Towards a Cost-effective Estimation of Uncaught Exceptions in SML Programs*

[The 4th International Static Analysis Symposium, Paris, Sept. 1997]

Kwangkeun Yi and Sukyoung Ryu

{kwang,puppy}@cs.kaist.ac.kr
Dept. of Computer Science
Korea Advanced Institute of Science and Technology(KAIST)

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 SML's only one "safety hole."
Even though SML program's control flow and exception flow are in general mutually dependent, analyzing the two flows are safely decoupled. Program's control-flow is firstly estimated from a set of equations defined 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.
A preliminary 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 4.58 seconds and it reports 4 may-uncaught exceptions, among which 3 exceptions 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 to 7 seconds) and to analyze modules in isolation.

1 Introduction

Exception handling facilities in Standard ML [MTH90] 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.

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 [Ar996].

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.

* 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, and by Samsung Electronics Corp.

1.1 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) = ···`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.

Caveat: Our analysis cannot analyze programs that utilize the generative nature of the exception (and the datatype) declarations. This limitation is not severe; exceptions (and datatypes) are largely declared at the global scope[2] or at a module level, or we can move existing local declarations out to the global level without affecting the "observational" behavior of the programs. Programs where this hoisting is impossible cannot be analyzed correctly by our analysis.

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

1.2 Our Approach

In the earlier work [Yi94], 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 one hour to analyze the ML-Lex program, for example.

For a better cost-effective analysis, we surveyed SML codes and found that such a full-fledged analysis may be an overkill in almost all cases. In particular, we found that such case as (*) almost never happened (Figure 1)[3]. This suggests that, in most cases, the call-graph estimation can be done independent of the exception analysis. Preparing for the rare case that exceptions carry functions would not pay-off in practice.

This does *not* mean that we don't guarantee the safety of our call-graph estimation. For such cases when functions to call are brought by uncaught exceptions, we choose to do a crude approximation, believing that this "large" approximation would be rarely detrimental to the call-graph accuracy. Please note that we cannot use standard techniques (e.g., 0CFA) for closure analysis [Shi91, Hei93, PS92, JW96] because their correctness does not consider languages with function-carrying exceptions.

---

[2] In CAML, only top-level exception declarations are permitted.

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

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

Program's call-graph is defined as the least solution satisfying a set of simultaneous equations. These equations are defined by case analysis of function expressions. For example, "`x(0)`" calls functions that are bound to `x` when λ`x`.e is called (programs are alpha-converted), "`(f 0) 1`" calls functions `(f 0)` that the `f`'s body (a function expression) represents. The crude approximation is when an exception's argument is the function to call. In this case we look up the places where exceptions are made and collect functions that are paired at those places. To fine-tune this approximation, among such functions we collect those whose types unify with the call expression's function type. The exact definition is in Proposition 4, after we fix the source language.

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

- One class is for set P_f of `f`'s uncaught exceptions[4]. For example, P_f of the following function

  `fun f(x) =` e`(0)+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.[5] For example, P_f of the following function

  `fun f(x) = raise x`

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

[4] In SML, uncaught exceptions are called exception packets. Hence "P_f."
[5] A value v is *available* in a function f iff, during program execution, the v is a result of a sub-expression of the function body.

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

$$
\begin{array}{rll}
\tau ::= & \tau\ \mathit{exn} & \text{exception type with argument type } \tau \\
| & \tau \rightarrow \tau & \text{function type} \\
| & \iota & \text{constant type}
\end{array}
$$

Fig. 2. L's Abstract Syntax

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

1. The solution of P_f for each top-level function f. The existence of such exceptions implies that the program can terminate abnormally.
2. Uncaught exceptions from each handle expression. From this information the programmer can check the completeness of the handler patterns.

2 Language L

For presentation brevity we present our analysis for an imaginary language L. The language is a monomorphically typed, call-by-value, high-order language. The language L's abstract syntax is shown in Figure 2. The usual monomorphic type rules are omitted.

Values in L are either exceptions or functions. An exception value "$\texttt{exn}\ \kappa\ e$" is constructed from an exception name κ and an expression e 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. Otherwise, e_1's normal value is returned.[6]

[6] Note that this handle expression can handle only one type of exceptions. To be more realistic, handle expression must have multiple functions ($\texttt{handle}\ e_1\ [\lambda x_{\tau_i}.e_2]^+$) for different exception types τ_i. Considering such handle expression, however, unnecessarily complicates our presentation.

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

Definition 1 **(*type*(e)).** For a program p, $type_p(e)$ denotes the type of an expression e in p. We write $type(e)$ when it is clear from the context which program the expression e belongs to.

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 Figure 2. For convenience we use numbers, multiple branches with the wild pattern, and exceptions with no argument.)

`-raise` and `+raise` expressions are used inside the handler expression to make the exception-handling situation explicit in the text, which facilitates our analysis design.

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

```
                          handle e λx_exn.
e handle                    case x
  ERROR => 1        is        ERROR 1
| FAIL => 2         ==>       FAIL  2
                              _ -raise x ERROR FAIL
```

  "`-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`:

```
                                handle e λx_(ιlist)exn.
                                  case x
exception E of int list             E (λy_ιlist.
...                       is               case y
e handle E nil => 1       ==>                NIL 1
                                             _ +raise x E) (decon x)
                                    _ -raise x E
```

  Above program's handler can handle exception `E` only with `nil` list. Program in L makes this situation explicit by re-raising the `E` exceptions if their arguments are non-empty list. "`+raise` x `E`" indicates the re-raised exception shall only be the `E` exceptions.
- Exception constructors that need arguments are translated into a function, which is β-reduced whenever appropriate. For example,

```
exception E of int     is
··· E, ···             ==> ··· (λx_ι.(exn E x)), ···
```

- Every functor applications are in-lined. That is, functor definitions and applications disappear and are replaced by in-lined structures.[7]

[7] 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 Set-Constraint Systems

Our exception analysis is presented in the set-constraint framework[Hei92, AH95]. We use this framework not because we will use its computation method (infinite solution as a grammar form) but because the modular nature of the constraints makes our presentation easy. Our exception analysis is computed by the conventional iterative fixpoint method because our solution space is finite: exception names in the program. Correctness proves are done by the fixpoint induction over the continuous functions that are derived [CC95] from our constraint systems.

We present three set-constraint systems: $\triangleright_1$, $\triangleright_2$, and $\triangleright_3$. Our analysis is the last one $\triangleright_3$. 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.

In presenting the set-constraint rules, we assume that 1) L programs are well-typed and 2) all variables in L programs are uniquely named.

To review some notions of set constraint formalism, an interpretation $\mathcal{I}$ is a *model* of a conjunction $\mathcal{C}$ of constraints if, for each constraint $\mathcal{X} \supseteq se$ (set variable $\mathcal{X}$ and set expression se) in $\mathcal{C}$, $\mathcal{I}(se)$ is defined and $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$. We write $lm(\mathcal{C})$ for the least model of $\mathcal{C}$. All the 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 [Hei92].

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 Figure 3 we show only three cases: application, raise, and handle expression. Other cases are straightforward adaptation of [Hei93]. We index V and P sometimes with expressions, sometimes with numbers. The subscript "e" of set variables "V_e" and "P_e" denotes the current expression to which the current rule applies. For constraint variables for e's i-th sub-expression, we write "V_i" as a shorthand for "V_{e_i}".

Our analysis could have been computing the solution of the $\triangleright_1$ constraints by using Heintze's method [Hei93]. 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, $\triangleright_2$ and $\triangleright_3$.

Proposition 2 (Typefulness of $\triangleright_1$). *For a closed term e, let $\triangleright_1 e{:}\mathcal{C}_1$. Then its least model $lm(\mathcal{C}_1)$ preserves types: $\forall e_i \in e.lm(\mathcal{C}_1)(V_i) \subseteq Val_{type(e_i)}$ (the set of values of $type(e_i)$).*

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

$$\begin{array}{llll}
\mathcal{I}(V_e) & \subseteq \mathit{Val} & \mathcal{I}(P_e) & \subseteq \mathit{Packet} \\
\mathcal{I}(\lambda x_\tau.e) & = \{\lambda x_\tau.e\} & \mathcal{I}(se \cup se') & = \mathcal{I}(se) \cup \mathcal{I}(se')
\end{array}$$

$$\begin{array}{ll}
\mathcal{I}(\mathit{app}_V(V_1)) & = \{v \mid \lambda x_\tau.e \in \mathcal{I}(V_1), v \in \mathcal{I}(V_e)\} \\
\mathcal{I}(\mathit{app}_P(V_1)) & = \{\kappa\, v \mid \lambda x_\tau.e \in \mathcal{I}(V_1), \kappa\, v \in \mathcal{I}(P_e)\} \\
\mathcal{I}(\texttt{-}\mathit{raise}(V_1, \kappa_1, \cdots, \kappa_n)) & = \{\kappa' v \mid \kappa' v \in \mathcal{I}(V_1), \forall i.\kappa_i \neq \kappa'\} \\
\mathcal{I}(\texttt{+}\mathit{raise}(V_1, \kappa)) & = \{\kappa\, v \mid \kappa\, v \in \mathcal{I}(V_1)\}
\end{array}$$

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

$$[\mathrm{RS}_{\rhd_1}] \quad \frac{\rhd_1\ e_1\colon \mathcal{C}_1}{\rhd_1\ \texttt{raise}\, e_1\colon \{P_e \supseteq V_1 \cup P_1\} \cup \mathcal{C}_1}$$

$$[\text{-}\mathrm{RS}_{\rhd_1}] \quad \frac{\rhd_1\ e_1\colon \mathcal{C}_1}{\rhd_1\ \texttt{-raise}\, e_1\, \kappa_1 \cdots \kappa_n\colon \{P_e \supseteq \texttt{-}\mathit{raise}(V_1, \kappa_1, \cdots, \kappa_n) \cup P_1\} \cup \mathcal{C}_1}$$

$$[\text{+}\mathrm{RS}_{\rhd_1}] \quad \frac{\rhd_1\ e_1\colon \mathcal{C}_1}{\rhd_1\ \texttt{+raise}\, e_1\, \kappa\colon \{P_e \supseteq \texttt{+}\mathit{raise}(V_1, \kappa) \cup P_1\} \cup \mathcal{C}_1}$$

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

Fig. 3. Constructing Concrete Constraints: $\rhd_1$

This typefulness is important for the consistency of the forthcoming constraint system $\rhd_2$.

The correctness of $\rhd_1$ (i.e., the least model of $\rhd_1$-constraints for a program includes the actual values and uncaught exceptions) is assumed; the proof must not be much different from those outlined in [Hei93], using a "set-based operational semantics" as an intermediate form.

3.2 Separate Call-graph Estimation

Definition 3. *Lam* is safe with respect to $\rhd_1$ iff $\rhd_1\ e\colon \mathcal{C}$ implies that, for every function-typed expression e' of the program e, $\mathit{Lam}(e') \supseteq \mathit{lm}(\mathcal{C})(V_{e'})$.

Proposition 4 (A safe call table *Lam*). *Given a program e, let FtnExpr be the set of function-typed expressions in e, and 2^{Lambda} be the powerset of lambda expressions in e. Define*

$$\mathit{Lam}\colon \mathit{FtnExpr} \to 2^{\mathit{Lambda}}$$

to be the least table satisfying the following equations:

$$
\begin{array}{lll}
Lam(\lambda x_\tau.e) & = \{\lambda x_\tau.e\} & \textit{constant function} \\
Lam(f) & = \{\lambda x_\tau.e \mid \mathtt{fix}\ f\ \lambda x_\tau.e\ \textit{in program}\} & \textit{named function} \\
Lam(x) & = \cup\{Lam(e_2) \mid e_1\ e_2\ \textit{in program} \wedge \lambda x_\tau.e \in Lam(e_1)\} & \textit{ftn var} \\
Lam(e_1\ e_2) & = \cup\{Lam(e) \mid \lambda x_\tau.e \in Lam(e_1)\} & \textit{function from application} \\
\multicolumn{3}{l}{Lam(\mathtt{decon}\ e) = \cup\{Lam(e') \mid \mathtt{exn}\ \kappa\ e'\ \textit{in program} \wedge \textit{type}(e') = \textit{type}(\mathtt{decon}\ e)\}} \\
\multicolumn{3}{l}{Lam(\mathtt{fix}\ f\ \lambda x_\tau e_1\ \mathtt{in}\ e_2) = Lam(e_2)} \\
\multicolumn{3}{l}{Lam(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3) = Lam(e_2) \cup Lam(e_3)} \\
\multicolumn{3}{l}{Lam(\mathtt{handle}\ e_1\ \lambda_\tau.e_2) = Lam(e_1) \cup Lam(e_2)}
\end{array}
$$

Then Lam is safe with respect to $\triangleright_1$.

Note that the last case: if functions to call are the arguments of exceptions then we look up the exception construction expressions in the program and collect functions that are paired. Among those exception expressions we selectively choose those whose argument type is equal to the target function. (Note that the L language is monomorphic. In SML, the "is equal to" must be "unifies with.")

3.3 Exception Constraint Construction $\triangleright_2$

We now consider a new system $\triangleright_2$(Figure 4) 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: $X_e \supseteq se$ (for its exception values) and $P_e \supseteq se$ (for uncaught exceptions during its evaluation). For solutions of X_e and P_e we will consider only exception names. That is, X_e is for the projection $|V_e|$ of e's values V_e:

$$\mathcal{I}(X_e) \supseteq |V_e| \stackrel{\text{def}}{=} \{\kappa \mid \kappa\ v \in V_e\} \cup |\{v \mid \kappa\ v \in V_e\}|.$$

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: $FtnExpr \to 2^{Lambda}$(Proposition 4).

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

Consider the rule for `-raise` expression:

$$[\text{-RS}_{\triangleright_2}]\ \frac{\triangleright_2\ e_1 : \mathcal{C}_1}{\triangleright_2\ \mathtt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n : \{P_e \supseteq (X_1 \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\}) \cup P_1\} \cup \mathcal{C}_1}$$

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

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

If an exception can have other exceptions as its arguments then the exclusion $\setminus_e$ has no effect. If blindly excluded, exceptions that are hidden as arguments of the escaping exception are considered caught. This makes the analysis unsafe. (The same reason is for the definition of $\cap_e$.)

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

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

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

$$\begin{array}{ll} \mathcal{I}(var(x)) & = \{\kappa \mid \text{“}e_1\ e_2\text{”} \in e, \lambda x_\tau.\bullet \in Lam(e_1), \kappa \in \mathcal{I}(X_2)\} \\ \mathcal{I}(app_X(e_1)) & = \{\kappa \mid \lambda x_\tau.e \in Lam(e_1), \kappa \in \mathcal{I}(X_e)\} \\ \mathcal{I}(app_P(e_1)) & = \{\kappa \mid \lambda x_\tau.e \in Lam(e_1), \kappa \in \mathcal{I}(P_e)\} \\ isExn(\tau) & = \text{true iff } \tau = \tau'\ exn \\ \mathcal{I}(X_1 \setminus_e \{\kappa_1, \cdots, \kappa_n\}) & = \begin{cases} \mathcal{I}(X_1) & \text{if } type(e) = \tau'\,exn \ \wedge\ isExn(\tau') \\ \mathcal{I}(X_1) \setminus \{\kappa_1, \cdots, \kappa_n\} & \text{otherwise} \end{cases} \\ \mathcal{I}(X_1 \cap_e \{\kappa\}) & = \begin{cases} \mathcal{I}(X_1) & \text{if } type(e) = \tau'\,exn \ \wedge\ isExn(\tau') \\ \mathcal{I}(X_1) \cap \{\kappa\} & \text{otherwise} \end{cases} \\ \mathcal{I}(\kappa) & = \{\kappa\} \end{array}$$

$\mathcal{I}(\lambda x_\tau.e)$ and $\mathcal{I}(se \cup se)$ are the same as in $\rhd_1$ (Figure 3).

$[\mathrm{VAR}_{\rhd_2}]\ \rhd_2\ x\colon \{X_x \supseteq var(x)\} \qquad [\mathrm{C}_{\rhd_2}] \qquad \rhd_2\ \mathtt{1}\colon \emptyset$

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

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

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

$$[\mathrm{APP}_{\rhd_2}]\ \frac{\rhd_2\ e_1\colon \mathcal{C}_1 \qquad \rhd_2\ e_2\colon \mathcal{C}_2}{\rhd_2\ e_1\ e_2\colon \{X_e \supseteq app_X(e_1), P_e \supseteq app_P(e_1) \cup P_1 \cup P_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

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

$$[\mathrm{RS}_{\rhd_2}]\ \frac{\rhd_2\ e_1\colon \mathcal{C}_1}{\rhd_2\ \mathtt{raise}\ e_1\colon \{P_e \supseteq X_1 \cup P_1\} \cup \mathcal{C}_1}$$

$$[\text{-}\mathrm{RS}_{\rhd_2}]\ \frac{\rhd_2\ e_1\colon \mathcal{C}_1}{\rhd_2\ \mathtt{-raise}\ e_1\ \kappa_1 \cdots \kappa_n\colon \{P_e \supseteq (X_1 \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\}) \cup P_1\} \cup \mathcal{C}_1}$$

$$[\text{+}\mathrm{RS}_{\rhd_2}]\ \frac{\rhd_2\ e_1\colon \mathcal{C}_1}{\rhd_2\ \mathtt{+raise}\ e_1\ \kappa\colon \{P_e \supseteq (X_1 \cap_{e_1} \{\kappa\}) \cup P_1\} \cup \mathcal{C}_1}$$

$$[\mathrm{HNDL}_{\rhd_2}]\ \frac{\rhd_2\ e_1\colon \mathcal{C}_1 \qquad \rhd_2\ e_2\colon \mathcal{C}_2}{\begin{array}{l} \rhd_2\ \mathtt{handle}\ e_1\ \lambda x_\tau.e_2\colon \{\ X_e \supseteq app_X(\lambda x_\tau.e_2) \cup X_1, \\ \qquad\qquad P_e \supseteq app_P(\lambda x_\tau.e_2), X_x \supseteq P_1 \\ \qquad \} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Fig. 4. Constructing Exception Constraints: $\rhd_2$

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

$$\mathcal{I}_2(X_i) \supseteq |\mathcal{I}_1(V_i)| \quad \text{and} \quad \mathcal{I}_2(P_i) \supseteq |\mathcal{I}_1(P_i)|$$

where $|\mathcal{V}|$ is the set of exception names constituting exception values in $\mathcal{V}$

$$|\mathcal{V}| \stackrel{\text{def}}{=} \{\kappa \mid \kappa\, v \in \mathcal{V}\} \cup |\{v \mid \kappa\, v \in \mathcal{V}\}|.$$

3.4 Function's Exception Constraint Construction $\rhd_3$

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 $\rhd_2$ as our stepping stone, we arrive at our constraint system $\rhd_3$(Figure 5) 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 $\rhd_3$-constraints for an input program is our analysis result: uncaught exceptions from each function.

In $\rhd_3$, set variables are indexed by the lambda and handlee expressions of the input program. We assume that all lambda 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 "`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 in f, and P_f is for uncaught exceptions during the call to f.

Consider the rule for application expression:

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

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

One noticeable rule is $[\text{VAR}_{\rhd_3}]$. Because the constraint granularity is a function, constraint for a variable x must be expressed in terms of two functions: function f that variable x provides with exceptions and another function g that provides x with exceptions. The f is the function that appears in the left-hand-side of "$\rhd_3\ x$" and the g is the function $(Owner(x))$ that has x as its argument. Therefore,

$$[\text{VAR}_{\rhd_3}] \quad f\ \rhd_3\ x\colon \{X_f \supseteq X_{Owner(x)}\}.$$

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

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

$$\begin{array}{lll} \kappa \in \mathit{Exn} & = \{\kappa_1, \cdots, \kappa_N\} & \text{exception constructors} \\ \kappa \in \mathit{Packet} & = \mathit{Exn} & \text{raised exceptions} \end{array}$$

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

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

$\mathcal{I}(X_f \setminus_e \{\kappa_1, \cdots, \kappa_n\})$, $\mathcal{I}(X_f \cap_e \{\kappa\})$, $\mathcal{I}(\kappa)$, and $\mathcal{I}(\lambda x_\tau.e)$ are the same as in $\rhd_2$ (Figure 4)

$[\mathrm{VAR}_{\rhd_3}]$ $\quad f \ \rhd_3\ x\colon \{X_f \supseteq \mathit{var}(x)\}$ $\qquad$ $[\mathrm{C}_{\rhd_3}]$ $\quad f\ \rhd_3\ \mathtt{1}\colon \emptyset$

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

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

$$[\mathrm{APP}_{\rhd_3}]\ \frac{f\ \rhd_3\ e_1\colon \mathcal{C}_1 \qquad f\ \rhd_3\ e_2\colon \mathcal{C}_2}{f\ \rhd_3\ e_1\, e_2\colon \{X_f \supseteq \mathit{app}_X(e_1, X_f), P_f \supseteq \mathit{app}_P(e_1, X_f)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

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

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

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

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

$$[\mathrm{HNDL}_{\rhd_3}]\ \frac{g\ \rhd_3\ e_g\colon \mathcal{C}_1 \qquad h\ \rhd_3\ e_2\colon \mathcal{C}_2}{\begin{array}{l} f\ \rhd_3\ \mathtt{handle}\, e_g\, \lambda_h x_\tau.e_2\colon \{\ X_f \supseteq \mathit{app}_X(\lambda_h x_\tau.e_2, P_g) \cup X_g, \\ \qquad\qquad\qquad\qquad\qquad\quad P_f \supseteq \mathit{app}_P(\lambda_h x_\tau.e_2, P_g) \\ \qquad\qquad\qquad\qquad\quad\ \} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Fig. 5. Constructing Function's Exception Constraints: $\rhd_3$

Proposition 6 (Correctness of $\rhd_3$). *For a closed term e, let $\rhd_2\ e\colon \mathcal{C}_2$ and* $\mathtt{main}\ \rhd_3\ e\colon \mathcal{C}_3$ *with their least models, $\mathcal{I}_2 = \mathit{lm}(\mathcal{C}_2)$ and $\mathcal{I}_3 = \mathit{lm}(\mathcal{C}_3)$. Then, if $g \rhd_3 e'\colon \mathcal{C}'$ occurs during* $\mathtt{main}\ \rhd_3\ e\colon \mathcal{C}_3$ *then*

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

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
```

$$
\begin{array}{ll}
\text{From line (1)} & \left\{\begin{array}{ll} X_m \supseteq \kappa & \\ X_f \supseteq X_m,\ X_m \supseteq X_f & (\text{from } X_m \supseteq \mathit{app}_X(f, X_m)) \\ P_m \supseteq P_f & (\text{from } P_m \supseteq \mathit{app}_P(f, X_m)) \end{array}\right. \\
\text{From line (2)} & \left\{\begin{array}{ll} X_g \supseteq X_f,\ X_f \supseteq X_g & (\text{from } X_f \supseteq \mathit{app}_X(g, X_f)) \\ P_f \supseteq P_h,\ X_h \supseteq P_g & (\text{from } P_f \supseteq \mathit{app}_P(h, P_g)) \end{array}\right. \\
\text{From line (3)} & P_g \supseteq X_g
\end{array}
$$

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

$$X_m = \{\kappa\}, X_f = X_m \cup X_g, X_g = X_f, X_h = P_g, P_m = P_f, P_f = P_h, P_g = X_g.$$

The least solution is:

$$X_m = \{\kappa\}, X_f = \{\kappa\}, X_g = \{\kappa\}, P_m = P_f = \emptyset, P_g = \{\kappa\}.$$

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:

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

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

The correctness of this new $\triangleright_3$ can be proved with respect to a typeful version of $\triangleright_2$ that 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 $[\mathrm{DCON}_{\triangleright_2}]$ rule:

$$[\mathrm{DCON}_{\triangleright_2}]\ \frac{\triangleright_2\ e_1 \colon \mathcal{C}_1}{\triangleright_2\ \mathtt{decon}\, e_1 \colon \{X_e \supseteq (X_1 | \mathit{isExn}(\mathit{type}(e))), P_e \supseteq P_1\} \cup \mathcal{C}_1}$$

3.6 Adapting $\triangleright_3$ to Standard ML

Because of SML's polymorphic types,

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

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

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

program	lines	cfa + setup(sec)[a]	solve(sec)[b]	analysis result
Knuth-Bendix.sml	519	8.62[c]	0.09[d]	1 (1x,1r,10h)[e]
ml-lex.sml	1204	4.58	0.01	4 (10x,19r,10h)
pathname.sml(library)	426	0.34	0.00[f]	4 (4x,6r,3h)
string-cvt.sml(library)	454	0.21	0.00	1 (1x,10r,4h)
class compiler	3511	47.77	0.04	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 C, run on Sparc 20, compiled by GCC
[c] SML user+system+gc time
[d] C user+system time
[e] 1 may-uncaught exceptions from top-level functions among 1 exns(1x), 1 raise exprs(1r), and 10 handlers(10h).
[f] $\leq$ 10 milli-seconds

Fig. 6. Experimental Results

– *Lam*'s last case must test for type unifiability instead of type equality: (In this case, *type*(e) - for an expression e of a program - indicates the SML type of the expression, determined during the program's let-polymorphic type inference[Mil78, MTH90].)

$$\begin{array}{ll} Lam(\texttt{decon}\ e) & = \bigcup\{Lam(e') \mid \texttt{exn}\ \kappa\ e' \text{ in pgm} \wedge type(e') \text{ unifies with } type(\texttt{decon}\ e)\} \\ Lam(\texttt{!}e) & = \bigcup\{Lam(e') \mid \texttt{\_:=}\ e', \texttt{ref}\ e' \text{ in pgm} \wedge type(e') \text{ unifies with } type(\texttt{!}e)\} \\ Lam(\texttt{\#}k\ e) & = \bigcup\{Lam(e_k) \mid \texttt{(}e_1, \cdots, e_n\texttt{)} \text{ in pgm} \wedge type(e_k) \text{ unifies with } type(\texttt{\#}k\ e)\} \end{array}$$

4 Experimental Results

A prototype's preliminary performance is shown in Figure 6.

Currently, the analysis speed ranges from 60 to 2160 SML-lines/sec. ([Yi94] ran at 0.2 SML-lines/sec and [FA96] at about 10 SML-lines/sec). We expect still a considerable improvement 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. We currently investigate a way for adapting Heintze's technique [HM97] of quadruple complexity. For this adaptation we must re-formulate his approach for languages where exceptions and functions are intertwined. Computing the constraints' least solution also uses the conventional fixpoint iteration[8] of cubic complexity.

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

[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.

To our knowledge, the cost-accuracy performance of Guzmán and Suárez [GS94]'s exception type system is not available to compare with ours.

5 Handling of Exception's Arguments

Because our analysis does not recognize exception's arguments unless the arguments were also exceptions, the analysis leads into a too conservative result for some programs.

Example 2. Consider the following program that has no uncaught exception:

```
exception Fail of int
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 and our analysis conclude that `f` has an escaping exception `Fail`.

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

Our proposal is to make $\triangleright_3$ bring extra data by which the programmer trim overly conservative results. We pair an exception name κ with the index of the expression ("$\texttt{exn}\ \kappa\ e$") at which the exception value is made.[9] 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 decide 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 $\triangleright_3$. The exception space becomes the set of tuples: $Exn = \{\kappa_1, \cdots, \kappa_N\} \times Expr$. The rule for `exn` expression becomes:

$$[\text{EXN}_{\triangleright 3}]\ \frac{f\ \triangleright_3\ e_1\colon \mathcal{C}_1}{f\ \triangleright_3\ \texttt{exn}\,\kappa\, e_1\colon \{X_f \supseteq \langle \kappa, e\rangle\} \cup \mathcal{C}_1}\ \text{where } e \equiv \texttt{exn}\,\kappa\, e_1$$

And $X_1 \setminus_e \{\kappa_1, \cdots, \kappa_n\}$ and $X_1 \cap_e \{\kappa\}$ removes (respectively, selects) tuples headed by κ_i's (respectively, by κ).

6 Conclusion

We safely and cost-effectively decoupled the exception flow and control flow analyses. 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

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

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 proves 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 [CC95] (Section A). Our method may be seen as a kind of abstract interpretation [CC77]. 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 currenlty implementing the idea of Section 5 and streamlining the control flow analysis part. Other than the analysis speed, for the analysis to be realistic, it must be able to analyze SML modules in isolation. We are working on our preliminary ideas.

Acknowledgment

We thank Dave MacQueen for his encouragement and keen interest in this work. We thank Nevin Heintze for sending us a draft of [HM97].

References

[AH95] Alex Aiken and Nevin Heintze. Constraint-based program analysis. POPL'95 Tutorial, January 1995.

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

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC95] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308. 1995.

[FA96] Manuel Fahndrich and Alexander Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints*, August 1996.

[GS94] Juan Carlos Guzmán and Ascánder Suárez. A type system for exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1994.

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

[Hei93] Nevin Heintze. Set based analysis of ml programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, July 1993.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[JW96] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–205, May 1996.
[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
[MTH90] Robin Milner, Mads Tofte, and Robert Haper. *The Definition of Standard ML*. MIT Press, 1990.
[PS92] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 1992.
[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.
[Yi94] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Lecture Notes in Computer Science*, volume 864, pages 238–254. Proceedings of the first international static analysis symposium edition, 1994.

A Proof of Proposition 6 (Correctness of $\rhd_3$)

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

Note that the least model $\mathcal{I}_2$ is equivalent to the $\subseteq$-least fixpoint fix$\mathcal{F}_2$ [CC95]. The continuous function $\mathcal{F}_2$ is derived from $\mathcal{C}_2$ as follows:

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

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

Similarly for $\mathcal{F}_2(\varphi)(P_i)$.

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

$$\forall model \mathcal{I} of \mathcal{C}_3.\text{“}f \rhd_3 e_i\colon \mathcal{C}\text{” } occurs\, during(\mathtt{main} \rhd_3 e\colon \mathcal{C}_3) \Rightarrow \mathcal{I}(X_f) \supseteq \varphi(X_i) \wedge \mathcal{I}(P_f) \supseteq \varphi(P_i).$$

Base case $Q(\emptyset)$ trivially holds. We claim that $Q(\mathcal{F}_2(\varphi))$ holds given the induction hypothesis $Q(\varphi)$. We consider only the X_f case for `handle`. Other cases are similarly done.

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

$$\begin{array}{lll}
\mathcal{I}(X_f) & \supseteq \{\kappa \mid \kappa \in \mathcal{I}(X_h), \mathcal{I}(X_h) \supseteq \mathcal{I}(P_g)\} \cup \mathcal{I}(X_g) & (\text{by [HNDL}_{\rhd_3}]) \\
& = \mathcal{I}(X_h) \cup \mathcal{I}(X_g) & (\text{because any model } \mathcal{I} \text{ of } \mathcal{C}_3 \text{ satisfies } \mathcal{I}(X_h) \supseteq \mathcal{I}(P_g)) \\
& \supseteq \varphi(X_2) \cup \mathcal{I}(X_g) & (\text{because “}h \rhd_3 e_2\text{” occurs and by I.H.}) \\
& \supseteq \varphi(X_2) \cup \varphi(X_g) & (\text{because “}g \rhd_3 e_g\text{” occurs and by I.H.}) \\
& = \mathcal{F}_2(\varphi)(X_e) & (\text{by difinition of } \mathcal{F}_2(\varphi)(X_e))
\end{array}$$

□

This article was processed using the LaTeX macro package with LLNCS style