 \cup \mathcal{C}_2) & = & \mathcal{I}(\mathcal{C}_1) \wedge \mathcal{I}(\mathcal{C}_2)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{I}(se) & \subseteq & \mathit{Val} + \mathit{Packet} \\
\mathcal{I}(\mathcal{X}) & \subseteq & \mathit{Val} \\
\mathcal{I}(\kappa\ \mathcal{X}) & = & \{\kappa\, v \mid v \in \mathcal{I}(\mathcal{X})\} \\
\mathcal{I}(\kappa^{-1}\ \mathcal{X}) & = & \{v \mid \kappa\, v \in \mathcal{I}(\mathcal{X})\} \\
\mathcal{I}(\bar{\kappa}\ \mathcal{X}) & = & \{\kappa'\, v \mid v \in \mathcal{I}(\mathcal{X}),\ \kappa' \neq \kappa\} \\
\mathcal{I}((se,\ \mathcal{X})) & = & \{(v_1,\ v_2) \mid v_1 \in \mathcal{I}(se),\ v_2 \in \mathcal{I}(\mathcal{X})\} \\
\mathcal{I}((\mathcal{X},\ se)) & = & \{(v_1,\ v_2) \mid v_1 \in \mathcal{I}(\mathcal{X}),\ v_2 \in \mathcal{I}(se)\} \\
\mathcal{I}((\mathcal{X},\ \mathcal{Y})) & = & \{(v_1,\ v_2) \mid v_1 \in \mathcal{I}(\mathcal{X}),\ v_2 \in \mathcal{I}(\mathcal{Y})\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{I}(a) & \subseteq & \mathit{Val} \\
\mathcal{I}(\mathtt{0}) & = & \{\mathtt{0}\} \\
\mathcal{I}(\lambda x.e) & = & \{\lambda x.e\} \\
\mathcal{I}(\kappa\ a) & = & \{\kappa\, v \mid v \in \mathcal{I}(a)\} \\
\mathcal{I}((a_1,\ a_2)) & = & \{(v_1,\ v_2) \mid v_1 \in \mathcal{I}(a_1),\ v_2 \in \mathcal{I}(a_2)\}
\end{array}
$$

Figure 2: Set Constraints

Let's consider the four rules for `0` in Figure 3. `0` is always included in the set $\{\mathtt{0}\}$, hence:

$$\texttt{[C-1a]} \qquad \mathtt{0} \ \triangleright\ \mathtt{0} : \{\mathtt{true}\}.$$

In order for `0` to be in $\mathcal{X}$, the necessary constraint is obviously $\{\mathtt{0} \subseteq \mathcal{X}\}$:

$$\texttt{[C-2a]} \qquad \mathcal{X} \ \triangleright\ \mathtt{0} : \{\mathtt{0} \subseteq \mathcal{X}\}.$$

In order for `0` to be included in $\kappa^{-1}\ \mathcal{X}$, $\mathcal{X}$ has to include κ `0`:

$$\texttt{[C-3a]} \qquad \kappa^{-1}\ \mathcal{X} \ \triangleright\ \mathtt{0} : \{\kappa\ \mathtt{0} \subseteq \mathcal{X}\}.$$

There's no way for `0` to be included in the other kinds of sets:

$$\texttt{[C-4a]} \qquad se \ \triangleright\ \mathtt{0} : \{\mathtt{false}\} \quad \text{where } se \notin \{\mathtt{0},\ \mathcal{X},\ \kappa^{-1}\ \mathcal{X}\}.$$

$$\texttt{[C-1a]} \quad \texttt{0} \rhd \texttt{0} : \{\texttt{true}\}$$
$$\texttt{[C-2a]} \quad \mathcal{X} \rhd \texttt{0} : \{\texttt{0} \subseteq \mathcal{X}\}$$
$$\texttt{[C-3a]} \quad \kappa^{-1}\ \mathcal{X} \rhd \texttt{0} : \{\kappa\ \texttt{0} \subseteq \mathcal{X}\}$$
$$\texttt{[C-4a]} \quad se \rhd \texttt{0} : \{\texttt{false}\} \qquad se \notin \{\texttt{0},\ \mathcal{X},\ \kappa^{-1}\ \mathcal{X}\}$$

$$\texttt{[NVN-a]} \quad se \rhd x : \{\mathcal{X} \subseteq se\} \qquad se \notin \{\hat{\texttt{E}}\}$$
$$\texttt{[NVX-a]} \quad \hat{\texttt{E}} \rhd x : \{\texttt{false}\}$$

$$\texttt{[RSN-a]} \quad se \rhd \texttt{raise E} : \{\texttt{false}\} \qquad se \notin \{\hat{\texttt{E}}\}$$
$$\texttt{[RSX-1a]} \quad \hat{\texttt{E}} \rhd \texttt{raise E} : \{\texttt{true}\}$$
$$\texttt{[RSX-2a]} \quad \hat{\texttt{E}'} \rhd \texttt{raise E} : \{\texttt{false}\} \qquad \texttt{E}' \neq \texttt{E}$$

$$\texttt{[CONN-1a]} \quad \frac{\kappa^{-1}\ \mathcal{X} \rhd e : \mathcal{C}}{\mathcal{X} \rhd \kappa\ e : \mathcal{C}} \qquad \texttt{[CONN-2a]} \quad \frac{\mathcal{X} \rhd e : \mathcal{C}}{\kappa\ \mathcal{X} \rhd \kappa\ e : \mathcal{C}}$$

$$\texttt{[CONN-3a]} \quad \frac{\kappa^{-1}\ \mathcal{Y} \rhd e : \mathcal{C}}{\kappa'^{-1}\ \mathcal{X} \rhd \kappa\ e : \mathcal{C} \cup \{\mathcal{Y} \subseteq \kappa'^{-1}\ \mathcal{X}\}} \quad \text{fresh } \mathcal{Y}$$

$$\texttt{[CONN-4a]} \quad \frac{\mathcal{X} \rhd e : \mathcal{C}}{\overline{\kappa'}\ \mathcal{X} \rhd \kappa\ e : \mathcal{C}} \quad \kappa' \neq \kappa$$

$$\texttt{[CONN-5a]} \quad \frac{a \rhd e : \mathcal{C}}{\kappa\ a \rhd \kappa\ e : \mathcal{C}} \qquad \texttt{[CONX-a]} \quad \frac{\hat{\texttt{E}} \rhd e : \mathcal{C}}{\hat{\texttt{E}} \rhd \kappa\ e : \mathcal{C}}$$

$$\texttt{[CONN-6a]} \quad se \rhd \kappa\ e : \{\texttt{false}\} \quad se \notin \{\mathcal{X},\ \kappa\ \mathcal{X},\ \kappa'^{-1}\ \mathcal{X},\ \overline{\kappa'}\ \mathcal{X} (\kappa' \neq \kappa),\ \kappa\ a,\ \hat{\texttt{E}}\}$$

$$\texttt{[DCONN-1a]} \quad \frac{\kappa\ se \rhd e : \mathcal{C}}{se \rhd \kappa^{-1}\ e : \mathcal{C}} \quad se \in \{\mathcal{X},\ a\} \qquad \texttt{[DCONX-a]} \quad \frac{\hat{\texttt{E}} \rhd e : \mathcal{C}}{\hat{\texttt{E}} \rhd \kappa^{-1}\ e : \mathcal{C}}$$

$$\texttt{[DCONN-2a]} \quad \frac{\kappa\ \mathcal{Y} \rhd e : \mathcal{C}}{se \rhd \kappa^{-1}\ e : \mathcal{C} \cup \{\mathcal{Y} \subseteq se\}} \quad \text{fresh } \mathcal{Y},\ se \notin \{\mathcal{X},\ a,\ \hat{\texttt{E}}\}$$

$$\texttt{[FSTN-a]} \quad \frac{(se,\ \mathcal{X}) \rhd e : \mathcal{C}}{se \rhd \texttt{fst}\ e : \mathcal{C}} \quad \begin{array}{l}\text{fresh } \mathcal{X},\\ se \notin \{\hat{\texttt{E}}\}\end{array} \qquad \texttt{[FSTX-a]} \quad \frac{\hat{\texttt{E}} \rhd e : \mathcal{C}}{\hat{\texttt{E}} \rhd \texttt{fst}\ e : \mathcal{C}}$$

$$\texttt{[SNDN-a]} \quad \frac{(\mathcal{X},\ se) \rhd e : \mathcal{C}}{se \rhd \texttt{snd}\ e : \mathcal{C}} \quad \begin{array}{l}\text{fresh } \mathcal{X},\\ se \notin \{\hat{\texttt{E}}\}\end{array} \qquad \texttt{[SNDX-a]} \quad \frac{\hat{\texttt{E}} \rhd e : \mathcal{C}}{\hat{\texttt{E}} \rhd \texttt{snd}\ e : \mathcal{C}}$$

$$\texttt{[TUPN-1a]} \quad \frac{\mathcal{Y} \rhd e_1 : \mathcal{C}_1 \quad \mathcal{Z} \rhd e_2 : \mathcal{C}_2}{\mathcal{X} \rhd (e_1,\ e_2) : \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{X} \subseteq (\mathcal{Y},\ \mathcal{Z})\}} \quad \text{fresh } \mathcal{Y},\ \mathcal{Z}$$

$$\texttt{[TUPN-2a]} \quad \frac{\mathcal{Y} \rhd e_1 : \mathcal{C}_1 \quad \mathcal{Z} \rhd e_2 : \mathcal{C}_2}{\kappa^{-1}\ \mathcal{X} \rhd (e_1,\ e_2) : \mathcal{C}_1 \cup \mathcal{C}_2 \quad \cup\{\mathcal{X} \subseteq \kappa\ \mathcal{W}\} \quad \cup\{\mathcal{W} \subseteq (\mathcal{Y},\ \mathcal{Z})\}} \quad \text{fresh } \mathcal{W},\ \mathcal{Y},\ \mathcal{Z}$$

$$\texttt{[TUPN-3a]} \quad se \rhd (e_1,\ e_2) : \{\texttt{false}\} \quad se \in \{\kappa\ \mathcal{X},\ \bar{\kappa}\ \mathcal{X}\}$$

Figure 3: Analysis Rules (Part I)

[TUPN-4a] $$\frac{se \rhd e_1 : \mathcal{C}_1 \quad \mathcal{X} \rhd e_2 : \mathcal{C}_2}{(se, \mathcal{X}) \rhd (e_1, e_2) : \mathcal{C}_1 \cup \mathcal{C}_2}$$

[TUPN-5a] $$\frac{\mathcal{X} \rhd e_1 : \mathcal{C}_1 \quad se \rhd e_2 : \mathcal{C}_2}{(\mathcal{X}, se) \rhd (e_1, e_2) : \mathcal{C}_1 \cup \mathcal{C}_2}$$

[TUPN-6a] $$\frac{\mathcal{X} \rhd e_1 : \mathcal{C}_1 \quad \mathcal{Y} \rhd e_2 : \mathcal{C}_2}{(\mathcal{X}, \mathcal{Y}) \rhd (e_1, e_2) : \mathcal{C}_1 \cup \mathcal{C}_2}$$

[TUPN-7a] $$\frac{a_1 \rhd e_1 : \mathcal{C}_1 \quad a_2 \rhd e_2 : \mathcal{C}_2}{(a_1, a_2) \rhd (e_1, e_2) : \mathcal{C}_1 \cup \mathcal{C}_2}$$

[TUPN-8a] $$a \rhd (e_1,\ e_2) : \{\texttt{false}\} \quad a \notin \{(a_1,\ a_2)\}$$

[TUPX-1a] $$\frac{\hat{\mathtt{E}} \rhd e_1 : \mathcal{C}}{\hat{\mathtt{E}} \rhd (e_1,\ e_2) : \mathcal{C}}$$

[TUPX-2a] $$\frac{\mathcal{X} \rhd e_1 : \mathcal{C}_1 \quad \hat{\mathtt{E}} \rhd e_2 : \mathcal{C}_2}{\hat{\mathtt{E}} \rhd (e_1,\ e_2) : \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{fresh } \mathcal{X}$$

[APPX-1a] $$\frac{\hat{\mathtt{E}} \rhd e_2 : \mathcal{C}}{\hat{\mathtt{E}} \rhd (\lambda x.e_1)\ e_2 : \mathcal{C}}$$

[APPX-2a] $$\frac{\hat{\mathtt{E}} \rhd e_2 : \mathcal{C}}{\hat{\mathtt{E}} \rhd (\texttt{fix}\ f\ \lambda x.e_1)\ e_2 : \mathcal{C} \cup \{\mathcal{F} \subseteq \lambda x.e_1\}}$$

[APPX-3a] $$\frac{\hat{\mathtt{E}} \rhd e_2 : \mathcal{C}}{\hat{\mathtt{E}} \rhd f\ e_2 : \mathcal{C}} \quad \texttt{fix}\ f\ \lambda x.e_1 \text{ in a program}$$

[APP-1a] $$\frac{\mathcal{X}_n \rhd e_2 : \mathcal{C}_1 \quad se \rhd [x_n/x]e_1 : \mathcal{C}_2}{se \rhd (\lambda x.e_1)\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{fresh } n$$

[APP-2a] $$\frac{\mathcal{X}_n \rhd e_2 : \mathcal{C}_1 \quad se \rhd [x_n/x]e_1 : \mathcal{C}_2}{se \rhd (\texttt{fix}\ f\ \lambda x.e_1)\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{F} \subseteq \lambda x.e_1\}} \quad \text{fresh } n$$

[APP-3a] $$\frac{\mathcal{X}_n \rhd e_2 : \mathcal{C}_1 \quad se \rhd [x_n/x]e_1 : \mathcal{C}_2}{se \rhd f\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2} \quad \texttt{fix}\ f\ \lambda x.e_1 \text{ in a program, fresh } n,\ \mathit{Used}_f() \leq \mathcal{N}$$

[CASEX-a] $$\frac{\hat{\mathtt{E}} \rhd e_1 : \mathcal{C}}{\hat{\mathtt{E}} \rhd \texttt{case}\ e_1\ \kappa\ e_2\ e_3 : \mathcal{C}}$$

[CASE-1a] $$\frac{\kappa\ \mathcal{X} \rhd e_1 : \mathcal{C}_1 \qquad se \rhd e_2 : \mathcal{C}_2}{se \rhd \texttt{case}\ e_1\ \kappa\ e_2\ e_3 : \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{fresh } \mathcal{X}$$

[CASE-2a] $$\frac{\bar{\kappa}\ \mathcal{X} \rhd e_1 : \mathcal{C}_1 \qquad se \rhd e_3 : \mathcal{C}_2}{se \rhd \texttt{case}\ e_1\ \kappa\ e_2\ e_3 : \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{fresh } \mathcal{X}$$

[HNDLN-a] $$\frac{se \rhd e_1 : \mathcal{C}}{se \rhd \texttt{handle}\ e_1\ \mathtt{E}\ e_2 : \mathcal{C}} \quad se \notin \{\hat{\mathtt{E}}\}$$

[HNDLX-a] $$\frac{\hat{\mathtt{E}'} \rhd e_1 : \mathcal{C}}{\hat{\mathtt{E}'} \rhd \texttt{handle}\ e_1\ \mathtt{E}\ e_2 : \mathcal{C}} \quad \mathtt{E}' \neq \mathtt{E}$$

[HNDL-a] $$\frac{\hat{\mathtt{E}} \rhd e_1 : \mathcal{C}_1 \qquad se \rhd e_2 : \mathcal{C}_2}{se \rhd \texttt{handle}\ e_1\ \mathtt{E}\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2}$$

Figure 4: Analysis Rules (Part II)

Rule `[CONN-1a]` dictates that because the value of $\kappa\ e$ should be included in $\mathcal{X}$ the value of e should be included in $\kappa^{-1}\ \mathcal{X}$:

$$\texttt{[CONN-1a]} \qquad \frac{\kappa^{-1}\ \mathcal{X} \rhd e\ :\ \mathcal{C}}{\mathcal{X} \rhd \kappa\ e\ :\ \mathcal{C}}.$$

Rules `[CONN-3a]`, `[DCONN-2a]`, `[TUPN-1a]`, and `[TUPN-2a]` in Figure 3 introduce new set variables. For example, consider:

$$\texttt{[CONN-3a]} \qquad \frac{\kappa^{-1}\ \mathcal{Y} \rhd e\ :\ \mathcal{C}}{\kappa'^{-1}\ \mathcal{X} \rhd \kappa\ e\ :\ \mathcal{C} \cup \{\mathcal{Y} \subseteq \kappa'^{-1}\ \mathcal{X}\}} \quad \text{fresh } \mathcal{Y}.$$

In order for $\kappa\ e$ to be included in $\kappa'^{-1}\ \mathcal{X}$, the value of e has to be included in $\kappa^{-1}(\kappa'^{-1}\ \mathcal{X})$. Because our set expression syntax does not allow κ^{-1} in front of $\kappa'^{-1}\ \mathcal{X}$, we replace $\kappa'^{-1}\ \mathcal{X}$ by a fresh variable $\mathcal{Y}$ and add a new constraint $\mathcal{Y} \subseteq \kappa'^{-1}\ \mathcal{X}$.

Unlike those rules in Figure 3, some rules in Figure 4 are alternatives, in the sense that for a given se and e, constraints $\mathcal{C}$ for $se \rhd e : \mathcal{C}$ can vary.

Consider rules `[CASEX-a]`, `[CASE-1a]`, and `[CASE-2a]` in Figure 4. If a `case` expression raises an exception, there are three possibilities depending on where the exception is raised from: e_1, e_2, and e_3. Rule `[CASEX-a]` is for a raised exception from e_1:

$$\texttt{[CASEX-a]} \qquad \frac{\hat{\texttt{E}} \rhd e_1\ :\ \mathcal{C}}{\hat{\texttt{E}} \rhd \texttt{case}\ e_1\ \kappa\ e_2\ e_3\ :\ \mathcal{C}}.$$

It dictates that if $\mathcal{C}$ is necessary for e_1 to raise the exception `E`, then $\mathcal{C}$ is necessary for the case expression `case` $e_1\ \kappa\ e_2\ e_3$ to raise the exception. Rule `[CASE-1a]` is for a raised exception from e_2:

$$\texttt{[CASE-1a]} \qquad \frac{\kappa\ \mathcal{X} \rhd e_1\ :\ \mathcal{C}_1 \qquad \hat{\texttt{E}} \rhd e_2\ :\ \mathcal{C}_2}{\hat{\texttt{E}} \rhd \texttt{case}\ e_1\ \kappa\ e_2\ e_3\ :\ \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{fresh } \mathcal{X}.$$

if $\mathcal{C}_1$ is necessary for e_1 to be in $\kappa\ \mathcal{X}$ for some $\mathcal{X}$ and $\mathcal{C}_2$ is necessary for e_2 to raise the exception `E`, then $\mathcal{C}_1 \cup \mathcal{C}_2$ is necessary for the case expression `case` $e_1\ \kappa\ e_2\ e_3$ to raise the exception. And rule `[CASE-2a]` is for a raised exception from e_3:

$$\texttt{[CASE-2a]} \qquad \frac{\bar{\kappa}\ \mathcal{X} \rhd e_1\ :\ \mathcal{C}_1 \qquad \hat{\texttt{E}} \rhd e_3\ :\ \mathcal{C}_2}{\hat{\texttt{E}} \rhd \texttt{case}\ e_1\ \kappa\ e_2\ e_3\ :\ \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{fresh } \mathcal{X}.$$

It dictates that if $\mathcal{C}_1$ is necessary for e_1 to be in $\bar{\kappa}\ \mathcal{X}$ for some $\mathcal{X}$ and $\mathcal{C}_2$ is necessary for e_3 to raise the exception `E`, then $\mathcal{C}_1 \cup \mathcal{C}_2$ is necessary for the case expression `case` $e_1\ \kappa\ e_2\ e_3$ to raise the exception.

Rules `[APPX-2a]` and `[APP-2a]` in Figure 4 make a constraint for the function variable f. For example, consider:

$$\texttt{[APP-2a]} \qquad \frac{\mathcal{X}_n \rhd e_2\ :\ \mathcal{C}_1 \qquad se \rhd [x_n/x]e_1\ :\ \mathcal{C}_2}{se \rhd (\texttt{fix}\ f\ \lambda x.e_1)\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{F} \subseteq \lambda x.e_1\}} \quad \text{fresh } n.$$

The rule forces the value of $\mathcal{F}$ to be included in $\{\lambda x.e_1\}$. These two rules are the only rules that make a constraint for each function variable, so every $\mathcal{F}$ represents a singleton set $\{\lambda x.e_1\}$. In order to uniquely name all the variables in the derivation tree, we replace all occurrences of the argument variable x in the function body e_1 by a fresh variable x_n: $[x_n/x]e_1$.

Rule `[APP-3a]` in Figure 4 for recursive calls cannot be used an indefinite number of times. The counter $Used_f()$ for each recursive function f records the number of times that the rule

for recursive calls is used. If the counter hits a finite, pre-fixed, limit $\mathcal{N}$ the rule cannot be applied. For example, consider:

$$\texttt{[APP-3a]} \quad \frac{\mathcal{X}_n \rhd e_2 : \mathcal{C}_1 \quad se \rhd [x_n/x]e_1 : \mathcal{C}_2}{se \rhd f\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2} \qquad \begin{array}{c}\texttt{fix}\ f\ \lambda x.e_1 \text{ in a program,} \\ \text{fresh } n,\ \mathit{Used}_f() \leq \mathcal{N}\end{array} .$$

That the side condition $\mathit{Used}_f() \leq \mathcal{N}$ is imposed only for recursive call expressions (rule [APP-3a]) implies that we bound the number of recursive call tracings while we propagate the constraints. This arbitrary bound for recursive call may fail to generate some constraints that would otherwise allow non-empty test data. Notice that the fixed bound for recursive derivations will not violate the safety of our analysis because failing to find test data vacuously satisfies the soundness condition: every generated test datum will cause the program to execute the designated raise expression.

Rules [FSTN-a], [SNDN-a], [TUPX-2a], [CASE-1a], and [CASE-2a] introduce new set variables in order to collect constraints for subexpressions. For example, consider:

$$\texttt{[TUPX-2a]} \qquad \frac{\mathcal{X} \rhd e_1 : \mathcal{C}_1 \quad \hat{\mathtt{E}} \rhd e_2 : \mathcal{C}_2}{\hat{\mathtt{E}} \rhd (e_1,\ e_2) : \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{fresh } \mathcal{X}$$

If $\mathcal{C}_1$ is necessary for e_1 to be in $\mathcal{X}$ for some $\mathcal{X}$ and $\mathcal{C}_2$ is necessary for e_2 to raise the exception E, then $\mathcal{C}_1 \cup \mathcal{C}_2$ is necessary for the tuple expression $(e_1,\ e_2)$ to raise the exception E.

3.2 Solving Set Constraints

Several set-constraint solving algorithms[HJ90, HJ91, CP97, CP98] are reported for a number of subclasses of set constraints. Our set constraints are *co-definite*, as defined by Charatonik and Podelski [CP97, CP98], hence, whenever satisfiable, have the *greatest* solution.

Computing the constraints' greatest solution uses the conventional fixpoint iteration. For a set of constraints $\mathcal{C}$ which is an analysis result of a given program, we initialize each set variable $\mathcal{X}$ in $\mathcal{C}$ to be the universe set. We get the constraints' greatest solution $gm(\mathcal{C})$ by repeatedly computing the values of set variables according to the set constraints in $\mathcal{C}$.

4 Soundness

We prove that if we evaluate the program with the test data which is the $\mathcal{C}$'s model of our analysis $\hat{\mathtt{E}} \rhd e : \mathcal{C}$ (where $\mathcal{C} \neq \{\texttt{false}\}$), then the program raises the exception E. Let us start with some definitions.

Definition 1 ($[\![\hat{\mathtt{E}} \rhd e : \mathcal{C}\,]\!]$) *For a program e, we write* $[\![\hat{\mathtt{E}} \rhd e : \mathcal{C}\,]\!]$ *for the derivation tree of the analysis* $\hat{\mathtt{E}} \rhd e : \mathcal{C}$.

Definition 2 ($[\![\sigma \vdash e \to o\,]\!]$) *For a program e, we write* $[\![\sigma \vdash e \to o\,]\!]$ *for the derivation tree of the evaluation* $\sigma \vdash e \to o$.

We prove the soundness of our analysis by simulating the evaluation of the program under a fixed environment σ^v which is induced by a test data v from our analysis:

Definition 3 (σ^v) *For a program* e_0 *with a free variable* x_0, *let* $[x_0 \mapsto v] \vdash e_0 \to o$. *We write* σ^v *for the environment satisfying the followings:*
For every σ *occurring in* $[\![[x_0 \mapsto v] \vdash e_0 \to o\,]\!]$,
(1) every $[x \mapsto v']$ *in* σ *is collected, and*
(2) for each $[f \mapsto \langle \sigma', \texttt{fix}\ f\ \lambda x.e\rangle]$ *in* σ, $[f \mapsto \langle \sigma^v, \texttt{fix}\ f\ \lambda x.e\rangle]$ *is collected.*

Fixed environment σ^v preserves the semantics of e_0:

Lemma 1 *For a program e_0 with a free variable x_0, let $[x_0 \mapsto v] \vdash e_0 \to o$. Then, σ^v does not change during the evaluation $[\![\sigma^v \vdash e_0 \to o\,]\!]$.*

Proof. During the evaluation of a program, environments can change only in function applications. Because x_0 is the only one free variable of e_0 and $[x_0 \mapsto v](x_0) = \sigma^v(x_0)$, $[\![[x_0 \mapsto v] \vdash e_0 \to o\,]\!]$ and $[\![\sigma^v \vdash e_0 \to o\,]\!]$is identical modulo environments. Thus, for each $\sigma \vdash e \to o'$ in $[\![\sigma^v \vdash e_0 \to o\,]\!]$, $\sigma' \vdash e \to o'$ exists in $[\![[x_0 \mapsto v] \vdash e_0 \to o\,]\!]$, $dom(\sigma') \subseteq dom(\sigma^v)$ by the definition of σ^v, and $\sigma'(x) = \sigma^v(x)$ for every variable $x \in dom(\sigma')$. By using these facts, the proof shows that the changed environment in each function application is the same as σ^v. That is, we prove that all value bindings introduced in each function application are already included in σ^v. □

We will use two inductions for $se \rhd e : \mathcal{C}$ occurring in $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$: one on the evaluation order of $se \rhd e : \mathcal{C}$ and the other on the number of relations $se_i \rhd e_i : \mathcal{C}_i$ in $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$ that contribute to the set expression se. Since se of $se \rhd e : \mathcal{C}$ is determined by $se_i \rhd e_i : \mathcal{C}_i$ in the path from $\hat{\mathrm{E}} \rhd e_0 : \mathcal{C}_0$ to $se \rhd e : \mathcal{C}$ in the analysis $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$, the number of relations that contribute to se is n of $e_0 \overset{n}{\sim} e$:

Definition 4 **($e_0 \overset{n}{\sim} e$)** *For a program e_0, let $\hat{\mathrm{E}} \rhd e_0 : \mathcal{C}_0$. We write $e_0 \overset{n}{\sim} e$ for the path of length n from the root $\hat{\mathrm{E}} \rhd e_0 : \mathcal{C}_0$ to any node $se \rhd e : \mathcal{C}$ in the analysis $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$.*

$$\textit{(1)}\ e_0 \overset{0}{\sim} e_0 \qquad \textit{(2)}\ \frac{e_0 \overset{n-1}{\sim} e' \qquad \frac{\cdots se \rhd e : \mathcal{C} \cdots}{se' \rhd e' : \mathcal{C}'} \in [\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]}{e_0 \overset{n}{\sim} e}$$

For a program e_0, let $\sigma \vdash e_0 \to o$. We similarly define $e_0 \overset{n}{\sim} e$ for the path of length n from the root $\sigma \vdash e_0 \to o$ to any node $\sigma' \vdash e \to o'$ in the evaluation $[\![\sigma \vdash e_0 \to o\,]\!]$.

Every relation $se \rhd e : \mathcal{C}$ in $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$ is correct, i.e., "$\mathcal{C}$ is necessary for e's value to be included in se":

Lemma 2 *For a program e_0 with a free variable x_0, let $\hat{\mathrm{E}} \rhd e_0 : \mathcal{C}_0$. Then, every relation $se \rhd e : \mathcal{C}$ occurring in $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$ satisfies the followings:*
For every $v \in gm(\mathcal{C}_0)(\mathcal{X}_0)$,
(1) $\sigma^v \vdash e \to o'$ exists in $[\![\sigma^v \vdash e_0 \to o\,]\!]$, and (2) $o' \in gm(\mathcal{C}_0)(se)$.

Proof. We prove by induction on the evaluation order of $se \rhd e : \mathcal{C}$ occurring in $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$. The proof consists of two main parts.

(1) We have to show that $\sigma^v \vdash e \to o'$ exists in $[\![\sigma^v \vdash e_0 \to o\,]\!]$. Because $se \rhd e : \mathcal{C}$ occurs in $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$, $e_0 \overset{n}{\sim} e$ exists in $[\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$. By showing that the same $e_0 \overset{n}{\sim} e$ exists in $[\![\sigma^v \vdash e_0 \to o\,]\!]$, we prove that $\sigma^v \vdash e \to o'$ exists in $[\![\sigma^v \vdash e_0 \to o\,]\!]$. The proof is done by induction on n of $e_0 \overset{n}{\sim} e$.

(2) For $se \rhd e : \mathcal{C} \in [\![\hat{\mathrm{E}} \rhd e_0 \ : \mathcal{C}_0\,]\!]$, after showing that $\sigma^v \vdash e \to o'$ exists in $[\![\sigma^v \vdash e_0 \to o\,]\!]$, we show that $o' \in gm(\mathcal{C}_0)(se)$. This proof is done for each case of $se \rhd e : \mathcal{C}$. □

Theorem 1 (Soundness) *For a program e_0 with a free variable x_0, let $\hat{\mathrm{E}} \rhd e_0 : \mathcal{C}_0$. Then, for every $v \in gm(\mathcal{C}_0)(\mathcal{X}_0)$, $[x_0 \mapsto v] \vdash e_0 \to \hat{\mathrm{E}}$ holds.*

Proof. By lemma 2, (1) $\sigma^v \vdash e_0 \to o$ exists in $[\![\sigma^v \vdash e_0 \to o\,]\!]$ and (2) $o \in \{\hat{\mathrm{E}}\}$, thus $\sigma^v \vdash e_0 \to \hat{\mathrm{E}}$ holds. Because x_0 is the only one free variable of e_0 and $[x_0 \mapsto v](x_0) = \sigma^v(x_0)$, the evaluation of e_0 under $[x_0 \mapsto v]$ makes the identical result to the result of the evaluation of e_0 under σ^v. Thus, $[x_0 \mapsto v] \vdash e_0 \to \hat{\mathrm{E}}$ holds. □

5 Examples

We shall present some examples to show how the analysis works for a given program. We include datatype declarations to make explicit the data constructors available in a given example program. Throughout this section, we write $\mathcal{U}$ for the universe set.

5.1 Example 1

Because of alternative rules in Figure 4, there can be several derivation trees for a given expression. In this section, we shall present all the derivation trees for the following program. The program raises an exception `E`, when the input value x is constructed with `A`. Otherwise, the program returns its input value.

$$\texttt{datatype}\ t\ =\ \texttt{A of}\ t \mid \texttt{B of}\ t \mid \texttt{C of 0}$$
$$\texttt{case}\ x\ \texttt{A}\ (\texttt{raise E})\ x$$

In order for a `case` expression to raise an exception `E`, we can apply `[CASEX-a]`, `[CASE-1a]`, or `[CASE-2a]`.

- A possible derivation:

$$\frac{\hat{\texttt{E}} \rhd x : \{\texttt{false}\}}{\hat{\texttt{E}} \rhd \texttt{case}\ x\ \texttt{A}\ (\texttt{raise E})\ x : \{\texttt{false}\}}$$

  This case is when we apply `[CASEX-a]`. There's no way for x to be included in $\{\hat{\texttt{E}}\}$, hence the constraint is obviously $\{\texttt{false}\}$.

- A possible derivation:

$$\frac{\texttt{A}\ \mathcal{Y} \rhd x : \{\mathcal{X} \subseteq \texttt{A}\ \mathcal{Y}\} \qquad \hat{\texttt{E}} \rhd \texttt{raise E} : \{\texttt{true}\}}{\hat{\texttt{E}} \rhd \texttt{case}\ x\ \texttt{A}\ (\texttt{raise E})\ x\ :\ \{\mathcal{X} \subseteq \texttt{A}\ \mathcal{Y},\ \texttt{true}\}}$$

  This case is when we apply `[CASE-1a]`. In order for x to be included in $\texttt{A}\ \mathcal{Y}$, $\mathcal{X}$ has to be included in $\texttt{A}\ \mathcal{Y}$. Because `raise E` always evaluates into $\hat{\texttt{E}}$, the constraint is $\{\texttt{true}\}$. The final constraint is:

$$\{\mathcal{X} \subseteq \texttt{A}\ \mathcal{Y},\ \texttt{true}\}$$

  and, as we expected, its greatest solution is $\mathcal{X} = \texttt{A}\ \mathcal{U}$ because a variable without constraints has the universe as its greatest model.

- A possible derivation:

$$\frac{\bar{\texttt{A}}\ \mathcal{Y} \rhd x : \{\mathcal{X} \subseteq \bar{\texttt{A}}\ \mathcal{Y}\} \qquad \hat{\texttt{E}} \rhd x : \{\texttt{false}\}}{\hat{\texttt{E}} \rhd \texttt{case}\ x\ \texttt{A}\ (\texttt{raise E})\ x\ :\ \{\mathcal{X} \subseteq \bar{\texttt{A}}\ \mathcal{Y},\ \texttt{false}\}}$$

  This case is when we apply `[CASE-2a]`. In order for x to be included in $\bar{\texttt{A}}\ \mathcal{Y}$, $\mathcal{X}$ has to be included in $\bar{\texttt{A}}\ \mathcal{Y}$. Because x can't raise any exception, the constraint is $\{\texttt{false}\}$.

5.2 Example 2

In the following program, the function f raises an exception `E` when the input is constructed with `Zero`, otherwise, calls itself with a decreased input.

We shall consider the derivation tree where the recursive call is traced only once. (We set the $\mathcal{N}$ in rule `[APP-3a]` to be one.)

$$\texttt{datatype}\ n\ =\ \texttt{Zero of 0}\ |\ \texttt{Suc of}\ n$$

$$(\texttt{fix}\ f\ \lambda x.\ \overbrace{\texttt{case}\ x\ \texttt{Zero}\ (\texttt{raise E})\ (\underbrace{f\ (\texttt{Suc}^{-1}\ x)}_{e_2})}^{e_1}\ )\ y$$

By applying `[APP-2a]` and `[NVN-a]` to the program:

$$\frac{\mathcal{X}_1 \rhd y : \{\mathcal{Y} \subseteq \mathcal{X}_1\} \qquad \hat{\texttt{E}} \rhd [x_1/x]e_1 : \mathcal{C}_1}{\hat{\texttt{E}} \rhd (\texttt{fix}\ f\ \lambda x.\ e_1)\ y : \{\mathcal{Y} \subseteq \mathcal{X}_1\} \cup \mathcal{C}_1}.$$

By applying `[CASE-2a]` and `[NVN-a]` to $\hat{\texttt{E}} \rhd [x_1/x]e_1 : \mathcal{C}_1$:

$$\frac{\overline{\texttt{Zero}}\ \mathcal{Z} \rhd x_1 : \{\mathcal{X}_1 \subseteq \overline{\texttt{Zero}}\ \mathcal{Z}\} \qquad \hat{\texttt{E}} \rhd [x_1/x]e_2 : \mathcal{C}_2}{\hat{\texttt{E}} \rhd \underbrace{\texttt{case}\ x_1\ \texttt{Zero}\ (\texttt{raise E})\ [x_1/x]e_2}_{[x_1/x]e_1} : \mathcal{C}_1 = \{\mathcal{X}_1 \subseteq \overline{\texttt{Zero}}\ \mathcal{Z}\} \cup \mathcal{C}_2},$$

and by applying `[APP-3a]`, `[DCONN-1a]`, and `[NVN-a]` to $\hat{\texttt{E}} \rhd [x_1/x]e_2 : \mathcal{C}_2$:

$$\frac{\dfrac{\texttt{Suc}\ \mathcal{X}_2 \rhd x_1 : \{\mathcal{X}_1 \subseteq \texttt{Suc}\ \mathcal{X}_2\}}{\mathcal{X}_2 \rhd \texttt{Suc}^{-1}\ x_1 : \{\mathcal{X}_1 \subseteq \texttt{Suc}\ \mathcal{X}_2\}} \qquad \hat{\texttt{E}} \rhd [x_2/x]e_1 : \mathcal{C}_3}{\hat{\texttt{E}} \rhd \underbrace{f\ (\texttt{Suc}^{-1}\ x_1)}_{[x_1/x]e_2} : \mathcal{C}_2 = \{\mathcal{X}_1 \subseteq \texttt{Suc}\ \mathcal{X}_2\} \cup \mathcal{C}_3}.$$

By applying `[CASE-1a]`, `[NVN-a]`, and `[RSX-1a]` to $\hat{\texttt{E}} \rhd [x_2/x]e_1 : \mathcal{C}_3$:

$$\frac{\texttt{Zero}\ \mathcal{W} \rhd x_2 : \{\mathcal{X}_2 \subseteq \texttt{Zero}\ \mathcal{W}\} \qquad \hat{\texttt{E}} \rhd \texttt{raise E} : \{\texttt{true}\}}{\hat{\texttt{E}} \rhd \underbrace{\texttt{case}\ x_2\ \texttt{Zero}\ (\texttt{raise E})\ [x_2/x]e_2}_{[x_2/x]e_1} : \mathcal{C}_3 = \{\mathcal{X}_2 \subseteq \texttt{Zero}\ \mathcal{W},\ \texttt{true}\}}.$$

Thus the constraint at the bottom of the derivation tree is:

$$\{\mathcal{Y} \subseteq \mathcal{X}_1,\ \mathcal{X}_1 \subseteq \overline{\texttt{Zero}}\ \mathcal{Z},\ \mathcal{X}_1 \subseteq \texttt{Suc}\ \mathcal{X}_2,\ \mathcal{X}_2 \subseteq \texttt{Zero}\ \mathcal{W},\ \texttt{true}\},$$

and its greatest solution is:

$$\mathcal{Y} = \texttt{Suc}\ (\texttt{Zero}\ \mathcal{U}),\ \mathcal{X}_1 = \texttt{Suc}\ (\texttt{Zero}\ \mathcal{U}),\ \mathcal{Z} = \mathcal{U},\ \mathcal{X}_2 = \texttt{Zero}\ \mathcal{U},\ \mathcal{W} = \mathcal{U}.$$

When we evaluate the program with the test data $\mathcal{Y} = \texttt{Suc}\ (\texttt{Zero}\ \mathcal{U})$, the recursive call is done only once and the program raises an exception `E`.

6 Conclusion

For a call-by-value, first-order language with the ML-style exception mechanism, we have presented a test data generation to cover all the exception-raise expressions in the input program. Our test data derivation is defined as a set-constraint propagation. Given a syntax tree representing the program and the initial constraint that the program has a unique uncaught exception (which will be raised by a particular raise expression), our analysis backwardly propagates constraints towards the leaves of the tree, to yield constraints about the input variable.

Our constraint propagation maintains a necessary-condition for each sub-expression by tracing back a single execution flow of the program. If multiple execution flows can be traced, only one flow is taken at a time. Thus we can avoid approximate constraints ("may" information in contrast to "must" information) that will be unavoidable when we try to subsume multiple possibilities into a single choice.

Even though in the worst case we have to trace all the possible execution traces, the large number of constraint-propagation trees would be quickly trimmed. First, because expressions that contribute to the uncaught exception would be sparse in the program, constraints along the majority of execution traces will quickly meet a contradicting result hence are immediately removed from consideration. Second, because it is sufficient for us to find at least one test datum to execute the designated raise expression, we can stop once the first-ever non-contradicting constraints are derived. Lastly, the constraint propagation takes time linearly proportional to the input program size.

We rigorously prove that every value that satisfies the constraints at the input variable will necessarily cause the program to execute the designated raise expression. For recursive calls we bound the number of repetitive tracings of the function's body by a fixed number, and this arbitrary bound for recursive propagations may fail to generate some constraints that would otherwise allow non-empty test data. However, the analysis safety (every generated test datum will cause the program to execute the designated raise expression) will still vacuously hold for empty test data result.

6.1 Related Works

As far as we know, our method is the first approach to automatic test data generation for exceptions. One salient feature of our method is its formal approach to the test data generation. We present a precise definition of the constraint propagation system and rigorously prove its soundness with respect to the operational semantics of the source language.

Because our analysis covers a particular path (a path to execute an exception-raise expression) in the program, the analysis falls into the path-wise test data generators. There are three kinds of test data generators: path-wise test data generators to cover certain structural elements in the program [Kor90, DO91, OJP99, Cla76, BKM91, How77, RbFHC76, BBS$^+$79], data specification generators to generate test data from a formal grammar [Mau90], and random test data generators [VMM91].

Most of the path-wise test data generators [Cla76, BKM91, How77, RbFHC76, BBS$^+$79] have used symbolic execution and they do not consider languages with exception mechanisms nor rigorously prove their soundness. Given a program and a designated path, the analysis symbolically executes the path and creates a set of constraints on the program's input variables. Depending on the target languages of the analyses, the constraints are integer-intervals for variables [Cla76] (Fortran), [BKM91] (Pascal-like language), numeric formula of variables [How77] (Fortran), types and integer-intervals for variables [RbFHC76] (Fortran), or inequalities for data fields [BBS$^+$79] (Cobol-like language). None of them considers exception mechanisms or mentioned their soundness with respect to the operational semantics of the language.

6.2 Adopting Our Analysis for the Higher-order ML

In order for our analysis to be adopted for the higher-order ML, we have to consider the following problems:

- We need to lower-approximate the function-call-graph of programs. In order to maintain the necessary condition of the test data, the call-graph estimation should not have

spurious information in the sense that a call edge should not be included when in doubt.

- Exceptions are first-class objects in ML. They are treated just like any 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. Therefore, our constraints have to express the flows of exception values intermingled with other normal values.

We are currently implementing this analysis. Our prototype has L programs as its input programs, construct set constraints by analyzing input programs, and solve set constraints by fixpoint iteration.

References

[AH95] Alex Aiken and Nevin Heintze. Constraint-based program analysis. Tutorial of the ACM Symposium on Principles of Programming Languages, January 1995.

[Ar996] Ariane 5: Flight 501 Failure. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, July 1996.

[BBS+79] Jānis Bičevskis, Juris Borzovs, Uldis Straujums, Andris Zarinš, and JR. Edward F. Miller. SMOTL – a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, 5:60–66, January 1979.

[BKM91] Juris Borzovs, Audris Kalninš, and Inga Medvedis. Automatic construction of test sets: Practical approach. volume 502 of *Lecture Notes in Computer Science*, pages 360–432. 1991.

[CDJ+89] Luca Cardelli, J. Donahue, Michael Jordan, B. Kalsow, and Greg Nelson. The modula-3 type system. In *ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, TX, January 1989.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[CP97] Witold Charatonik and Andreas Podelski. Solving set constraints for greatest models. Technical Report MPI-I-97-2-004, Max-Planck-Institut für Informatik, April 1997.

[CP98] Witold Charatonik and Andreas Podelski. Co-definite set constraints. In *Lecture Notes in Computer Science*, volume 1379, pages 211–225. Springer-Verlag, Proceedings of the 9th International Conference on Rewriting Techniques and Applications - RTA'98 edition, 1998.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[GJS96] James Gosling, Bill Joy, and Guy L. Jr. Steele. *The Java Language Specification (Java Series)*. Addison-Wesley, September 1996.

[Hei92] Nevin Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, October 1992.

[HJ90] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. Technical Report IBM Technical Report RC 16089 (# 71415), IBM, August 1990.

[HJ91] 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.

[How77] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 4(4):266–278, 1977.

[HP83] A. Nico Habermann and Dewayne E. Perry. *Ada for Experienced Programmers*. Addison-Wesley, 1983.

[Kah88] G. Kahn. Natural semantics. In K. Fuchi and M. Nivaat, editors, *Programming of Future Generation Computers*, pages 237–257. Elsevier Science Publishers (North-Holland), 1988.

[Kor90] Bokdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[Mau90] Peter M. Maurer. Generating testing data with enhanced context-free grammars. *IEEE Software*, 7(4), July 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[OJP99] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 1999.

[PL99] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *ACM Symposium on Principles of Programming Languages*, pages 276–290, January 1999.

[RbFHC76] C. V. Ramamoorthy, Siu bun F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

[VMM91] Jeffrey Voas, Larry Morell, and Keith Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–58, March 1991.

[YC99] Kwangkeun Yi and Byeong-Mo Chang. Exception analysis for java. In *ECOOP'99 Workshop on Formal Techniques for Java Programs*, June 1999.

[YR97] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Lecture Notes in Computer Science*, volume 1302, pages 98–113. Springer-Verlag, Proceedings of the 4th International Static Analysis Symposium edition, 1997.

[YR98] Kwangkeun Yi and Sukyoung Ryu. SML/NJ Exception Analyzer 0.98. http://compiler.kaist.ac.kr/pub/exna/exna-README.html, December 1998.

[YRon] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, (invited submission).