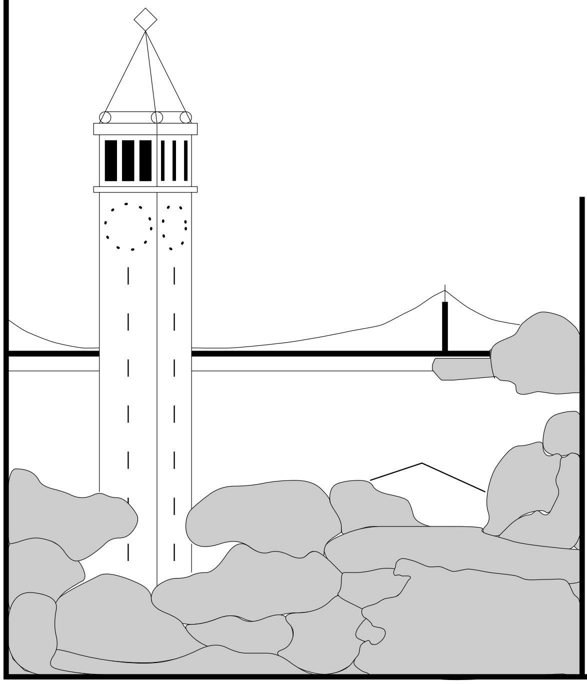
Tracking down Exceptions in Standard ML Programs

Manuel Fähndrich Jeffrey S. Foster Alexander Aiken Jason Cu



Report No. UCB/CSD-98-996

February 1998

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Tracking down Exceptions in Standard ML Programs

Manuel Fähndrich* Jeffrey S. Foster† Alexander Aiken† Jason Cu

EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
{manuel, jfoster, aiken}@cs.berkeley.edu
(510)-642-6509

Abstract

We describe our experiences with an exception analysis tool for Standard ML. Information about exceptions gathered by the analysis is visualized using PAM, a program visualization tool for EMACS. We study the results of the analysis of three well-known programs, classifying exceptions as assertion failures, error exceptions, control-flow exceptions, and pervasive exceptions. Even though the analysis is often conservative and reports many spurious exceptions, we have found it useful for checking the consistency of error and control-flow exceptions. Furthermore, using our tools, we have uncovered two minor exception-related bugs in the three programs we scrutinized.

1 Introduction

The ML type system guarantees that ML programs never terminate abnormally. Normal termination, however, may be caused by an exception, which is often undesirable. ML compilers infer a type for each expression in a program, but compute no information about exceptions produced or raised by an expression. In [FA97] we proposed an exception inference system to fill this gap. It subsumes the type inference by inferring ML types annotated with exceptions.

Based on this exception inference, we have built an exception analysis tool (EAT) that allows the programmer to display uncaught exceptions at certain program points while browsing code. The implementation of EAT is based on two components, called BANE and PAM that we have developed over the last few years. BANE (Berkeley ANalysis Engine) is a general framework for implementing constraint-based program analyses. PAM (Program Analysis Mode) is a point-and-click hypertext system based on EMACS for viewing the results of program analyses.

This paper describes our experience applying the exception analysis tool EAT to three Standard ML [MTH90] programs distributed with the SML/NJ [App92] compiler. With the help of PAM, we classify the exceptions used in ML-LEX, ML-YACC, and ML-BURG into four categories and study the precision of the exception analysis with respect to the uncaught exceptions reported for the main function of each program. Any sound exception analysis is necessarily conservative in that it may report exceptions that cannot be raised in any actual run of the program. It is thus necessary to *resolve* every exception that is reported as potentially uncaught to an *actual uncaught exception* by exhibiting some input to the program that causes the uncaught exception, or to a *spurious exception* by showing that the `raise` expression in question constitutes dead code. PAM helps in the resolution of uncaught exceptions by allowing the programmer to view exception information at each lambda expression, function application, and exception handler. Using this technique, we found two minor undocumented bugs, one in ML-LEX, and one in ML-BURG.

We divide exceptions raised by ML programs into four general categories:

1. *Assertion failures*: Exceptions raised when a program enters an unexpected state, *e.g.*, when an internal data-structure is found to be inconsistent;

2. *Error exceptions*: Exceptions raised after reporting an error, *e.g.*, a parser may report a syntax error and then raise an exception;

*Supported in part by NSF Young Investigator Award CCR-9457812, NSF Grant CCR-9416973, an NDSEG fellowship, and a gift from Rockwell Corporation.

3. *Control-flow exceptions*: Exceptions raised without any error reporting, *e.g.*, `Fifo.Empty` for a `get` operation on an empty queue; and

4. *Pervasive exceptions*: Exceptions raised by primitives or predefined functions, *e.g.*, `Subscript` raised by `Array.sub` when the array index is out of bounds.

These categories are approximate, and not every exception corresponds exactly to one of them. Assertion failures correspond roughly to errors that would be signaled in C programs using the `assert` macro. A typical example appears in ML-LEX, whose scanner can be in one of three integer modes, 0, 1, or 2. If the scanner finds itself in any other mode (*i.e.*, in the default branch of a particular `case` expression), an assertion failure exception is raised. The user of a program does not expect to see any assertion failures. Instead, assertion failures alert the programmer during development about missing cases or violations of invariants.

Error exceptions are used to terminate execution and correspond to non-zero integers returned to the operating system by a C program. These exceptions should be documented in the interface as possible results of the program.

Control-flow exceptions should always be caught at some level. Otherwise, uncaught control-flow exceptions terminate the program without an error message. Note that some programs turn control-flow exceptions into error exceptions using a top-level catch-all handler that prints out the name of the uncaught exception and then reraises the exception.

Pervasive exceptions are control-flow exceptions raised by built-in routines. Common examples are `Overflow` and range errors. Since our exception analysis does not model integer constants or arithmetic, such exceptions are assumed to arise from any call to particular built-in routines.

The rest of this paper is organized as follows. Section 2 informally describes the exception analysis we use. Section 3 outlines the PAM visualization mode for EMACS. We discuss our experience using EAT in Section 4. Related work appears in Section 5, and Section 6 concludes.

2 Exception Inference

This section informally describes the exception inference used in our study and discusses some issues in working with real programs. A more formal treatment for a subset of SML can be found in [FA97].

The analysis is structured as a *type and effect* system [LG88]. For every expression e in the program, the analysis infers a type τ and an effect σ. The type τ corresponds closely to the SML type of e, except that exception types, datatypes, and function types carry extra type parameters for exception sets. Effects σ describe the set of exceptions that are potentially raised during evaluation of e. For example, the SML basis function `List.hd` returning the first element of a list

```
val hd = fn l =>
           case l of
              nil => raise Empty
            | (x::y) => x
```

has type `'a list` $\xrightarrow{\texttt{Empty@}p}$ `'a`, *i.e.*, a function whose domain is lists of element type `'a`, whose range is `'a`, and whose possible exceptions are `Empty` raised at position p. Effects are modeled as a pair containing an exception name (or set of names) and a position in the source code. The lambda expression for `hd` has no effect itself, since evaluating the expression cannot raise any exceptions.

The next example is an excerpt from the hash table functor of the SML/NJ library.

```
datatype 'a hash_table =
      HT of {not_found : exn,
             table     : ...,
             n_items   : ...}

fun mkTable (sizeHint,notFound) =
      (HT {not_found = notFound,
           table = ...,
           n_items = ref 0})
```

The function `mkTable` is used to create an empty hash table. It takes an exception argument `notFound`, which is stored as part of the hash table data structure. This exception is raised during `lookup` and `remove` operations on keys that are not part of the table. In order to correctly report the exception raised by `lookup` or `remove`, we need to attach the exception used when creating the hash table to the type of the hash table. In general, we augment types with exception information by parameterizing types with an extra exception argument. In the example, the `hash_table` type constructor takes a second argument denoting the set of exceptions potentially stored in the hash table structure. Thus, the type of `mkTable` is

$$\texttt{int} * \texttt{exn}(\epsilon) \rightarrow (\texttt{'a}, \epsilon)\ \texttt{hash\_table}$$

which states that `mkTable` can be applied to a pair consisting of an integer and an exception (whose exception names are bound to ϵ), and it returns a hash table data structure containing elements of type `'a` and exceptions ϵ. The *exception variable* ϵ can be instantiated to any set of exception names. The dependency between the hash_table type and the exceptions raised by the lookup function appears clearly in the type of lookup:

$$(\texttt{'a},\epsilon)\ \texttt{hash\_table} \rightarrow \texttt{key} \xrightarrow{\epsilon @ p} \texttt{'a}$$

Any exceptions carried by the hash table may be raised at position p when calling lookup (p is the position of the raise expression within lookup).

In general, we infer for each datatype whether or not it carries any exception names and effects. Our inference conflates all exceptions carried by a datatype. It is possible to relax this restriction, and in fact we have analyzed programs other than the ones reported on here, where this restriction causes severe loss of precision.

Standard ML has parameterized modules called functors. Our exception inference cannot directly analyze functors due to unresolved exception aliasing. We use a tool to expand all functor applications prior to performing the analysis.

Readers familiar with SML may wonder about the generativity of exception declarations, which in general makes it impossible to statically name all exceptions. Generativity only comes into play if exception declarations appear within functions. Our analysis reports such declarations and treats these exceptions very conservatively by never filtering them in exception handlers.

2.1 Basis Library

In order to produce meaningful results, any exception inference must know the exception signatures of primitive (pervasive) functions. For example, in our system the array indexing function Array.sub has type $\texttt{'a array * int} \xrightarrow{\texttt{Subscript@array-sig.sml:20.52-20.56}} \texttt{'a}$.

We have manually generated signatures annotated with exceptions for large parts of the SML basis library. To make this tedious job significantly easier, we make use of the signatures already provided in the SML/NJ compiler source. Functions of type $\texttt{'d} \rightarrow \texttt{'r}$ are manually given new signatures ('d, 'r, 'e) efun, where efun is a new primitive type constructor for function types carrying exceptions. The last element of the type, 'e, is an exception constructor C that stands for the exception type $\texttt{exn}(C)$. For example, the specification for Array.sub is transformed to

```
signature ARRAY =
  sig
    ...
    val sub : ('a array * int, 'a,
               Subscript) efun
    ...
  end
```

Notice that not only is the change relatively minor, but the signature is still parseable by the front end.

A number of pervasive exceptions are not modeled by our inference, in particular Overflow due to its omnipresence. It would however be trivial to add it to the pervasive signatures.

2.2 Implementation

We have implemented our exception inference on top of the Berkeley ANalysis Engine.[1] BANE is a framework for writing program analyses based on a mixture of constraints [FA97], including set-constraints [AW93]. BANE separates the specification (constraint generation) of an analysis from its implementation (constraint solving). Our exception inference traverses the source code of the input program, generating type constraints and constraints that capture the local flow of exceptions. BANE handles the representation and resolution of constraints.

The solutions to the constraints model the global flow of exceptions. We extract the set of uncaught exceptions for functions, applications, and handle expressions. This information is then written into a description file suitable for visualization with PAM.

3 Visualization

PAM is a program analysis mode for EMACS, providing a textual point-and-click interface for displaying the results of a program analysis. PAM takes as input a *program analysis description file*, which contains a sequence of overlays onto the source text of the program. Each overlay specifies a character range, a highlight color for the region, and a pointer to the information shown when this overlay is selected. When a source file is opened in PAM mode, the text is colored according to the overlays. A key press or mouse click on an overlay displays the associated text in a separate EMACS window. The textual information associated with an overlay can also contain hypertext cross-links to other information or to positions in the source text file.

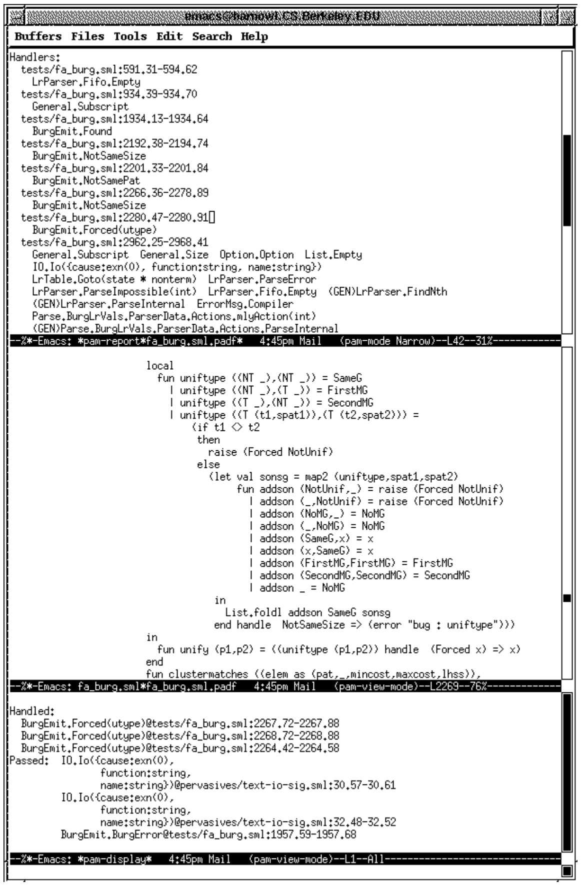
Besides the overlays, description files also contain a report section, which is the text shown first when viewing a PAM description file. For our exception inference, the report contains

- the list of declared exceptions, cross-linked to their positions in the source text,
- a list of handlers, each cross-linked to the source text and showing the set of exceptions handled,

[1] http://bane.cs.berkeley.edu

Program	Assertion Failures	Error Exceptions	Control-flow Exceptions	Pervasive Exceptions	Unused
ML-LEX (3.5s)	`LexError Match`	`Error`	`eof LOOKUP notfound SyntaxError`	`Chr Io Size Subscript`	`ParseError`
ML-YACC (28.1s)	`Bind Find FindNth Goto Lalr LexerError LexHackingError Match MkTable mlyAction ParseImpossible ParseInternal Produces Shift`	`ParseError Semantic`	`Done Fifo.Empty LexError select_arb`	`List.Empty Io Size Subscript`	
ML-BURG (14.7s)	`Compiler FindNth Goto LexerError LexHackingError mlyAction ParseImpossible ParseInternal`	`BurgError ParseError`	`Fifo.Empty Forced Found LexError NotSamePat NotSameSize NotThere`	`List.Empty Io Option Size Subscript`	

Table 1: Declared or used exceptions and their classification



```
Handlers:
  tests/fa_burg.sml:591.31-594.62
    LrParser.Fifo.Empty
  tests/fa_burg.sml:934.39-934.70
    General.Subscript
  tests/fa_burg.sml:1934.13-1934.64
    BurgEmit.Found
  tests/fa_burg.sml:2192.38-2194.74
    BurgEmit.NotSameSize
  tests/fa_burg.sml:2201.33-2201.84
    BurgEmit.NotSamePat
  tests/fa_burg.sml:2266.36-2278.89
    BurgEmit.NotSameSize
  tests/fa_burg.sml:2280.47-2280.91
    BurgEmit.Forced(utype)
  tests/fa_burg.sml:2962.25-2968.41
    General.Subscript  General.Size  Option.Option  List.Empty
    IO.Io({cause:exn(0), function:string, name:string})
    LrTable.Goto(state * nonterm)  LrParser.ParseError
    LrParser.ParseImpossible(int)  LrParser.Fifo.Empty  (GEN)LrParser.FindNth
    (GEN)LrParser.ParseInternal  ErrorMsg.Compiler
    Parse.BurgLrVals.ParserData.Actions.mlyAction(int)
    (GEN)Parse.BurgLrVals.ParserData.Actions.ParseInternal
--%*-Emacs: *pam-report*fa_burg.sml.padf*   4:45pm Mail   (pam-mode Narrow)--L42--31%-----------

                        local
                          fun uniftype ((NT _),(NT _)) = SameG
                            | uniftype ((NT _),(T _)) = FirstMG
                            | uniftype ((T _),(NT _)) = SecondMG
                            | uniftype ((T (t1,spat1)),(T (t2,spat2))) =
                                (if t1 <> t2
                                 then
                                   raise (Forced NotUnif)
                                 else
                                   (let val sonsg = map2 (uniftype,spat1,spat2)
                                        fun addson (NotUnif,_) = raise (Forced NotUnif)
                                          | addson (_,NotUnif) = raise (Forced NotUnif)
                                          | addson (NoMG,_) = NoMG
                                          | addson (_,NoMG) = NoMG
                                          | addson (SameG,x) = x
                                          | addson (x,SameG) = x
                                          | addson (FirstMG,FirstMG) = FirstMG
                                          | addson (SecondMG,SecondMG) = SecondMG
                                          | addson _ = NoMG
                                    in
                                      List.foldl addson SameG sonsg
                                    end handle  NotSameSize => (error "bug : uniftype")))
                        in
                          fun unify (p1,p2) = ((uniftype (p1,p2)) handle  (Forced x) => x)
                        end
                        fun clustermatches ((elem as (pat,_,mincost,maxcost,lhss)),
--%*-Emacs: fa_burg.sml*fa_burg.sml.padf   4:45pm Mail   (pam-view-mode)--L2269--76%-----------

Handled:
  BurgEmit.Forced(utype)@tests/fa_burg.sml:2267.72-2267.88
  BurgEmit.Forced(utype)@tests/fa_burg.sml:2268.72-2268.88
  BurgEmit.Forced(utype)@tests/fa_burg.sml:2264.42-2264.58
Passed:  IO.Io({cause:exn(0),
                function:string,
                name:string})@pervasives/text-io-sig.sml:30.57-30.61
         IO.Io({cause:exn(0),
                function:string,
                name:string})@pervasives/text-io-sig.sml:32.48-32.52
         BurgEmit.BurgError@tests/fa_burg.sml:1957.59-1957.68

--%*-Emacs: *pam-display*   4:45pm Mail   (pam-view-mode)--L1--All----------------------------
```

Figure 1: Screen shot of PAM viewing ML-BURG

- a list of function declarations, cross-linked to the source text, and

- a list of exceptions that the inference reports as potentially uncaught during compile/load time.

Overlays are generated for all lambda expressions, function declarations, applications, and handle expressions. Exceptions are displayed in the form `Name@`p, where p is the position in the source file where the exception is potentially raised. The position part is cross-linked to the source file for easy navigation. PAM allows backtracking, similarly to a web browser.

By using this system it is easy to start at the body of the main function of a program and follow the uncaught exceptions backwards to see where they were raised. In this way one can decide whether the inferred exceptions are errors or whether they are results of the conservatism of the analysis.

Figure 1 shows a screen shot of PAM viewing the analysis result of ML-BURG. The top window shows the report section, focused on the list of handle expressions. The middle window shows the source text as it would be displayed when clicking on the second to last handler in the report section (cursor position). The handler in question appears near the bottom of the middle window, handling `Forced`. The bottom window shows the result of clicking on the `handle` keyword of the handler in the middle window. This display shows that the handled `Forced` exception is raised at three positions (also visible in the middle window). Two `Io` exceptions and the `BurgError` exception (raised by the call to `error` visible in the middle window) fall through the handler.

Program	Assertion Failures	Error Exceptions	Control Flow	Pervasive
ML-LEX	LexError− Match?	Error+	eof+	Chr? Io+ Size− Subscript?
ML-YACC	Bind? Find? FindNth? Goto? Lalr? LexerError− Match? MkTable− mlyAction? ParseImpossible? ParseInternal? Produces− Shift?	ParseError+ Semantic+	Fifo.Empty− LexError−	List.Empty− Io+ Size− Subscript?
ML-BURG	Compiler− FindNth? Goto? LexerError− mlyAction? ParseImpossible? ParseInternal?	BurgError+ ParseError+	Fifo.Empty− LexError+	List.Empty− Io+ Option− Size− Subscript?

Table 2: Exceptions reported for the main function of each program

4 Results

We have applied EAT to three programs distributed with version 109.31 of the SML/NJ compiler: the lexer generator ML-LEX, the parser generator ML-YACC, and the tree-rewrite generator ML-BURG. Table 1 categorizes the set of exceptions declared in each program as well as the pervasive exceptions used indirectly through predefined functions.[2] The number below each program name is the time in seconds needed to perform the exception inference on an UltraSparc. ML-LEX contains a declaration of exception ParseError that is subsequently never used. Note the large number of assertion failure exceptions for ML-YACC and ML-BURG. The majority of them are part of the automatically generated lexer and parser contained in ML-YACC and ML-BURG. There are relatively few error exceptions in all programs.

Table 2 lists the uncaught exceptions reported by our analysis for the main function of each program. For each category, the reported uncaught exception name is annotated with one of the symbols $\{+, -, ?\}$. A (+) symbol means that the exception can actually be raised by providing suitable parameters to the program. We have verified these using suitable example inputs. A (−) symbol means that the exception cannot be raised in any execution of the program, *i.e.* the analysis reports a spurious exception. We identified these exceptions by inspecting the code. Finally, a (?) means that the exception is likely not raised, but proving so requires more than a simple code inspection.

In the category of assertion failures, EAT reports all exceptions as uncaught. This is not surprising, since these exceptions are never handled in the code. Showing that such exceptions do not occur requires proving that the respective `raise` expressions are dead code. Our analysis cannot tell which branches of a `case` or `if` expression are taken and is thus very conservative in this respect. Constant propagation or simple set-based analysis could be used to remove a few of the spurious exceptions (marked by −). Similarly, our analysis is conservative with respect to pervasive exceptions. For example, showing that the `Subscript` exception is never raised requires *range analysis* [SI77].

On the other hand, EAT is useful for detecting problems with control flow exceptions, and to infer the set of error exceptions. It helped us detect a minor bug in the ML-LEX program: the `eof` (end-of-file) exception can escape if the lexer generator is supplied with a lex specification that contains no occurrence of the % sign. One could argue that the `eof` exception is therefore an error exception, but looking at the code, this does not seem to be the programmer's intention, since no error message is printed in this case.

For ML-YACC, EAT reports the spurious control-flow exceptions `Fifo.Empty` and `LexError`, neither of which can actually escape. Code inspection shows that the call to `Fifo.get` that potentially raises `Fifo.Empty` is called with a non-empty queue. The absence of `LexError` is more subtle, since it rests on the fact that the lexer used in ML-YACC handles all input characters.

For ML-BURG, EAT reports the same control-flow exceptions as for ML-YACC, but in this case only the

[2] We have manually moved four exception declarations from within functions to outer scopes to improve the precision of the analysis.

Fifo.Empty exception is spurious. The lex specification for BURG files does not handle all input characters and raises LexError on an invalid character (*e.g.* on @).[3] Since the LexError exception is raised without any error message, we claim that this uncaught exception constitutes a programming bug.

EAT also reports exceptions that are potentially raised at compile/load time. For ML-LEX, our analysis proves the absence of such exceptions. For ML-YACC and ML-BURG, a few spurious exceptions are reported, *e.g.*, LexHackingError.

Although our exception analysis is very conservative and reports many spurious exceptions, it has proven useful in uncovering two minor bugs in long-standing programs. Our visualization mode PAM has been key in understanding the flow of exceptions discovered by our analysis. Without a good visualization tool, results of program analyses are very difficult to interpret and validate.

5 Related Work

There are several other exception inference systems for ML. Yi [Yi94] uses an abstract interpretation [CC79] framework to perform a much more precise analysis than ours. Unfortunately, it scales poorly, requiring many hours of analysis time on ML-YACC, and the analysis results are very difficult to inspect. The systems described in [GS94, YR97] are simpler analyses than our own, and are in general less precise. To study the precision trade-off, we have also implemented a variation of [FA97] similar to [GS94]. On the three programs studied in this paper, the two versions produced identical results. However, for programs using more higher order features, the loss of precision in the second approach can be significant.

One problem with EAT is that it isn't useful to prove the absence of assertion failures. To do so requires proving that certain raise expressions constitute dead code. Several techniques can be used to improve the analysis in this area. [Yi94] models variants of datatypes and integers such that impossible branches can be pruned. Similarly, set-based analysis [Hei94] can also provide information to prune branches. Refinement types [FP91] are another approach to proving that certain branches of case statements are not needed. If datatypes are viewed as defining regular tree languages, then refinement types specify sub-languages of datatypes. Refinement types have potential to express complex data structure invariants.

Pervasive exceptions like Subscript are also modeled very conservatively by our analysis. In general, proving the absence of Subscript exceptions is equivalent to proving that no runtime check on array indexing is required. A long line of work on range-checking for array subscripts starts with [SI77].

Finally, in Java [GJS96], methods must declare the set of exceptions that might be thrown during a call. This allows Java compilers to perform a similar (although non-polymorphic) exception verification. That approach however has the same shortcomings as the one described here with respect to assertion failures and certain pervasive exceptions. As a result, such exceptions are called *unchecked* in Java and need not be listed in method signatures.

PAM is in part inspired by MrSpidey [FFK$^+$96], a static debugger for Scheme. MrSpidey performs set-based analysis on Scheme programs and presents the information to the programmer as graphical overlays over the source code.

6 Conclusions and Future Work

We have evaluated the precision and utility of an exception inference for Standard ML. Although the analysis cannot prove the absence of exceptions raised as a result of a failed assertion, it is useful to check the consistency of control-flow exceptions and to infer the set of error exceptions. Applying the analysis to three programs distributed with the SML/NJ compiler, we have discovered two minor exception related bugs.

We are also currently applying EAT to a very large program, a points-to analysis for C written in BANE [FFA97]. Two problems arose in this context. First, although the inference does scale, the description file of the analysis results uses more than 100MB of disk space. As a result, the visualization becomes impractical. Factoring repeated information in the description file might help curb this blowup. Second, the program we are looking at contains many higher order functions and stores functions in data structures. Due to the way we model datatypes carrying exceptions, the exceptions of many distinct functions are conflated, yielding very imprecise results. We are experimenting with ways to increase the precision of our analysis to make the results more useful.

References

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

[3] Due to a yet unresolved problem in the compiler, ML-BURG hangs instead of raising the exception.

[AW93] A. Aiken and E. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, January 1979.

[FA97] M. Fähndrich and A. Aiken. Program Analysis Using Mixed Term and Set Constraints. In *Proceedings of the 4th International Static Analysis Symposium*, 1997.

[FFA97] J. Foster, M. Fähndrich, and A. Aiken. Flow-Insensitive Points-to Analysis with Term and Set Constraints. Technical Report UCB//CSD-97-964, University of California, Berkeley, July 1997.

[FFK$^+$96] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[FP91] T. Freeman and F. Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 268–277. ACM Press, June 1991.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley Longman, Inc., 1996.

[GS94] J. C. Guzmán and A. Suárez. An Extended Type System for Exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 127–135, June 1994.

[Hei94] N. Heintze. Set Based Analysis of ML Programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–17, June 1994.

[LG88] J. Lucassen and D. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[SI77] N. Suzuki and K. Ishihata. Implementation of an Array Bound Checker. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977.

[Yi94] K. Yi. Compile-Time Detection of Uncaught Exceptions for Standard ML Programs. In *Proceedings of the 1st International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science.* Springer, 1994.

[YR97] K. Yi and S. Ryu. Towards a Cost-Effective Estimation of Uncaught Exceptions in SML Programs. In *Proceedings of the 4th International Static Analysis Symposium*, 1997.