Compile-time Detection of Uncaught Exceptions in Standard ML Programs

Kwangkeun Yi*

AT&T Bell Laboratories

Abstract. We present a static analysis that detects potential runtime exceptions that are raised and never handled inside Standard ML programs. This analysis enhances the software safety by predicting, *prior to* the program execution, the abnormal termination caused by unhandled exceptions.
Our analysis prototype has been implemented by using a semantics-based analyzer generator and has been successfully tested with real Standard ML programs consisting of thousand lines.
We introduce semantic sparse analysis to reduce the analysis cost without compromising the analysis accuracy. In this method, expressions will only be analyzed when their evaluations are relevant to our analysis.

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.

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 the program safety. A program can terminate abnormally when an exception is raised and never handled.

Our goal is to develop a compile-time tool to eliminate 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.

* kwang@research.att.com, AT&T Bell Laboratories, Rm 2A-421, 600 Mountain Avenue, Murray Hill, NJ 07974-0636, U.S.A.

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

```
Error(``at line 7'')
```

constructs the `Error` exception with the string argument. 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 called an exception packet. In this paper, 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 the 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 raised exceptions from the subcomputation of e. When the exception 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 paired properly 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 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 invoke `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]`[2] in

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

As an example of our analysis, consider the following program where exception constructor and its argument are passed as function parameters[3].

```
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 [a] => g(a+1, y, x)   (4)
          | EXIT nil => 0
fun main(c) = g(c, ERROR, EXIT)
```

When `g` is first called inside `main`, raised exceptions `ERROR [c]` and `EXIT nil` are handled by the handler inside `g`. Meanwhile, when `g` is called recursively (line `(4)`), the two exception constructors are swapped. Hence, the raised exceptions `EXIT [a+1]` and `ERROR nil`, at this time, cannot be handled by the handler. Our analysis detects this situation.

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 the exception declaration binds a new, unique name to the exception constructor. For example, in the following *incorrect* definition of the factorial function, each recursive call to `fact` generates a new instance of exception `ZERO`. Therefore, the handler in line `(3)`, which can handle exceptions declared only in its textual scope, does not handle the exception that is raised inside the recursive call `fact(n-1)`. Hence this function always stops with an uncaught exception `ZERO`.

```
fun fact(n) =
    let exception ZERO                          (1)
    in if n <= 0 then raise ZERO                (2)
       else n * fact(n-1) handle ZERO => 1      (3)
    end
```

Our analysis cannot analyze the programs that utilize the generative nature of the exception (and the datatype) declarations. This limitation is not severe; exceptions (and also datatypes) are largely declared at the global scope or at the structure[4] level, or we can hoist existing local declarations out to the global level without affecting the "observational" semantics of the programs. Programs where this hoisting is impossible cannot be analyzed correctly by our analysis.

[2] `[1]` is the singleton list of 1.

[3] We have found such cases in the source (in `src/env/pickle.sml`) of SML/NJ 1.01 compiler.

[4] A structure in SML is a unit for modular programming; it corresponds roughly to a file in C programming.

1.3 Analysis Methodology

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 the information about uncaught exceptions.

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

- Unhandled exceptions of global 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.

1.4 Implementation Status

Our analysis has been implemented by Z1 [YH93, Yi93] and has been successfully used to analyze "real" SML programs such as SML/NJ libraries, ML-YACC, and ML-LEX.

At the moment, however, the analysis is not fast enough to be used interactively. For example, the analysis prototype takes 6 hours for ML-LEX.[5] In average, 38 iterative evaluations for each expression are required to reach to the fixed-point (the analysis end point).

We are working on several ways to improve the analysis speed. In Sect. 8 we will present one idea (semantics-based sparse analysis) that will be embodied in our analysis.

1.5 Related Works

Guzmán and Suárez [GS94] reported an instrumented type-inference system to collect unhandled exceptions for a simplified core ML. Their approach may not be strong enough to deal with the realistic use of the SML exceptions. For example, they regarded exceptions as just names, without argument values. To handle exceptions with arguments, they may need an idea similar to the "regions" [TJ92] in order to approximate the range of exception arguments.

On the other hand, type-inference or, in general, constraints-resolution based program analysis [TJ92, TT94, LG88, JG91, Hei92] that uses unification as its computation method, seems to have some appealing characteristics: relatively small analysis cost and a natural support for separate analysis. We plan to have a comparative study of the two analysis methods for the instance of the exception analysis.

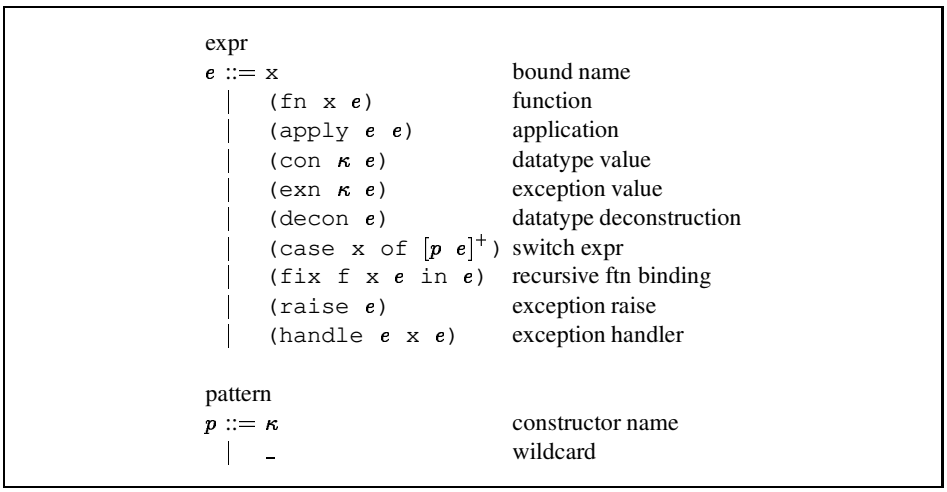
In the conventional data flow analysis framework, Hennesey [Hen81] discussed several optimization problems for the programs with exception handling facilities.

[5] ML-LEX program has 1229 lines. After being translated in our intermediate language, it has 14,502 expressions, 8 exceptions, 8 handlers, and 47 raise expressions. The analysis result shows that ML-LEX may have unhandled exceptions: `Subscript`, `error` and `lex_error`.

2 The 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 1 shows this intermediate language.

For brevity, we present a simplified version of the language. We have omitted numbers, strings, records, primitive arithmetic operators, and memory operators (like allocation, assignment and dereference).



expr		
e ::=	`x`	bound name
|	`(fn x` e`)`	function
|	`(apply` e e`)`	application
|	`(con` κ e`)`	datatype value
|	`(exn` κ e`)`	exception value
|	`(decon` e`)`	datatype deconstruction
|	`(case x of` $[p\ e]^+$`)`	switch expr
|	`(fix f x` e `in` e`)`	recursive ftn binding
|	`(raise` e`)`	exception raise
|	`(handle` e `x` e`)`	exception handler
pattern		
p ::=	κ	constructor name
|	_	wildcard

Fig. 1. (Simplified) Abstract Syntax of the Intermediate Language

The intermediate language is a call-by-value higher-order language (based on the Lambda [App92] of the SML New Jersey (SML/NJ) compiler). Informally, the semantics of the language is as follows. 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`)`. The case expression `(case x of` p_1 $e_1 \cdots$`)` branches to e_k when the value of `x` has a constructor name that matches with the p_k pattern. The handle expression `(handle` e_1 `x` e_2`)`, where e_2 will typically be a `case` expression, evaluates e_1 first. When the 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 `f` $= \lambda$`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 form are not supported by the abstract syntax in Fig. 1. For convenience, we use numbers, for example.)

– When case 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.
On the other hand, the handler patterns are always augmented with an extra raise expression, in order to re-raise exceptions that are not caught:

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

Note that the "x" has the exception value that was raised inside e. Hence the raise expression "`(raise x)`" 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:

```
                                        (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))
```

– 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.
– 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
... T, ...                ⟹      ... (fn x (con T x)), ...
```

– 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 is not desirable for the abstraction. Next, we tailor this semantics into one (termed *concrete semantics*) that is easier to abstract. Finally, we abstract the concrete semantics, resulting in an finite, approximate interpreter that is suitable for the compile-time collecting analysis.

4 Standard Semantics

Let us first review some notations. $A_\perp$ is the lifted cpo: bottom and incomparable elements of set A. For two cpos A and B, $A + B$ is the coalesced sum ($\perp_A = \perp_B = \perp_{A+B}$), $A \times B$ is the Cartesian product with the component-wise order, and $A \to B$ consists of strict, continuous functions with the point-wise order.

The standard evaluation function $\mathcal{E}$ returns a value of an expression for a given environment:

$$\mathcal{E}: Expr_\perp \to Env \to Value.$$

An environment

$$\sigma \in Env = Id_\perp \to Value$$

is a map from variables $Id_\perp$ 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 $v \in Value$ is either a closure $Closure$, a datatype value $Data$, an exception value Exn or an exception packet (a raised exception) $\underline{Exn}$:

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

The closure is, as usual, a pair of the function text and the environment at the function definition. The datatype value is a pair of a constructor name and its argument (similarly for the exception value):

$$\begin{aligned} Data &= DataCon_\perp \times Value \\ Exn &= ExnCon_\perp \times Value \end{aligned}$$

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

We do not include the standard semantics. It should be straightforward to derive the formal semantics from these domain definitions and from the informal description in Sect. 2.

5 Concrete Semantics

A semantics that is defined over recursively-defined domains is troublesome when we derive from it a finite, abstract interpreter, because we must find the abstractions that cut the reflexivity in order to achieve the finite domains. The standard semantics of the previous section is of such a case; it has the recursively-defined value domain $Value$:

$$\begin{aligned} Value &= Closure + Data + \cdots \\ &= (Expr_\perp \times (Id_\perp \to Value)) + (DataCon_\perp \times Value) + \cdots \end{aligned}$$

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[6]: 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):

$$\mathcal{E}: Expr_{\perp} \rightarrow Env \times Store \rightarrow Value \times Store$$

When a value v needs to be bound to a variable x, a new location ℓ is allocated in the store $s \in Store = Loc \rightarrow Value$ and the value is written in that location $s[v/\ell]$. The environment $\sigma \in Env = Id_{\perp} \rightarrow Loc$ 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 x's value is needed, 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 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 values and exceptions use, for the argument component, the location Loc in place of the $Value$ domain. That is, when a datatype value (a pair $\in DataCon \times Value$ 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. 2.

5.1 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

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

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.

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

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

$$
\begin{array}{lll}
\mathcal{F} = \lambda\mathcal{E}.\lambda e.\lambda\langle\sigma, s_0\rangle.\ \text{case } e \text{ of} & & \\
\texttt{x}: & s_0(\sigma(\texttt{x})) & \\
\texttt{(raise } e\texttt{)}: & \text{letx } \langle v, s_1\rangle = \mathcal{E}\ e\ \langle\sigma, s_0\rangle & \\
 & \text{in } \langle \underline{v}, s_1\rangle & \\
\texttt{(handle } e_1 \texttt{ x } e_2\texttt{)}: & \text{let } \langle v, s_1\rangle = \mathcal{E}\ e_1\ \langle\sigma, s_0\rangle & \\
 & \text{in if } v = \underline{v'} \in \underline{\mathit{Exn}}\ \ (\mathit{new}\ \ell) & \\
 & \quad \text{then } \mathcal{E}\ e_2\ \langle\sigma[\ell/\texttt{x}], s_1[v'/\ell]\rangle & \\
 & \quad \text{else } \langle v, s_1\rangle & \\
\texttt{(case x of } p_1\, e_1 \cdots p_n\, e_n\texttt{)}: & \mathcal{E}\ e_j\ \langle\sigma, s_1\rangle & \\
 & \quad (\kappa \overset{\text{match}}{=} p_j, \langle\kappa, \ell\rangle = s_1(\sigma(\texttt{x}))) & \\
\texttt{(apply } e_1\ e_2\texttt{)}: & \text{letx } \langle\langle\texttt{(fn x } e\texttt{)}, \sigma'\rangle, s_1\rangle = \mathcal{E}\ e_1\ \langle\sigma, s_0\rangle & \\
 & \quad \langle v, s_2\rangle = \mathcal{E}\ e_2\ \langle\sigma, s_1\rangle & \\
 & \text{in } \mathcal{E}\ e\ \langle\sigma'[\ell/\texttt{x}], s_2[v/\ell]\rangle\ \ (\mathit{new}\ \ell) & \\
\texttt{(fn x } e\texttt{)}: & \langle\langle(\text{fn x } e), \sigma\rangle, s_0\rangle & \\
\texttt{(fix f x } e_1 \texttt{ in } e_2\texttt{)}: & \mathcal{E}\ e_2\ \langle\sigma[\ell/\texttt{f}], s_0[\langle\texttt{(fn x } e\texttt{)}, \sigma[\ell/\texttt{f}]\rangle/\ell]\rangle & \\
 & (\mathit{new}\ \ell) & \\
\texttt{([con|exn] } \kappa\ e\texttt{)}: & \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\ \ (\mathit{new}\ \ell) & \\
\texttt{(decon } e\texttt{)}: & \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}
$$

We write $f[y/x]$ to represent the function that is identical to f, except at x, where its value is y.

Fig. 2. 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. The abstraction procedure consists of the abstractions of the concrete domains and of the concrete evaluation function.

We require that the abstract domains be finite lattices. Each element $\hat{x} \in \hat{D}$ in an abstract domain $\hat{D}$ denote an *ideal*[7] $\gamma(\hat{x}) \subseteq D$ of concrete values. The partial order

[7] 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

$\hat{x} \sqsubseteq \hat{y}$ in the abstract domain is when $\hat{x}$'s information is more precise than $\hat{y}'s$, 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 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.

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 correctness.

6.1 Abstracting Locations

In abstract semantics, we use a single location for each allocation sites 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. 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 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 to represent all locations allocated at that site, we can use multiple abstract locations each of which represents an exclusive subset of those locations. 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]).

---

upwardly complete (every chain in I has the least upper bound inside I). Because each ideal includes the bottom element of the concrete domain, when our concrete domain is a function space whose bottom element represents non-termination, our abstract element cannot exclude non-termination as a part of its meaning.

The 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.

6.2 Abstract Evaluation

Abstract interpreter for the exception analysis is shown in Fig. 3. 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 use $\mathtt{x}_e$ where the subscript can be the expression index.

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 the subcomputations, is collected in the value of the conclusion.

Consider the handle expression.

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

We first evaluate the expression e_1. The handler need to handle the raised exception $\hat{v_1}.\underline{\hat{X}}$, if any, during the evaluation. Hence we evaluate the second expression e_2, which is usually a `case` expression, with the store $\hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\mathtt{x}]$ that holds the exception value $\hat{v_1}.\underline{\hat{X}} : \hat{X}$ at `x`. The value in conclusion is either the value $\hat{v_1}$ after the evaluation of e_1, when this expression does not raise any exception, or the value $\hat{v_2}$ after the handling, when the first expression raises 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. If the handler patterns of e_2 is not complete enough to handle all cases, the exceptions bound to `x` is re-raised,[8] hence is captured inside $\hat{v_2}$.

Auxiliary operation

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

which is used for `case` expression, chooses among the data values $\hat{v}.\hat{D}$ and exceptions $\hat{v}.\hat{X}$ those that matches with a pattern in P but not with a pattern in Q.

Suppose `x` below contains two exceptions {`E`, `F`}.

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

[8] Note that when an SML source program is translated into the intermediate language, appropriate `raise` expressions added for non-complete matches – see discussions in Sect. 2.1.

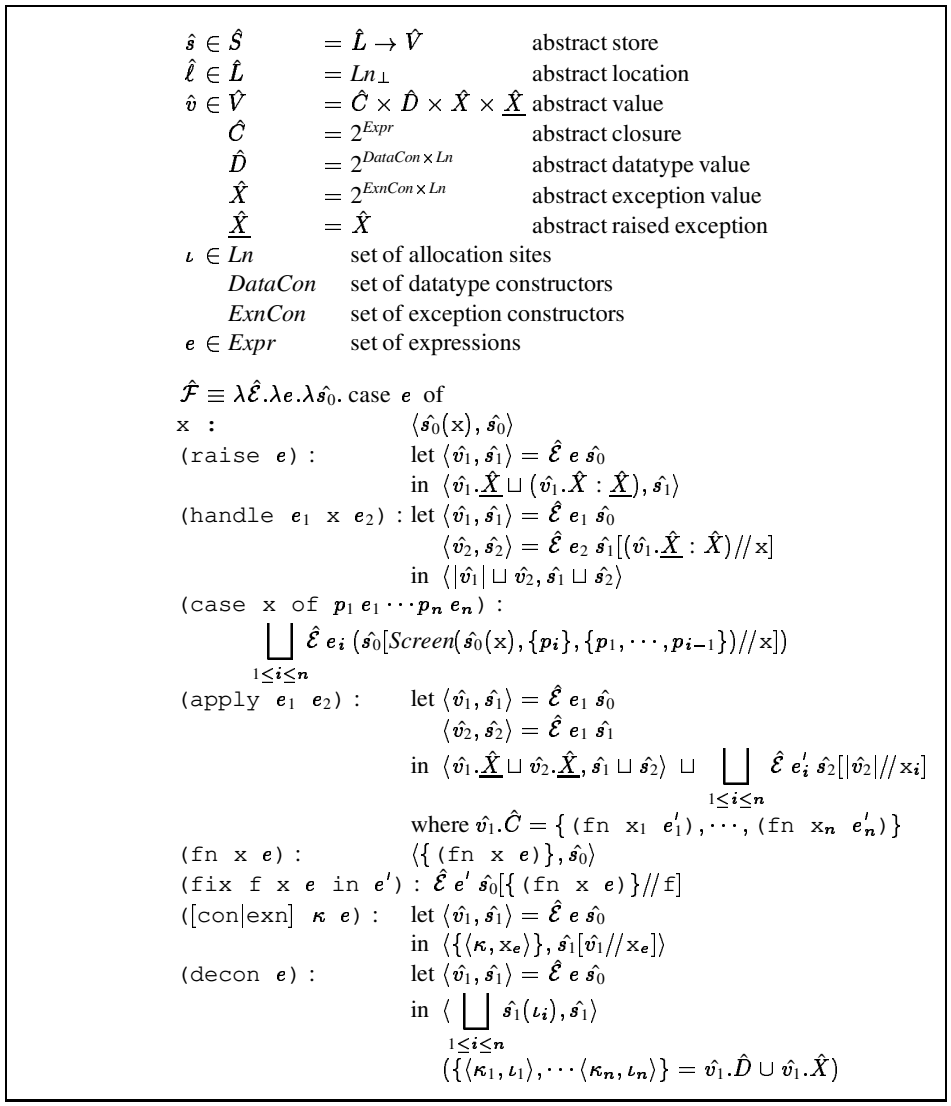


Fig. 3. Abstract Semantics for Exception Analysis

When we analyze the second branch `(raise x)`, exception `E` should not be bound to `x` because this exception matches with the first pattern. This effect is intended by the *Screen* operation.

The correctness of our abstract semantics can be proven by the fixed-point induction [Sto77]. A part of the proof is in Appendix A.

Accuracy Concern: An Implementation Details The rule for the `case` expression can have a bad effect on the accuracy, 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` (remember the notation: $f[y//x] = f[y \sqcup f(x)/x]$). Therefore, even though the $Screen$ operation removes some values of `x` that are not relevant to each branch, the store value for `x` still remains the same as before for each evaluation of the branches.

This problem is remedied, in our implementation, by using different location at each branch. Each trimmed value of `x` for each branch is bound to a unique name for that branch, instead of always to the same `x`. And the "`x`" inside each branch e_i is replaced by its unique name, say, "$\mathtt{x}_i$", and the abstract evaluation rule becomes

$$\texttt{(case x of } p_1\ e_1 \cdots p_n\ e_n\texttt{)} : \bigsqcup_{1 \le i \le n} \hat{\mathcal{E}}\ e_i\ (\hat{s_0}[Screen(\hat{s_0}(\mathtt{x}), \{p_i\}, \{p_1, \cdots, p_{i-1}\})//\mathtt{x}_i])$$

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 $\hat{\mathcal{F}}$ of Fig. 3. 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. The specification language of Z1 is a strongly-typed applicative language with user-defined types. A user-defined type is either a set or a lattice. The specification consists of four parts: lattice and set definitions (for abstract domains), auxiliary functions, abstract syntax tree interface, and the main interpreter definition.

The output from Z1 is a C program which becomes an executable analyzer when linked with libraries and the target language[9] parser. (This parser must be implemented by the user.)

The specification has 426 lines. Generated C code has 6965 lines. The executable size is 427 Kbytes.

The generated analysis computes a collecting analysis of a program P. The analysis result is a pair of two tables T_X and T_Y that have, for each expression $\sigma \in P$, a pair of a pre-state $T_X(\sigma)$ and a post-state $T_Y(\sigma)$ that describe run-time states that occur before and after that expression, respectively. In our case, the pre-state is $\hat{S}$ (store) (input to the abstract interpreter), and the post-state is $\hat{V} \times \hat{S}$ (output of the abstract interpreter).

The reader may want to see [YH93, Yi93] for a detailed description of Z1.

7.1 Analysis in Action

Some snapshots of the analysis of the example program in Sect. 1.2 are shown in the following table. This tables 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 called initially. It shows exception $\langle \texttt{ERROR}, \ell \rangle$ and $\langle \texttt{EXIT}, \ell' \rangle$ are raised, whose

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

arguments (at locations ℓ and ℓ') has constructor CONS[10] 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), raised exceptions ERROR and EXIT have both CONS and NIL (at locations ℓ and ℓ') as the arguments. Among these exceptions, $\langle$ERROR,$\ell'\rangle$ and $\langle$EXIT,$\ell\}$ escape the handler, which our analysis detects.

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

8 Semantic Sparse Analysis

It is 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). We need a sparse analysis technique like [DRZ92, CCF91, CFR+89, DGS94] that have been developed in the conventional flow analysis framework.

We will informally outline a semantics-based sparse analysis[11] technique for the exception analysis. We will discuss at the level of the concrete semantics. Deriving an abstract correspondence will be straightforward.

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 "reacheable" from the free variables $FV(e)$ of e. This reacheable 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 reacheable values during the evaluation of e.[12]

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

[10] List cell constructor.

[11] The conventional sparse analysis methods are problematic when applied to "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] 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.

- 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.[13]
- When the current expression e occurs during the computation of an exception argument. 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 to evaluate 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.

9 Conclusion

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

The analysis is a collecting analysis based on an abstract semantics of an intermediate language, into which Standard ML programs are translated before the analysis. The analysis has been implemented by using Z1 [YH93, Yi93] and has been successfully used to analyze SML/NJ Libraries, ML-YACC and ML-LEX programs.

We have proposed a semantics-based sparse analysis technique that analyzes an expression only when the expression generates the exception behavior.

Acknowledgement

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 encourgement. Also, thanks to the program committee for their helpful comments.

References

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

[13] 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.

[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.

[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.

[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.

[Hen81] John Hennessy. Program optimization and exception handling. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 200–206, 1981.

[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.

[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.

[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.

A Correctness Proof

The correctness of the abstract semantics $fix\hat{\mathcal{F}}$ (in Figure 3) with respect to the concrete semantics $fix\mathcal{F}$ (in Figure 2) is stated as

$$\alpha_Y \circ fix\mathcal{F} \sqsubseteq fix\hat{\mathcal{F}} \circ \alpha_X .$$

The α_X (respectively, α_Y) is the abstraction function of the pre-state $Env \times Store$ (respectively, the post-state $Value \times Store$). Abstraction functions for domains are as follows:

$$\begin{array}{ll}
\alpha_L : Loc \to \hat{L} & = \lambda \ell.\text{if } \ell = \bot \text{ then } \bot \text{ else } \iota \text{ where } \ell \in Allocated(\iota) \\
\alpha_C : Closure \to \hat{C} & = \lambda c.\text{if } c = \bot \text{ then } \{\} \text{ else } \{e\} \text{ where } c = \langle e, \sigma \rangle \\
\alpha_D : Data \to \hat{D} & = \lambda d.\text{if } d = \bot \text{ then } \{\} \\
 & \quad \text{else } \{\langle \kappa, \iota \rangle \,|\, d = \langle \kappa, \ell \rangle \ \wedge \ \ell \in Allocated(\iota)\} \\
\alpha_X : Exn \to \hat{X} & = \lambda x.\text{if } x = \bot \text{ then } \{\} \\
 & \quad \text{else } \{\langle \kappa, \iota \rangle \,|\, x = \langle \kappa, \ell \rangle \ \wedge \ \ell \in Allocated(\iota)\} \\
\alpha_{\underline{X}} : \underline{Exn} \to \underline{\hat{X}} & = \alpha_X \\
\alpha_V : Value \to \hat{V} & = \lambda v.\text{if } x = \bot \text{ then } \bot \text{ else } \langle \bot, \cdots, \alpha_\star(v), \bot, \cdots \rangle \text{ for } v \in \star \\
\alpha_S : Store \to \hat{S} & = \lambda s.\lambda \hat{\ell}. \bigsqcup_{\alpha_L(\ell) \sqsubseteq \hat{\ell}} \alpha_V(s(\ell))
\end{array}$$

We prove the correctness by means of the fixed-point induction method [Sto77]. The assertion $Q(f,g)$[14] that we will prove is

$$Q(f,g) = \alpha_Y \circ f \sqsubseteq g \circ \alpha_X .$$

We must show $Q(\bot,\bot)$ holds, which is trivial, and show that $Q(\mathcal{E},\hat{\mathcal{E}})$ implies $Q(\mathcal{F}(\mathcal{E}),\hat{\mathcal{F}}(\hat{\mathcal{E}}))$. Then, by the fixed-point induction, $Q(fix\mathcal{F}, fix\hat{\mathcal{F}})$ holds.

We will sketch the proof of the second part, $Q(\mathcal{E},\hat{\mathcal{E}}) \Rightarrow Q(\mathcal{F}(\mathcal{E}),\hat{\mathcal{F}}(\hat{\mathcal{E}}))$, for the `handle` expression. Proofs for other expressions can be done similarly.

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

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

By the induction hypothesis $Q(\mathcal{E},\hat{\mathcal{E}})$, $\alpha(\langle v, s_1 \rangle) \sqsubseteq \langle \hat{v_1}, \hat{s_1} \rangle$. Clearly, when v is not a raised exception, the concrete result $\alpha(\langle v, s_1 \rangle)$ is subsumed by the abstract part $\langle |\hat{v_1}|, \hat{s_1} \rangle$. On the other hand, when $v = \underline{v'} \in \underline{Exn}$,

$$\alpha(\langle \sigma[\ell/\mathrm{x}], s_1[v : Exn/\ell] \rangle) \sqsubseteq \hat{s_1}[(\hat{v_1}.\underline{\hat{X}} : \hat{X})//\mathrm{x}].$$

Hence, again by $Q(\mathcal{E},\hat{\mathcal{E}})$,

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

[14] The assertion must be inclusive, which is the case here, in order to apply the fixed-point induction.