ELSEVIER

Theoretical Computer Science 277 (2002) 185–217

Theoretical Computer Science

www.elsevier.com/locate/tcs

A cost-effective estimation of uncaught exceptions in Standard ML programs ☆

Kwangkeun Yi *, Sukyoung Ryu

Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1 Kusong-dong Yusong-gu, Taejon 305-701, South Korea

Abstract

We present a static analysis that detects potential runtime exceptions that are raised and never handled inside Standard ML (SML) programs. This analysis will predict abrupt termination of SML programs, which is SMLs 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.

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 1.36 s and it reports 3 may-uncaught exceptions, which are exactly the exceptions that can really escape. Our final goal is to make the analysis overhead less than 10% of the compilation time (compiling the ML-Lex takes 6–7 s) and to analyze modules in isolation.

1. Introduction

Exception handling facilities in Standard ML [13] allow the programmer to define, raise and handle exceptional conditions. Exceptional conditions are brought (by a raise expression) to the attention of another expression where the raised exceptions may be handled.

☆ This work is supported in part by KOSEF Grant 95-0100-54-3, by Korea Ministry of Information and Communication Grant 96151-IT2-12, by Korea Ministry of Science and Technology, by Samsung Electronics Corp., by LG Information and Communications, and by Creative Research Initiatives of the Korean Ministry of Science and Technology. [*A* preliminary version of this paper appeared in the proceedings of the 4*th* International Static Analysis Symposium.]

* Corresponding author. Tel.: +82-42-869-3536; fax: +82-42-869-3510.

E-mail addresses: kwang@cs.kaist.ac.kr (K. Yi), puppy@cs.kaist.ac.kr (S. Ryu).

0304-3975/02/$ - see front matter
PII: S0304-3975(00)00317-0

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

The exception facilities, however, can provide a hole for program safety. SML programs can abruptly halt when an exception is raised and never handled. This is the only one "safety hole" in well-typed SML programs. Uncaught exceptions are sometimes disastrous [2].

In this paper, we present a static analysis that detects exceptions that may cause this abrupt halt of SML programs. Our goal is to develop an effective such analysis that has less than 10% overhead of the total compilation time.

1.1. Exception mechanism in Standard ML

In SML, exceptions 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.

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

```
Error("at line 10")
```

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

```
exception Error of string
```

An exception is raised by

$$\texttt{raise}\ e$$

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 particularly called an exception packet. In this paper, however, when the context is clear we will use exception, exception value, and exception packet interchangeably.

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

$$e\ \texttt{handle}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n$$

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.

1.2. Analysis problems

- 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)··· (∗)

  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.

1.3. 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.

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).

Fig. 1. Exception use statistics in SML programs.

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.

1.4. Our approach

In the earlier work [20], all the above problems were tackled by a monolithic abstract interpreter. Functions, exceptions, and other data values were parts of the abstract values. The analysis was a collecting analysis that computed stable program states at each expression point of the input program. This monolithic approach was appealing because the analysis design and its correctness proof was done at once by a sound abstraction of the SML's concrete semantics. The collecting analyzer was, however, too expensive. It took about 1 h 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 (Fig. 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 do not guarantee the safety of our call-graph estimation. For such cases when functions to call are brought by uncaught exceptions, we choose to do a crude approximation, believing that this "large" approximation would be rarely

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

detrimental to the call-graph accuracy. Please note that we cannot use standard techniques for closure analysis [16, 7, 14, 10] 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 4.

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

  $$\texttt{fun f(x)} = e(\texttt{0}) + \texttt{1}$$

  includes the sum $\bigcup_g P_g$ 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 example, P_f of the following function:

  $$\texttt{fun f(x)} = \texttt{raise x}$$

  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 indicates that the program may terminate abnormally.
2. Uncaught exceptions from each handle expression. From this information the programmer can check the completeness of the handler patterns.

2. Language L

For presentation brevity, we present our analysis for an imaginary language L. The language is a monomorphically typed, call-by-value, higher-order language. The lan-

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

Abstract syntax

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

Type

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

Type rules

$\Gamma \in Var \xrightarrow{\text{fin}} Type \qquad ArgType(\kappa) =$ exception κ's argument type

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

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

[FIX] $$\frac{\Gamma[f \mapsto \tau_1] \vdash \lambda x_\tau.e_1 : \tau_1 \quad \Gamma[f \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{fix}\ f\ \lambda x_\tau.e_1\ \texttt{in}\ e_2 : \tau_2}$$

[EXN] $$\frac{\Gamma \vdash e : \tau \quad ArgType(\kappa) = \tau}{\Gamma \vdash \texttt{exn}\ \kappa\ e : \tau\ exn}$$

[DCON] $$\frac{\Gamma \vdash e : \tau\ exn}{\Gamma \vdash \texttt{decon}\ e : \tau}$$

[APP] $$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

[CASE] $$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau'\ exn\\ \Gamma \vdash e_2 : \tau\\ \Gamma \vdash e_3 : \tau\end{array}}{\Gamma \vdash \texttt{case}\ e_1\ \kappa\ e_2\ e_3 : \tau}$$

[RS] $$\frac{\Gamma \vdash e_1 : \tau\ exn}{\Gamma \vdash \texttt{raise}\ e : \tau'}$$

[-RS] $$\frac{\Gamma \vdash e : \tau\ exn}{\Gamma \vdash \texttt{-raise}\ e\ \kappa^+ : \tau'}$$

[+RS] $$\frac{\Gamma \vdash e : \tau\ exn}{\Gamma \vdash \texttt{+raise}\ e\ \kappa : \tau'}$$

[HNDL] $$\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma[x \mapsto \tau] \vdash e_2 : \tau' \quad \tau = \tau''\ exn}{\Gamma \vdash \texttt{handle}\ e_1\ \lambda x_\tau.e_2 : \tau'}$$

[C] $$\Gamma \vdash 1 : \iota$$

Fig. 2. L's abstract syntax and type rules.

guage L's abstract syntax and the usual monomorphic type rules are shown in Fig. 2. We use the usual notation that for a finite function $f \in A \xrightarrow{\text{fin}} B$, $f[a \mapsto b]$ 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, we work on the SML source level.[3]

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

Values in L are either exceptions or functions. An exception value is constructed by "$\texttt{exn}\ \kappa\ e$" where κ is an exception name and expression e is for its argument value. The argument of an exception is recovered by "$\texttt{decon}\ e$". "$\texttt{fix}\ f\ \lambda x_\tau.e_1\ \texttt{in}\ e_2$" binds recursive function $f = \lambda x_\tau.e_1$ in e_2. The case expression "$\texttt{case}\ e_1\ \kappa\ e_2\ e_3$" branches to e_2 if the value of e_1 is constructed with κ, otherwise, to e_3. "$\texttt{raise}\ e$" raises exception e. The `+raise` and `-raise` expressions are used in limited contexts, which we will discuss in the next section. The handle expression "$\texttt{handle}\ e_1\ \lambda x_\tau.e_2$", 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.

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

The operational semantics of L is in Fig. 3. Relation $\gamma \vdash e \Rightarrow v$ (resp. $\gamma \vdash e \Rightarrow \underline{\kappa\ v}$) is read: expression e evaluates into value v (resp. raises exception $\kappa\ v$). Note that except for the handle rule every rule

$$\frac{\begin{array}{c}\sigma_1 \vdash e_1 \Rightarrow v_1\\ \vdots\\ \sigma_n \vdash e_n \Rightarrow v_n\end{array}}{\sigma \vdash e \Rightarrow v}$$

represents the following n more extra rules for propagating a raised exception:

$$\frac{\sigma_1 \vdash e_1 \Rightarrow \underline{\kappa\ v}}{\sigma \vdash e \Rightarrow \underline{\kappa\ v}} \quad \frac{\begin{array}{c}\sigma_1 \vdash e_1 \Rightarrow v\\ \sigma_2 \vdash e_2 \Rightarrow \underline{\kappa\ v}\end{array}}{\sigma \vdash e \Rightarrow \underline{\kappa\ v}} \cdots \frac{\begin{array}{c}\sigma_1 \vdash e_1 \Rightarrow v_1\\ \vdots\\ \sigma_{n-1} \vdash e_{n-1} \Rightarrow v_{n-1}\\ \sigma_n \vdash e_n \Rightarrow \underline{\kappa\ v}\end{array}}{\sigma \vdash e \Rightarrow \underline{\kappa\ v}}$$

This indicates that evaluation of expressions e_i in the hypothesis stops with the first raised exception, and this is the result of the expression e in the conclusion.

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 $\Gamma \vdash e{:}\,\tau$ is a sub-deduction of $\vdash \wp{:}\,\tau_0$. We simply write $type(e)$ when it is clear from the context which program $\wp$ the expression e belongs to.

Note that the type $type_\wp(e)$ is uniquely defined. The typing rules for raise expressions ([RS], [−RS], and [+RS]), which can assign "arbitrary" types, will assign unique ones when the type τ_0 of the program $\wp$ is fixed.

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.

$$
\begin{array}{rlll}
\sigma \in & Env & = Var \xrightarrow{\text{fin}} Val & \text{environments} \\
v \in & Val & = Closure + Exn + \{1\} & \text{values} \\
& Closure & = Expr \times Env & \text{lambda exprs in in program } \wp \\
& & & \text{and environments} \\
\kappa\ v \in & Exn & = Con \times Val & \text{exceptions} \\
& Con & = \{\kappa_1, \ldots, \kappa_N\} & \text{exception names in program } \wp \\
\underline{\kappa\ v} \in & Packet & = Exn & \text{raised exceptions}
\end{array}
$$

$$\sigma \vdash 1 \Rightarrow 1 \qquad\qquad \sigma \vdash \lambda x_\tau.e \Rightarrow \langle \lambda x_\tau.e, \sigma \rangle$$

$$\frac{\sigma(x) = v}{\sigma \vdash x \Rightarrow v} \qquad\qquad \frac{\sigma[f \mapsto \langle \texttt{fix}\ f\ \lambda x_\tau.e_1, \sigma \rangle] \vdash e_2 \Rightarrow v}{\sigma \vdash \texttt{fix}\ f\ \lambda x_\tau.e_1\ \texttt{in}\ e_2 \Rightarrow v}$$

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash \texttt{exn}\ \kappa\ e \Rightarrow \kappa\ v} \qquad\qquad \frac{\sigma \vdash e \Rightarrow \kappa\ v}{\sigma \vdash \texttt{decon}\ e \Rightarrow v}$$

$$\frac{\begin{array}{c}\sigma \vdash e_1 \Rightarrow \langle \lambda x_\tau.e', \sigma' \rangle \\ \sigma \vdash e_2 \Rightarrow v_2 \\ \sigma'[x \mapsto v_2] \vdash e' \Rightarrow v\end{array}}{\sigma \vdash e_1\ e_2 \Rightarrow v} \qquad\qquad \frac{\begin{array}{c}\sigma \vdash e_1 \Rightarrow \kappa\ v \\ \sigma \vdash e_2 \Rightarrow v\end{array}}{\sigma \vdash \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \Rightarrow v}$$

$$\frac{\begin{array}{c}\sigma \vdash e_1 \Rightarrow \langle \texttt{fix}\ f\ \lambda x_\tau.e', \sigma' \rangle \\ \sigma \vdash e_2 \Rightarrow v_2 \\ \sigma'[f \mapsto \langle \texttt{fix}\ f\ \lambda x_\tau.e', \sigma' \rangle][x \mapsto v_2] \vdash e' \Rightarrow v\end{array}}{\sigma \vdash e_1\ e_2 \Rightarrow v} \qquad\qquad \frac{\begin{array}{c}\sigma \vdash e_1 \Rightarrow \kappa' v \quad \kappa' \neq \kappa \\ \sigma \vdash e_3 \Rightarrow v\end{array}}{\sigma \vdash \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \Rightarrow v}$$

$$\frac{\sigma \vdash e \Rightarrow \kappa\ v}{\sigma \vdash \texttt{raise}\ e \Rightarrow \underline{\kappa\ v}} \qquad\qquad \frac{\sigma \vdash e \Rightarrow \kappa\ v \quad \forall i \in \{1, \ldots, n\}.\kappa_i \neq \kappa}{\sigma \vdash \texttt{-raise}\ e\ \kappa_1 \cdots \kappa_n \Rightarrow \underline{\kappa\ v}}$$

$$\frac{\sigma \vdash e \Rightarrow \kappa\ v}{\sigma \vdash \texttt{+raise}\ e\ \kappa \Rightarrow \underline{\kappa\ v}} \qquad\qquad \frac{\sigma \vdash e_1 \Rightarrow \underline{v} \quad \sigma[x \mapsto v] \vdash e_2 \Rightarrow}{\sigma \vdash \texttt{handle}\ e_1\ \lambda x_\tau.e_2 \Rightarrow v}$$

Fig. 3. L's operational semantics.

The second condition requires that the exceptions that are raised and thrown to a handler should have the same type as the handler function's argument type.

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

Proof. Note that the typefulness of L requires the raised exceptions, as well as expression's values, to be typeful (the second condition of Definition 2).

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 $\{\tau_1,\ldots,\tau_n\}$ be the set of exception types in an SML program. Every SML handle expression is translated into an L handler:

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

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).

2.1. 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 Fig. 2. For convenience we use numbers and multiple branches with the wild-card pattern, for example.)

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

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

  "-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:

$$\begin{array}{lcl} & & \texttt{handle}\ e\ \lambda\ x_{(\iota list)exn}. \\ & & \texttt{case}\ x \\ \texttt{exception E of int list} & & \texttt{E}\ (\lambda\ y_{\iota list}. \\ \cdots & \stackrel{\text{is}}{\Rightarrow} & \texttt{case}\ y \\ e\ \texttt{handle E nil}\ \Rightarrow 1 & & \texttt{NIL 1} \\ & & \_\ +\texttt{raise}\ x\ \texttt{E})\ (\texttt{decon}\ x) \\ & & \_\ -\texttt{raise}\ x\ \texttt{E} \end{array}$$

The 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,

$$\begin{array}{lcl} \texttt{exception E of int} & & \\ \cdots\ \texttt{E},\ \cdots & \overset{\text{is}}{\Rightarrow} & \cdots\ (\lambda\, x_\iota.(\texttt{exn E}\ x)),\ \cdots \end{array}$$

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

3. Set-constraint systems

Our exception analysis is presented in the set-constraint framework [6, 1]. 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 [17] over the continuous functions that are derived [4] 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 3.3) decouples control-flow analysis (Section 3.2) from exception analysis. $\triangleright_3$ (Section 3.4) 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 3.1) is assumed correct with respect to the standard semantics of L.

To review some notions of set constraint formalism, an interpretation $\mathscr{I}$ (a map from set expressions to sets) is a *model* (a solution) of a conjunction $\mathscr{C}$ of constraints if, for each constraint $\mathscr{X} \supseteq se$ (set variable $\mathscr{X}$ and set expression se) in $\mathscr{C}, \mathscr{I}(se)$ is defined and $\mathscr{I}(\mathscr{X}) \supseteq \mathscr{I}(se)$. We write $lm(\mathscr{C})$ for the least model of $\mathscr{C}$. 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 [6].

[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.

3.1. Concrete constraint construction $\triangleright_1$

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".

In Fig. 4 we index V and P sometimes with expressions, sometimes with numbers. For example, in

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

the $\mathscr{C}_1$ has constraints, among others, for e_1. The set variable for e_1 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 2.1), 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:

$$\mathscr{I}(var(x)) = \{v \mid \text{“}e_1\ e_2\text{”} \in e, \lambda x_\tau.e \in \mathscr{I}(V_1), v \in \mathscr{I}(V_2)\}$$

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

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

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

Consider the rule for the `handle` expression:

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

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 reraised from the handler function.

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

$$
\begin{array}{lcl}
\mathscr{I}(V_e) & \subseteq & \mathit{Val} \\
\mathscr{I}(\lambda x_\tau.e) & = & \{\lambda x_\tau.e\} \\
\mathscr{I}(\mathit{exn}(\kappa, V_1)) & = & \{\kappa\ v \mid v \in \mathscr{I}(V_1)\}
\end{array}
\qquad \mathscr{I}(1) = \{1\} \qquad
\begin{array}{lcl}
\mathscr{I}(P_e) & \subseteq & \mathit{Packet} \\
\mathscr{I}(se \cup se') & = & \mathscr{I}(se) \cup \mathscr{I}(se') \\
\mathscr{I}(\mathit{decon}(V_1)) & = & \{v \mid \kappa\ v \in \mathscr{I}(V_1)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{I}(\mathit{var}(x)) & = & \{v \mid e_1\ e_2 \in \wp, \lambda x_\tau.e \in \mathscr{I}(V_1), v \in \mathscr{I}(V_2)\} \\
\mathscr{I}(\mathit{app}_V(V_1)) & = & \{v \mid \lambda x_\tau.e \in \mathscr{I}(V_1), v \in \mathscr{I}(V_e)\} \\
\mathscr{I}(\mathit{app}_P(V_1)) & = & \{\kappa\ v \mid \lambda x_\tau.e \in \mathscr{I}(V_1), \kappa\ v \in \mathscr{I}(P_e)\} \\
\mathscr{I}(\mathit{case}(V_1, \kappa, V_2, V_3)) & = & \{v \mid\ \in \mathscr{I}(V_2), \kappa\ v' \in \mathscr{I}(V_1)\} \cup \{v \mid v \in \mathscr{I}(V_3),\ \kappa' v' \in \mathscr{I}(V_1), \kappa' \neq \kappa\} \\
\mathscr{I}(-\mathit{raise}(V_1, \kappa_1, \ldots, \kappa_n)) & = & \{\kappa'\ v \mid \kappa'\ v \in \mathscr{I}(V_1),\ \forall i.\kappa_i \neq \kappa'\} \\
\mathscr{I}(+\mathit{raise}(V_1, \kappa)) & = & \{\kappa\ v \mid \kappa\ v \in \mathscr{I}(V_1)\}
\end{array}
$$

$$
[\mathrm{VAR}_{\rhd_1}] \quad \rhd_1 x\colon \{V_x \supseteq \mathit{var}(x)\} \qquad [\mathrm{C}_{\rhd_1}] \quad \rhd_1 1\colon \{V_e \supseteq 1\} \qquad [\mathrm{ABS}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1}{\rhd_1 \lambda x_\tau.e_1\colon \{V_e \supseteq \lambda x_\tau.e_1\} \cup \mathscr{C}_1}
$$

$$
[\mathrm{FIX}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1 \quad \rhd_1 e_2\colon \mathscr{C}_2}{\rhd_1 \texttt{fix}\ f\ \lambda x_\tau.e_1\ \texttt{in}\ e_2\colon \{V_e \supseteq V_2,\ P_e \supseteq V_2,\ V_f \supseteq \lambda x_\tau.e_1\} \cup \mathscr{C}_1 \cup \mathscr{C}_2}
$$

$$
[\mathrm{EXN}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1}{\rhd_1 \texttt{exn}\ \kappa\ e_1\colon \{V_e \supseteq \mathit{exn}(\kappa, V_1), P_e \supseteq P_1\} \cup \mathscr{C}_1}
$$

$$
[\mathrm{DCON}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1}{\rhd_1 \texttt{decon}\ e_1\colon \{V_e \supseteq \mathit{decon}(V_1),\ P_e \supseteq P_1\} \cup \mathscr{C}_1}
$$

$$
[\mathrm{APP}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1 \quad \rhd_1 e_2\colon \mathscr{C}_2}{\rhd_1 e_1\ e_2\colon \{V_e \supseteq \mathit{app}_V(V_1),\ P_e \supseteq \mathit{app}_P(V_1) \cup P_1 \cup P_2\} \cup \mathscr{C}_1 \cup \mathscr{C}_2}
$$

$$
[\mathrm{CASE}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1 \quad \rhd_1 e_2\colon \mathscr{C}_2 \quad \rhd_1 e_3\colon \mathscr{C}_3}{\rhd_1 \texttt{case}\ e_1\ \kappa\ e_2\ e_3\colon \{V_e \supseteq \mathit{case}(V_1, \kappa, V_2, V_3),\ P_e \supseteq P_1 \cup P_2 \cup P_3\} \cup \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3}
$$

$$
[\mathrm{RS}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1}{\rhd_1 \texttt{raise}\ e_1\colon \{P_e \supseteq V_1\} \cup \mathscr{C}_1}
$$

$$
[-\mathrm{RS}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1}{\rhd_1 \texttt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n\colon \{P_e \supseteq -\mathit{raise}(V_1, \kappa_1, \ldots, \kappa_n)\} \cup \mathscr{C}_1}
$$

$$
[+\mathrm{RS}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1}{\rhd_1 \texttt{+raise}\ e_1\ \kappa\colon \{P_e \supseteq +\mathit{raise}(V_1, \kappa)\} \cup \mathscr{C}_1}
$$

$$
[\mathrm{HNDL}_{\rhd_1}] \quad \frac{\rhd_1 e_1\colon \mathscr{C}_1 \quad \rhd_1 e_2\colon \mathscr{C}_2}{\begin{array}{ll}\rhd_1 \texttt{handle}\ e_1\ \lambda x_\tau.e_2\colon \{ & V_e \supseteq \mathit{app}_V(\lambda x_\tau.e_2) \cup V_1, \\ & P_e \supseteq \mathit{app}_P(\lambda x_\tau.e_2), V_x \supseteq P_1 \\ \qquad\qquad\qquad\qquad \} & \cup \mathscr{C}_1 \cup \mathscr{C}_2 \end{array}}
$$

Fig. 4. Constructing concrete constraints: $\rhd_1$.

We could have used Heintze's method [7, 8] 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 forthcoming constraint systems, $\rhd_2$ and $\rhd_3$

Proposition 2 (Correctness of $\triangleright_1$). *For a program (a closed expression) $\wp$, let $\triangleright_1 \wp : \mathscr{C}_1$ and let $lm(\mathscr{C}_1)$ be the least model of $\mathscr{C}_1$. Then for every sub-expression e of $\wp$, $lm(\mathscr{C}_1)(V_e)$ (respectively $lm(\mathscr{C}_1)(P_e)$) includes all the values that results from e (respectively all the exceptions that escapes from e) during the execution of $\wp$.*

Proof Sketch. This correctness can be proved by following the steps outlined in [7, 6]. The key idea is to define a "set-based operational semantics" that is defined over a fixed set environment (a map from variables to the sets of values). A set environment is defined to be safe if the environment includes all the values that are bound to each variable during the program's standard execution (Fig. 3). Among the safe set environments, there exists the least safe set environment. Our least model $lm(\mathscr{C}_1)$ is proved to be equivalent to the least safe set environment: $lm(\mathscr{C}_1(V_e)$ (resp. $lm(\mathscr{C}_1)(P_e)$) is exactly the set of values (resp. the set of escaping exceptions) that are derived for e by the set-based operational semantics with the least safe set environment. $\square$

Not only is $\triangleright_1$ correct but typeful. The following typefulness is important for the consistency of the forthcoming constraint system $\triangleright_2$.

Proposition 3 (Typefulness of $\triangleright_1$). *For a program (a closed expression) $\wp$, let $\triangleright_1 \wp : \mathscr{C}_1$. Then its least model $lm(\mathscr{C}_1)$ preserves types: $\forall e \in \wp . lm(\mathscr{C}_1)(V_e) \subseteq Val_{type(e)}$ (the set of values of $type(e)$).*

Proof. The least model $lm(\mathscr{C}_1)$ is equivalent to the $\subseteq$-least fixpoint fix $\mathscr{F}_1$ of the following continuous function $\mathscr{F}_1$ derived from $\mathscr{C}_1$ as follows [4]:

$$\begin{aligned}
\mathscr{V} &= \{V_e \mid e \in \wp\} \cup \{P_e \mid e \in \wp\} \text{ the set of constraint variables for program } \wp \\
2^{Val} &= \text{the powerset of } Val, \text{ordered by} \subseteq \\
\mathscr{F}_1 &: (\mathscr{V} \to 2^{Val}) \to (\mathscr{V} \to 2^{Val})
\end{aligned}$$

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

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

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 $\supseteq$ in constraints is $=$.

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

$$typeful(\rho)$$

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

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

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

[C] $e = 1$. $\mathscr{F}_1(\rho)(V_e) = \{1\} \subseteq Val_\iota$.

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

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

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

$\mathscr{F}_1(\rho)(V_x) = \rho(P_1)$ by definition. Because the program $\wp$ is typeful, the e_1's uncaught exn is bound to x_τ. Thus by IH $\rho(P_1) \subseteq Val_\tau$.

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

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

Other cases are similarly proved.

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

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

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

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

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

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

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

Fig. 5. Call-graph estimation rules.

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

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

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

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

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

Other cases are similarly proved. □

3.2. Separate call-graph estimation

Call-graph estimation methods [9, 10, 14, 16, 18] in the literature cannot be directly used in our exception analysis, because their high-order source languages do not have exceptions, not to mention the function-carrying exceptions.

Fig. 5 shows our rules to estimate the call graph of a program $\wp$. An edge $e \rightarrow \lambda x.e'$ indicates that during the execution of $\wp$ the e may evaluate into a closure of the lambda $\lambda x.e'$.

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

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

We estimate that an exception's argument functions are those whose types are equal 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 4 (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 expressions in* $\wp$. *Define* $Lam : FtnExpr \to 2^{Lambda}$ *to be*

$$Lam(e) = \{\lambda x_\tau.e' \mid e \to \lambda x_\tau.e' \text{ is deducible by the rules in Fig. 5}\}.$$

Then Lam is safe: $\rhd_1 \wp : \mathscr{C} \to \forall e \in FtnExpr.\ Lam(e) \supseteq lm(\mathscr{C})(V_e)$.

Proof. Note that the *Lam* is equivalent to the $\subseteq$-least fixpoint of the following continuous function $\mathscr{L}$ [4]:

$$\mathscr{L} : (FtnExpr \to 2^{Lambda}) \to (FtnExpr \to 2^{Lambda})$$

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

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

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

Note that we include the $typeful(\rho)$ assertion that we used in the proof of Proposition 3. 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 $Q(\mathscr{L}(\ell), \mathscr{F}_1(\rho))$.

Case e of a normal variable x:

$$\begin{aligned} \mathscr{L}(\ell)(x) &= \bigcup\{\ell(e_2) \mid e_1\ e_2 \in \wp, \lambda x.e \in \ell(e_1)\} && \text{by definition} \\ &\supseteq \bigcup\{\ell(e_1) \mid e_1\ e_2 \in \wp, \lambda x.e \in \rho(V_{e_1})\} && \text{by IH} \\ &\supseteq \bigcup\{\rho(V_{e_1}) \mid e_1\ e_2 \in \wp, \lambda x.e \in \rho(V_{e_1})\} && \text{by IH} \\ &= \mathscr{F}_1(\rho)(V_x) \quad \text{by definition.} \end{aligned}$$

[5] Note that the L language is monomorphic. In SML, the "is equal to" most be "unifies with".

Other cases are done similarly, except for the case e of `decon` e_1:

$$\begin{aligned}
\mathscr{L}(\ell)(e) &= \bigcup\{\lambda x.e' \mid \lambda x.e' \in \wp, type(e) = type(\lambda x.e')\} \quad \text{by definition}\\
&= Val_{type(e)}, \text{the set of lambdas of } type(e) \text{ in } \wp \quad \text{by definition}\\
&\supseteq \mathscr{F}_1(\rho)(V_e) \quad \text{by that } typeful(\rho) \Rightarrow typeful(\mathscr{F}_1(\rho)), \text{which is}\\
&\quad \text{proven in Proposition 3.} \qquad \square
\end{aligned}$$

3.3. Exception constraint construction $\triangleright_2$

We now consider a new system $\triangleright_2$ (Fig. 6) where only exceptions are considered. Constraints for function (non-exception) values are removed and instead, a pre-computed, safe call-graph table *Lam* (Proposition 4) 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 $|\mathscr{I}(V_e)|$ of exception names in e's values $\mathscr{I}(V_e)$:

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

Definition 3. $|\mathscr{I}(V)| = \{\kappa \mid \kappa\ v \in V\} \cup |\{v \mid \kappa\ v \in \mathscr{I}(V)\}|$.

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 4).

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

Consider the rule for `-raise` expression:

$$[-\mathrm{RS}_{\triangleright_2}] \quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1}{\triangleright_2\ -\texttt{raise}\ e_1\ \kappa_1 \cdots \kappa_n\colon\ \{P_e \supseteq (X_1 \backslash_{e_1} \{\kappa_1, \ldots, \kappa_n\})\} \cup \mathscr{C}_1}$$

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

$$\mathscr{I}(X_1 \backslash_e \{\kappa_1, \ldots, \kappa_n\}) = \begin{cases} \mathscr{I}(X_1) & \text{if } type(e) = \tau' exn \wedge isExn(\tau') \\ \mathscr{I}(X_1) \backslash \{\kappa_1, \ldots, \kappa_n\} & \text{otherwise} \end{cases}$$

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

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

The expression raises the exception $\kappa_1(\kappa_2(1))$ 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

$$\begin{array}{lll}
\kappa \in \mathit{Exn} & = \{\kappa_1, \ldots, \kappa_N\} & \text{exception names in program } \wp \\
\kappa \in \mathit{Packet} & = \mathit{Exn} & \text{raised exceptions}
\end{array}$$

$$\mathscr{I}(X_e) \subseteq \mathit{Exn} \qquad \mathscr{I}(P_e) \subseteq \mathit{Packet}$$

$$\begin{array}{ll}
\mathscr{I}(\mathit{var}(x)) & = \{\kappa \mid e_1\ e_2 \in \wp, \lambda x_\tau.e \in \mathit{Lam}(e_1), \kappa \in \mathscr{I}(X_2)\} \\
\mathscr{I}(\mathit{app}_X(e_1)) & = \{\kappa \mid \lambda x_\tau.e \in \mathit{Lam}(e_1), \kappa \in \mathscr{I}(X_e)\} \\
\mathscr{I}(\mathit{app}_P(e_1)) & = \{\kappa \mid \lambda x_\tau.e \in \mathit{Lam}(e_1), \kappa \in \mathscr{I}(P_e)\} \\
\mathit{isExn}(\tau) & = \text{true iff } \tau = \tau'\ \mathit{exn} \\
\mathscr{I}(X_1 \backslash_e \{\kappa_1, \ldots, \kappa_n\}) & = \begin{cases} \mathscr{I}(X_1) & \text{if } \mathit{type}(e) = \tau'\ \mathit{exn} \wedge \mathit{isExn}(\tau') \\ \mathscr{I}(X_1) \backslash \{\kappa_1, \ldots, \kappa_n\} & \text{otherwise} \end{cases} \\
\mathscr{I}(X_1 \cap_e \{\kappa\}) & = \begin{cases} \mathscr{I}(X_1) & \text{if } \mathit{type}(e) = \tau'\ \mathit{exn} \wedge \mathit{isExn}(\tau') \\ \mathscr{I}(X_1) \cap \{\kappa\} & \text{otherwise} \end{cases} \\
\mathscr{I}(\kappa) & = \{\kappa\}
\end{array}$$

$\mathscr{I}(\lambda x_\tau.e)$ and $\mathscr{I}(se \cup se)$ are the same as in $\triangleright_1$ (Fig. 4, p. 12).

$$[\mathrm{VAR}_{\triangleright_2}]\quad \triangleright_2\ x\colon\ \{X_x \supseteq \mathit{var}(x)\} \qquad [C_{\triangleright_2}]\quad \triangleright_2\ 1\colon\ \emptyset$$

$$[\mathrm{ABS}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1}{\triangleright_2\ \lambda x_\tau.e_1\colon\ \mathscr{C}_1} \qquad [\mathrm{FIX}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1 \quad \triangleright_2\ e_2\colon\ \mathscr{C}_2}{\triangleright_2\ \mathtt{fix}\ f\ \lambda x_\tau.e_1\ \mathtt{in}\ e_2\colon\ \{X_e \supseteq X_2,\ P_e \supseteq P_2\} \cup \mathscr{C}_1 \cup \mathscr{C}_2}$$

$$[\mathrm{DCON}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon \mathscr{C}_1}{\triangleright_2\ \mathtt{decon}\ e_1\colon\ \{X_e \supseteq X_1,\ P_e \supseteq P_1\} \cup \mathscr{C}_1}$$

$$[\mathrm{EXN}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1}{\triangleright_2\ \mathtt{exn}\ \kappa\ e_1\colon\ \{X_e \supseteq \kappa \cup X_1,\ P_e \supseteq P_1\} \cup \mathscr{C}_1}$$

$$[\mathrm{APP}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon \mathscr{C}_1 \quad \triangleright_2\ e_2\colon\ \mathscr{C}_2}{\triangleright_2\ e_1\ e_2\colon\ \{X_e \supseteq \mathit{app}_X(e_1),\ P_e \supseteq \mathit{app}_P(e_1) \cup P_1 \cup P_2\} \cup \mathscr{C}_1 \cup \mathscr{C}_2}$$

$$[\mathrm{CASE}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1 \quad \triangleright_2\ e_2\colon\ \mathscr{C}_2 \quad \triangleright_2\ e_3\colon\ \mathscr{C}_3}{\triangleright_2\ \mathtt{case}\ e_1\ \kappa\ e_2\ e_3\colon\ \{X_e \supseteq X_2 \cup X_3,\ P_e \supseteq P_1 \cup P_2 \cup P_3\} \cup \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3}$$

$$[\mathrm{RS}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1}{\triangleright_2\ \mathtt{raise}\ e_1\colon\ \{P_e \supseteq X_1\} \cup \mathscr{C}_1}$$

$$[-\mathrm{RS}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1}{\triangleright_2\ \mathtt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n\colon\ \{P_e \supseteq (X_1 \backslash_{e_1} \{\kappa_1, \ldots, \kappa_n\})\} \cup \mathscr{C}_1}$$

$$[+\mathrm{RS}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1}{\triangleright_2\ \mathtt{+raise}\ e_1\ \kappa\colon\ \{P_e \supseteq (X_1 \cap_{e_1} \{\kappa\})\} \cup \mathscr{C}_1}$$

$$[\mathrm{HNDL}_{\triangleright_2}]\quad \frac{\triangleright_2\ e_1\colon\ \mathscr{C}_1 \quad \triangleright_2\ e_2\colon\ \mathscr{C}_2}{\triangleright_2\ \mathtt{handle}\ e_1\ \lambda x_\tau.e_2\colon\ \left\{ \begin{array}{l} X_e \supseteq \mathit{app}_X(\lambda x_\tau.e_2) \cup X_1, \\ P_e \supseteq \mathit{app}_P(\lambda x_\tau.e_2), X_x \supseteq P_1 \end{array} \right\} \cup \mathscr{C}_1 \cup \mathscr{C}_2}$$

Fig. 6. Constructing exception constraints: $\triangleright_2$.

$X_e = \{\kappa_1, \kappa_2\}$ 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 $\backslash_e$ 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 $\cap_e$.)

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

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

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

$\mathscr{I}(X_f \setminus_e \{\kappa_1, \ldots, \kappa_n\}), \mathscr{I}(X_f \cap_e \{\kappa\}), \mathscr{I}(\kappa)$, and $\mathscr{I}(\lambda x_\tau.e)$ are the same as in $\rhd_2$ (Fig. 6, p. 18).

$$[\text{VAR}_{\rhd_3}] \quad f \rhd_3 x : \{X_f \supseteq var(x)\} \qquad [\text{C}_{\rhd_3}] \quad f \rhd_3 1 : \emptyset$$

$$[\text{ABS}_{\rhd_3}] \ \frac{g \rhd_3 e_1 : \mathscr{C}_1}{f \rhd_3 \lambda_g x_\tau.e_1 : \mathscr{C}_1} \qquad [\text{FIX}_{\rhd_3}] \ \frac{g \rhd_3 e_1 : \mathscr{C}_1 \quad f \rhd_3 e_2 : \mathscr{C}_2}{f \rhd_3 \texttt{fix } g\lambda_g x_\tau.e_1 \texttt{in } e_2 : \mathscr{C}_1 \cup \mathscr{C}_2}$$

$$[\text{DCON}_{\rhd_3}] \ \frac{f \rhd_3 e_1 : \mathscr{C}_1}{f \rhd_3 \texttt{decon } e_1 : \mathscr{C}_1} \qquad [\text{EXN}_{\rhd_3}] \ \frac{f \rhd_3 e_1 : \mathscr{C}_1}{f \rhd_3 \texttt{exn } \kappa\ e_1 : \{X_f \supseteq \kappa\} \cup \mathscr{C}_1}$$

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

$$[\text{CASE}_{\rhd_3}] \quad \frac{f \rhd_3 e_1 : \mathscr{C}_1 \quad f \rhd_3 e_2 : \mathscr{C}_2 \quad f \rhd_3 e_3 : \mathscr{C}_3}{f \rhd_3 \texttt{case } e_1\ \kappa\ e_2\ e_3 : \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3}$$

$$[\text{RS}_{\rhd_3}] \quad \frac{f \rhd_3 e_1 : \mathscr{C}_1}{f \rhd_3 \texttt{raise } e_1 : \{P_f \supseteq X_f\} \cup \mathscr{C}_1}$$

$$[-\text{RS}_{\rhd_3}] \quad \frac{f \rhd_3 e_1 : \mathscr{C}_1}{f \rhd_3 \texttt{-raise } e_1\ \kappa_1 \cdots \kappa_n : \{P_f \supseteq X_f \setminus_{e_1} \{\kappa_1, \ldots, \kappa_n\}\} \cup \mathscr{C}_1}$$

$$[+\text{RS}_{\rhd_3}] \quad \frac{f \rhd_3 e_1 : \mathscr{C}_1}{f \rhd_3 \texttt{+raise } e_1\ \kappa : \{P_f \supseteq X_f \cap_{e_1} \{\kappa\}\} \cup \mathscr{C}_1}$$

$$[\text{HNDL}_{\rhd_3}] \quad \frac{g \rhd_3 e_g : \mathscr{C}_1 \quad h \rhd_3 e_2 : \mathscr{C}_2}{\begin{array}{l} f \rhd_3 \texttt{handle } e_g\ \lambda_h x_\tau.e_2 : \{\ X_f \supseteq app_X(\lambda_h x_\tau.e_2, P_g) \cup X_g, \\ \qquad\qquad\qquad\qquad\quad P_f \supseteq app_P(\lambda_h x_\tau.e_2, P_g) \\ \qquad\qquad\qquad\qquad\ \} \cup \mathscr{C}_1 \cup \mathscr{C}_2 \end{array}}$$

Fig. 7. Constructing function's exception constraints: $\rhd_3$.

The constraint-construction rule $\rhd_2$ is a safe approximation of $\rhd_1$:

Proposition 5 (Correctness of $\rhd_2$). *For a program $\wp$, let $\rhd_1 \wp : \mathscr{C}_1$ and $\rhd_2 \wp : \mathscr{C}_2$ with their least models, $\mathscr{I}_1 = lm(\mathscr{C}_1)$ and $\mathscr{I}_2 = lm(\mathscr{C}_2)$. If Lam is safe with respect to $\rhd_1$,* *then for every sub-expression* $e \in \wp$:

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

Proof. The least models $\mathscr{I}_1$ and $\mathscr{I}_2$ are equivalent to the $\subseteq$-least fixpoints $\operatorname{fix}\mathscr{F}_1$ and $\operatorname{fix}\mathscr{F}_2$, respectively [4]. The $\mathscr{F}_1$ is defined in the proof of Proposition 3. The continuous function $\mathscr{F}_2$ is derived from $\mathscr{C}_2$ as follows:

$$\Psi = \{X_e \mid e \in \wp\} \cup \{P_e \mid e \in \wp\} \text{ the set of constraint variables for a program } \wp$$

$$2^{Exn} = \text{the powerset of } Exn, \text{ ordered by } \subseteq$$

$$\mathscr{F}_2 : (\Psi \to 2^{Exn}) \to (\Psi \to 2^{Exn})$$

$$\mathscr{F}_2(\varphi)(X_e) =$$

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

$$\mathscr{F}_2(\varphi)(P_e) =$$

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

It is straightforward to derive this function because the $\rhd_2$ generates at most one constraint per X_i and P_i. That is, each $\supseteq$ in constraints is $=$.

We prove $Q(\text{fix}\,\mathscr{F}_2, \text{fix}\,\mathscr{F}_1)$ by the 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 3. 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 $Q(\mathscr{F}_2(\varphi), \mathscr{F}_1(\rho))$ holds given the induction hypothesis $Q(\varphi, \rho)$. That is, we need to show $\mathscr{F}_2(\varphi)(X_e) \supseteq |\mathscr{F}_1(\rho)(V_e)|$ and $\mathscr{F}_2(\varphi)(P_e) \supseteq |\mathscr{F}_1(\rho)(P_e)|$.

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

$$\begin{aligned} \mathscr{F}_2(\varphi)(X_x) &= \varphi(P_1) && \text{(by definition)} \\ &\supseteq |\rho(P_1)| && \text{(by IH)} \\ &= |\mathscr{F}_1(\rho)(V_x)| && \text{(by definition).} \end{aligned}$$

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

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

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

$$\begin{aligned} \mathscr{F}_2(\varphi)(X_e) &= \{\kappa\} \cup \varphi(X_1) && \text{(by definition)} \\ &\supseteq \{\kappa\} \cup |\rho(V_1)| && \text{(by IH)} \end{aligned}$$

By definition, $|\mathscr{F}_1(\rho)(V_e)| = |\{\kappa\ v \mid v \in \rho(V_1)\}| = \{\kappa\} \cup |\rho(V_1)|$.
Therefore, $\mathscr{F}_2(\varphi)(X_e) \supseteq |\mathscr{F}_1(\rho)(V_e)|$.

$$\begin{aligned} \mathscr{F}_2(\varphi)(P_e) &= \varphi(P_1) && \text{(by definition)} \\ &\supseteq |\rho(P_1)| && \text{(by IH)} \\ &= |\mathscr{F}_1(\rho)(P_e)| && \text{(by definition).} \end{aligned}$$

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

$$\begin{aligned} \mathscr{F}_2(\varphi)(X_e) &= \varphi(X_1) && \text{(by definition)} \\ &\supseteq |\rho(V_1)| && \text{(by IH)} \\ &\supseteq |\mathscr{F}_1(\rho)(V_e)| && \text{(by definition).} \end{aligned}$$

$$\begin{aligned}
\mathscr{F}_2(\varphi)(P_e) &= \varphi(P_1) && \text{(by definition)}\\
&\supseteq |\rho(P_1)| && \text{(by IH)}\\
&= |\mathscr{F}_1(\rho)(P_e)| && \text{(by definition).}
\end{aligned}$$

[APP] $e = e_1\ e_2$.

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

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

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

$$\begin{aligned}
\mathscr{F}_2(\varphi)(X_e) &= \varphi(X_2) \cup \varphi(X_3) && \text{(by definition)}\\
&\supseteq \{v \mid v \in |\,\rho(V_2)|, \kappa\ v' \in \rho(V_1)\}\\
&\quad \cup \{v | v \in |\,\rho(V_3)|, \kappa' v' \in \rho(V_1),\ \kappa' \neq \kappa\} && \text{(by IH)}\\
&= |\mathscr{F}_1(\rho)(V_e)| && \text{(by definition).}
\end{aligned}$$

$$\begin{aligned}
\mathscr{F}_2(\varphi)(P_e) &= \varphi(P_1) \cup \varphi(P_2) \cup \varphi(P_3) && \text{(by definition)}\\
&\supseteq |\rho(P_1)| \cup |\rho(P_2)| \cup |\rho(P_3)| && \text{(by IH)}\\
&= |\mathscr{F}_1(\rho)(P_e)| && \text{(by definition).}
\end{aligned}$$

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

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

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

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

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

$$\begin{aligned}\mathscr{F}_2(\varphi)(P_e) &= \varphi(X_1)\backslash\{\kappa_1,\ldots,\kappa_n\} \quad \text{(by definition)}\\ &\supseteq |\rho(V_1)|\backslash\{\kappa_1,\ldots,\kappa_n\} \quad \text{(by IH)}\end{aligned}$$

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

Therefore, $\mathscr{F}_2(\varphi)(P_e)\supseteq|\mathscr{F}_1(\rho)(P_e)|$.
[$+$RS] $e=$ `+raise` $e_1\ \kappa$ where $type\ (e_1)=\tau'\ exn \wedge isExn(\tau')$.

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

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

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

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

Therefore, $\mathscr{F}_2(\varphi)(P_e) \supseteq |\mathscr{F}_1(\rho)(P_e)|$.
[HNDL] $e = \texttt{handle}\ e_1 \lambda x_\tau.e_2$.

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

$$\begin{aligned} \mathscr{F}_2(\varphi)(P_e) &= \varphi(P_2) && \text{(by definition)} \\ &\supseteq |\rho(P_2)| && \text{(by IH)} \\ &= |\mathscr{F}_1(\rho)(P_e)| && \text{(by definition).} \quad \square \end{aligned}$$

3.4. 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.

In $\triangleright_3$, 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:

$$[\text{APP}_{\triangleright_3}] \quad \frac{f \triangleright_3 e_1\colon \mathscr{C}_1 \quad f \triangleright_3 e_2\colon \mathscr{C}_2}{f \triangleright_3 e_1\ e_2\colon \{X_f \supseteq app_X(e_1, X_f), P_f \supseteq app_P(e_1, X_f)\} \cup \mathscr{C}_1 \cup \mathscr{C}_2}$$

The left-hand side f of "$f \triangleright_3 e$" 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 include the exceptions $app_P(e_1, X_f)$ uncaught during the call.

One noticeable rule is $[\text{VAR}_{\triangleright_3}]$. Because the constraint granularity is a function, constraints 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,

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

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 $\mathscr{I}(X_g) \supseteq \mathscr{I}(X_f)$ (in terms of caller f and callee g):

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

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

Proposition 6 (Correctness of $\triangleright_3$). *For a closed term e, let $\triangleright_2 e$: $\mathscr{C}_2$ and* `main` $\triangleright_3 e$: $\mathscr{C}_3$ *with their least models, $\mathscr{I}_2 = lm(\mathscr{C}_2)$ and $\mathscr{I}_3 = lm(\mathscr{C}_3)$. Then, if $g \triangleright_3 e'$: $\mathscr{C}'$ occurs during* `main` $\triangleright_3 e$: $\mathscr{C}_3$ *then*

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

Proof. We will prove that for any model $\mathscr{I}$ of $\mathscr{C}_3$, the above holds. Note that the least model $\mathscr{I}_2$ is equivalent to the $\subseteq$-least fixpoints fix $\mathscr{F}_2$. The $\mathscr{F}_2$ is defined in the proof of Proposition 5.

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

$$\forall \textit{ model } \mathscr{I} \textit{ of } \mathscr{C}_3.\text{“}f \triangleright_3 e{:}\mathscr{C}'' \textit{ occurs during } (\texttt{main} \triangleright_3 \wp{:}\ \mathscr{C}_3)$$

$$\Rightarrow \mathscr{I}(X_f) \supseteq \varphi(X_e) \wedge \mathscr{I}(P_f) \supseteq \varphi(P_e).$$

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

In the following proof, for each case $f \triangleright_3 expr$ we abbreviate the *expr* by e.

[VAR] $f \triangleright_3 x$ (handle variable) where `handle` e_g $\lambda_h x_\tau.e_2 \in \wp$.

$$\begin{aligned}
\mathscr{I}(X_f) &\supseteq \mathscr{I}(X_{Owner(x)}) = \mathscr{I}(X_h) && \text{(by [VAR}_{\triangleright_3}\text{])}\\
&\supseteq \mathscr{I}(P_g) && \text{(by [HNDL}_{\triangleright_3}\text{])}\\
&\supseteq \varphi(P_g) && \text{(because “}g \triangleright_3 e_g\text{” occurs and by IH)}\\
&= \mathscr{F}_2(\varphi)(X_x) && \text{(by definition)}
\end{aligned}$$

[VAR] $f \triangleright_3 x$(normal variable).

By [VAR$_{\triangleright_3}$], $\mathscr{I}(X_f) \supseteq \mathscr{I}(X_{Owner(x)})$. Let a function g be the owner of x: $\lambda_g x_\tau.e'$. For each “$k \triangleright_3 e_1\ e_2$” that occurs at the $Owner(x)$'s call site, “$k \triangleright_3 e_2$” occurs. Thus by IH,

$$\begin{aligned}
\varphi(X_2) &\subseteq \mathscr{I}(X_k)\\
&\subseteq \mathscr{I}(X_g) \quad \text{(by [APP}_{\triangleright_3}\text{])}
\end{aligned}$$

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

[EXN] $f \rhd_3$ `exn` $\kappa\ e_1$.

$$\begin{aligned} \mathscr{I}(X_f) &\supseteq \{\kappa\} && \text{(by [EXN}_{\rhd_3}\text{])} \\ \mathscr{I}(X_f) &\supseteq \varphi(X_1) && \text{(because "}f \rhd_3 e_1{''}\text{ occurs and by IH)} \\ \text{Therefore, } \mathscr{I}(X_f) &\supseteq \{\kappa\} \cup \varphi(X_1) \\ &= \mathscr{F}_2(\varphi)(X_e) && \text{(by definition)} \end{aligned}$$

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

$$\begin{aligned} \mathscr{I}(X_f) &\supseteq \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \mathscr{I}(X_g), \mathscr{I}(X_g) \supseteq \mathscr{I}(X_f)\} \quad \text{(by [APP}_{\rhd_3}\text{])} \\ &\supseteq \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \varphi(X_{e'})\} \\ &\quad \text{(because "}g \rhd_3 e'\text{" occurs and by IH)} \\ &= \mathscr{F}_2(\varphi)(X_e) \quad \text{(by definition).} \end{aligned}$$

$$\begin{aligned} \mathscr{I}(P_f) &\supseteq \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \mathscr{I}(P_g), \mathscr{I}(X_g) \supseteq \mathscr{I}(X_f)\} \quad \text{(by [APP}_{\rhd_3}\text{])} \\ &\supseteq \{\kappa \mid \lambda_g x_\tau.e' \in Lam(e_1), \kappa \in \varphi(P_{e'})\} \\ &\quad \text{(because "}g \rhd_3 e'\text{" occurs and by IH)} \\ \mathscr{I}(P_f) &\supseteq \varphi(P_1) \quad \text{(because "}f \rhd_3 e_1\text{" occurs and by IH)} \\ \mathscr{I}(P_f) &\supseteq \varphi(P_2) \quad \text{(because "}f \rhd_3 e_2\text{" occurs and by IH)} \end{aligned}$$

Therefore,

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

[RS] $e = f \rhd_3$ `raise` e_1.

$$\begin{aligned} \mathscr{I}(P_f) &\supseteq \mathscr{I}(X_f) && \text{(by [RS}_{\rhd_3}\text{])} \\ &\supseteq \varphi(X_1) && \text{(because "}f \rhd_3 e_1\text{" occurs and by IH)} \\ &= \mathscr{F}_2(\varphi)(P_e) && \text{(by definition).} \end{aligned}$$

[−RS] $e =$ `-raise` $e_1\ \kappa_1 \cdots \kappa_n$ where $type(e_1) = \tau'\ exn \wedge isExn(\tau')$.

$$\begin{aligned} \mathscr{I}(P_f) &\supseteq \mathscr{I}(X_f \backslash_{e_1} \{\kappa_1, \ldots, \kappa_n\}) && \text{(by [}-\text{RS}_{\rhd_3}\text{])} \\ &= \mathscr{I}(X_f) && \text{(by definition of } \backslash_{e_1}\text{)} \\ &\supseteq \varphi(X_1) && \text{(because "}f \rhd_3 e_1\text{" occurs and by IH)} \\ &= \mathscr{F}_2(\varphi)(P_e) && \text{(by definition).} \end{aligned}$$

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

$$
\begin{aligned}
\mathscr{I}(P_f) &\supseteq \mathscr{I}(X_f \backslash_{e_1} \{\kappa_1, \ldots, \kappa_n\}) && \text{(by [}-\text{RS}_{\rhd_3}\text{])}\\
&= \mathscr{I}(X_f) \backslash \{\kappa_1, \ldots, \kappa_n\} && \text{(by definition of } \backslash_{e_1}\text{)}\\
&\supseteq \varphi(X_1) \backslash \{\kappa_1, \ldots, \kappa_n\} && \text{(because ``} f \rhd_3 e_1\text{'' occurs and by IH)}\\
&= \mathscr{F}_2(\varphi)(P_e) && \text{(by definition).}
\end{aligned}
$$

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

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

[+RS] $e = $ `+raise` $e_1\ \kappa_1 \cdots \kappa_n$ where $type(e_1) = \tau'\ exn \wedge \neg isExn(\tau')$.

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

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

$$
\begin{aligned}
\mathscr{I}(X_f) &\supseteq \{\kappa \mid \kappa \in \mathscr{I}(X_h), \mathscr{I}(X_h) \supseteq \mathscr{I}(P_g)\} \cap \mathscr{I}(X_g) \quad \text{(by [HNDL}_{\rhd_3}\text{])}\\
&= \mathscr{I}(X_h) \cup \mathscr{I}(X_g)\\
&\qquad \text{(because any model } \mathscr{I} \text{ of } \mathscr{C}_3 \text{ satisfies } \mathscr{I}(X_h) \supseteq \mathscr{I}(P_g))\\
&\supseteq \varphi(X_2) \cup \mathscr{I}(X_g) \quad \text{(because ``} h \rhd_3 e_2\text{'' occurs and by IH)}\\
&\supseteq \varphi(X_2) \cup \varphi(X_g) \quad \text{(because ``} g \rhd_3 e_g\text{'' occurs and by IH)}\\
&= \mathscr{F}_2(\varphi)(X_e) \qquad \text{(by definition).}
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{I}(P_f) &\supseteq \{\kappa \mid \kappa \in \mathscr{I}(P_h), \mathscr{I}(X_h) \supseteq \mathscr{I}(P_g)\} \quad \text{(by [HNDL}_{\rhd_3}\text{])}\\
&= \mathscr{I}(P_h) \quad \text{(because any model } \mathscr{I} \text{ of } \mathscr{C}_3 \text{ satisfies } \mathscr{I}(X_h) \supseteq \mathscr{I}(P_g))\\
&\supseteq \varphi(P_2) \quad \text{(because ``} h \rhd_3 e_2\text{'' occurs and by IH)}\\
&= \mathscr{F}_2(\varphi)(P_e) \quad \text{(by definition).} \qquad \square
\end{aligned}
$$

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

```
(1) fun m() = f(exn κ 1)
(2) fun f(x) = handle g(x) λh y.1   (in SML g(x) handle _ ⇒ 1)
(3) fun g(x) = raise x
```

From line (1),

$$X_m \supseteq \kappa$$

$$X_f \supseteq X_m, X_m \supseteq X_f \quad (\text{from } X_m \supseteq app_X(f, X_m))$$

$$P_m \supseteq P_f \quad (\text{from } P_m \supseteq app_P(f, X_m))$$

From line (2)

$$X_g \supseteq X_f, X_f \supseteq X_g \quad (\text{from } X_f \supseteq app_X(g, X_f))$$

$$P_f \supseteq P_h, X_h \supseteq P_g \quad (\text{from } P_f \supseteq app_P(h, P_g))$$

From line (3)

$$P_g \supseteq X_g$$

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

$$X_m = \{\kappa\}, \quad X_f = X_m \cup X_g, \quad X_g = X_f, \quad X_h = P_g,$$

$$P_m = P_f, \quad P_f = P_h, \quad P_g = X_g.$$

The least solution is

$$X_m = \{\kappa\}, \quad X_f = \{\kappa\}, \quad X_g = \{\kappa\},$$

$$P_m = \emptyset, \quad P_f = \emptyset, \quad P_g = \{\kappa\}.$$

3.5. Typeful constraints for improved accuracy

Some constraint rules of $\triangleright_3$ can be safely sharpened using types. Our actual analysis 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:

$$\mathscr{I}(app_X(e_1, \mathscr{X})) = \{\kappa \mid \lambda_f x_\tau.e \in Lam(e_1),\ \kappa \in \mathscr{I}(X_f \mid isExn(type(e))),$$

$$\mathscr{I}(X_f) \supseteq \mathscr{I}(\mathscr{X} | isExn(\tau))\}$$

$$\mathscr{I}(app_P(e_1, \mathscr{X})) = \{\kappa \mid \lambda_f x_\tau.e \in Lam(e_1),\ \kappa \in \mathscr{I}(P_f),\ \mathscr{I}(X_f) \supseteq \mathscr{I}(\mathscr{X} \mid isExn(\tau))\}$$

$$\mathscr{I}(var(x)) = \mathscr{I}(X_{Owner(x)} | isExn(type(x)))$$

$$X_f \supseteq app_X(\cdots) \cup (X_g | isExn(type(e_g))) \quad \text{in } [\mathrm{HNDL}_{\triangleright_3}]$$

where $\mathscr{I}(\mathscr{X}|cond) = \mathscr{I}(\mathscr{X})$ 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:

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

Example 2. Consider the following program:
(1) fun f(x) = $\cdots$ (exn κ_1 1)$\cdots$g(1)$\cdots$
(2) fun g(x) = raise (exn κ_2 x)

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

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

From line (2),

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

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.

Similar techniques of type-directed improvement of analyses have been reported: accuracy improvement of control flow analysis [11] and stratification of alias analysis [15].

3.6. Handling of exception's arguments

Because the analysis does not recognize exception's arguments 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 for 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.

Our analysis reports the pair of an exception name κ and the index of the expression ($\texttt{exn}\ \kappa\ e$) where the exception is made.[7] Given a pair of exception name and its birth place information, the programmer can decide which may-uncaught exceptions are real, assuming that the birth place expression has the argument data explicit in the text. In the above example, the programmer safely decides the may-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:

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

The rule for `exn` expression becomes

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

And

$$X_1 \backslash_e \{\kappa_1, \ldots, \kappa_n\} \quad \text{and} \quad X_1 \cap_e \{\kappa\}$$

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

3.7. Adapting $\rhd_3$ to Standard ML

Because of SML's polymorphic types,

- *isExn* needs to be conservative. The new definition is:

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

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

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

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

program	lines	cfa + setup(sec)[a]	solve(sec)[b]	analysis result
Knuth-Bendix.sml	519	0.54[c]	0.07[d]	1 (1x,1r,10h)[e]
ml-lex.sml	1204	0.89	0.47	3 (10x,19r,10h)
instantiate.sml	1384	2.74	0.04	2 (7x,8r,18h)
typecheck.sml	648	6.07	0.03	0 (1x,2r,17h)
moduleutil.sml	847	4.37	0.08	3 (3x,25r,23h)
pathname.sml	426	0.09	0.01	4 (4x,6r,3h)
string-cvt.sml	454	0.13	0.03	1 (1x,10r,4h)
class compiler	3511	3.65	0.10	3 (11x,34r,4h)

[a] Control-flow analysis and constraints set-up: in SML, run on DEC Alpha Server1000(4/200), compiled by SML/NJ 108.13
[b] Solving constraints: in SML, run on DEC Alpha Server1000(4/200), compiled by SML/NJ 108.13
[c] SML user+system+gc time
[d] C user+system time
[e] I may un-caught exceptions from top-level functions among 1 exns(1x), 1 raise exprs(1r), and 10 handlers(10h).

Fig. 8. Experimental results.

- *Lam*'s 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 [12, 13, 19].

4. Experimental results

A prototype's preliminary performance is shown in Fig. 8.

Currently, the analysis speed ranges from 110 to 4000 SML-lines/s ([20] ran at 0.2 SML-lines/s and [5] at about 10 SML-lines/s). 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.

[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.

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 3 may-uncaught exceptions are exactly those that can really escape.

5. Conclusion

We found that even though the exception flow and control flow are in general intertwined in SML 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. Our early experimental evidence suggests that this separation is not detrimental to the accuracy of the exception analysis, while it makes the analysis significantly faster than the earlier methods. We are optimistic that we are near to a right balance of the cost-accuracy performance.

We showed 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 were done by showing the consistencies between the three constraint systems. We used the fixpoint induction for continuous functions that were derived from the constraint rules [4]. Our method may be seen as a kind of abstract interpretation [3]. This paper's technique for enlarging the constraint granularity and proving its consistency with smaller-grained constraint systems can be applied to other analysis problems where the data to analyze are sparse in programs.

We are currently working on analyzing SML modules in isolation, which will be the last thing to make the analysis realistic.

Acknowledgements

We thank Dave MacQueen for his encouragement and keen interest in this work.

References

[1] A. Aiken, N. Heintze, Constraint-based program analysis, Tutorial ACM Symp. on Principles of Programming Languages, January 1995.

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

[3] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, ACM Symp. on Principles of Programming Languages, 1977, pp. 238–252.

[4] P. Cousot, R. Cousot, Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, Proc. 7th Internat. Conf. on Computer-Aided Verification, Lecture Notes in Computer Science, Vol. 939, Springer, Berlin, 1995, pp. 293–308.

[5] M. Fahndrich, A. Aiken, Making set-constraint program analyses scale, Workshop on Set Constraints, August 1996.

[6] N. Heintze, Set based program analysis, Ph.D. Thesis, Carnegie Mellon University, October 1992.
[7] N. Heintze, Set based analysis of ML programs, Tech. Report CMU-CS-93-193, Carnegie Mellon University, July 1993.
[8] N. Heintze, J. Jaffar, A decision procedure for a class of set constraints, Tech. Report CMU-CS-91-110, Carnegie-Mellon University, February 1991.
[9] N. Heintze, D. McAllester, Linear-time subtransitive control flow analysis, Proc. SIGPLAN Conf. on Programming Language Design and Implementation, June 1997, pp. 261–272.
[10] S. Jagannathan, A. Wright, Flow-directed inlining, Proc. SIGPLAN Conf. on Programming Language Design and Implementation, May 1996, pp. 193–205.
[11] S. Jagannathan, S. Weeks, A. Wright, Type-directed flow analysis for typed intermediate languages, Proc. 4th Internat. Static Analysis Symp., Lecture Notes in Computer Science, Vol. 1302, Springer, Berlin, 1997, pp. 232–249.
[12] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (1978) 348–375.
[13] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Defintion of Standard ML (Revised), MIT Press, Cambridge, MA, 1997.
[14] J. Palsberg, M.I. Schwartzbach, Safety analysis versus type inference, Inform. and Comput. 118 (1) (1992) 128–141.
[15] E. Ruf, Partitioning dataflow analyses using types, ACM Symp. on Principles of Programming Languages, January 1997, pp. 15–26.
[16] O. Shivers, Control-flow analysis of higher-order languages, Ph.D. Thesis, Carnegie Mellon University, May 1991, Tech. Report CMU-CS-91-145.
[17] J.E. Stoy, Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, MA, 1977.
[18] Y.M. Tang, P. Jouvelot, Separate abstract interpretation for control-flow analysis, Proc. Theoretical Aspect Comput. Sci., Lecture Notes in Computer Science, Vol. 789, Springer, Berlin, 1994, pp. 224–243.
[19] M. Tofte, Type inference for polymorphic references, Inform. and Comput. 89 (1990) 1–34.
[20] K. Yi, Compile-time detection of uncaught exceptions for Standard ML programs, Proc. 1st Internat. Static Analysis Symp., Lecture Notes in Computer Science, Vol. 864, Springer, Berlin, 1994, pp. 238–254.