Estimating Uncaught Exceptions in Standard ML Programs from Type-based Equations*

Kwangkeun Yi† and Sukyoung Ryu‡ and Ki-Hyun Pyun§
Dept. of Computer Science
Korea Advanced Institute of Science and Technology
Taejon 305-701, Korea

Abstract

We present a static analysis that detects potential runtime exceptions that are raised and never handled inside Standard ML (SML) programs.

Contrary to our earlier method [Yi94] (based on abstract interpretation [CC77]) where the input program's control flow is simultaneously computed while our exception analysis progresses, we separate the two phases in a manner similar to conventional data flow analysis. Before the exception analysis begins, we first estimate the input program's control flow from the type information from SML/NJ compiler. Based on this call-graph structure, exception flow is specified as a set of equations, whose solution is computed using an iterative least fixpoint method.

At time of this writing, a prototype of this analysis is applied to two realistic SML programs (ML-LEX and OR-SML core) and is 3 *or* 40 *times faster than the earlier method and saves memory by* 35 *or* 65 *percent.*

Key words: *program safety, exception handling, high-order exceptions, data flow analysis, control flow analysis, abstract interpretation, Standard ML, types*

1 Introduction

Exception handling facilities in programming languages 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 was supported, in part, by KOSEF Grant 95-0100-20-01-3 and by Korea Ministry of Information and Communications.

†`kwang@cs.kaist.ac.kr`,
phone:+82.42.869.3536, fax:+82.42.869.3510, address:Dept. of Computer Science, KAIST, 373-1 Kusong-dong Yusong-Gu, Taejon 305-701, Korea

‡`puppy@cs.kaist.ac.kr`

§`nature@cs.kaist.ac.kr`

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 or effects 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. A program can terminate abnormally when an exception is raised and never handled.

Our goal is to develop a compile-time tool for eliminating this safety hole. The tool will detect, *prior to* the program execution, potential runtime exceptions that may be astray. In this paper, we present one such tool for Standard ML (SML) [MTH90] programs.

1.1 Exception Mechanism in Standard ML

In SML, exceptions are treated just like any other value (until they are raised). They can be passed as function arguments, returned as the results of function applications, bound to identifiers, stored in locations and etc.

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

```
Error("at line 7")
```

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

`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:

$$e \texttt{ handle } p_1 \texttt{ => } e_1 \mid \cdots \mid p_n \texttt{ => } e_2$$

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

Since SML exceptions are first-class objects, it is not straightforward from the program texts whether a handler and a raise expression are properly paired to handle all potential exceptions.

Consider the following program fragment:

```
f(x) = ···raise x···
```

In order to find which exceptions are raised inside `f`, we must determine which exceptions are bound to `x`. We must also analyze which handlers are provided for expressions that call `f`, in order to deactivate exceptions that can be handled. For another example that has a higher-order function, consider:

```
f(g) = ···g(x) handle E =>···
```

We must analyze which procedures are bound to `g` in order to determine which exceptions `g(x)` can raise. As in the previous case, we must also analyze which handlers are provided for expressions that call `f`, in order to deactivate exceptions that may escape from the handler inside `f`.

Because of this high-order nature of SML functions and exceptions, abstract interpretation was a natural choice in our earlier work [Yi94]. In this paper, we propose a new method that is less elaborate (and rather brute-force) but has a promising cost-effectiveness. This new approach is similar to conventional data-flow analysis in the sense that the input program's control flow is considered separately from its exception flow.

2 Caveat

Our analysis has the following limitations:

- Our analysis cannot analyze programs that utilize the generative nature of the exception (and the datatype) declarations. This limitation seems not severe; exceptions (and datatypes) are largely declared at the global scope or at the structure 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). Exceptions from primitive functions (such as `+` and `/`) are not considered.

3 Our Approach

A prerequisite for a program's flow analysis is to estimate the program's control flow. If a program is written in a high-order language such as SML, estimating the program's control flow is not straightforward.

Our earlier method[Yi94] uses a monolithic approach (an abstract interpretation[CC77, AH87]) where the closure analysis is done while exception analysis progresses, by treating closures as well as exceptions as parts of the analysis values.

In this work, we present a simple yet effective approach. Firstly, we separate the two analyses. The control flow of an SML program is estimated before our exception analysis begins. A simple control flow estimation is done by using the information from the SML/NJ's type inference phase: for each call expression "$e_1\ e_2$", we assume that functions(lambdas) in a program whose types can unify with that of e_1's are called. Actually, this estimation is used only when it is not clear from the text (e.g. if e_1 is not a function's declared name nor an immediate function expression "`fn x =>` $\cdots$") which function e_1 represents.

This rather crude closure analysis is reasonable because SML programs tend to have a variety of function types. We examined some SML programs and found that their functions are uniformly distributed across a wide spectrum of types and that for each function f there are relatively small number of functions whose types unify with f's type. See Figure 1.

This call graph information is used in estimating the set of raised exceptions from a function. Suppose a function f has a raise expression `raise` e. If the e is not an immediate exception then we assume that all

program	#fn-tys[a] (#fns)[c]	#fns/#fn-tys μ[d]	σ[e]	#unif-tys[b] μ[f]	σ[g]
KB.sml[h]	129 (177)	1.30	1.03	4.68	7.92
mlex.sml[i]	105 (173)	1.56	1.69	2.78	10.27
cgut.sml[j]	108 (186)	1.43	1.94	2.82	5.00
inst.sml[k]	75 (117)	1.31	0.89	1.97	8.37
mc.sml[l]	115 (146)	1.22	0.62	0	0
tchk.sml[m]	53 (89)	1.36	2.06	0	0

[a]Number of distinct function-types

[b]Number of unifiable function-types t' per a function-type t, $= |\{t' | t' \neq t, \mathit{unify}(t', t)\}|$.

[c]Number of functions (`FNexp`'s in program's `absyn` representation inside SML/NJ 108.13)

[d]Mean

[e]Standard deviation

[f]Mean

[g]Standard deviation

[h]Knuth-Bendix.sml

[i]ML-Lex source in SML/NJ 108.13

[j]This paper's call-graph analysis utility source

[k]in `src/module/instantiate.sml` of the SML/NJ 108.13

[l]in `src/translate/` of the SMl/NJ 108.13

[m]in `src/typing/typecheck.sml` of the SMl/NJ 108.13

Figure 1: Functions and types in SML programs

exceptions that may be available at the current function `f` are raised. This set of exceptions available at a function f includes the exceptions passed to f from its callers and those returned from functions called by f.

Such relations regarding exceptions among functions are expressed as two equations per function. One equation is for the set of possibly uncaught exceptions P_f during f's application. For example, P_f of the following function

```
fun f(g) = g(0)+1
```

is the sum $\bigcup_h P_h$ of P_h's for h that may be bound to `g`. Because `g`'s type is `int`$\rightarrow$`int`, all user functions of that type are assumed to be bound to g.

The other equation is for the set of exceptions (not raised) X_f that are available during f's application. For example, X_f of the following function

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

is the sum $\bigcup_h X_h$ of X_h's for h that may call `f`, because the caller h passes its available exceptions to `f` through `x`. A clear superset of such functions h is the set of functions that call a function whose argument type is `exn`. For example, the following `h` is considered to call the `f`

```
fun h(g) = ···g(Error)···.
```

Please note that when a function f has `raise` expressions, P_f is composed of X_f:

$$P_f = \cdots \cup X_f \cup \cdots.$$

Our exception analysis is to solve this set of equation pairs (P_f and X_f).

In the remaining of this paper, we will use the following terms:

Definition 1 For handle expression,

$$\texttt{(handle } e_1 \texttt{ (fn x } e_2\texttt{))}$$

the function (`fn x` e_2) that receives raised exceptions from e_1 is called *handler function*. The e_1 is called *handlee expression*.

4 Use of Type Information from the SML/NJ Compiler

We use type information from the SML/NJ compiler in estimating the control flow (call graph) of a program: for a function call expression (e_1 e_2), if it is not clear which function may be called, or in other words, if e_1 is not an immediate function (i.e., $e_1 \neq$ (`fn x =>` e)), we assume that all functions that have the same type as e_1 are called. This is certainly a safe approximation to the call graph.

We simplify the SML/NJ compiler's type information so that only two kinds of types (exception and non-exception types) are considered. This abstract types are annotated to expressions. Simplified type t is:

t	::=	`nexn`	non-exception type
	|	`exn`	may-exception type
	|	$t \times \cdots \times t$	tuple type
	|	$t \rightarrow t$	function type

Meaning of each type is:

- `nexn`: definitely not an exception type
- `exn`: may-be an exception type. This is when it is the exception type or a polymorphic type.
- $t_1 \times \cdots \times t_n$: tuple type, each of which is of type t_i
- $t_1 \rightarrow t_2$: function type, whose argument and result types are t_1 and t_2, respectively.

A function may receive (or return) an exception if the function's argument (or result) type is `exn`, or is a tuple type whose component is `exn`, or is a tuple type whose component is a tuple type whose component is `exn`, *ad infinitum.*

We use "[`exn`]" for such t that may have the exception type:

Definition 2 A type t is [exn] iff

$$\begin{array}{rcl} t & = & \texttt{exn} \quad \text{or} \\ t & = & t_1 \times \cdots t_n \text{ where } \exists i : t_i \text{ is } [\texttt{exn}] \end{array}$$

Definition 3 A type t is [nexn] iff

$$\begin{array}{rcl} t & = & \texttt{nexn} \quad \text{or} \\ t & = & t_1 \times \cdots t_n \text{ where } \forall i : t_i \text{ is } [\texttt{nexn}] \quad \text{or} \\ t & = & t_1 \rightarrow t_2 \end{array}$$

Definition 4 For a function expression e of type $t \rightarrow t$, $\tau(e)$ is its type after its domain and range types are simplified into [exn] or [nexn]. For example, for $f : \texttt{exn} \rightarrow \texttt{exn} \times \texttt{nexn}$, $\tau(f) = [\texttt{exn}] \rightarrow [\texttt{exn}]$.

We build a table that has four entries: [exn]→[exn], [nexn]→[exn], [exn]→[nexn], and [nexn]→[nexn]. When we estimate functions that may be called at an application "$(e_1\ e_2)$", we assume all the functions in the table's entry for the type of $\tau(e_1)$ be called.

5 Equations for Uncaught Exceptions

For each function f, what we want to know is the set P_f of raised exceptions that may be uncaught during f's application. This set of exception packets comes from three origins inside f.

- `raise` expressions inside `f`

  $$\texttt{fun f() = } \cdots \texttt{(raise } e \texttt{)} \cdots$$

  $$\bigcup_{e \in R(f)} Trim(e, X_f)$$

  where

  $$\mathrm{R}(f) = \{e | e \text{ is a raise expression in } f\}$$

  $$Trim(e, X_f) = \begin{cases} \{\kappa\} & \text{if } e = \text{“}\texttt{raise } \kappa\text{”} \\ X_f & \text{if } e = \text{“}\texttt{raise } e\text{”} \\ X_f \cap \{k\} & \text{if } e = \text{“}\texttt{+raise x } k\text{”} \\ X_f - \{k_i\} & \text{if } e = \text{“}\texttt{-raise x} \\ & \qquad k_1 \cdots k_n\text{”} \end{cases}$$

  `+raise`
  If a handler's pattern for exception's argument part is not complete, the exception is explicitly re-raised by `+raise`.
  `-raise`
  The handler patterns are always augmented with an extra raise(`-raise`) expression, in order to re-raise exceptions that are not caught.

- unhandled application inside `f`

  $$\texttt{fun f() = } \cdots \texttt{(app h x)} \cdots$$

  $$\bigcup_{g \in N(f)} P_g$$

  where

  $$\mathrm{N}(f) = \{g | f \text{ calls } g \text{ without handler shield}\}$$

- handler functions $(\texttt{fn x } e')$ inside f

  $$\texttt{fun f() = } \cdots \texttt{(handle } e \texttt{ (fn x } e'\texttt{))} \cdots$$

  $$\bigcup_{h \in H(f)} P_h$$

  where

  $$\mathrm{H}(f) = \{g | g \text{ is a handle in } f\}$$

  For handle expression

  $$\cdots \texttt{(handle } e_1 \texttt{ (fn x } e_2\texttt{))} \cdots$$

  , raised exceptions from functions called inside the handlee expression e_1 are passed to the handler function $(\texttt{fn x } e_2)$. Exceptions that cannot be handled are re-raised inside e_2. Such re-raised exceptions are those expressed in the third part.

Therefore, the equation for P_f is the sum (union) of three components:

$$P_f = \bigcup_{e \in R(f)} Trim(e, X_f) \ \cup \bigcup_{g \in N(f)} P_g \ \cup \bigcup_{h \in H(f)} P_h. \quad (1)$$

6 Equations for Available Exceptions

X_f, the set of exception values (not raised exceptions) that are available inside f, consists of three parts:

- exceptions explicitly available inside f

  $$K_f = \{\kappa | \text{“}\texttt{(exn } \kappa\texttt{)}\text{” in } f\}$$

- exceptions returned from functions called by f

  $$\bigcup_{g \in \mathrm{From}(f)} X_g$$

  where

  $$\mathrm{From}(f) = \{g | f \text{ may call } g \text{ and } \tau(g) = (\_ \rightarrow [\texttt{exn}])\}$$

- exceptions passed to f For this, we need to consider two cases depending on kinds of f. If f is a normal function (not a handler function), then passed exceptions to f are from functions that call f

  $$\bigcup_{g \in \mathrm{To}(f)} X_g$$

where

$$\text{To}(f) = \{g | g \text{ may call } f \text{ and } \tau(f) = ([\texttt{exn}] \rightarrow \_)\}.$$

The other case is when f is a handler function (`fn x` e') with handlee expression e, as in

$$(\texttt{handle}\ e\ (\texttt{fn x}\ e')).$$

Then the passed exceptions are the raised exceptions from e. Exceptions raised in the handlee expression e is simply

$$P_e.$$

(Even though P is defined only for functions, we subscript P with an expression e, in which case we mean P for an imaginary function whose body is e.)

- exceptions available through free variables o f f
  $\texttt{fun g(x) = } \cdots \underbrace{\texttt{(fn (y)} \cdots \texttt{x}\cdots\texttt{)}}_{f} \cdots$

$$\bigcup_{g \in Free(f)} X_g$$

$Free(f) =$
$\quad \cup\{To(f') | f' \in Owner(FV(f)) \cap Scope(f)$ and $\tau(f') = ([\texttt{exn}] \rightarrow \_)\}.$
$FV(f) = \{v | v \text{ is a free variable in } f\}$
$Owner(x) = \{f | x \text{ is an argument of } f\}$,
$Owner(V) = \cup_{v \in V} Owner(v)$
$Scope(f) = \{g | g \text{ is in } f\text{'s lexical scope}\}$
(i.e., the set of functions that enclose f's definition).

In summary,

$$X_f = K_f \cup \bigcup_{g \in From(f)} X_g \cup \bigcup_{g \in To(f)} X_g \cup P_{e(f)} \cup \bigcup_{g \in Free(f)} X_g. \tag{2}$$

7 Discussion

7.1 Prototype Implementation

We implemented our analysis in the following way. We add a probing module into SML/NJ 1.01 compiler. This module gets internal representation of the input SML program from the compiler, and translates it into our own intermediate form that has type annotations for each function. From this intermediate form, the module estimates the call graph, from which the equation pairs P_f and X_f are derived and printed out. Up until to this step is done by the hooked-up module.

The generated equations are passed to Z1 [YH93], from which a C program that computes the least solution of the equation set is generated. This C program's output is our analysis results: least solution for P_f and X_f for each function f. Among these solutions, we are interested in P_f (uncaught exceptions) for top-level functions f.

program	statistics[a]			
	# lns[b]	# exs [c]	# hds[d]	# ras[e]
ML-LEX	1.2K	8x	8h	47r
OR-SML[f]	1.8K	21x	14h	65r

program	analysis cost	
	new method	earlier method
ML-LEX	1.5min[g]/1.18Mb[h]	53min/3.1Mb
OR-SML	9.5min/1.97Mb	28min/2.7Mb

[a]After the source is translated into the intermediate form.
[b]Number of SML source lines
[c]Number of exception constructors
[d]Number of handle expressions
[e]Number of raise expressions
[f]Core of an DB query interpreter for disjunctive data
[g]minutes on SPARC20(128MB)
[h]mega bytes

Figure 2: Preliminary performance figures

At time of this writing, we have applied this prototype analysis to two realistic SML programs: ML-LEX and OR-SML. As shown in Figure 2 our new method is 3 or 40 times faster than the earlier method and saves memory by 35 or 65 percent.

7.2 Analysis Accuracy

Even though our analysis saves much time and space comparing to the old one [Yi94] based on abstract interpretation (Figure 2 at page 5), its accuracy is decreased. For example, in analyzing ML-LEX program, new analysis decides three more exceptions may be uncaught than those reported by the abstract interpretation.

This accuracy decrease is because the current version of type-based control flow estimation is based not on SML types but on simplified types. Thus many spurious edges are considered in a program's control flow graph.

We beleive this accuracy problem can be easily resolved. Without type simplification, we can keep SML types as they are and estimate the call graph based on those types. We are currently working on this and expect this use of intact SML types can also speed-up our fixpoint computation, because each equation's right-hand-side will have smaller unknowns, which is in turn because smaller number of functions are assumed to be called from each call sites.

7.3 Related Works

Recently, there were several reports that used the type information of statically-typed functional programs in order to generate optimized codes. Shao and Appel [SA95] used the type information of SML programs in optimizing data representations. Tarditi et.al. [TMC+96] reported an optimizing SML compiler where the type information of the input SML programs were used in almost all of their compilation phases. Similarly, Glasgow Haskell Compiler keeps, at its every transformation phase, the type information for optimizations and correctness checks. Our approach is(call-graph estimation from types) inspired from these works.

Also, there were some analysis approaches that compute static information from instrumented type-inference systems [AW93, JG89, TJ91, TJ92, TT94]. In particular, Tang and Jouvelot [TJ94] reported a type-inference-based control-flow analysis.

For exception analysis, Guzmán and Suárez[GS94] reported a type system for the exception analysis. Aiken and Fahndrich [FA96] tackles the same problem by instrumenting their set-constraint-based type inference system. Unfortunately, we cannot have their performance figures to compare with ours.

References

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[FA96] Manuel Fahndrich and Alex Aiken, May 1996. (personal communication).

[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.

[JG89] Pierre Jouvelog and David K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–226, 1989.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, 1991.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Haper. *The Definition of Standard ML.* MIT Press, 1990.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[SA95] Zhong Shao and Andrew Appel. A type-based compiler for standard ml. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[TJ91] Jean-Pierre Talpin and Pierre Jouvelot. Type, effect and region reconstruction in polymorphic functional languages. In *Proceedings of Functional Programming Languages and Computer Architecture*, 1991.

[TJ92] Jean-Piere Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TJ94] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *Lecture Notes in Computer Science*, volume 789, pages 224–243. Springer-Verlag, proceedings of the theoretical aspect in computer science edition, 1994.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, January 1994.

[YH93] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 246–259 (also as CSRD Report No. 1260), January 1993.

[Yi94] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Lecture Notes in Computer Science*, volume 864. Proceedings of the first international static analysis symposium edition, 1994.