An Abstract Interpretation for Estimating Uncaught Exceptions in Standard ML Programs*

Kwangkeun Yi†
kwang@cs.kaist.ac.kr
Korea Advanced Institute of Science & Technology

Abstract

We present a static analysis that detects potential runtime exceptions that are raised and never handled inside Standard ML (SML) programs. This analysis enhances the software safety by predicting, *prior to* the program execution, the abnormal termination caused by unhandled exceptions.

The analysis is specified as a finite, abstract semantics of an intermediate language. The intermediate language, into which SML programs are translated before the analysis begins, is defined such that the mechanism of SML's exception propagation becomes explicit in its text. This syntactic manipulation makes our analysis easy.

Our analysis prototype has been implemented by using an analyzer generator called Z1 and has been used to analyze SML programs consisting of thousand lines. Our analysis is limited to SML programs that are type-correct and are operationally invariant even if the generative nature of SML's data-type and exception declarations is not considered.

(*To appear in Science of Computer Programming, North-Holland Publishing Co., Netherlands*)

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.

*A preliminary version of this paper was presented in the 1st International Static Analysis Symposium (SAS'94) and appeared in *Lecture Notes in Computer Science* Vol. 864.

†This work was done while the author was associated with AT&T Bell Laboratories. Current address: Dept. of Computer Science, KAIST, Taejon, 305-701, South Korea.

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

$$\texttt{raise}\ e$$

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:

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

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`.

Lastly, we must take the exception arguments into account. This is in order to catch, for example, the escaping exception `Error[1]`[1] in

```
(··· raise Error[1] ···) handle Error nil => 1
```

1.2.1 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)`). Therefore, the handler in line `(3)`, which can only handle exceptions declared in its lexical scope, cannot handle another instance of `ZERO` that is newly declared and raised inside the recursive call `fact(n-1)`. Hence this `fact` function fails with an uncaught exception `ZERO`.

```
fun fact(n) =
```

[1] `[1]` is the singleton list of 1.

```
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 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 the structure[2] 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.

Another limitation of our analysis is that we consider only exceptions that appear in the program's text (including library sources). Thus, hidden exceptions from primitive functions[3] are not considered. For example, for an integer division expression "e_1 **div** e_2" we do not report the possibly-uncaught exception `Div`, which is raised when the value of e_2 is zero. This limitation can easily be lifted if our analysis is equipped with a table of primitive operators and their exceptions.

1.3 Analysis Examples

Consider a program where a handler is not complete enough to handle all cases.

```
exception NEGA and ZERO
fun f(x) = if x<0 then raise NEGA
           else if x=0 then raise ZERO
           else x
fun g h x = h(x) handle NEGA => x           (1)
fun main(x) = g f x
```

The handler inside `g` (line `(1)`) cannot handle exception `ZERO` that may be raised inside `f`. Our analysis detects this.

Consider another program where a handler is complete but some exceptions can still escape.

```
exception NEGA and ZERO
fun f(x) = if x<0 then raise NEGA
           else if x=0 then raise ZERO
           else x
fun g h x = h(x)                            (1)
            handle NEGA => h(x+1)           (2)
                 | ZERO => h(1)             (3)
fun main(x) = g f x
```

The handler inside `g` is complete enough to catch all exceptions from `h(x)` (line `(1)`). However, because of the repeated call to `h` (lines `(2)`) inside a handle

[2] A structure in SML is a unit for modular programming; it may be said that an SML structure is analogous to a file in C programming.

[3] Functions of the SML/NJ's base environment.

branch, exceptions ZERO and NEGA can be raised again without being handled. Our analysis detects these uncaught exceptions.

Lastly, consider the following example where exception constructor and its argument are passed as function parameters.[4]

```
exception ERROR of int list
exception EXIT of int list
fun f(n, x, y) =
     if n<0 then raise (x [n])          (1)
     else if n=0 then raise (y nil)     (2)
     else n
fun g(m, x, y) =
     f(m, x, y)                         (3)
     handle ERROR [n] => g(n, y, x)     (4)
          | EXIT nil => 0               (5)
fun main(c) = g(c, ERROR, EXIT)
```

When g is first called inside main, a raised exception ERROR [n] or EXIT nil are handled by the handler inside g (line (4) and (5)). Meanwhile, when g is called recursively (line (4)), the two exception constructors are swapped. Hence, raised exceptions ERROR nil and EXIT [n], at this time, cannot be handled by the handler. Our analysis detects this situation.[5]

1.4 Analysis Implementation

We use the collecting analyzer generator Z1 [YH93, Yi93] in specifying and implementing our analysis. The analysis specification is an abstract interpreter [CC77, CC92]. From this specification, Z1 generates an executable *collecting analyzer*. The collecting analysis computes, for each expression of the input program, a value that characterizes the run-time states that occur at that expression. The program state, in our case, contains a collection of uncaught exceptions. Details of our implementation by Z1 is discussed in Section 7.

After the analysis, the following information is conveyed to the programmer:

- Unhandled exceptions of top-level functions. The existence of such exceptions implies that the program can terminate abnormally.
- Raised exceptions at each handle expression. Using this information the programmer can check if the handler patterns are complete to cover all cases.

Our analyzer has been used to analyze some programs including SML/NJ libraries, ML-YACC, and ML-LEX. See Figure 1 for preliminary performance figures.

---

[4] Such cases are found in the source of SML/NJ 1.01 compiler's environment module.

[5] Actually, these two exceptions are *included* in the analysis result; two other spurious exceptions ERROR [n] and EXIT nil are reported too. See Section 7.1.

program	statistics[a]
ML-LEX	1.2K[b], 8x[c], 8h[d], 47r[e]
ML-YACC	7.7K, 21x, 6h, 122r
OR-SML[f]	1.8K, 21x, 14h, 65r

program	analysis cost	result[g]
ML-LEX	53min[h]/3.1Mb[i]	`eof,Subscript,Match,error,lex_error`
ML-YACC	617min/26.5Mb	`Subscript,Semantic,MkTable`
OR-SML	28min/2.7Mb	`Badobject, Dontunify, Badtypeorunion Iscrewedup,Badtypealpha,Badtypemap Badtyperho,Cannotsort,Unknownnilset`

[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] possibly-uncaught exceptions
[h] minutes on SGI Challenger
[i] mega bytes

Figure 1: Preliminary performance figures

1.5 Related Works

Guzmán and Suárez [GS94] reported an instrumented type-inference system to collect unhandled exceptions for a simplified core ML. Their system does not allow exceptions with arguments. In order to consider exception arguments, they may need an idea similar to the “regions” [TJ92] for approximating the set of argument values accompanying an exception.

On the other hand, such type-inference or, in general, constraints-resolution based program analysis [TJ92, TT94, LG88, JG91, Hei92, AW93] seems to have some appealing characteristics: relatively small analysis cost (because, for some problems and target languages, it is possible to use the unification [Rob65] process rather than the iterative fixpoint method) and a natural support for separate analysis (as reported in [TJ94]). It remains to have a comparative study of the two analysis methods (unification of instrumented type inference versus iterative approach based on abstract interpreter) for the instance of the exception analysis.

2 The Intermediate Language

Our analysis does not directly analyze the SML programs. We have an intermediate language into which the SML programs are translated before the analysis begins. Figure 2 shows this intermediate language.

expression			
e	::=	`x`	variable
	|	`(fn x` e`)`	function
	|	`(apply` e e`)`	application
	|	`(con` κ e^0`)`	datatype value
	|	`(exn` κ e^0`)`	exception value
	|	`(decon` e`)`	datatype deconstruction
	|	`(case x of` $\{p\ e\}^+$`)`	switch expression
	|	`(fix f x` e `in` e`)`	recursive function binding
	|	`(raise` e`)`	exception raise
	|	`(handle` e `x` e`)`	exception handler
pattern			
p	::=	κ	constructor name
	|	`_`	wild-card

$\{\}$ for grouping, $\bullet^+$ for one or more $\bullet$'s, and $\bullet^0$ if $\bullet$ is optional.

Figure 2: (Simplified) Abstract Syntax of the Intermediate Language

In this paper we present a simplified version of the language. We do not show numbers, strings, records, primitive arithmetic operators, and memory operators (like allocation, assignment and dereference). In our implementation though, all these omitted features are supported.

The intermediate language is an applicative higher-order language (based on Lambda [App92] of the SML New Jersey (SML/NJ) compiler). An informal semantics of the language is as follows. (Formal semantics is presented in Section 4.) A datatype value `(con` κ e`)` or an exception value `(exn` κ e`)` is constructed from a constructor name κ and an expression e for its argument value. The argument of a datatype or of an exception is recovered by the deconstruction expression `(decon` e`)`. That is, `(decon (con` κ e`))` is equal to e. The case expression `(case x of` p_1 $e_1 \cdots$`)` branches to e_i when the value of `x` has a constructor name that matches with p_i. For example, `(case x of A 1 _ 2)` is 1 if `x` is a value `(con A` e`)` or `(exn A` e`)`. The wild-card pattern `_` matches with every name. The handle expression `(handle` e_1 `x` e_2`)`, where e_2 will typically be a `case` expression, evaluates e_1 first. If e_1's result is a raised exception $\underline{v}$, the exception value v, not the exception packet $\underline{v}$, is bound to `x` inside e_2. Otherwise, e_1's value is returned. Expression `(fix f x` e_1 `in` e_2`)` binds the recursive function $\mathtt{f} = \lambda \mathtt{x}.e_1$ inside e_2.

2.1 Translation

The translation of the SML programs into their intermediate forms does the following noteworthy things. (Note that, in this section, some examples in the intermediate forms are not legitimate according to the abstract syntax of Figure 2. For convenience, we use numbers, for example.)

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

```
                                 (handle e x
e handle                           (case x
  ERROR => 1       translate          of ERROR 1
| FAIL => 2           ==>                FAIL  2
                                         _ (raise x)))
```

  Note that the "x" has the exception value that was raised inside e. Hence the raise expression "(raise x)" in the last branch has the effect of propagating the exception packets that cannot be handled by the current handler.

  A translation example for a handler of an argument-carrying exception is:

```
                                       (handle e x
                                         (case x
                                           of E (apply
                                                   (fn y
exception E of int list                             (case y
...                         translate                 of NIL 1
e handle E nil => 1            ==>                       _  (raise x)))
                                                   (decon x))
                                               _ (raise x)
                                         ))
```

  Note that (decon x) considers the arguments of exceptions bound to x.

- When patterns in an SML source are not complete enough to cover all cases, the translation makes this situation manifest in the intermediate form. For example,

```
datatype t = A | B | C            (case x
case x                 translate     of A 1
  of A => 1               ==>           B 2
   | B => 2                             _ (raise (exn MATCH)))
```

  Note that the incomplete patterns for a datatype can be statically detected. Our translation resorts to the SML/NJ compiler for this detection.

- Alpha conversion is done: every identifier for variable and function is made distinct.

- Functors in the SML module system are translated into ordinary functions. A functor's argument and result are represented as records (as explained in [App92]). The record construct in our intermediate language is omitted for brevity in this paper.

- A datatype or exception constructor that requires an argument is translated into a function, which is β-reduced whenever appropriate. For example,

```
datatype t = T of int     translate     ··· (fn x (con T x)), ···
··· T, ···                   ⟹
```

- The input SML program is assumed to be type-correct. This condition is easily supported in our case because the program translation occurs after the program passes the type inference phase of the SML/NJ compiler.

3 Roadmap

We take the following steps to arrive at an abstract interpreter for the exception analysis. We start from a *standard semantics* of the language. This standard semantics is natural and simple, but it is difficult to create a finite semantics from it. Thus, we will tailor this standard semantics into one called *concrete semantics* that becomes easier to abstract (make finite). Finally, we abstract the concrete semantics, resulting in a finite, approximate interpreter that is suitable for the compile-time computation. We prove the correctness of our abstract interpreter against the concrete semantics.

4 Standard Semantics

The standard semantics is shown in Figure 3. Note that our semantics is not denotational in that the semantics of function application is not defined compositionally. Our semantics function

$$\mathcal{E}\colon Expr \rightarrow Env \rightarrow Value = \mathit{fix}\,\mathcal{F}$$

is defined to be the least fixpoint of the functional

$$\mathcal{F}\colon (Expr \rightarrow Env \rightarrow Value) \rightarrow (Expr \rightarrow Env \rightarrow Value).$$

Therefore an expression e's semantics is not defined from the semantics of e's subparts but is defined to be the image $\mathit{fix}\,\mathcal{F}[\![e]\!]$ of the least fixpoint $\mathit{fix}\,\mathcal{F}$ of $\mathcal{F}$.

Let us briefly review our notations. $A_\perp$ is a lifted cpo[6] : bottom ($\perp$) and incomparable elements of set A. For two cpos A and B, $A + B$ is the coalesced

[6] Complete partial ordering. A partial-order set X is cpo iff X has a least element and every chain in X has a least upper bound in X.

Semantic domains

$$
\begin{array}{lllll}
\sigma & \in & \mathit{Env} & = \mathit{Id} \rightarrow \mathit{Value} & \text{environment} \\
v & \in & \mathit{Value} & = \mathit{Closure} + \mathit{Data} + \mathit{Exn} + \underline{\mathit{Exn}} & \text{value} \\
 & & \mathit{Closure} & = \mathit{Expr}_\bot \times \mathit{Env} & \text{closure} \\
 & & \mathit{Data} & = \mathit{DataCon}_\bot \times \mathit{Value} & \text{datatype value} \\
 & & \mathit{Exn} & = \mathit{ExnCon}_\bot \times \mathit{Value} & \text{exception} \\
 & & \underline{\mathit{Exn}} & = \mathit{Exn} & \text{raised exception} \\
e & \in & \mathit{Expr} & \text{set of expressions} & \\
\kappa & \in & \mathit{DataCon} & \text{set of datatype constructors} & \\
\kappa & \in & \mathit{ExnCon} & \text{set of exception constructors} & \\
 & & \mathit{Id} & \text{set of variables} &
\end{array}
$$

Semantic function $\mathcal{E}\colon \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Value}$ is the least fixpoint $\mathit{fix}\mathcal{F}$ of

$$\mathcal{F}\colon (\mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Value}) \rightarrow (\mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Value})$$

$$
\begin{array}{lcll}
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{x}]\!]\,\sigma & = & \sigma(\mathtt{x}) & \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(raise}\ e\mathtt{)}]\!]\,\sigma & = & \text{letx} & v = \mathcal{E}[\![e]\!]\,\sigma \\
 & & \text{in} & \underline{v} \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(handle}\ e_1\ \mathtt{x}\ e_2\mathtt{)}]\!]\,\sigma & = & \text{let} & v_1 = \mathcal{E}[\![e_1]\!]\,\sigma \\
 & & \text{in} & \text{if } v_1 = \underline{v} \in \underline{\mathit{Exn}} \\
 & & & \text{then } \mathcal{E}[\![e_2]\!]\,\sigma[v/\mathtt{x}] \\
 & & & \text{else } v_1 \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(case\ x\ of}\ p_1\,e_1 \cdots p_n\,e_n\mathtt{)}]\!]\,\sigma & = & \multicolumn{2}{l}{\mathcal{E}[\![e_j]\!]\,\sigma} \\
 & & \multicolumn{2}{l}{(\kappa \overset{\text{match}}{=} p_j \text{ where } \langle\kappa, v\rangle = \sigma(\mathtt{x}))} \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(apply}\ e_1\ e_2\mathtt{)}]\!]\,\sigma & = & \text{letx} & \langle\mathtt{(fn\ x}\ e\mathtt{)}, \sigma'\rangle = \mathcal{E}[\![e_1]\!]\,\sigma \\
 & & & v \;=\; \mathcal{E}[\![e_2]\!]\,\sigma \\
 & & \text{in} & \mathcal{E}[\![e]\!]\,\sigma'[v/\mathtt{x}] \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(fn\ x}\ e\mathtt{)}]\!]\,\sigma & = & \multicolumn{2}{l}{\langle\mathtt{(fn\ x}\ e\mathtt{)}, \sigma\rangle} \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(fix\ f\ x}\ e_1\ \mathtt{in}\ e_2\mathtt{)}]\!]\,\sigma & = & \multicolumn{2}{l}{\mathcal{E}[\![e_2]\!]\,\sigma'} \\
 & & \multicolumn{2}{l}{(\sigma' = \mathit{fix}\,\lambda\nu.\sigma[\langle\mathtt{(fn\ x}\ e_1\mathtt{)}, \nu\rangle/\mathtt{f}])} \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(}\{\mathtt{con}|\mathtt{exn}\}\ \kappa\ e\mathtt{)}]\!]\,\sigma & = & \text{letx} & v = \mathcal{E}[\![e]\!]\,\sigma \\
 & & \text{in} & \langle\kappa, v\rangle \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(decon}\ e\mathtt{)}]\!]\,\sigma & = & \text{letx} & \langle\kappa, v\rangle = \mathcal{E}[\![e]\!]\,\sigma \\
 & & \text{in} & v
\end{array}
$$

Figure 3: Standard Semantics for Exception Evaluation

sum ($\perp_A = \perp_B = \perp_{A+B}$), $A \times B$ is the Cartesian product with the component-wise order, and $A \rightharpoonup B$ consists of strict, continuous functions with the point-wise order. For $f \in X \rightharpoonup Y$, we write $f[y/x]$ to represent the function that is identical to f, except at x, where its value is y.

The standard evaluation function $\mathcal{E}$ of an expression e returns a value of the expression for a given environment. An environment

$$\sigma \in \mathit{Env} = \mathit{Id} \rightharpoonup \mathit{Value}$$

is a map (a continuous function) from variables Id to their values Value. Set Id consists of the names for functions, arguments and exception binders (`x`'s in the handle expression (`handle` e_1 `x` e_2)) A value $v \in \mathit{Value}$ is either a closure $\mathit{Closure}$, a datatype value Data, an exception value Exn or an exception packet (a raised exception) $\underline{\mathit{Exn}}$:

$$v \in \mathit{Value} = \mathit{Closure} + \mathit{Data} + \mathit{Exn} + \underline{\mathit{Exn}}.$$

The closure is, as usual, a pair of the function text and the environment at the function definition:

$$\mathit{Closure} = \mathit{Expr}_\perp \times \mathit{Env}.$$

The datatype value is a pair of a constructor name and its argument (similarly for the exception value):

$$\begin{array}{lcl} \mathit{Data} & = & \mathit{DataCon}_\perp \times \mathit{Value} \\ \mathit{Exn} & = & \mathit{ExnCon}_\perp \times \mathit{Value} \end{array}$$

An exception packet $\underline{\mathit{Exn}}$ is the same as an exception value except that we mark it with the underline.

4.1 Expressing the SML Exception Convention

To express the exception convention, we use the "letx" notation

$$\text{"letx } v = []_1 \text{ in } []_2\text{"}$$

as a shorthand for

$$\text{"let } v = []_1 \text{ in if } v \in \underline{\mathit{Exn}} \text{ then } v \text{ else } []_2.\text{"}$$

That is, the evaluation of the "letx" bindings terminates with the first whose result is a raised exception. This raised exception becomes the result in conclusion of the "letx" expression. When no exception is raised, "letx" is the same as "let." Note that in the semantics we do not use the "letx" for the handle expression, because a handler is the only way to stop the propagation of an exception.

5 Concrete Semantics

A semantics that is defined over recursively-defined domains is troublesome when we derive from it a finite, abstract interpreter.

The standard semantics of the previous section has infinite domains that are recursively defined. Consider the value domain *Value*:

$$\begin{array}{rcl} \mathit{Value} & = & \mathit{Closure} + \mathit{Data} + \cdots \\ & = & (\mathit{Expr}_\perp \times (\mathit{Id} \rightarrow \mathit{Value})) + (\mathit{DataCon}_\perp \times \mathit{Value}) + \cdots \end{array}$$

In this section, we will develop a new semantics (called concrete semantics) that uses no recursively-defined domains hence becomes easier to abstract than the standard semantics.

Our solution is to use the store[7]: a map from locations to values, upon which some effects of the evaluation function are accumulated (i.e., the store is a part of both the input and the output of the evaluation function):

$$\mathcal{E}: \mathit{Expr} \rightarrow \mathit{Env} \times \mathit{Store} \rightarrow \mathit{Value} \times \mathit{Store}$$

When a value v needs to be bound to a variable $\mathtt{x}$, a new location ℓ is allocated in the store $s \in \mathit{Store}$

$$s \in \mathit{Store} = \mathit{Loc} \rightarrow \mathit{Value}$$

and the value is written in that location $s[v/\ell]$. The environment $\sigma \in \mathit{Env}$

$$\sigma \in \mathit{Env} = \mathit{Id} \rightarrow \mathit{Loc}$$

then maps the identifier to the location $\sigma[\ell/\mathtt{x}]$. Thus, for example, the argument of a function is mapped to different locations, one for each invocation of the function. When variable $\mathtt{x}$'s value is needed, $\mathtt{x}$'s location $e(\mathtt{x})$ is fetched from the current environment e and the store entry $s(e(\mathtt{x}))$ of the location has the value of $\mathtt{x}$.

By using the locations and the stores, the value domain can be defined non-recursively. The domain for the closure is defined without the *Value* domain, because the environment component is now a map from identifiers to locations. The domains for the datatype and exception values has, for the argument component, the location *Loc* in place of the *Value* domain. That is, when a datatype value (a pair $\in \mathit{DataCon}_\perp \times \mathit{Value}$ in the standard semantics) is constructed, a new location is allocated in the current store to hold the argument value, and this new location (rather than the argument value itself) is paired with the constructor name.

The concrete semantics is shown in Fig. 4.

[7] Actually, in order to handle the allocation, assignment and dereference expressions that are included in the real intermediate language, we need the store domain anyway.

Semantic domains

$$
\begin{array}{llllll}
s & \in & \mathit{Store} & = & \mathit{Loc} \rightarrow \mathit{Value} & \text{store} \\
\sigma & \in & \mathit{Env} & = & \mathit{Id} \rightarrow \mathit{Loc} & \text{environment} \\
v & \in & \mathit{Value} & = & \mathit{Closure} + \mathit{Data} + \mathit{Exn} + \underline{\mathit{Exn}} & \text{value} \\
 & & \mathit{Closure} & = & \mathit{Exp}_{\perp} \times \mathit{Env} & \text{closure} \\
 & & \mathit{Data} & = & \mathit{DataCon}_{\perp} \times \mathit{Loc} & \text{datatype value} \\
 & & \mathit{Exn} & = & \mathit{ExnCon}_{\perp} \times \mathit{Loc} & \text{exception value} \\
 & & \underline{\mathit{Exn}} & = & \mathit{Exn} & \text{raised exception} \\
 & & \mathit{Loc} & = & \{\ell \mid \text{location } \ell\}_{\perp} & \text{location} \\
e & \in & \mathit{Expr} & & \text{set of expressions} & \\
\kappa & \in & \mathit{DataCon} & & \text{set of datatype constructors} & \\
\kappa & \in & \mathit{ExnCon} & & \text{set of exception constructors} & \\
 & & \mathit{Id} & & \text{set of variables} &
\end{array}
$$

Semantic function $\mathcal{E}\colon \mathit{Expr} \rightarrow \mathit{Env} \times \mathit{Store} \rightarrow \mathit{Value} \times \mathit{Store}$ is the least fixpoint $\mathit{fix}\mathcal{F}$ of

$$
\begin{array}{ll}
\mathcal{F}\colon & (\mathit{Expr} \rightarrow \mathit{Env} \times \mathit{Store} \rightarrow \mathit{Value} \times \mathit{Store}) \rightarrow \\
 & (\mathit{Expr} \rightarrow \mathit{Env} \times \mathit{Store} \rightarrow \mathit{Value} \times \mathit{Store})
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{x}]\!]\,\langle\sigma, s_0\rangle & = & \multicolumn{2}{l}{s_0(\sigma(\mathtt{x}))} \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(raise}\ e\mathtt{)}]\!]\,\langle\sigma, s_0\rangle & = & \text{letx} & \langle v, s_1\rangle = \mathcal{E}[\![e]\!]\,\langle\sigma, s_0\rangle \\
 & & \text{in} & \langle \underline{v}, s_1\rangle \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(handle}\ e_1\ \mathtt{x}\ e_2\mathtt{)}]\!]\,\langle\sigma, s_0\rangle & = & \text{let} & \langle v, s_1\rangle = \mathcal{E}[\![e_1]\!]\,\langle\sigma, s_0\rangle \\
 & & \text{in} & \text{if } v = \underline{v'} \in \underline{\mathit{Exn}} \quad (\mathit{new}\ \ell) \\
 & & & \text{then } \mathcal{E}[\![e_2]\!]\,\langle\sigma[\ell/\mathtt{x}], s_1[v'/\ell]\rangle \\
 & & & \text{else } \langle v, s_1\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(case\ x\ of}\ p_1\, e_1 \cdots p_n\, e_n\mathtt{)}]\!]\,\langle\sigma, s_0\rangle = \mathcal{E}[\![e_j]\!]\,\langle\sigma, s_0\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad (\kappa \overset{\text{match}}{=} p_j \text{ where } \langle\kappa, \ell\rangle = s_0(\sigma(\mathtt{x}))) \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(apply}\ e_1\ e_2\mathtt{)}]\!]\,\langle\sigma, s_0\rangle = \text{letx } \langle\langle\mathtt{(fn\ x}\ e\mathtt{)}, \sigma'\rangle, \mathrm{s}_1\rangle = \mathcal{E}[\![\mathrm{e}_1]\!]\,\langle\sigma, \mathrm{s}_0\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle v, s_2\rangle = \mathcal{E}[\![e_2]\!]\,\langle\sigma, s_1\rangle \\
\qquad\qquad\qquad\qquad\qquad \text{in } \ \mathcal{E}[\![\mathrm{e}]\!]\,\langle\sigma'[\ell/\mathtt{x}], \mathrm{s}_2[\mathrm{v}/\ell]\rangle \quad (\mathit{new}\ \ell)
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(fn\ x}\ e\mathtt{)}]\!]\,\langle\sigma, s_0\rangle & = & \multicolumn{2}{l}{\langle\langle(\text{fn x } e), \sigma\rangle, s_0\rangle} \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(fix\ f\ x}\ e_1\ \mathtt{in}\ e_2\mathtt{)}]\!]\,\langle\sigma, s_0\rangle & = & \text{let} & \sigma' = \sigma[\ell/\mathtt{f}]\ (\mathit{new}\ \ell) \\
 & & & s' = s_0[\langle\mathtt{(fn\ x}\ e\mathtt{)}, \sigma'\rangle/\ell] \\
 & & \text{in} & \mathcal{E}[\![e_2]\!]\,\langle\sigma', s'\rangle \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(\{con|exn\}}\ \kappa\ e\mathtt{)}]\!]\,\langle\sigma, s_0\rangle & = & \text{letx} & \langle v, s_1\rangle = \mathcal{E}[\![e]\!]\,\langle\sigma, s_0\rangle \\
 & & \text{in} & \langle\langle\kappa, \ell\rangle, s_1[v/\ell]\rangle \quad (\mathit{new}\ \ell) \\
\mathcal{F}\,\mathcal{E}\,[\![\mathtt{(decon}\ e\mathtt{)}]\!]\,\langle\sigma, s_0\rangle & = & \text{letx} & \langle\langle\kappa, \ell\rangle, s_1\rangle = \mathcal{E}[\![e]\!]\,\langle\sigma, s_0\rangle \\
 & & \text{in} & \langle s_1(\ell), s_1\rangle
\end{array}
$$

Figure 4: Concrete Semantics for Exception Evaluation

6 Abstract Exception Evaluation

The abstraction of the concrete semantics is needed to make the resulting interpretation computable at compile-time. This abstraction consists of abstracting both the semantic domains and the interpreter function.

We makes the abstract domains be finite lattices[8]. Each element $\hat{x} \in \hat{D}$ in an abstract domain $\hat{D}$ denotes an *ideal*[9] $\gamma(\hat{x}) \subseteq D$ of concrete values. The partial order $\hat{x} \sqsubseteq \hat{y}$ in the abstract domain is when $\hat{x}$'s information is more precise than that of $\hat{y}$, i.e., when $\gamma(\hat{x}) \subseteq \gamma(\hat{y})$. The lattice structure ensures the existence of a safe element $\hat{x} \sqcup \hat{y}$ whose information $\gamma(\hat{x} \sqcup \hat{y})$ is consistent with the others $\gamma(\hat{x})$ and $\gamma(\hat{y})$.

The abstract evaluation function must be monotonic and be a upper approximate of its concrete correspondence. A function $\hat{f}: \hat{A} \to \hat{B}$ is a upper approximation of its concrete counterpart $f: A \to B$ when the abstract result $\hat{f}(\hat{x})$ over $\hat{x}$ must include the concrete result $f(x)$ for every x meant by $\hat{x}$. The monotonicity requires that $\hat{f}$'s results for consistent inputs be consistent.

Both the finiteness of the abstract domains and the monotonicity of the abstract evaluation guarantee the termination of the induced program analysis. The upper approximate-ness is necessary for the soundness.

6.1 Abstracting Locations

In abstract semantics, we use a single location for each allocation site of the source program. Note that new locations are allocated at four places. When a function is defined (inside the `fix` expression), a new location for the function name is allocated to hold the closure. When a function is applied, a new location to hold its argument. When a handler is applied, a new location to hold, if any, exception value. Lastly, when a datatype or exception value is created, a new location to hold its argument.

We uniquely name the allocation sites of a program, and use these names for abstract locations. Let Ln be the set of unique names for the allocation sites. An abstract location $\iota \in Ln$ represents the set $Allocated(\iota)$ of all concrete locations that are allocated at site ι during the execution of the program. Formally, the abstract location $\hat{L}$ and its abstraction map $\alpha_L: Loc \to \hat{L}$ are:

$$\begin{aligned} \hat{L} &= Ln_\perp \\ \alpha_L &= \lambda\ell.\text{if } \ell = \perp \text{ then } \perp \text{ else } \iota \text{ such that } \ell \in Allocated(\iota). \end{aligned}$$

Generally, that a single abstract location ℓ represents multiple, concrete locations can deteriorate the analysis accuracy. This is because storing a value

[8] In general, abstract domains need not be finite. Even for infinite lattices, if their every chain is bounded we can have terminating abstract interpretation[CC77, Bou93, Bou92] by means of applying "widening" operators at flow cycles. However, in Z1 we cannot specify such operators that are selectively applied only to some flow points.

[9] An ideal I of a cpo D is a subset of D that is downwardly closed ($x \sqsubseteq y \in I$ implies $x \in I$) and upwardly complete (every chain in I has the least upper bound inside I).

$$
\begin{array}{lcl}
\alpha_L : \mathit{Loc} \rightarrow \hat{L} & = & \lambda \ell.\text{if } \ell = \bot \text{ then } \bot \text{ else } \iota \text{ where } \ell \in \mathit{Allocated}(\iota) \\
\alpha_C : \mathit{Closure} \rightarrow \hat{C} & = & \lambda c.\text{if } c = \bot \text{ then } \{\} \text{ else } \{e\} \text{ where } c = \langle e, \sigma \rangle \\
\alpha_D : \mathit{Data} \rightarrow \hat{D} & = & \lambda d.\text{if } d = \bot \text{ then } \{\} \\
& & \quad \text{else } \{\langle \kappa, \iota \rangle | d = \langle \kappa, \ell \rangle \ \wedge \ \ell \in \mathit{Allocated}(\iota)\} \\
\alpha_X : \mathit{Exn} \rightarrow \hat{X} & = & \lambda x.\text{if } x = \bot \text{ then } \{\} \\
& & \quad \text{else } \{\langle \kappa, \iota \rangle | x = \langle \kappa, \ell \rangle \ \wedge \ \ell \in \mathit{Allocated}(\iota)\} \\
\alpha_{\underline{X}} : \underline{\mathit{Exn}} \rightarrow \underline{\hat{X}} & = & \alpha_X \\
\alpha_V : \mathit{Value} \rightarrow \hat{V} & = & \lambda v. \left\{ \begin{array}{ll}
\text{if } v = \bot \text{ then} & \langle \bot_{\hat{C}}, \bot_{\hat{D}}, \bot_{\hat{X}}, \bot_{\underline{\hat{X}}} \rangle \\
\text{else if } v \in \mathit{Closure} \text{ then} & \langle \alpha_C(v), \bot_{\hat{D}}, \bot_{\hat{X}}, \bot_{\underline{\hat{X}}} \rangle \\
\text{else if } v \in \mathit{Data} \text{ then} & \langle \bot_{\hat{C}}, \alpha_D(v), \bot_{\hat{X}}, \bot_{\underline{\hat{X}}} \rangle \\
\text{else if } v \in \mathit{Exn} \text{ then} & \langle \bot_{\hat{C}}, \bot_{\hat{D}}, \alpha_X(v), \bot_{\underline{\hat{X}}} \rangle \\
\text{else if } v \in \underline{\mathit{Exn}} \text{ then} & \langle \bot_{\hat{C}}, \bot_{\hat{D}}, \bot_{\hat{X}}, \alpha_{\underline{X}}(v) \rangle
\end{array} \right. \\
\alpha_S : \mathit{Store} \rightarrow \hat{S} & = & \lambda s.\lambda \hat{\ell}. \bigsqcup_{\alpha_L(\ell) \sqsubseteq \hat{\ell}} \alpha_V(s(\ell)) \\
\multicolumn{3}{l}{\alpha_{E \times S} : \mathit{Env} \times \mathit{Store} \rightarrow \hat{S} = \lambda \langle e, s \rangle.\alpha_S(s)} \\
\multicolumn{3}{l}{\alpha_{V \times S} : \mathit{Value} \times \mathit{Store} \rightarrow \hat{V} \times \hat{S} = \lambda \langle v, s \rangle.\langle \alpha_V(v), \alpha_S(s) \rangle}
\end{array}
$$

Figure 5: Abstraction functions for domains

to $\hat{\ell}$ must have the effect of raising the location's value in its lattice; we cannot overwrite the existing value at the location.

This accuracy deterioration is not avoidable but can be reduced, to some extent. For example, instead of using a single abstract location for each allocation site, we can use multiple abstract locations each of which represents an exclusive subset of the locations allocated at that site. One technique is to use the "abstract procedure string" [Har89] that classifies the locations according to the procedural movements (calls and returns) that they experience after their births. Depending on the abstractions of locations, we can achieve the effects of various cost-accuracy balances (such as "call/single", "dynamic/multiple" or "single/multiple" granularities [HDCM93]). We chose not to use these techniques because our exception analysis with our simple abstraction showed a satisfying accuracy.

Our abstraction of locations eliminates the use of the environment (a map from variables to locations) because only one abstract location is associated with each variable. The elimination of environments immediately entails an abstraction of closures. An abstract closure becomes a set of function definitions without the environment component.

The abstractions for other domains are straightforward. See Figure 5.

6.2 Abstract Evaluation

Abstract interpreter for the exception analysis is shown in Fig. 6. Notations: $f[y//x] = f[f(x) \sqcup y/x]$. $x.D$ selects D component of x. x for $\langle \bot, \cdots, x, \bot \cdots \rangle$ in proper contexts. $x : D$ casts $x \in D'$ into D (only when D and D' are equivalent except for names). $|\hat{v}|$ is identical to $\hat{v}$ except for $|\hat{v}|.\underline{\hat{X}} = \bot$ (raised exception component). For abstract locations we use the program's variable names (assuming that every variable is named uniquely). When an allocation site has no variable (such as the datatype and exception construction expressions), we choose a unique name for such sites.

Note that the abstract evaluation does not use the "letx" notation. That is, when an exception is raised during a subcomputation, the remaining evaluation is not aborted. Rather, the evaluation continues and its result, together with the exceptions raised during subcomputations, is collected in the value of the conclusion.

Consider the raise expression.

$$\begin{array}{rcll} \hat{\mathcal{E}}[\![\texttt{(raise}\ e\texttt{)}]\!]\ \hat{s_0} & = & \text{let} & \langle \hat{v_1}, \hat{s_1} \rangle = \hat{\mathcal{E}}[\![e]\!]\ \hat{s_0} \\ & & \text{in} & \langle \hat{v_1}.\underline{\hat{X}} \sqcup (\hat{v_1}.\hat{X} : \underline{\hat{X}}), \hat{s_1} \rangle \end{array}$$

We first evaluate the exception expression e. Any raised exception during this evaluation is collected in $\hat{v_1}.\underline{\hat{X}}$. By the current raise expression, the exception values $\hat{v_1}.\hat{X}$ are raised $\hat{v_1}.\hat{X} : \underline{\hat{X}}$ and are collected (joined) with the already raised exceptions $\hat{v_1}.\underline{\hat{X}}$.

Consider the handle expression.

$$\begin{array}{rcll} \hat{\mathcal{E}}[\![\texttt{(handle}\ e_1\ \texttt{x}\ e_2\texttt{)}]\!]\ \hat{s_0} & = & \text{let} & \langle \hat{v_1}, \hat{s_1} \rangle = \hat{\mathcal{E}}[\![e_1]\!]\ \hat{s_0} \\ & & & \langle \hat{v_2}, \hat{s_2} \rangle = \hat{\mathcal{E}}[\![e_2]\!]\ \hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\texttt{x}] \\ & & \text{in} & \langle |\hat{v_1}| \sqcup \hat{v_2}, \hat{s_1} \sqcup \hat{s_2} \rangle \end{array}$$

We first evaluate the expression e_1. The handler needs to handle, if any, raised exceptions $\hat{v_1}.\underline{\hat{X}}$ inside e_1. With the store $\hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\texttt{x}]$ that holds the exceptions $\hat{v_1}.\underline{\hat{X}} : \hat{X}$ at `x`. we evaluate the second expression e_2, which is usually a `case` expression. The value in conclusion is either the value $\hat{v_1}$ if expression e_1 did not raise any exception or the value $\hat{v_2}$ after the handling if expression e_1 raised some exceptions. These two possibilities are accommodated by the join operation $|\hat{v_1}| \sqcup \hat{v_2}$. We do not return the raised exceptions of $\hat{v_1}$ because they are considered inside the evaluation of e_2. (Hence $|\hat{v_1}|$, not $\hat{v_1}$, in $|\hat{v_1}| \sqcup \hat{v_2}$.) Note that if the handler patterns of e_2 is not complete enough to handle all cases, the exceptions bound to `x` are re-raised,[10] hence is captured inside $\hat{v_2}$.

Consider the case expression.

$$\begin{array}{l} \hat{\mathcal{E}}[\![\texttt{(case x of}\ p_1\, e_1 \cdots p_n\, e_n\texttt{)}]\!]\ \hat{s_0} = \\ \quad \bigsqcup_{1 \le i \le n} \hat{\mathcal{E}}[\![e_i]\!]\ (\hat{s_0}[\mathit{Screen}(\hat{s_0}(\texttt{x}), \{p_i\}, \{p_1, \cdots, p_{i-1}\})//\texttt{x}]) \end{array}$$

[10] Note that when an SML source program is translated into the intermediate language, appropriate `raise` expressions are added for incomplete patterns – see discussions in Sect. 2.1.

Semantic domains

$$
\begin{array}{lllll}
\hat{s} & \in & \hat{S} & = \hat{L} \rightarrow \hat{V} & \text{abstract store} \\
\hat{\ell} & \in & \hat{L} & = Ln_{\perp} & \text{abstract location} \\
\hat{v} & \in & \hat{V} & = \hat{C} \times \hat{D} \times \hat{X} \times \underline{\hat{X}} & \text{abstract value} \\
 & & \hat{C} & = 2^{Expr} & \text{abstract closure} \\
 & & \hat{D} & = 2^{DataCon \times Ln} & \text{abstract datatype value} \\
 & & \hat{X} & = 2^{ExnCon \times Ln} & \text{abstract exception value} \\
 & & \underline{\hat{X}} & = \hat{X} & \text{abstract raised exception} \\
\iota & \in & Ln & \text{set of allocation sites} & \\
 & & DataCon & \text{set of datatype constructors} & \\
 & & ExnCon & \text{set of exception constructors} & \\
e & \in & Expr & \text{set of expressions} &
\end{array}
$$

Semantic function $\hat{\mathcal{E}}: Expr \rightarrow \hat{S} \rightarrow \hat{V} \times \hat{S}$ is the least fixpoint $\mathit{fix}\hat{\mathcal{F}}$ of

$$\hat{\mathcal{F}}: (Expr \rightarrow \hat{S} \rightarrow \hat{V} \times \hat{S}) \rightarrow (Expr \rightarrow \hat{S} \rightarrow \hat{V} \times \hat{S})$$

$$
\begin{array}{lll}
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{x} \rrbracket\,\hat{s_0} & = & \langle \hat{s_0}(\mathtt{x}), \hat{s_0} \rangle \\
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(raise}\ e\mathtt{)} \rrbracket\,\hat{s_0} & = & \text{let}\ \ \langle \hat{v_1}, \hat{s_1} \rangle = \hat{\mathcal{E}}\llbracket e \rrbracket\,\hat{s_0} \\
 & & \text{in}\ \ \ \langle \hat{v_1}.\underline{\hat{X}} \sqcup (\hat{v_1}.\hat{X} : \underline{\hat{X}}), \hat{s_1} \rangle \\
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(handle}\ e_1\ \mathtt{x}\ e_2\mathtt{)} \rrbracket\,\hat{s_0} & = & \text{let}\ \ \langle \hat{v_1}, \hat{s_1} \rangle = \hat{\mathcal{E}}\llbracket e_1 \rrbracket\,\hat{s_0} \\
 & & \phantom{\text{let}}\ \ \langle \hat{v_2}, \hat{s_2} \rangle = \hat{\mathcal{E}}\llbracket e_2 \rrbracket\,\hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\mathtt{x}] \\
 & & \text{in}\ \ \ \langle |\hat{v_1}| \sqcup \hat{v_2}, \hat{s_1} \sqcup \hat{s_2} \rangle \\
\multicolumn{3}{l}{\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(case\ x\ of}\ p_1\,e_1 \cdots p_n\,e_n\mathtt{)} \rrbracket\,\hat{s_0} =} \\
\multicolumn{3}{l}{\qquad \bigsqcup_{1 \le i \le n} \hat{\mathcal{E}}\llbracket e_i \rrbracket\ \ (\hat{s_0}[Screen(\hat{s_0}(\mathtt{x}), \{p_i\}, \{p_1, \cdots, p_{i-1}\})//\mathtt{x}])} \\
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(apply}\ e_1\ \ e_2\mathtt{)} \rrbracket\,\hat{s_0} & = & \text{let}\ \ \langle \hat{v_1}, \hat{s_1} \rangle = \hat{\mathcal{E}}\llbracket e_1 \rrbracket\,\hat{s_0} \\
 & & \phantom{\text{let}}\ \ \langle \hat{v_2}, \hat{s_2} \rangle = \hat{\mathcal{E}}\llbracket e_1 \rrbracket\,\hat{s_1} \\
 & & \text{in}\ \ \ \langle \hat{v_1}.\underline{\hat{X}} \sqcup \hat{v_2}.\underline{\hat{X}}, \hat{s_1} \sqcup \hat{s_2} \rangle \sqcup \\
 & & \phantom{\text{in}}\ \ \ \bigsqcup_{1 \le i \le n} \hat{\mathcal{E}}\llbracket e'_i \rrbracket\ \hat{s_2}[|\hat{v_2}|//\mathtt{x}_i] \\
 & & \text{where}\ \hat{v_1}.\hat{C} = \{\mathtt{(fn\ x}_1\ \ e'_1\mathtt{)}, \cdots, \mathtt{(fn\ x}_n\ \ e'_n\mathtt{)}\} \\
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(fn\ x}\ \ e\mathtt{)} \rrbracket\,\hat{s_0} & = & \langle \{\mathtt{(fn\ x}\ \ e\mathtt{)}\}, \hat{s_0} \rangle \\
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(fix\ f\ x}\ \ e\ \ \mathtt{in}\ e'\mathtt{)} \rrbracket\,\hat{s_0} & = & \hat{\mathcal{E}}\llbracket e' \rrbracket\ \hat{s_0}[\{\mathtt{(fn\ x}\ \ e\mathtt{)}\}//\mathtt{f}] \\
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(\{con|exn\}}\ \ \kappa\ \ e\mathtt{)} \rrbracket\,\hat{s_0} & = & \text{let}\ \ \langle \hat{v_1}, \hat{s_1} \rangle = \hat{\mathcal{E}}\llbracket e \rrbracket\,\hat{s_0} \\
 & & \text{in}\ \ \ \langle \{\langle \kappa, \mathtt{x}_e \rangle\}, \hat{s_1}[\hat{v_1}//\mathtt{x}_e] \rangle \\
\hat{\mathcal{F}}\,\hat{\mathcal{E}}\,\llbracket \mathtt{(decon}\ e\mathtt{)} \rrbracket\,\hat{s_0} & = & \text{let}\ \ \langle \hat{v_1}, \hat{s_1} \rangle = \hat{\mathcal{E}}\llbracket e \rrbracket\,\hat{s_0} \\
 & & \text{in}\ \ \ \langle \bigsqcup_{1 \le i \le n} \hat{s_1}(\iota_i), \hat{s_1} \rangle \\
 & & \text{where}\ \hat{v_1}.\hat{D} \cup \hat{v_1}.\hat{X} = \{\langle \kappa_1, \iota_1 \rangle, \cdots \langle \kappa_n, \iota_n \rangle\}
\end{array}
$$

Figure 6: Abstract Semantics for Exception Analysis

We evaluate every branch of the case expression and collect the results. Inside each branch e_i, the value of `x` will be appropriately trimmed (by the *Screen* operation) according to the case patterns $\{p_1, \cdots, p_{i-1}\}$ that appear in the previous branches. The auxiliary operation

$$Screen(\hat{v}, P, Q)$$

chooses among the data values $\hat{v}.\hat{D}$ and exceptions $\hat{v}.\hat{X}$ those that match with a pattern in P but not with any pattern in Q.

$$\begin{array}{l}
Screen\colon \hat{V} \times 2^{Pattern} \times 2^{Pattern} \rightarrow \hat{V} \\
Screen(\hat{v}, P, Q) = \\
\quad \text{let} \quad screen = \lambda\langle x, P\rangle.\{\langle\kappa, \iota\rangle \in x \,|\, \exists p \in P : \kappa \stackrel{\text{match}}{=} p\} \\
\quad \text{in} \quad \langle \quad \bot_{\hat{C}}, \\
\qquad\qquad screen(\hat{v}.\hat{D}, P) - screen(\hat{v}.\hat{D}, Q), \\
\qquad\qquad screen(\hat{v}.\hat{X}, P) - screen(\hat{v}.\hat{X}, Q), \\
\qquad\qquad \bot_{\underline{\hat{X}}} \ \rangle
\end{array}$$

This auxiliary operation is used to sharpen our analysis as follows. When we analyze a program, we carry an approximate store whose entry is a collection of approximate values that a variable can have during execution. This collection always denotes a superset of actual values. The larger the collection, the less accurate the analysis becomes. In each branch of a `case` expression, blind use of this collection may result in overly conservative (hence very inaccurate) analysis.

For example, suppose that the current abstract store $\hat{s}$ has two exceptions $\{$`E`, `F`$\}$ for `x` below.

```
(case x
      E e1
      _ (raise x))
```

When we analyze the second branch `(raise x)` exception `E` should not be considered for `x`, because this exception matches with the first pattern. This trimming is achieved by the *Screen* operation:

$$Screen(\hat{s}(\mathtt{x}), \{\_\}, \{\mathtt{E}\})$$

6.2.1 Accuracy Concern: An Implementation Details

Even with the *Screen* operation, the rule for the `case` expression has no effect on improving the analysis accuracy. This is because the *Screen* result does not replace the existing value of `x` in the store. Instead, the result is joined with the existing value of `x` (recall the notation: $f[y//x] = f[y \sqcup f(x)/x]$). We cannot overwrite the existing value because an abstract location `x` represents multiple concrete locations.

Therefore, even with the *Screen* operation the store value at `x` after

$$\hat{s}[Screen(\cdots)//\mathtt{x}]$$

remains unchanged.

This problem is simply solved by using different names for `x` inside each branch. Each trimmed value of `x` for each branch e_i is bound to a unique name, say $\mathtt{x}_i$, instead of always to the same `x`. And every "`x`" inside each branch e_i is replaced by its unique name "$\mathtt{x}_i$". This replacement is straightforward because the source was already alpha-converted when translated from SML. For example,

<pre>(case x
  of E (apply
         (fn y
          (case y
            of NIL y
               _ (raise x)))
         (decon x))
     _ (raise x)
)</pre>
	$\overset{\text{becomes}}{\Longrightarrow}$	<pre>(case x
		  of E (apply
		         (fn y
		          (case y
		            of NIL y1
		               _ (raise x1)))
		         (decon x1))
		     _ (raise x2)
		)</pre>

Let e'_i be the result of such replacement for i-th branch e_i in the case expression

$$\texttt{(case x of } p_1\, e_1 \cdots p_n\, e_n\texttt{)}$$

Then the new abstract evaluation rule for `case` expression becomes

$$\bigsqcup_{1\le i\le n} \hat{\mathcal{E}}[\![e'_i]\!]\ (\hat{s_0}[Screen(\hat{s_0}(\mathtt{x}), \{p_i\}, \{p_1, \cdots, p_{i-1}\})//\mathtt{x}_i]).$$

6.3 Correctness of the Abstract Semantics

Theorem 1 *For a given program e, its abstract semantics $fix\hat{\mathcal{F}}[\![e]\!]$ can be computed in a finite time.*

Proof. This fact immediately follows from that every operation used in the abstract semantics of Figure 6 is monotonic and all the abstract domains where the *Expr* is the set of expressions of the given program e are finite. □

Proving the soundness of the abstract semantics needs the following two lemmas.

Lemma 1 *All abstraction functions (Figure 5) are strict and continuous.*

Proof. The strictness of α_L, α_C, α_D, α_X, $\alpha_{\underline{X}}$, and α_V is obvious from their definitions. Store abstraction α_S is strict because α_L and α_V are strict. $\alpha_{E\times S}$ and $\alpha_{V\times S}$ are strict because α_S and α_V are strict.

Similarly, it is trivial to see that every abstraction function is monotonic. Continuity immediately follows from that every chain of concrete domains is of finite length. □

Lemma 2 *For every store s, value v, and location ℓ,*

$$\alpha(s[v/\ell]) \sqsubseteq \alpha(s)[\alpha(v)//\alpha(\ell)].$$

Proof. Recall the store abstraction:

$$\alpha_S = \lambda s.\lambda\hat{\ell}.\ \bigsqcup_{\alpha_L(\ell)\sqsubseteq\hat{\ell}} \alpha_V(s(\ell))$$

Thus,

$$\begin{array}{rcll}
\alpha(s[v/\ell])(\alpha(\ell)) & = & \bigsqcup_{\alpha(\ell')\sqsubseteq\alpha(\ell)} \alpha(s[v/\ell](\ell')) & \text{by definition of } \alpha_S \\
& \sqsubseteq & \alpha(v) \sqcup \bigsqcup_{\alpha(\ell')\sqsubset\alpha(\ell)} \alpha(s(\ell')) & \\
& \sqsubseteq & \alpha(v) \sqcup \bigsqcup_{\alpha(\ell')\sqsubseteq\alpha(\ell)} \alpha(s(\ell')) & \text{by monotonicity of } \alpha \\
& = & \alpha(v) \sqcup \alpha(s)(\alpha(\ell)) & \text{by definition of } \alpha_S \\
& = & (\alpha(s)[\alpha(v)//\alpha(\ell)])(\alpha(\ell)) & \text{by definition of } //
\end{array}$$

On the other hand, for $\ell' \neq \ell$,

$$\begin{array}{rcll}
\alpha(s[v/\ell])(\alpha(\ell')) & = & \bigsqcup_{\alpha(\ell'')\sqsubseteq\alpha(\ell')} \alpha(s[v/\ell](\ell'')) & \\
& = & \bigsqcup_{\alpha(\ell'')\sqsubseteq\alpha(\ell')} \alpha(s(\ell'')) & \text{because } \ell' \neq \ell \\
& = & \alpha(s)(\alpha(\ell')) & \\
& \sqsubseteq & \alpha(s)[\alpha(v)//\alpha(\ell)](\alpha(\ell')). &
\end{array}$$

By the above two cases, $\alpha(s[v/\ell]) \sqsubseteq \alpha(s)[\alpha(v)//\alpha(\ell)]$. □

The soundness of the abstract semantics $\mathit{fix}\hat{\mathcal{F}}$ (in Figure 6) with respect to the concrete semantics $\mathit{fix}\mathcal{F}$ (in Figure 4) is that for an arbitrary expression e and input $x \in E \times S$ the concrete evaluation result $\mathit{fix}\mathcal{F}[\![e]\!]\,x$ must be implied by its abstract correspondence $\mathit{fix}\hat{\mathcal{F}}[\![e]\!]\,(\alpha_{E\times S}(x))$:

Theorem 2 *For any expression e*

$$\alpha_{V\times S} \circ \mathit{fix}\mathcal{F}[\![e]\!] \sqsubseteq \mathit{fix}\hat{\mathcal{F}}[\![e]\!] \circ \alpha_{E\times S}.$$

Proof. We can prove by Cousot's inductive soundness proof method [CC92, Proposition 4.3]. However, a very similar yet simpler proof method is applicable: fixpoint induction [Sto77, page 213].

Let $Q(f,g)$ be an assertion

$$Q(f,g) = \forall \text{expression } e : \alpha_Y \circ f[\![e]\!] \sqsubseteq g[\![e]\!] \circ \alpha_X.$$

Base case: $Q(\bot,\bot)$ holds because all abstraction functions are strict (Lemma 1).

Induction step: assuming that for continuous functions $\mathcal{E}$ and $\hat{\mathcal{E}}$, $Q(\mathcal{E},\hat{\mathcal{E}})$ holds we will show $Q(\mathcal{F}(\mathcal{E}),\hat{\mathcal{F}}(\hat{\mathcal{E}}))$ holds. Then, by the fixpoint induction, the goal $Q(\mathit{fix}\mathcal{F},\mathit{fix}\hat{\mathcal{F}})$ holds.

We will present the proof of the induction step for the `raise`, `handle`, and `case` expressions. Proofs for other expressions can be done similarly. For abstract functions, we will simply write α without its type subscript; from the context, it is clear which domain abstraction α indicates. For easy reference, we juxtapose the concrete and abstract interpretation rules separated by $\|$.

- `(raise` e`)`:

$$\begin{array}{ll} \text{letx} & \langle v, s_1\rangle = \mathcal{E}\ e\ \langle \sigma, s_0\rangle \\ \text{in} & \langle \underline{v}, s_1\rangle \end{array} \quad \| \quad \begin{array}{ll} \text{let} & \langle \hat{v_1}, \hat{s_1}\rangle = \hat{\mathcal{E}}\ e\ \hat{s_0} \\ \text{in} & \langle \hat{v_1}.\underline{\hat{X}} \sqcup (\hat{v_1}.\hat{X} : \underline{\hat{X}}), \hat{s_1}\rangle \end{array}$$

  By the induction hypothesis ($Q(\mathcal{E}, \hat{\mathcal{E}})$), $\alpha(\langle v, s_1\rangle) \sqsubseteq \langle \hat{v_1}, \hat{s_1}\rangle$. Thus, the final abstract result

$$\langle \hat{v_1}.\underline{\hat{X}} \sqcup (\hat{v_1}.\hat{X} : \underline{\hat{X}}), \hat{s_1}\rangle$$

  is consistent with the two possibilities of the concrete evaluation because

  - When $v \in \underline{Exn}$, $\alpha(\langle v, s_1\rangle) \sqsubseteq \langle \hat{v_1}.\underline{\hat{X}}, \hat{s_1}\rangle$.
  - When $v \in Exn$, $\alpha(\langle \underline{v}, s_1\rangle) \sqsubseteq \langle \hat{v_1}.\hat{X} : \underline{\hat{X}}, \hat{s_1}\rangle$.

- `(handle` e_1 `x` e_2`)`:

$$\begin{array}{ll} \text{let} & \langle v, s_1\rangle = \mathcal{E}\ e_1\ \langle \sigma, s_0\rangle \\ \text{in} & \text{if } v = \underline{v'} \in \underline{Exn} \\ & \text{then } \mathcal{E}\ e_2\ \langle \sigma[\ell/\mathtt{x}], s_1[v'/\ell]\rangle \\ & \quad (new\ \ell) \\ & \text{else } \langle v, s_1\rangle \end{array} \quad \| \quad \begin{array}{ll} \text{let} & \langle \hat{v_1}, \hat{s_1}\rangle = \hat{\mathcal{E}}\ e_1\ \hat{s_0} \\ & \langle \hat{v_2}, \hat{s_2}\rangle = \hat{\mathcal{E}}\ e_2\ \hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\mathtt{x}] \\ \text{in} & \langle |\hat{v_1}| \sqcup \hat{v_2}, \hat{s_1} \sqcup \hat{s_2}\rangle \end{array}$$

  By the induction hypothesis $Q(\mathcal{E}, \hat{\mathcal{E}})$, $\alpha(\langle v, s_1\rangle) \sqsubseteq \langle \hat{v_1}, \hat{s_1}\rangle$.

  - When v is not a raised exception, the concrete result $\langle v, s_1\rangle$ is subsumed by the abstract part $\langle |\hat{v_1}|, \hat{s_1}\rangle$.
  - On the other hand, when $v = \underline{v'} \in \underline{Exn}$,

$$\begin{array}{rcll} \alpha(\langle \sigma[\ell/\mathtt{x}], s_1[v'/\ell]\rangle) & \sqsubseteq & \alpha(s_1)[\alpha(v')//\mathtt{x}] & \text{by Lemma 2} \\ & \sqsubseteq & \hat{s_1}[\alpha(v')//\mathtt{x}] & \text{by monotonicity of } \hat{s_1} \\ & \sqsubseteq & \hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\mathtt{x}] & \text{because } \alpha(v) \sqsubseteq \hat{v_1} \end{array}$$

    By $Q(\mathcal{E}, \hat{\mathcal{E}})$ and by the monotonicity of $\hat{\mathcal{E}}$,

$$\begin{array}{rcl} \alpha(\mathcal{E}\ e_2\ \langle \sigma[\ell/\mathtt{x}], s_1[v'/\ell]\rangle) & \sqsubseteq & \hat{\mathcal{E}}\ e_2\ \alpha(\langle \sigma[\ell/\mathtt{x}], s_1[v'/\ell]\rangle) \\ & \sqsubseteq & \hat{\mathcal{E}}\ e_2\ \hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\mathtt{x}] = \langle \hat{v_2}, \hat{s_2}\rangle. \end{array}$$

- `(case x of` $p_1\ e_1 \cdots p_n\ e_n$`)`:

$$\begin{array}{lcl}
\mathcal{E}\ e_j\ \langle \sigma, s_0\rangle & & \\
(\kappa \stackrel{\text{match}}{=} p_j \text{ where} & \sqsubseteq & \bigsqcup_{1\le i\le n} \hat{\mathcal{E}}\ e_i\ (\hat{s_0}[\mathit{Screen}(\hat{s_0}(\mathtt{x}), \cdots)//\mathtt{x}]) \\
\ \langle \kappa, \ell\rangle = s_0(\sigma(\mathtt{x}))) & &
\end{array}$$

By the induction hypothesis $Q(\mathcal{E}, \hat{\mathcal{E}})$,

$$\begin{array}{rcl}
\alpha(\mathcal{E}\ e_j\ \langle \sigma, s_0\rangle) & \sqsubseteq & \hat{\mathcal{E}}\ e_j\ \hat{s_0} \\
& \sqsubseteq & \hat{\mathcal{E}}\ e_j\ (\hat{s_0}[\mathit{Screen}(\cdots)//\mathtt{x}]) \quad \text{because } \forall y : f \sqsubseteq f[y//x] \\
& \sqsubseteq & \bigsqcup_{1\le i\le n} \hat{\mathcal{E}}\ e_i\ (\hat{s_0}[\mathit{Screen}(\cdots)//\mathtt{x}])
\end{array}$$

- We will prove the case of Section 6.2.1 where we use distinct (subscripted) $\mathtt{x}_i$'s in each `case` branch. For `(case x of` $p_1\, e_1 \cdots p_n\, e_n$`)`, e_i' is equivalent to e_i except that every "`x`" inside e_i is replaced by "$\mathtt{x}_i$." Concrete semantics is: before a selected branch e_j is evaluated we allocate a new location for $\mathtt{x}_j$ and use this location inside e_j'.

$$\begin{array}{lcl}
\mathcal{E}\ e_j'\ \langle \sigma[\ell'/\mathtt{x}_j], s_0[v/\ell']\rangle & & \\
(\text{new } \ell',\ v \text{ as } \langle \kappa, \ell\rangle = s_0(\sigma(\mathtt{x})), & \sqsubseteq & \bigsqcup_{1\le i\le n} \hat{\mathcal{E}}\ e_i'\ (\hat{s_0}[\mathit{Screen}(\hat{s_0}(\mathtt{x}), \cdots)//\mathtt{x}_i]) \\
\ \text{and } \kappa \stackrel{\text{match}}{=} p_j) & &
\end{array}$$

Because $\alpha(s_0) = \hat{s_0}$,

$$\alpha(s_0(\sigma(\mathtt{x})) \stackrel{\text{let}}{=} v) \sqsubseteq \hat{s_0}(\mathtt{x}) \stackrel{\text{let}}{=} \hat{v}.$$

Furthermore, because v matches with p_j,

$$\alpha(v) \sqsubseteq \mathit{Screen}(\hat{v}, \{p_j\}, \{p_1, \cdots, p_{j-1}\}) \qquad (1)$$

Thus

$$\begin{array}{rcll}
\alpha(s_0[v/\ell']) & \sqsubseteq & \hat{s_0}[\alpha(v)//\alpha(\ell')] & \text{by Lemma 2} \\
& & \text{By (1) and monotonicity of } \hat{s_0} & \\
& \sqsubseteq & \hat{s_0}[\mathit{Screen}(\hat{v}, \{p_j\}, \{p_1, \cdots, p_{j-1}\}//\mathtt{x}_j] & (2)
\end{array}$$

By the induction hypothesis $Q(\mathcal{E}, \hat{\mathcal{E}})$,

$$\begin{array}{rcl}
\alpha(\mathcal{E}\ e_j'\ \langle \sigma[\ell'/\mathtt{x}_j], s_0[v/\ell']\rangle) & \sqsubseteq & \hat{\mathcal{E}}\ e_j'\ \alpha(\langle \sigma[\ell'/\mathtt{x}_j], s_0[v/\ell']\rangle) \\
& & \text{By (2) and monotonicity of } \hat{\mathcal{E}} \\
& \sqsubseteq & \hat{\mathcal{E}}\ e_j'\ (\hat{s_0}[\mathit{Screen}(\cdots)//\mathtt{x}_j]) \\
& \sqsubseteq & \bigsqcup_{1\le i\le n} \hat{\mathcal{E}}\ e_i'\ (\hat{s_0}[\mathit{Screen}(\cdots)//\mathtt{x}_i]).
\end{array}$$

□

7 Implementing the Analysis by Z1

Our analysis has been implemented by Z1 [YH93, Yi93].

The input to Z1 is a specification of the abstract interpreter of Figure 6. Neither the standard semantics nor the concrete semantics are processed by Z1. These non-abstract semantics are only necessary for us to derive a safe abstract interpreter.

An abstract interpreter specification in Z1 consists of three parts: lattice and set definitions (for abstract domains), auxiliary function definitions, and the main interpreter definition. The abstract domains ($\hat{S}, \hat{L}, \hat{V}$, and etc.) of the exception analysis are exactly defined as lattices in Z1. For example,

`(lattice S (-> L V))`	for $\hat{S} = \hat{L} \rightarrow \hat{V}$
`(lattice L (flat Ln))`	for $\hat{L} = Ln_\perp$
`(lattice V (* C D X R))`	for $\hat{V} = \hat{C} \times \hat{D} \times \hat{X} \times \underline{\hat{X}}$
...	
`(set Ln (index numIds))`	for $Ln = \{i \in \mathbf{Z} \mid 0 \leq i \leq \texttt{numIds}()\}$

defines the three abstract domains. Note that, in the definition of set `Ln`, `numIds` is a procedure that is implemented by us to return the number of allocation sites of an input program. Over these lattices and sets, the abstract interpreter $\hat{\mathcal{F}}$ is specified.

The output from Z1 is a C program that becomes an executable analyzer when linked with the target language [11] parser and syntax-tree interface procedures. This parser and the interface procedures must be implemented in C by us.

The specification of our abstract interpreter has 426 lines. Generated C code has 6965 lines. The executable size is 427 Kbytes.

The generated analyzer computes a *collecting analysis* of an input program. The collecting analyzer computes, for each program point of the input program, an abstract state that characterizes the run-time states that can occur at that point during execution. In Z1 a program state is a pair of the pre-state and the post-state, and the program points are the nodes of the program's abstract syntax tree. As an example, for a function application expression "`(apply` e_1 e_2`)`" the pre-state at the program point `(apply` $\cdots$`)` is the program state immediately before the beginning of the application. The post-state is the state after the completion of the application.

Z1's derivation of a collecting analysis from an abstract interpreter functional $\hat{\mathcal{F}}$ is straightforward. Note that an abstract interpreter is a function that defines, for each language construct, its evaluation rule: a state transformer from a pre-state to a post-state. In our case, the abstract interpreter $\hat{\mathcal{E}}$ has therefore the

[11] The target language is the language in which the programs to analyze are written. In our case, the intermediate language in Section 2.

following type

$$\hat{\mathcal{E}}: \Sigma \to \hat{S} \to \hat{V} \times \hat{S}$$

where Σ is the set of program points, and $\hat{S}$ (abstract store) is the lattice of pre-states, $\hat{V} \times \hat{S}$ (pair of abstract value and store) is the lattice of post-states. This $\hat{\mathcal{E}}$ function, which is usually recursively defined, is embedded in its associated functional $\hat{\mathcal{F}}$:

$$\begin{aligned}\hat{\mathcal{F}} = \lambda\hat{\mathcal{E}}.\lambda e.\lambda\hat{s_0}.\ & \text{case } e \text{ of} \\ & (\texttt{raise}\ e')\text{: } \cdots \hat{\mathcal{E}}(e', \hat{s_0}) \cdots \\ & (\texttt{handle}\ e_1\ \texttt{x}\ e_2)\text{: } \cdots \hat{\mathcal{E}}(e_1, \hat{s_0}) \cdots \\ & \cdots\end{aligned}$$

This abstract interpreter functional $\hat{\mathcal{F}}$, an input program P (actually, P's set of program points Σ_P), and the initial pre-state $\hat{s_0}$ that is valid at the P's start point, are three inputs to the collecting analysis computation

$$\mathit{Tabulate}(\hat{\mathcal{F}}, \Sigma_P, \hat{s_0}). \qquad \text{(see Figure 7)}$$

The analysis results (two tables T_X and T_Y) have, for each program point p, a pre-state $T_X(p) \in \hat{S}$ and a post-state $T_Y(p) \in \hat{V} \times \hat{S}$ that characterize run-time states that occur before and after that point during execution. The safeness of this collecting analysis algorithm is proven in [CHY95]. Note that the fixpoint algorithm is one that can be used only if the lattices are finite and the functions are monotonic. The monotonicity of $\hat{\mathcal{F}}$ in Figure 6 is straightforward to show.

Our collecting analyzer uses the same idea as minimal function graph [JM86] or collecting interpretation [HY88]. It iterates, given an initial state valid at the program's start point, until all reachable states are computed for each program point. At each iteration, the algorithm computes, for each program point, a new state value reachable from the state values computed by the previous iteration. Note that our method is not the denotational approach [Bur87, BHA85, Nie82], where the table of the program's semantic function is computed across the entire argument spaces.

Figure 7 presents a simplified version of our collecting analysis algorithm. In reality, Z1 uses a worklist algorithm that invokes $\hat{\mathcal{F}}$ only for a subset of program points whose T_X and T_Y entries were changed by the previous iteration. The fixpoint computation performance may vary, depending on the order in which elements are selected from the worklist. Z1 uses the heuristics in [CH93] for the selection order, which is guided by the structure of the dependence graph (an expression e_1 depends on another expression e_2 if the evaluation of e_1 requires that of e_2) in order to approximate the optimal order of selecting an element from the worklist. (Similar fixpoint algorithms are reported in [CDMH93, Jør93]. Our algorithm may be seen as a mixture of the top-down and bottom-up fixpoint algorithms [CH92].) The reader may refer to [CHY95, CH93] for the complete algorithm and the proof of its correctness.

```
eval(p: Σ, x: X): Y
begin
        if x ⋢ T_X(p) then
              T_X(p)  = T_X(p) ⊔ x;      /* join the pre-state */
        return T_Y(p);
end

Tabulate(F: (Σ → X → Y) → Σ → X → Y, Σ_P: 2^Σ, x_0: X): void
T_X, T'_X: Σ_P → X;                      /* pgm point to pre-state */
T_Y, T'_Y: Σ_P → Y;                      /* pgm point to post-state */
begin
        ∀p ∈ Σ_P : T_X(p) = ⊥_X,   T_Y(p) = ⊥_Y;
        T_X(p_0) = x_0;                  /* pre-state at the pgm's entry point */
        repeat
              ⟨T'_X, T'_Y⟩ = ⟨T_X, T_Y⟩;   /* remember the previous iteration */
              foreach p ∈ Σ_P            /* for each pgm point */
                    T_Y(p)  = F(eval, p, T_X(p));  /* compute the post-state */
        until  (T_X ⊑ T'_X) ∧ (T_Y ⊑ T'_Y)        /* repeat until stable */
end
```

In the exception analysis, F is $\hat{\mathcal{F}}$, *eval* is $\hat{\mathcal{E}}$, X is $\hat{S}$, and Y is $\hat{V} \times \hat{S}$ of the abstract semantics in Figure 6.

Figure 7: A primitive algorithm to compute collecting analysis from an abstract interpreter functional F

7.1 Analysis Snapshots

Some snapshots of our analysis of the last example in Section 1.3 are shown in the following table. For convenience, the program is shown here again:

```
exception ERROR of int list
exception EXIT of int list
fun f(n, x, y) =
     if n<0 then raise (x [n])          (1)
     else if n=0 then raise (y nil)     (2)
     else n
fun g(m, x, y) =
     f(m, x, y)                         (3)
     handle ERROR [n] => g(n, y, x)     (4)
          | EXIT nil => 0               (5)
fun main(c) = g(c, ERROR, EXIT)
```

The following table shows the raised exception and the store at the point right after the call `f(m,x,y)` at line `(3)`. The column "non fixpoint" shows the case when `f` is initially called. It shows exception $\langle$`ERROR`,$\ell\rangle$ and $\langle$`EXIT`,$\ell'\rangle$ are raised, whose arguments (at locations ℓ and ℓ') have constructors `CONS`[12]and `NIL`, respectively. Location ℓ (respectively ℓ') is the one allocated for "`[n]`" (resp. "`nil`") in line `(1)` (resp. line `(2)`). After the multiple calls to `f` (by recursive call to `g` at line `(4)`), raised exceptions `ERROR` and `EXIT` have both `CONS` and `NIL` as their arguments. Among these exceptions, $\langle$`ERROR`,$\ell'\rangle$ and $\langle$`EXIT`,$\ell\}$ escape the handler, which our analysis detects.

	non fixpoint	fixpoint
$\hat{v}.\underline{X}$	$\{\langle\texttt{ERROR}, \ell\rangle, \langle\texttt{EXIT}, \ell'\rangle\}$	$\{\langle\texttt{ERROR}, \ell\rangle, \langle\texttt{ERROR}, \ell'\rangle, \langle\texttt{EXIT}, \ell\rangle, \langle\texttt{EXIT}, \ell'\rangle\}$
$\hat{s}$ (x)	$\{\lambda x.(\texttt{exn ERROR } x)\}$	$\{\lambda x.(\texttt{exn ERROR } x), \lambda x.(\texttt{exn EXIT } x)\}$
$\hat{s}$ (y)	$\{\lambda x.(\texttt{exn EXIT } x)\}$	$\{\lambda x.(\texttt{exn ERROR } x), \lambda x.(\texttt{exn EXIT } x)\}$
$\hat{s}$ (ℓ)	$\{\langle\texttt{CONS}, -\rangle\}$	$\{\langle\texttt{CONS}, -\rangle\}$
$\hat{s}$ (ℓ')	$\{\langle\texttt{NIL}, \bot\rangle\}$	$\{\langle\texttt{NIL}, \bot\rangle\}$

8 Discussion

8.1 Semantic Sparse Analysis

We need a sparse analysis technique for reducing our analysis cost. It seems wasteful to trace all expressions of the input program, because only a small subset of the expressions may generate the exception behavior (creating, raising and handling). In conventional data flow analysis framework, many techniques [DRZ92, CCF91, CFR$^+$89, DGS94] have been developed. However, these methods are problematic for "higher-order" languages like SML, because the SML program's flow graph, which is a prerequisite of the conventional methods, is not available *prior to* the analysis.

[12] List constructor name.

We will informally outline a semantics-based sparse analysis technique for the exception analysis. We will discuss at the level of the concrete semantics. Deriving an abstract correspondence will be straightforward. This sparse analysis technique is not implemented for our analysis. A similar idea was discussed in [Har89] for interprocedural dependence analysis of Scheme programs.

Proposition 1 *Before we evaluate an expression, we can conservatively decide whether the evaluation will have the exception behavior or not, by examining the expression text with respect to the current environment and the store.*

Before we evaluate an expression e

$$\mathcal{E}\ e\ \langle \sigma, s \rangle$$

we can collect all values that might be used during this evaluation. These values consist of those that are "reachable" from the free variables $FV(e)$ of e. This reachable set R is constructed as follows. First, it is initialized with the values $\{s(\sigma(\mathtt{x})) | \mathtt{x} \in FV(e)\}$ of the free variables of e. For each closure value $\langle e', \sigma' \rangle$ in R, we add to R the values $\{s(\sigma'(\mathtt{x}')) | \mathtt{x}' \in FV(e')\}$ of the closure's free variables, and so on. The final, transitively closed set R will contain the reachable values during the evaluation of e. This process of constructing the R set is analogous to the mark phase of the garbage collection. The root set of our case is the free variables of the expression e.

Conservative conditions under which the evaluation "$\mathcal{E}\ e\ \langle \sigma, s \rangle$" may cause exception behavior are as follows: (We consider, for simplicity, that the expression e is also included in R as a closure $\langle e, \sigma \rangle$.)

- When there exists a closure in R whose body has a `raise`, `handle` or `exn` expression.
- When there is an expression that receives a exception value, manipulates it and returns it, without raising, handling nor creating a new exception. That is, when there exists a closure in R whose type has the exception type or a polymorphic type. Note that the intermediate expression can have the type information imported from the type inference phase (of the SML/NJ compiler) for its SML source.
- When the current expression e occurs during the computation of an exception argument (like in "`(exn E` e`)`"). This is because our analysis must take the exception arguments into account during the handler matches.

When we evaluate an expression, we check the above conditions. If any one of the conditions holds, we evaluate the expression. Otherwise, we skip the evaluation. This method will reduce the analysis cost, assuming that the time spent computing R is less than the time spent evaluating unnecessarily many expressions.

The computation cost of R for every expression may offset the gain we expect. In this case, we may apply the sparse evaluation rule only for, say, the function applications.

8.2 Conclusion

We have presented a static analysis that detects exceptions that are raised and never handled inside Standard ML programs. This analysis improves software safety by predicting, before program execution, the abnormal termination caused by potentially unhandled exceptions.

The analysis is specified as a finite, abstract semantics of an intermediate language. From this semantics, an executable collecting analyzer is derived. This derivation is done by a tool called Z1 [YH93, Yi93]. The generated analyzer was used to analyze SML/NJ Libraries, ML-YACC and ML-LEX programs.

The intermediate language is defined such that the mechanism of SML's exception propagation becomes explicit in its text. For example, every handler expression is augmented with a raise expression that will re-raise the exceptions that are not caught by the handler patterns.

Our analysis is limited to SML programs that are type-correct and are operationally invariant even if the generative nature of SML's data-type and exception declarations is not considered.

Acknowledgment

I thank Lal George, Carl Gunter, Lorenz Huelsbergen, Dave MacQueen, John Reppy, Jon Riecke and Zhong Shao for discussions and helps during this work. In particular, my thanks to Dave for his support and encouragement. I thank Manuel Fahndrich for correcting some typos. Also, thanks to anonymous referees for their helpful comments.

References

[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.

[BHA85] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. Technical Report DoC 85/6, Imperial College, 1985.

[Bou92] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992. Also as a tech report: DEC Paris Research Lab, March, 1992.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Lecture Notes in Computer Science*, volume 735 of *Proceedings of Formal Methods in Programming and Their Applications*, pages 128–141. 1993.

[Bur87] Geoffrey L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Department of Computing, Imperial College, University of London, March 1987.

[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.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992. Also as a tech report: Ecole Polytechnique, no. LIX/RR/92/10.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[CDMH93] B. Le Charlier, O. Degimbe, L. Michel, and P. Van Hentenryck. Optimization techniques for general purpose fixpoint algorithms: practical efficiency for the abstract interpretation of prolog. In *Lecture Notes in Computer Science*, volume 724, pages 15–26. Springer-Verlag, proceedings of the third international workshop on static analysis edition, September 1993.

[CFR+89] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and control dependence graph. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, 1989.

[CH92] Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, May 1992.

[CH93] Liling Chen and Luddy Harrison. Efficient computation of fixpoints that arise in complex program analysis. Technical Report CSRD Report No. 1245, Center for Supercomputing R & D, University of Illinois at Urbana-Champaign, 1993.

[CHY95] Liling Chen, Luddy Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 1995.

[DGS94] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *International Confererence on Compiler Construction*, April 1994.

[DRZ92] D. Dhamdhere, B. Rosen, and F. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[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.

[Har89] Williams L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[HDCM93] Pascal Van Hentenryck, Olivier Degimbe, Baudouin Le Charlier, and Laurent Michel. The impact of granularity in abstract interpretation of prolog. In Patrick Cousot, Moreno Falaschi, Gilberto File, and Antoine Rauzy, editors, *Lecture Notes in Computer Science*, volume 724. Springer-Verlag, proceedings of the third international workshop on static analysis edition, 1993.

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

[HY88] Paul Hudak and Jonathan Young. A collecting interpretation of expressions (without powerdomains). In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 107–118, 1988.

[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.

[JM86] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs: Abridged version. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, 1986.

[Jør93] N. Jørgensen. Chaotic fixpoint iteration guided by dynamic dependency. In *Lecture Notes in Computer Science*, volume 724, pages 27–44. Springer-Verlag, proceedings of the third international workshop on static analysis edition, September 1993.

[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.

[Nie82] Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18, 1982.

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

[Sto77] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[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.

[Yi93] Kwangkeun Yi. *Automatic Generation and Management of Program Analyses*. PhD thesis, University of Illinois at Urbana-Champaign, August 1993. Report UIUCDCS-R-93-1828.