Dynamic State Restoration Using Versioning Exceptions

V. Krishna Nandivada (nvk@cs.ucla.edu)
Department of Computer Science
University of California, Los Angeles

Suresh Jagannathan (suresh@cs.purdue.edu)
Department of Computer Science
Purdue University, West Lafayette

Abstract.

We explore the semantics and analysis of a new kind of control structure called a *versioning* exception that ensures the state of the program, at the point when an exception handler is invoked, reflects the program state at the point when the handler is installed. Versioning exceptions provide a transaction-like versioning semantics to the code protected by a handler: modifications performed within the dynamic context of the corresponding handler are versioned, and committed to the store only if the computation completes normally. Similar to the role of backtracking in logic programming, this facility allows unwanted effects of computations to be discarded when exceptional or undesirable conditions are detected.

We define a novel points-to analysis to efficiently track changes to the store within handler-protected scopes. The role of the analysis is to facilitate optimizations that minimize the number of locations which must be restored when a versioning exception is raised. The analysis is defined by a reachability approximation over locations that indicates which objects have been potentially modified within a handler scope. The analysis is defined for programs which support first-class procedures, locations, and exceptions.

1. Introduction

cxceptions found in languages like SML, Java, or Haskell are a structured way of expressing non-local control-flow when unexpected conditions arise. Exception mechanisms allow programs to exit gracefully from error conditions without requiring the entire computation to halt. The implementation of exceptions thus changes the control stack to reflect the continuation in effect at the point where the handler is defined. However, the store is usually left untouched. This means that although control reverts to a meaningful program point, modifications to the store performed between the installation of the handler and the point where the exception is raised, are not undone. For many real-world applications, the inability to restore modified locations automatically requires programmers to carefully inject such operations manually, significantly complicating program structure.

Logic programming languages such as Prolog provide a backtracking mechanism that reverts program state to a consistent point; when the unification of a clause and goal fails, backtracking occurs, discarding all bindings established in the process of the failed unification. However, backtracking occurs implicitly when unification fails, and users have only a limited degree of control (e.g., using cut predicates) in influencing backtracking decisions. In particular, while backtracking in the logic programming context discards bindings established during goal resolution, we are interested in selectively discarding store values for a given location.

Languages like Argus [28] provide linguistic support for committing critical data only when the transaction succeeds, by means of a *guardian* structure. Although programmers have significantly more control than in logic programming about when state changes should be committed, guardians are heavyweight abstractions whose commit action is defined in terms of a two-phase commit protocol.

We are interested in exploring the implications of enriching first-class lightweight control-structures such as exceptions or continuations found in SML, Scheme, or Java with a controlled form of versioning that can be used to express robust computation of the kind available in transaction-oriented or logic programming languages. While exceptions are an expressive linguistic device to capture non-local control-flow, transactions and backtracking are powerful mechanisms to revert computation to well-defined states in response to unexpected or undesirable conditions. By combining essential aspects of both features, programs can respond to unexpected error or failure conditions by reverting control to a handler that can, among other actions, revert program state to an earlier well-defined version.

This paper studies the semantics and analysis of a control structure called a *versioning* exception that provides a transaction-like versioning semantics to the code protected by an exception handler. A versioned exception ensures that the content of the store at the point when the exception is raised reflects the program state at the point when the corresponding handler is defined. The data generated in code protected by such exceptions are implicitly versioned. Each version is associated with a particular generative exception value. When an exception is raised, the version corresponding to the associated exception value is restored. Handlers may choose to re-execute the protected code, or revert control-flow to another program point. In either case, all effects to the program store and environment made within the protected region are undone. Thus, the semantics of a versioning exception is similar to transactional models [26, 20, 37] in which raising an exception is tantamount to an abort. However, unlike typical transaction systems in which aborts occur due to the conflicts among multiple concurrent threads accessing shared data or because of unexpected change in the external environment (e.g., processor failure), state restoration using versioning exceptions occur when a software exception signaling a non-local transfer of control is raised.

2. Motivation

Fig. 1 gives an example of how versioned exceptions may be used to simplify complex program structure. The procedure `update` traverses a binary tree, updating the nodes of the tree, provided every visited node satisfies a supplied predicate, `compare`. If the predicate fails, all updates must be discarded, and the search retried, using an alternative predicate. This function abstracts the functionality of many common graph coloring and search algorithms.

The constructor `vExn` creates a new instance of a version exception type, binding this instance to `failure`. Its argument is a procedure which is applied when the exception is raised, and serves as the handler for the exception. The result of evaluating the handler is supplied as the argument to the continuation of the corresponding `try` expression in which the expression was raised. Versioned exceptions are used to capture the state of the tree immediately prior to the commencement of the search and update procedure. The `try` expression protects the dynamic context of the expression it encloses. When an update fails, an exception is raised via `restore`, and control transfers to the exception handler defined by the matching `try` expression. All modifications done by the procedure from the point at which control entered the `try` expression are discarded.

In this example, raising the `failure` exception while visiting and updating nodes in function `visit` in the tree results in the state of the tree being restored to its state extant at the point where the exception scope was entered; this corresponds to the state at the point the call to `update` was made within the `try` exception. Thus, raising a failure exception results in control returning to the handler's continuation. Since the value supplied to the continuation is `Fail`, a new call to `attempt` is performed, with a new comparison function.

This program uses versioned exceptions to reduce the overhead of preserving the invariant that no node can be updated unless all nodes in the tree satisfy the supplied predicate. Without them, the program would either need to traverse the tree twice, checking the state of each node in a prepass before performing updates, or would need to maintain copies of original node values to restore in case the predicate ultimately

```
let fun handler(x) = x
    val failure = vExn(handler)
    fun update(compare,tree) =
        let fun visit(n) =
                  (case n of
                        Node (data,left,right) =>
                           if compare(data)
                              then (modify!(n,f(data));
                                    visit(left);
                                    visit(right))
                              else restore(failure,Fail)
                      | Empty => Success)
         in visit(tree)
         end
    fun attempt(compare) =
      (case try(failure,update(compare,getTree())) of
        Fail => attempt( new comparison function)
      | Success => ())
in attempt(compare)
end
```

Figure 1. Versioning exceptions may be used to revert selected elements of the store to a previous known state.

failed on some node during a traversal of some other portion of the tree. Continuations found in Scheme or exception facilities found in Java or Standard ML [30] would allow control to be transferred in interesting ways as in the example, but do not support implicit restoration of updates performed on heap-allocated objects. Transaction systems maintain rollback logs that reflect updates which are committed only if the transaction commits, but these mechanisms are not integrated with exception-style control-flow operators; recent work by Shinnar *et. al.* [35] is a notable counterexample that uses transaction-style undo logs to restore program state modified within the scope of an exception handler. Backtracking facilities found in logic programming languages can be used to revert the environment to an earlier state, but do not consider reverting the contents of the store. In this example, `data` is a reference to a store location that is repeatedly modified by the `update` procedure. A failed comparison does not result in discarding the binding of `data` to its reference, but only discarding the contents of that reference.

Of course, this simple example could presumably be rewritten in a purely functional style to avoid the use of the store altogether allowing the use of continuations or backtracking to restore bindings when the comparison operation fails. For many real-world programs however, such a translation may be impractical or infeasible.

We have discovered several examples in production-quality systems code where versioned exceptions would be useful. One such example, taken from Linux 2.4 kernel code is shown in Fig. 2(a). The program fragment is a snippet of code taken from the implementation of the `do_fork` function, presented in pseudo-code form. This function invokes a number of other procedures, all of whom modify various global data structures; if any of these operations fail, modifications performed thus far must be undone. Rollback is implemented via a complex set of jumps to labels that implement restore code, of which only a few are shown here. The full implementation is quite complicated and arguably error-prone. Note that having traditional exceptions or continuations would be of only marginal use here. They would eliminate the need for having unstructured jumps, but would still require complex clean-up code to be executed to restore proper values for modified globals.

In contrast, Fig. 2(b) shows the code for the `do_fork` function using versioned exceptions. The block of code to be protected is encapsulated within a `try` block. The exception variable s is supplied as an argument to the procedures called by this function; the handler returns its argument to the continuation of the `try` block and is invoked by a `restore` expression. When invoked, control will revert to the `try` expression's continuation, with all variables live at that point restored to their values upon entry to this block. This code is not only arguably more compact and modular, but also less liable to programming errors.

A naive implementation of versioned exceptions is easily achieved: we need simply to snapshot the store whenever a new handler is installed, copying back this saved version if the handler is invoked. Such an implementation, however, scales poorly as the size of the store increases and is costly in terms of the overhead incurred to save potentially multiple versions of modified objects. Moreover since it is probable that not all live locations may have changed in the code protected by a versioned exception, implementations that do not incorporate such observations in their realization of the `restore` operation are likely to be too inefficient in practice.

One might imagine an alternative strategy that only logs the original values of updated locations. In the presence of complex control and dataflow, and because locations are first-class, it will often not be possible to determine whether logging an update at a particular program point is required by simple syntactic examination of the program. On the other hand, delaying the decision about whether an update requires logging to runtime is equally problematic, since the references accessed

```
do_fork(...){
 if (copy_files(···))
    goto bad_fork_cleanup;
 if (copy_fs(···))
    goto bad_fork_cleanup_files;
 ...
 if any error
    goto fork_out;
 ...
 goto fork_out:
 bad_fork_cleanup_files:
   exit_files(p);
 bad_fork_cleanup:
   do some cleanup;
 fork_out:
}
```

(*a*)

```
fun do_fork(...) =
  let fun handler(x) = x
    val s = vExn(handler)
    val status =
      try(s,
          (copyFiles(s,...);
           copyFs(s,...);
           ...
           if error(...)
             then restore(s,Fail)))
  in (case status of
       Fail => ...
     | Success => ...)
  end
```

(*b*)

Figure 2. Versioned exceptions may have utility in simplifying the structure for systems programming code. (a) shows a snippet of function *do_fork* found in the Linux 2.4 kernel. (b) shows the same function rewritten to use versioned exceptions.

by a handler's continuation will not be known at the time an update in a handler-protected scope occurs.

In this paper we consider a program analysis that tracks the changes to locations in the store within a block of code protected by a handler. The analysis suggests restoration of only those locations that are modified within the dynamic context defined by the block at the point of handler invocation and which might have some use in the continuation of block. Our analysis provides a weak form of context sensitivity useful in defining a reachability approximation defined over store locations to produce an approximation to the set of values that must be saved and restored when exceptions are raised. We propose a syntax-directed translation derived from classical flow analysis, to capture the interactions among locations of the store within handler protected regions to permit versioned exceptions to be implemented efficiently.

3. Language

We define a simple higher-order language with first-class procedures, references and exceptions. The syntax for programs is given in A-normal form [12] and defined below. For simplicity, we omit recursive function definitions and tail-call application; their addition poses no interesting complications to the definition and analysis of versioning exceptions. While the language is simple, it serves as a reasonable intermediate representation for realistic call-by-value mostly-functional languages like Scheme or ML [24, 3], and shares central aspects of the representations used in these implementations, namely, first-class procedures, exceptions, and locations.

We also assume that variables are appropriately α-renamed. We use w, x, y, z to range over variables, Pr to range over primitive functions, and c to range over constants.

$$\begin{array}{l} s \in \ \mathit{Simp} ::= c \mid \mathit{Pr}(x_1,\ldots,x_n) \mid \mathsf{ref}\, x \mid\ !x \mid \lambda\, x.e \mid x_0(x_1) \mid \mathsf{vExn}(x) \\ e \in \mathit{Exp} ::= x \mid \mathsf{let}\ \ x{=}s\ \ \mathsf{in}\ \ e \mid x_1 := x_2 \mid \mathsf{if}\ \ x\,\mathsf{then}\ \ e_1\ \ \mathsf{else}\, e_2 \mid \mathsf{try}(x,e) \mid \mathsf{restore}(x,y) \end{array}$$

We define $Prog$ to be the set of closed expressions. Constants, primitive functions, reference constructors (defined by ref) dereferences (!), abstraction, application, and exception constructors (vExn) define simple expressions. The argument to an exception constructor is a variable bound to a procedure that defines the handler for the exception. One can imagine a more sophisticated exception handling facility that permits a given exception to be associated with different handlers in different contexts, but such extensions do not affect the structure of the analysis defined in the following sections, and are omitted here. Expressions introduce local bindings via let, define variables, perform assignments, define conditional test and branch, introduce try-blocks or perform a restore operation.

The set of types for our language is generated by the following grammar:

$$\tau ::= \mathbf{Int}\ \mid \mathbf{Bool}\ \mid \tau \rightarrow \tau \mid \mathbf{Ptr}\ \tau \mid \mathbf{Exn}\ \tau$$

where $\mathbf{Ptr}\ \ \tau$ denotes the type of locations holding values of type τ, and $\mathbf{Exn}\ \tau$ denotes the type of exception constructors whose handler is of type τ. Types are used to establish and filter points-to information as we describe in Section 5. The type system, which is standard, is shown in Fig 3.

The expression "$\mathsf{try}(x,e)$" evaluates x to an exception E and then evaluates e (called the *exception expression*) in the context of E. Evaluating "$\mathsf{restore}(y,z)$", when the value of y is E, *raises* exception E and control is transferred to the closest enclosing try expression for E in the program's dynamic context at the point where the raise occurs. E's handler is then evaluated with the value of z as its argument, and the result is supplied to the continuation of the try expression that catches the raised exception.

$$\frac{}{A \vdash c \,:\, \mathit{typeOf}(c)} \qquad \frac{\begin{array}{c} A \vdash x_i \,:\, \tau_i \\ \mathit{typeOf}(Pr) = \tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \end{array}}{A \vdash Pr(x_1, \ldots, x_n) \,:\, \tau}$$

$$\frac{A \vdash x \,:\, \tau}{A \vdash \mathsf{ref}\, x \,:\, \mathbf{Ptr}\,\tau} \qquad \frac{A \vdash x \,:\, \mathbf{Ptr}\,\tau}{A \vdash !x \,:\, \tau}$$

$$\frac{A[x \mapsto \tau_1] \vdash e \,:\, \tau_2}{A \vdash \lambda\, x.e \,:\, \tau_1 \rightarrow \tau_2} \qquad \frac{\begin{array}{c} A \vdash x_1 \,:\, \tau_1 \rightarrow \tau_2 \\ A \vdash x_2 \,:\, \tau_1 \end{array}}{A \vdash x_1(x_2) \,:\, \tau_2}$$

$$\frac{A \vdash x \,:\, \tau_1 \rightarrow \tau_2}{A \vdash \mathsf{vExn}(x) \,:\, \mathbf{Exn}\ (\tau_1 \rightarrow \tau_2)} \qquad \frac{\begin{array}{c} A \vdash s \,:\, \tau \\ A[x \mapsto \tau] \vdash e \,:\, \tau' \end{array}}{A \vdash \mathsf{let}\, x = s\, \mathsf{in}\, e \,:\, \tau'}$$

$$\frac{\begin{array}{c} A \vdash x_1 \,:\, \mathbf{Ptr}\,\tau_1 \\ A \vdash x_2 \,:\, \tau_1 \end{array}}{A \vdash x_1 := x_2 \,:\, \tau_1} \qquad \frac{\begin{array}{c} A \vdash x \,:\, \mathbf{Bool} \\ A \vdash e_1 \,:\, \tau \\ A \vdash e_2 \,:\, \tau \end{array}}{A \vdash \mathsf{if}\, x\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \,:\, \tau}$$

$$\frac{\begin{array}{c} A \vdash x \,:\, \mathbf{Exn}\ (\tau_1 \rightarrow \tau_2) \\ A \vdash e \,:\, \tau_2 \end{array}}{A \vdash \mathsf{try}(x, e) \,:\, \tau_2} \qquad \frac{\begin{array}{c} A \vdash v \,:\, \tau_1 \\ A \vdash x \,:\, \mathbf{Exn}\ (\tau_1 \rightarrow \tau_2) \end{array}}{A \vdash \mathsf{restore}(x, v) \,:\, \tau_2}$$

Figure 3. Type System.

Our syntax and definition of try and restore is superficially similar to the way exceptions are defined and used in Scheme or ML [24, 3], but simplifies some key aspects. In particular, our definition binds an exception to its handler at the point where the exception is defined, rather than at the point where the exception is handled. Thus, when a restore action is performed, the handler invoked is the one associated with the raised exception. This simplification, while easily relaxed, allows us to focus attention on the key contribution of versioning exceptions, namely, their support for capturing and restoring state within a protected scope. The actual handler chosen to deal with an exception is less significant in the context of this work than the mechanisms used to identify and restore changes in state. For programs in which an exception is associated with a unique handler, the expression,

```
let E = vExn (λ x.e₂)
in e₁
```

is equivalent to the Standard ML expression

```
let exception E of  type of x
```

```
in $e_1$
handle E x => $e_2$
end
```

Unlike ordinary exceptions, however, the state in which the try expression's continuation is evaluated after an exception is raised does not reflect modifications performed on the store in the course of evaluating e. Of course, if computation proceeds normally, and no exception is raised, the continuation operates in a context where all side-effects to the store performed by e are visible. Our semantics restores *all* locations modified within an exception scope, and used in the scope's continuation. It is conceivable that applications may sometimes wish to propagate information to the continuation even if an exception is thrown. Linguistic extensions that provide such fine-grained discrimination of locations fall outside the scope of this paper.

4. Operational Semantics

We define a small-step semantics for the language in the style of the CESK-machine [11]. The machine is specified in terms of *states* and a transition relation $\longrightarrow$ on states:

$$CE^2SK = \langle \mathit{State} \times \longrightarrow \rangle$$

Every evaluation step in the machine[1] yields a new state that reflects the contents of the store, environment, continuation, and exception stack. In the following, we write $X + Y$ and $X \times Y$ to mean the cartesian sum and product of sequences (ordered sets) X and Y, respectively. We write $X \rightarrow Y$ to denote the set of partial functions from X to Y. The notation $f[x \mapsto v]$ denotes the function that is identical to f on all elements except x for which it returns v. We write X^* to denote the finite sequences of elements of X, $X \oplus Y$ to denote the append of sequence X onto sequence Y. We also write $\mathit{State} \longrightarrow \mathit{State}'$ to reflect a transition from state State to State'. For functions f and g, $(f + g)(x) = g(x)$ if $x \in \mathit{Dom}(g)$ and $f(x)$ otherwise; similarly, $(f - g)(x) = f(x)$ if $x \in \mathit{Dom}(g)$ and $f(x) \neq g(x)$, and is undefined otherwise.

Given CE^2SK, we define a partial function *eval* :

$$\mathit{Prog} \rightarrow \langle \mathit{Env}, \mathit{Store}, \mathbf{halt}\, \langle v \rangle \rangle$$

that defines the semantics of all terminating programs $P \in \mathit{Prog}$ under the evaluation rules defined by CE^2SK.

[1] The E^2 corresponds to Environment and Exception-stack.

$$\begin{array}{rcll}
s & \in & \mathit{State} & = \mathit{ReturnState} + \mathit{ControlState} \\
 & & & \quad + \mathit{ExnState} + \mathit{HaltState} \\
\mathit{returnState} & \in & \mathit{ReturnState} & = \mathit{Cont} \times \mathit{Value} \times \mathit{Store} \times \mathit{ExnStack} \\
\mathit{controlState} & \in & \mathit{ControlState} & = \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Cont} \times \mathit{ExnStack} \\
\mathit{exnState} & \in & \mathit{ExnState} & = \mathit{Cont} \times \mathit{Store} \times \mathit{ExnStack} \\
k & \in & \mathit{Cont} & = \mathit{Frame}^* \\
\Sigma & \in & \mathit{ExnStack} & = (\mathit{ExnVal} \times \mathit{Cont} \times \mathit{Store})^* \\
\mathit{frame} & \in & \mathit{Frame} & = \mathbf{ret}\,\langle \mathit{Var}, \mathit{Exp}, \mathit{Env}\rangle + \mathbf{exn}\,\langle \mathbf{Int}\ , \mathit{Value}\rangle \\
 & & & \quad + \mathbf{sto}\,\langle \mathit{Store}\rangle \\
\rho & \in & \mathit{Env} & = \mathit{Var} \rightarrow \mathit{Value} \\
\sigma & \in & \mathit{Store} & = \mathit{Loc} \rightarrow \mathit{Value} \\
\mathsf{clo}\,\langle \lambda\, x.e, \rho\rangle & \in & \mathit{Closure} & = \mathit{Lambda\ Exp} \times \mathit{Env} \\
\mathsf{exnVal}\,\langle n, v\rangle & \in & \mathit{ExnVal} & = \mathbf{Int}\ \times \mathit{Value} \\
\mathbf{halt}\,\langle v\rangle & \in & \mathit{HaltState} & = \mathit{Value} \\
v & \in & \mathit{Value} & = \mathit{Constant} + \mathit{Closure} + \mathit{Loc} + \mathit{ExnVal}
\end{array}$$

Figure 4. Domain Equations.

State domains for the machine are defined in Fig. 4. The semantics for the non-exception core is standard and given in Fig. 5. Let expressions are used to augment the environment. Here, abstractions evaluate to closures that capture the lexical environment, application is expressed as a non-tail call that builds a new continuation frame, references are created and dereferences. Assignments and conditional statements manipulate the store and control respectively in the obvious way.

Rules for exception constructors, try, and restore expressions are presented in Fig. 6. Constructors for versioning exceptions create a new exception (generative) value that includes a unique identifier for the exception, and the closure that defines its handler. Evaluating a try expression pushes onto the current exception stack a structure that contains the expression's exception argument, the expression's continuation, and the current store. When an exception is raised, the stack is traversed to find an entry for the raised exception; the handler for the exception is evaluated in the context of the current store, a frame consisting of saved store and continuation, and an exception stack in which all frames between the point when the handler was defined and the exception raised have been discarded. If the execution of the handler finishes successfully, then the continuation is executed with the restored store. The critical observation here is that the store is preserved when the expression e protected by a try block (e.g., try(E,e)) begins evalu-

$$
\begin{array}{l}
\langle x, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle k, \rho(x), \sigma, \Sigma\rangle \\
\langle \mathsf{let}\, x = c\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho[x \mapsto c], k, \sigma, \Sigma\rangle \\
\langle \mathsf{let}\, x = Pr(x_1, \ldots, x_n)\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma\rangle \longrightarrow \\
\qquad\qquad\qquad \langle e, \rho[x \mapsto \mathtt{Pr}(\rho(x_1), \rho(x_2), \ldots, \rho(x_n))], k, \sigma, \Sigma\rangle \\
\langle \mathsf{let}\, x = \mathsf{ref}\, y\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho[x \mapsto l], k, \sigma[l \mapsto \rho(y)], \Sigma\rangle \\
\qquad\qquad\qquad \text{for fresh } l \\
\langle \mathsf{let}\, x = !y\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho[x \mapsto \sigma(\rho(y))], k, \sigma, \Sigma\rangle \\
\langle \mathsf{let}\, x = \lambda\, y.e'\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho[x \mapsto \mathsf{clo}\,\langle \lambda\, y.e', \rho\rangle], k, \sigma, \Sigma\rangle \\
\langle \mathsf{let}\, x = y(z)\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma\rangle \longrightarrow \\
\qquad\qquad\qquad \langle e', \rho'[w \mapsto \rho(z)], \{\mathbf{ret}\,\langle x, e, \rho\rangle\} \oplus k, \sigma, \Sigma\rangle \\
\qquad\qquad\qquad \text{provided } \rho(y) = \mathsf{clo}\,\langle \lambda\, w.e', \rho'\rangle \\
\langle x_1 := x_2, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle k, \rho(x_2), \sigma[\rho(x_1) \mapsto \rho(x_2)], \Sigma\rangle \\
\langle \mathsf{if}\, x\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e_1, \rho, k, \sigma, \Sigma\rangle \\
\qquad\qquad\qquad \text{provided } \rho(x) = \text{true} \\
\langle \mathsf{if}\, x\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e_2, \rho, k, \sigma, \Sigma\rangle \\
\qquad\qquad\qquad \text{provided } \rho(x) = \text{false} \\
\langle \{\mathbf{ret}\,\langle x, e, \rho\rangle\} \oplus k, v, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho[x \mapsto v], k, \sigma, \Sigma\rangle
\end{array}
$$

Figure 5. Small-step exact semantics for core language.

ation. Any exception raised by e will cause all modifications made to the store to be discarded when control reverts to the try expression's continuation.

Note that the semantics executes the exception handler in the context of the store present at the point the restore expression is evaluated. However, when an exception is thrown, the continuation of the try expression is evaluated in the context of the store present at the point

$$
\begin{array}{l}
\langle \mathsf{let}\, x = \mathsf{vExn}(y)\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho[x \mapsto \mathsf{exnVal}\,\langle n, \rho(y)\rangle], k, \sigma, \Sigma\rangle \\
\qquad\qquad\qquad \text{for fresh } n \\
\langle \mathsf{try}(x, e), \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho, k, \sigma, \{\langle \rho(x), k, \sigma\rangle\} \oplus \Sigma\rangle \\
\langle \mathsf{restore}(x, y), \rho, k, \sigma, \Sigma\rangle \longrightarrow \langle \{\mathbf{exn}\,\langle n, \rho(y)\rangle\} \oplus k, \sigma, \Sigma\rangle \\
\qquad \text{provided } \rho(x) = \mathsf{exnVal}\,\langle n, v\rangle \\
\langle \{\mathbf{exn}\,\langle n, v\rangle\} \oplus k, \sigma, \Sigma\rangle \longrightarrow \langle e, \rho[x \mapsto v], \{\mathbf{sto}\,\langle \sigma'\rangle\} \oplus k', \sigma, \Sigma''\rangle \\
\qquad \text{provided } \Sigma = \Sigma' \oplus \{\langle \mathsf{exnVal}\,\langle n, \mathsf{clo}\,\langle \lambda\, x.e, \rho\rangle\rangle, k', \sigma'\rangle\} \oplus \Sigma'' \\
\qquad \text{and } \langle \mathsf{exnVal}\,\langle n, v\rangle, k', \sigma'\rangle \notin \Sigma' \\
\langle \{\mathbf{sto}\,\langle \sigma'\rangle\} \oplus k, v, \sigma, \Sigma\rangle \longrightarrow \langle k, v, \sigma', \Sigma\rangle
\end{array}
$$

Figure 6. Small-step semantics for exceptions.

when the exception handler was installed. That is, changes made to the store in the exception handler are not visible in the continuation. By changing the definition of handler invocation slightly, we derive an alternative semantics in which modifications done in the handler are visible in the continuation. This can be useful if updates performed in the handler are to be visible in its continuation.

$$\begin{array}{l}\langle\{\mathbf{exn}\,\langle n,v\rangle\}\oplus k,\sigma,\Sigma\rangle \longrightarrow \langle e,\rho[x\mapsto v],\{\mathbf{sto}\,\langle\sigma\rangle\}\oplus\{\mathbf{sto}\,\langle\sigma'\rangle\}\oplus k',\sigma,\Sigma''\rangle\\ \qquad\text{provided } \Sigma=\Sigma'\oplus\{\langle\mathsf{exnVal}\,\langle n,\mathsf{clo}\,\langle\lambda\,x.e,\rho\rangle\rangle,k',\sigma'\rangle\}\oplus\Sigma''\\ \qquad\text{and } \langle\mathsf{exnVal}\,\langle n,v\rangle,k',\sigma'\rangle\notin\Sigma'\end{array}$$

$$\langle\{\mathbf{sto}\,\langle\sigma''\rangle\}\oplus\{\mathbf{sto}\,\langle\sigma'\rangle\}\oplus k,v,\sigma,\Sigma\rangle \longrightarrow \langle k,v,\sigma'+(\sigma-\sigma''),\Sigma\rangle$$

In this paper, we use the semantics shown in Fig. 6. The alternative interpretation does not pose any interesting challenges for our analysis.

Example

To illustrate the semantics, consider the program shown in Fig. 7. Some of the expressions are annotated with labels, as superscripts over the expressions enclosed in square brackets. For readability, we use syntactic sugar where appropriate; e.g. sequencing expressed using a semicolon(;) separator, is shorthand for nested let expressions.

After declaring variables s, x_1, x_2, x_3, p_1 and f, the program wraps a call to f (with argument x_3) inside an exception-protected scope. The function f raises an exception using restore after modifying various store locations. There is a dereference of x_1 in the try block's continuation.

Note that x_1 and x_1' are aliases and hence the assignment to x_1' at expression 7 affects x_1 as well. In addition, function f actually only modifies locations x_1 (via the call to procedure g) and p_1 within its definition. However, only x_1 is live in the continuation of the try block. Hence, only x_1 needs to be saved and restored upon entry and exit from the try block at label 15. Since the language supports first-class locations and procedures, these inferences are not easily derived by simple syntactic examination of the program. Instead, a static analysis capable of recording assignments within exception-protected regions is required. The following section defines such an analysis.

5. Analysis

The exact semantics naively implements versioning exceptions by saving the complete store at a try block and restoring it when an exception is raised. Another naive implementation would be to log all assignment statements. However both of these are excessively conservative since

```
[let $s$ = vExn($\lambda z.z$) in
    [let $x_1$ = [ref 0]$^1$ in
        [let $x_2$ =[ref $x_1$]$^3$ in
            [let $x_3$= [$\lambda p$. [let $x'_1$ =!$x_2$ in [$x'_1$:=1]$^7$; [$p$]$^8$]$^6$]$^5$ in
                [let $p_1$=[ref 2]$^{10}$ in
                    [let f=[$\lambda g$.let $p_2$=$g$(4)
                                  in $p_1$ := $p_2$; restore(s,100)]$^{12}$
                        in [try($s$, [$f$($x_3$)]$^{14}$)]$^{15}$; [!$x_1$]$^{16}$
                    ]$^{13}$ ]$^{11}$ ]$^9$ ]$^4$ ]$^2$ ]$^0$
```

Figure 7. Example program.

not all store locations may be modified and not all of these may be live in the continuation. We want to save/restore only those locations that are modified in the sub-expression of the try block and are used in its continuation. A try expression may have many continuations since it may occur within a procedure invoked from different call-sites. Thus, an interprocedural program analysis is required to approximate the continuation set and the set of locations modified in the dynamic context of the expression the try expression protects.

To do so, our analysis builds the abstract maps defined in Fig. 8. To simplify the presentation we assume that every expression in program P is uniquely labeled.

$\hat{v} \in \widehat{Value}$	$= \mathbf{Int} \mid \mathbf{Bool} \mid ALamExp$ $\mid ARefExp \mid AExcpn$	$AExcpn$	$= \widehat{\mathsf{Excpn}}\langle l\rangle$
		$ALamExp$	$= \widehat{\mathsf{LamExp}}\langle l\rangle$
		$ARefExp$	$= \widehat{\mathsf{RefExp}}\langle l\rangle$
$\mathcal{F} \in Flowset$	$= Var + Label \to \mathcal{P}(\widehat{Value})$	$\mathcal{K} \in ContMap$	$= Label \to \mathcal{P}(Exp)$
$\mathcal{U} \in Useset$	$= Label \to \mathcal{P}(Var)$	$\mathcal{M} \in Defset$	$= Label \to \mathcal{P}(Var)$
$\mathcal{A} \in IMap$	$= Var \to \mathcal{P}\ (Var^{\{-1,+1\}})$		

Figure 8. Abstract maps.

Abstract values include abstract base types, closures, locations, and exception values. An abstract closure *ALamExp* is the label of the λ-expression of its exact counterpart, and an abstract location *ARefExp* is the label of the ref expression that creates the corresponding exact location, while *AExcpn* abstracts exception values by approximating all exceptions created at label l. (We use $\widehat{constructor}$ to denote that the *constructor* generates abstract values.)

$\mathcal{F}$ is a flow function that abstracts the environment, and produces control and data flow information: for any variable x, $\mathcal{F}(x)$ returns the abstract values that x may acquire; for any label l, $\mathcal{F}(l)$ returns the abstract values produced by the expression with label l.

Four maps are used to produce necessary reachability information:

1. For any expression $[e]^l$, the *continuation* map $\mathcal{K}(l)$ returns the set of e's continuations.

2. For any expression $[e]^l$, the *use* map $\mathcal{U}(l)$ gives the set of variables referenced in e.

3. For any expression $[e]^l$, the *def* map $\mathcal{M}(l)$ gives the list of variables bound to locations that are modified in this expression by an assignment statement.

4. Map $\mathcal{A}$ defines *points-to* and *pointed-by* relations for each variable. These help in capturing data dependencies among store locations, which allows us to inductively build reachability information through elements in the store. The integer superscript is used to differentiate between points-to and pointed-by relations.

We elaborate on the structure of these maps below.

Flow Function

The specification of the flow function is defined in terms of subset constraints for a typical monovariant (flow- and context-insensitive) flow analysis [19, 31]. Its specification relies on a helper function $\mathit{last} : \mathit{Exp} \to \mathcal{P}(\mathit{Exp})$ that returns the set of the last sub-expression(s) of an expression. The function is defined as follows:

$$\begin{array}{ll}
\mathit{last}(\mathsf{let}\ x{=}s\ \mathsf{in}\ e) & = \mathit{last}(e) \\
\mathit{last}(\mathsf{try}(x{,}e)) & = \mathit{last}(e) \\
\mathit{last}(\mathsf{if}\ x\,\mathsf{then}\ e_1\ \mathsf{else}\,e_2) & = \mathit{last}(e_1) \cup \mathit{last}(e_2) \\
\mathit{last}(x_1\ :=\ x_2^l) & = \{x_2^l\} \\
\mathit{last}(\mathsf{restore}(x, y)) & = \cup_{\mathsf{Excpn}\langle l\rangle \in \mathcal{F}(x), [\mathsf{vExn}(z)]^l \in P, \lambda\, w.e \in \mathcal{F}(z)}\ \mathit{last}(e) \\
\mathit{otherwise}\ \mathit{last}(e^l) & = \{e^l\}
\end{array}$$

Note that for the restore expression, *last* returns the last subexpression(s) of the set of exception handlers that could be called.

To avoid enforcing constraints on dead code, we only require constraints to hold on *initialized* expressions, where $Init_F$ is the smallest set of subexpressions of P that includes P and is closed under the following rules:

1. If $\mathsf{let}\, x = s\, \mathsf{in}\, e \in \mathit{Init}_F$ and $s \neq y(z)$, $\{s, e\} \in \mathit{Init}_F$.

If $e \in \mathit{Init}_F$ and e is

- $\mathsf{let}\, x = c \,\mathsf{in}\, e'$, then $typeOf(c) \in \mathcal{F}(x)$.
- $\mathsf{let}\, x = Pr(x_1,\ldots,x_n) \,\mathsf{in}\, e'$, then $typeOf(Pr(x_1,\ldots,x_n)) \in \mathcal{F}(x)$.
- $\mathsf{let}\, x = [\lambda\, w.e']^l \,\mathsf{in}\, e$, then $\mathsf{LamExp}\langle l\rangle \in \mathcal{F}(x)$.
- $\mathsf{let}\, x = [\mathsf{ref}\, y]^l \,\mathsf{in}\, e'$, then $\mathsf{RefExp}\langle l\rangle \in \mathcal{F}(x)$, and $\mathcal{F}(y) \subseteq \mathcal{F}(l)$.
- $\mathsf{let}\, x = !y \,\mathsf{in}\, e'$, then $\forall \mathsf{RefExp}\langle l\rangle \in \mathcal{F}(y)$, $\mathcal{F}(l) \subseteq \mathcal{F}(x)$.
- $\mathsf{let}\, x = [\mathsf{vExn}(y)]^l \,\mathsf{in}\, e'$, then $\mathsf{Excpn}\langle l\rangle \in \mathcal{F}(x)$ and $\mathcal{F}(y) \subseteq \mathcal{F}(l)$.
- $\mathsf{let}\, x = y(z) \,\mathsf{in}\, e'$, then $\forall \mathsf{LamExp}\langle l\rangle \in \mathcal{F}(y), \forall[\lambda\, w.e'']^l \in P$, $\mathcal{F}(z) \subseteq \mathcal{F}(w)$, and if $\mathit{last}(e'') \in \mathit{Init}_F$, and $\mathit{last}(e'') = e_b^{l'}$, then $\mathcal{F}(l') \subseteq \mathcal{F}(x)$.
- $x_1 := x_2$, then $\forall \mathsf{RefExp}\langle l\rangle \in \mathcal{F}(x_1)$, $\mathcal{F}(x_2) \subseteq \mathcal{F}(l)$.
- $[\mathsf{try}(x, e')]^l$, then
  (a) if $\mathit{last}(e) \in \mathit{Init}_F$ and $\mathit{last}(e) = e_b^{l'}$, then $\mathcal{F}(l') \subseteq \mathcal{F}(l)$
  (b) $\forall[\mathsf{restore}(y, z)]^{l''} \in P$ such that $\mathcal{F}(x) \cap \mathcal{F}(y) \neq \phi$, $\mathcal{F}(l'') \subseteq \mathcal{F}(l)$.
- $[\mathsf{restore}(x, y)]^l$, then
  $\forall \mathsf{Excpn}\langle l'\rangle \in \mathcal{F}(x)$, $[\mathsf{vExn}(z)]^{l'} \in P$, $\forall[\lambda\, w.e_b]^{l''} \in \mathcal{F}(z)$
  (a)$\mathcal{F}(y) \subseteq \mathcal{F}(w)$
  (b) if $\mathit{last}(e_b) \in \mathit{Init}_F$ and $\mathit{last}(e_b) = e_t^{l'''}$, then $\mathcal{F}(l''') \subseteq \mathcal{F}(l)$.

Figure 9. Flow analysis constraints.

2. If $\mathsf{let}\, x = y(z) \,\mathsf{in}\, e \in \mathit{Init}_F$ and $\lambda\, w.e' \in \mathcal{F}(y)$ then:
   (a) $e' \in \mathit{Init}_F$.
   (b) if $\mathit{last}(e') \in \mathit{Init}_F$, then $e \in \mathit{Init}_F$.
3. If $\mathsf{if}\, x \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 \in \mathit{Init}_F$ then $\{e_1,e_2\} \in \mathit{Init}_F$.
4. If $\mathsf{try}(x, e) \in \mathit{Init}_F$, then $e \in \mathit{Init}_F$.
5. If $\mathsf{restore}(x, y) \in \mathit{Init}_F$, $\mathsf{Excpn}\langle l\rangle \in \mathcal{F}(x), [\mathsf{vExn}(z)]^l \in P, \lambda\, w.e \in \mathcal{F}(z)$ then $e \in \mathit{Init}_F$.

The flow constraints for the language are given in Fig 9. Most of the rules are standard. The flow analysis associates an abstract exception value with exception constructors; this value contains the program label of the constructor expression. A try block results in flows being computed for both the label of the block that stem from the expression protected by the block as well as the results of any handler for its exception argument that could be raised within its body. These abstract values are recorded in the expression's associated label. The computed flow for a restore expression establishes two subset constraints. The first propagates the argument value for handlers to the handler parameter. The second propagates the result of the handler to the expression's label. Constraints between try and restore expressions ensure that these values are propagated to the result label of the appropriate try block.

The Continuation Map

In order to make versioning exceptions practical, it is necessary to minimize the set of locations saved on entry to a try expression. Informally, only those locations that are both live in the expression's continuation and modified in the protected expression need to be preserved. To compute this intersection, we must determine the set of return points (or continuations) for expressions in the program. Because the language supports higher-order procedures and first-class exceptions, this set cannot be determined by simple syntactic examination of the program.

Instead, we build a continuation map that associates with every expression e^l, the set of expressions that constitutes e's continuations, $\mathcal{K}(l)$. This map is built in two steps. First, the original program is transformed into CPS (continuation-passing style) [12]. The translation to CPS given an input program in A-normal form is straightforward [12]. This transformation inserts continuation procedures and variables for all non-trivial expressions. In particular, the CPS conversion of a try expression of the form $\mathsf{try}(x,e)$ with continuation variable k will be translated into $\mathsf{try}(x, e')$ where every subexpression $e'' \in \mathit{last}(e)$ is transformed into a call to k. Second, we apply flow analysis on this transformed program to compute the continuation procedures bound to continuation variables. For example, the CPS transform of a procedure $p = \lambda\, x.e$ with continuation k is $\lambda\langle x, k\rangle.\mathit{CPS}[\![e]\!]k$. Each distinct call to p supplies a new continuation procedure argument for k. Thus, given $\mathsf{try}(x, e)$, e's continuation set represents the return points where control is transferred upon completion of the try expression.

The Variable Use Map

For any expression $[e]^l \in \mathit{Init}_F, \mathcal{U}(l)$ contains the set of variables referenced in e. In the case of restore expressions, all variables referenced in the handler belong to the set of variables that are used by the restore expression.

For all $e_0^l \in Init_F$, if e_0 is

- $\mathsf{let}\, x = e_1^{l_1}\, \mathsf{in}\, e_b^{l_b}$, then $\mathcal{U}(l_1) \cup \mathcal{U}(l_b) \subseteq \mathcal{U}(l)$.
- $\mathsf{vExn}(x)$, then $\{x\} \in \mathcal{U}(l)$.
- $x_0(x_1)$, then $\{x_0, x_1\} \in \mathcal{U}(l)$ and $\forall \mathsf{LamExp}\langle l'\rangle \in \mathcal{F}(x_0)$, $\mathcal{U}(l') \subseteq \mathcal{U}(l)$.
- x *or* $e_0 = \mathsf{ref}\, x$ *or* $e_0 = !x$, then $\{x\} \in \mathcal{U}(l)$.
- $Pr(x_1,\ldots,x_n)$, then $\{x_1, \ldots, x_n\} \in \mathcal{U}(l)$.
- $x_0 := x_1$, then $\{x_1\} \in \mathcal{U}(l)$.
- $\mathsf{restore}(x,y)$, then $\{x,y\} \in \mathcal{U}(l)$ and $\forall \mathsf{Excpn}\langle l_1\rangle \in \mathcal{F}(x)$, $\exists[\mathsf{vExn}(z)]^{l_1} \in P$, $\forall[\lambda\, w.[e_b]^{l_3}]^{l_2} \in \mathcal{F}(z)$, $\mathcal{U}(l_3) \in \mathcal{U}(l)$
- $\mathsf{if}\ x\, \mathsf{then}\ e_t^{l_t}\ \mathsf{else}\, e_f^{l_f}$, then $(\{x\} \cup \mathcal{U}(l_t) \cup \mathcal{U}(l_f)) \subseteq \mathcal{U}(l)$.
- $\mathsf{try}(x, e^{l'})$, then $\mathcal{U}(l') \subseteq \mathcal{U}(l)$.

The Def Map and Points-to Map

Besides determining the continuations and references of expressions, the analysis also requires information about where locations and bindings are defined, how they are dereferenced and related. The variable definition map $\mathcal{M}$ and the points-to map $\mathcal{A}$ provide this information.

$\mathcal{A}$ tracks both points-to and pointed-by information. To illustrate, if $\mathcal{A}(x)$ has a member

y^{-1}	then x points-to y.
	If $typeOf(y)$ is τ then $typeOf(x)$ would be $\mathbf{Ptr}\ \tau$.
y^{+1}	then x is pointed-by y.
	If $typeOf(y)$ is $\mathbf{Ptr}\ \tau$ then $typeOf(x)$ would be τ.

If dereferencing variable x returns the content of the variable y then we say x points-to y and y is pointed-by x. Note that map $\mathcal{A}$ is a weighted directed graph. If $y^w \in \mathcal{A}(x)$ then there is an edge from x to y, and the weight of the directed edge (x, y) is w.

Fig. 10 shows the definition of $\mathcal{A}$ and $\mathcal{M}$. Consider rule 3. The set of locations modified by a restore expression includes all the modifications performed by all the exception handlers that can be invoked by the $\mathsf{restore}$ expression.

To illustrate the construction of the points-to map, note that as an effect of the assignment expression (Rule 5) x_1 points-to x_2 and x_2 is pointed-by x_1. In rules $(5-8)$ we insert the points-to/pointed-by constraints only if both variables are bound to locations. This is accomplished by ensuring the type of the variable is $\mathbf{Ptr}\ \ \tau$.

In the let expression (8), for each possible procedure $\lambda x.e_1$ that can flow into x_0, x_3 (the argument) and x will share the same location.

(1)
$[\mathsf{if}\, x\, \mathsf{then}\, [e_1]^{l_1}\, \mathsf{else}\, [e_2]^{l_2}]^l$
– $\mathcal{M}(l_1) \cup \mathcal{M}(l_2) \subseteq \mathcal{M}(l)$

(2) $[\mathsf{try}(x, [e_1]^{l_1})]^l$
– $\mathcal{M}(l_1) \subseteq \mathcal{M}(l)$.

(3) $[\mathsf{restore}(x, y)]^l$
– $\forall \mathsf{Excpn}\langle l_1 \rangle \in \mathcal{F}(x)$,
  s.t. $[\mathsf{vExn}(z)]^{l_1} \in P$,
 $\forall [\lambda\, w.[e_b]^{l_3}]^{l_2} \in \mathcal{F}(z)$,
  $\mathcal{M}(l_3) \in \mathcal{M}(l)$

(4) $[\mathsf{let}\, x = s\, \mathsf{in}\, [e]^{l_1}]^l$
if $s \equiv c$ or $s \equiv Pr(x_1, \ldots, x_n)$
or $s \equiv \lambda\, y.e'$ or $s \equiv \mathsf{vExn}(y)$
– $\mathcal{M}(l_1) \subseteq \mathcal{M}(l)$.

(5) $[x_1 := x_2]^l$
– $\{x_1\} \in \mathcal{M}(l)$.
– if $typeOf(x_2)$=$\mathbf{Ptr}\ \tau$
 . $\{{x_1}^{+1}\} \subseteq \mathcal{A}(x_2)$
 . $\{{x_2}^{-1}\} \subseteq \mathcal{A}(x_1)$

(6) $[\mathsf{let}\, x_1 = !x_2\, \mathsf{in}\, [e]^{l_1}]^l$
– $\mathcal{M}(l_1) \subseteq \mathcal{M}(l)$.
– if $typeOf(x_1)$=$\mathbf{Ptr}\ \tau$
 . $\{{x_2}^{+1}\} \subseteq \mathcal{A}(x_1)$
 . $\{{x_1}^{-1}\} \subseteq \mathcal{A}(x_2)$

(7) $[\mathsf{let}\ x_1 = \mathsf{ref}\, x_2\ \mathsf{in}\ [e]^{l_1}]^l$
– $\mathcal{M}(l_1) \subseteq \mathcal{M}(l)$.
– if $typeOf(x_2)$=$\mathbf{Ptr}\ \tau$
 . $\{{x_2}^{-1}\} \subseteq \mathcal{A}(x_1)$
 . $\{{x_1}^{+1}\} \subseteq \mathcal{A}(x_2)$

(8) $[\mathsf{let}\ x_1 = x_0(x_3)\ \mathsf{in}\ [e]^{l_1}]^l$
– $\mathcal{M}(l_1) \subseteq \mathcal{M}(l)$.
– $\forall [\lambda x.[e_1]^{l_2}]^{l_3} \in \mathcal{F}(x_0)$
 . $\mathcal{M}(l_2) \subseteq \mathcal{M}(l)$.
 . if $typeOf(x) = \mathbf{Ptr}\ \tau$
  $\mathit{Unify}(\mathcal{A}(x), \mathcal{A}(x_3))$
 (a) if $x_2 \in \mathit{last}(e_1)$ and
   $typeOf(x_2)$=$\mathbf{Ptr}\ \tau$
    $\mathit{Unify}(\mathcal{A}(x_1), \mathcal{A}(x_2))$
 (b) if $!x_2 \in \mathit{last}(e_1)$ and
   $typeOf(x_1)$=$\mathbf{Ptr}\ \tau$
    $\{{x_2}^{+1}\} \subseteq \mathcal{A}(x_1)$
    $\{{x_1}^{-1}\} \subseteq \mathcal{A}(x_2)$.
 (c) if $\mathsf{ref}\ \ x_2 \in \mathit{last}(e_1)$ and
   $typeOf(x_2)$=$\mathbf{Ptr}\ \tau$
    $\{{x_2}^{-1}\} \subseteq \mathcal{A}(x_1)$
    $\{{x_1}^{+1}\} \subseteq \mathcal{A}(x_2)$.

Figure 10. Constraints for Abstract Def Map and Points-to map.

Hence we unify their points-to maps. The procedure *Unify*(a, b) generates two constraints: $a \subseteq b$ and $b \subseteq a$. The unification of points-to information between the argument and the formal parameter is essential. Within the function body, these variables effectively serve as aliases for the same location; thus, updates to one must be propagated to the other.

Different constraints are generated depending on the members of *last*(e). For example, if $x_2 = \mathit{last}(e)$ is bound to a location, we unify the points-to map for x_2 and the map for the result; this allows effects to be propagated bi-directionally between caller and callee. As before,

$\mathcal{F}(s)$	$=$	$\{\mathsf{Excpn}\langle 0\rangle\}$	$\mathcal{F}(1)$	$=$	$\{\mathbf{Int}\ \}$
$\mathcal{F}(x_1)$	$=$	$\{\mathsf{RefExp}\langle 1\rangle\}$	$\mathcal{F}(10)$	$=$	$\{\mathbf{Int}\ \}$
$\mathcal{F}(x'_1)$	$=$	$\{\mathsf{RefExp}\langle 1\rangle\}$	$\mathcal{M}(14)$	$=$	$\{x'_1, p_1\}$
$\mathcal{F}(x_2)$	$=$	$\{\mathsf{RefExp}\langle 3\rangle\}$	$\mathcal{M}(6)$	$=$	$\{x'_1\}$
$\mathcal{F}(x_3)$	$=$	$\{\mathsf{LamExp}\langle 5\rangle\}$	$\mathcal{U}(16)$	$=$	$\{x_1\}$
$\mathcal{F}(f)$	$=$	$\{\mathsf{LamExp}\langle 12\rangle\}$	$\mathcal{A}(x_1)$	$=$	$\{{x_2}^{+1}\}$
$\mathcal{F}(g)$	$=$	$\{\mathsf{LamExp}\langle 5\rangle\}$	$\mathcal{A}(x_2)$	$=$	$\{{x_1}^{-1}, {x'_1}^{-1}\}$
$\mathcal{F}(p_1)$	$=$	$\{\mathsf{RefExp}\langle 10\rangle\}$	$\mathcal{A}(x'_1)$	$=$	$\{{x_2}^{+1}\}$
$\mathcal{F}(p)$	$=$	$\{\mathbf{Int}\ \}$	$\mathcal{K}(15)$	$=$	$\{!x_1\}$
$\mathcal{F}(p_2)$	$=$	$\{\mathbf{Int}\ \}$			
$\mathcal{F}(z)$	$=$	$\{\mathbf{Int}\ \}$			

Figure 11. Abstract maps for the program shown in Fig. 7.

note that we cannot simply establish uni-directional constraints because there is a chance that both x_1 and x_2 maybe live simultaneously after the alias is created. The constraints generated for the other cases are analogous to the ones for the previous let expressions$(6, 7)$.

Example revisited

In Fig. 11 we show the effect of applying these constraints to the example shown in Fig. 7. We first compute the maps $\mathcal{F}$,$\mathcal{U}$,$\mathcal{M}$ and $\mathcal{K}$. We then build the points-to-map $\mathcal{A}$.

Consider the map $\mathcal{A}$. $\mathcal{A}(x_1)$ is $\{{x_2}^{+1}\}$. This suggests that x_1 is pointed-by x_2 and hence modifications done to the contents of the location bound to x_1 can affect the evaluation of the binding expression for x_2. As another example, $\mathcal{A}(x_2)$ is $\{{x_1}^{-1}, {x'_1}^{-1}\}$. This suggests that x_2 points-to x_1 and x'_1, and hence modifications to the store performed through an access of x_2 may affect the value of x_1 and x'_1.

We can interpret the points-to map pictorially as a directed graph as shown in Fig. 12. For example, it shows that, x_1 and x'_1 can be modified by dereferencing x_2 (due to the -1 edges) and modifying x_1 or x'_1 can affect a variable that is obtained by dereferencing x_2 (due to the $+1$ edges).

The graph also gives alias information: From node x_1 we can reach node x'_1 at cost $(-1 + 1 = 0)$. Hence x_1 and x'_1 are potential aliases.

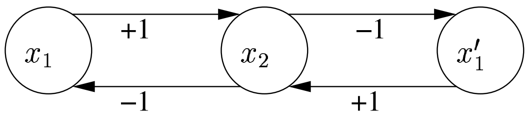

Figure 12. A points-to map can be depicted graphically with weights as edge labels.

Algorithm

The structure of the maps allow us to compute the subset of locations whose contents need to be saved upon entry to a try block protecting an expression which may raise a versioning exception.

For each try block of the form $[\mathsf{try}(x, [e_1]^{l_1})]^l$ we define $\Gamma(l)$ to be a subset of expressions that evaluate to locations whose contents need to be saved at the try expression. Let variable u be used in some continuation of the try block, i.e. $u \in \mathcal{U}(l')$ and $e^{l'} \in \mathcal{K}(l)$; assume further that u is also live in the exception expression e_1. If there is a variable v that is modified in e_1 and there exists a path from v to u in the points-to graph $\mathcal{A}$, then we need to save the location denoted by v. In other words, for each try expression, we want to save all locations which will be referenced in the continuation and may be modified in the exception expression. We can define this intuition formally thus:

DEFINITION 1. **Path**: Define $\mathit{path} \subseteq \mathit{Path}$: $(\mathit{Var}, \mathit{Var}) \rightarrow (\mathcal{P}(\mathit{Var}), \mathbf{Int}\ )$ as follows:

$$\mathit{path}(x_1,x_1) = \{\{x_1\},0\}$$
$$\mathit{path}(x_1,x_2) = \forall\ x^w\ \in\ \mathcal{A}(x_1),\ \{\ (p\ \cup\ \{x_1\}, j+w)\ |\ (p,j)\ \in\ \mathit{path}(x,x_2)\ \}$$

$\mathit{path}(x_1, x_2)$ computes a transitive points-to relation in terms of the path between the two variables and returns along with the path, the sum of weights on all the edges, indicating the total number of dereferences that need to be done to reach x_2 from x_1.

DEFINITION 2. **(Save/Restore)**. Let $e = [\mathsf{try}(x, e_b^{l_b})]^{l_1}$, and let $e^{l_k} \in \mathcal{K}(l)$. Then,

If $u \in \mathcal{U}(l_k) \cap \mathit{Live}(l_b)$,	*u is used in continuation and visible in e_b,*
and $\exists v \in \mathcal{M}(l_b)$	*and if v is modified in e_b,*
and $\mathit{path}(v, u) = (p, i)$,	*and there is a transitive path from v to u*
where $p \neq \phi$ and $i = 0$	*such that v and u may be aliases*
then $\Gamma(l) = \{\alpha \mid \alpha = \mathit{deref}(u, i)\}$	*then restore back $v = \mathit{deref}(u, i)$*

The function $\mathit{deref}(u, i)$ returns an expression which dereferences the variable u, i number of times. e.g. $\mathit{deref}(u, 2) =!(!(u))$. Note that, for any location v of the store, that need to be saved/restored by versioning exceptions, has to be accessible from some variable u, which is live in the exception expression and is referenced in the continuation. If no such u exists we need not save/restore v.

For the example shown in Fig. 7, $\Gamma(15)$ is $\{x_1\}$. Hence the contents of location x_1 upon entry to the try block must be saved.

A simple implementation of versioning exceptions would prepend bindings that capture the state preserved by Γ to each try block; and, would insert code at the end of the handler to restore the prepended state, and to transfer control to the continuation. Restore expressions would simply jump to the appropriate handler by raising the appropriate exception.

Precision and Limitations

Although our discussion thus far has focussed on the incorporation of versioning exceptions into a mostly functional language, the core language used in our study is sufficiently expressive to generalize our conclusions to other important (non-functional) languages like Java. While the addition of object-oriented features such as inheritance requires obvious extensions to our core semantics, these additions are completely orthogonal to the design of versioning exceptions. Indeed, the specification of such features within a lambda-calculus framework such as the one given here has been well-studied [13]. There are, however, three central features of Java that do impact the specification of versioning exceptions. The first has to do with the ability to invoke native methods; a native method call used within a versioning exception scope may modify shared locations accessible from Java methods. Since we cannot assume the ability to track control and dataflow through native methods, our reachability analysis can make no conclusion on the set of locations possibly modified therein. The second issue concerns dynamic class loading; if new class definitions can be loaded during program execution, then a whole-program interprocedural analysis of the kind presented here must be delayed until link-time when all class definitions are known. If performed earlier, the analysis may mistakenly assert reachability constraints that are violated by class and method definitions that are subsequently loaded and invoked. Finally, Java's support for multi-threading brings other issues to the forefront, some of which are discussed later in this section.

For the analysis presented here to be used in the context of a real programming language, it is required that we need to be able to translate all the semantic features of the language to an intermediate language similar to the language presented here. As a case study let us look at

Java. It can be observed that one main difference between Java and our language is the absence of high level data structures (objects, classes *etc.*) in the later. However such data structures can be easily abstracted out and the program can be translated to a intermediate form similar to the language present in the current paper. There are issues however. Somple simple ones are the absence of base types, absence of loop constructs, absence of control exceptions and so on. These things can be easily handled by adding new structures and types to our langaugage and the analysis would still apply with very little obvious modifications. There are also some issues like type casting, dynamic dispatching *etc.* that would require some fundamental additions to our language and possibly some latent changes to the computation of flow information in our analysis.

Most of the limitations of the analysis stem from its flow- and context-insensitive nature. For example, it critically relies on flow analysis to track dataflow through the dynamic context of a handler-protected scope. More accurate flow analyses would lead to more accurate refinement of the points-to sets it computes. Adding flow- or context-sensitivity is likely to improve the precision. Like other points-to analyses [2, 14], the accuracy of our approach depends on the accuracy of the underlying control-flow analysis. Loss of precision in accurately identifying the set of procedures that can be invoked at a call-site, or the set of continuations (i.e, return-points) of a procedure may lead to imprecise points-to information, resulting in more locations tagged for restoration when a versioning exception is raised than necessary.

Fig. 13 presents two cases that illustrate the imprecision caused by the context- and flow-insensitive nature of our analysis. The program fragment on the left defines two locations x_1 and x'_1, calls procedure f on these locations, assigns to x_1, and raises versioning exception s. Since the two calls to f are not disambiguated, the rules in Fig. 10 establish an unneeded alias among x_1 and x'_1. The program fragment on the right exhibits similar characteristics. The fourth let expression and the following assignment statement generates a graph as shown in 12. To see why, observe that the binding of x_2 to ref x_1 by Rule 7 in Fig. 10 induces a points-to constraint from x_2 to x_1. Now, when x_2 is subsequently modified by the assignment statement, $x_2 := x'_1$, a points-to constraint is also established from x_2 to x'_1. Because a path is established between x_1 and x'_1 (via x_2), both x_1 and x'_1 end up getting saved and restored even though neither need to be.

Because alias analysis is not integrated into our formulation, the analysis does not distinguish among references that are potential aliases for one another. To illustrate, consider references r_1 and r_2 created within the same exception scope, with each being assigned the contents

```
let $s$ = vExn($\lambda$ $z.z$) in
let $f$ = $\lambda$ $y$.1 in
let $x_1$ = ref 0 in
let $x'_1$ = ref 1 in
   try($s$,
      $f$($x_1$);
      $f$($x'_1$);
      $x_1 := 2$;
       restore($s, 5$));
   !$x'_1$
```

(a)

```
let $s$ = vExn($\lambda$ $z.z$) in
let $x_1$ = ref 0 in
let $x'_1$ = ref 1 in
let $x_2$ = ref $x_1$ in
   $x_2$ := $x'_1$;
   try($s$,
      $x_1$ := 3;
       restore($s, 5$));
   !$x'_1$
```

(b)

Figure 13. Analysis restores x'_1 because of context insensitivity in (a) and flow insensitivity in (b)

of variable v. Consider a use of v in the continuation of the try block under consideration. Our analysis would recommend the restoration of both r_1 and r_2, even though restoration of either one is suffficient. However, unlike unification-based points-to analyses [36], our analysis preserves full paths in the points-to maps, which would otherwise be lost because of path unification. As an example, for the program shown below, an alogrithm like [36] would suggest restoration of both x_1 and x'_1 because of path unification: That analysis would compute that x_1 may point to either of the two references RefExp$\langle 1 \rangle$ or RefExp$\langle 2 \rangle$. And hence to restore conservatively we have to restore both x_1 and x'_1. However our analysis would restore x_1 only.

```
let $s$ = vExn($\lambda$ $z.z$) in
let $x_1$ = [ ref 0]$^1$ in
let $x'_1$ = [ ref 1]$^2$ in
let $x_2$ = ref $x_1$ in
   $x_2$ := $x'_1$;
   try($s$, $x_1$ := 3; restore($s, 5$));
   !$x_1$
```

The analysis presented here has the obvious limitations for multithreaded applications that use shared variables. For example, say thread A changes location X within a try block, thread B reads X and changes it's state. Now, if X is restored in thread A, it is unclear whether B's state is still valid since it may be in the midst of performing computation that depends on the old, discarded value of X . One simple solution would be to copy all shared locations and variables to local structures just before the outermost try block and copy the updated local variables at the end of the try block. This method preserves sequential consistency, but is arguably inefficient. As part of our future work, we plan to examine extensions of our analysis to handle multi-threaded programs.

The semantics for versioning exceptions presented here are interesting in only those states of the program that can be 'undone'. For states like I/O it is not possible to restore the state in the absence of an agreed upon 'restore' definition. Recent work by Harris *et al* [17] formalize the separation between irrevocable actions such as I/O from revocable transactional behavior by generalizing the Haskell type system. Such an approach could be used to check the validity of raising a versioning exception in the presence of I/O.

Proof of safety

We show that saving and restoring the elements in Γ is *safe*. Our notion of safety captures the intuition that no modifications to the store in an exception expression are visible in the continuation of a try-block if a versioning exception is raised in the dynamic context of the block. We use two auxilary functions *Live* and *locs* in the safety theorem. *Live*:*Label* $\rightarrow$ $\mathcal{P}(\mathit{Var})$ that associates an expression label to the set of live variables at the beginning of that expression. For each expression present in the input set, *locs* returns the set of all the possible locations that could be returned by the evaluation of that expression. We state this safety property formally:

THEOREM 1. Let $[\mathsf{try}(x,[e_1]^{l_1})]^l \in P$, $\mathcal{K}(l) = \{k_1^{l_{k_1}}, \ldots, k_n^{l_{k_n}}\}$, and let $\mathit{eval}(P) = \langle \rho, \sigma, \textbf{halt}\ \langle v \rangle\rangle$ Then,
1. $\mathit{locs}(\Gamma(l)) \subseteq \mathit{Dom}(\sigma)$
2. $\mathit{locs}(\Gamma(l)) \supseteq \mathit{locs}(\mathit{Live}(l_1)) \cap \mathit{locs}(\mathcal{U}(l_{k_i})) \cap \mathit{locs}(\mathcal{M}(l_1))$, for $1 \leq i \leq n$.

Proof. The first part of the theorem is trivial because the locations obtained in *path* are always in the store σ. We prove the second part of the theorem, by induction on the structure of expressions.

The only interesting case to consider is assignment. Say the assignment expression is given by $x_1 := x_2$. There are two cases:

1. $x_1 \in \mathcal{U}(l_{k_i})$: Due to the constraints added, for the assignment statement (Rule 5 in Fig. 10) and the definition of Γ, it is ensured that the location of x_1 is restored.

2. $x_1' \in \mathcal{U}(l_{k_i})$ where x_1' is an alias of x_1: There are three subcases depending on the way in which the alias is created:

   a) The alias is created by expressions related to the rule $8(a)$ in Fig 10: The *Unify* procedure ensures in $\mathit{path}(x_1, x_1') = (p, i)$, p is not null and i is 0.

b) The alias is created by the presence of an even number of let expressions corresponding to the rules 6 and $8(b)$ in Fig 10 (e.g. let $x_1 =!x_2 \ \ldots$ let $x'_1 =!x_2 ...$): The constraints added ensures that in $path(x_1, x'_1) = (p, i)$, p is not null and i is $0(-1+1-1+1 \cdots$ *even number of times*).

c) The alias is created because of interaction among expressions corresponding to rules $5-8$ in Fig 10: This type of alias gets created due to expressions 5 and 6, 6 and 7, 5 and $8(b)$, and 6 and $8(c)$. As in the previous case, we can see that in $path(x_1, x'_1) = (p, i)$, p is not null and i is 0.

In all these cases (individually or a combination $locs(\Gamma(l)) \supseteq locs(Live(l_1)) \cap locs(\mathcal{U}(l_{k_i})) \cap locs(\mathcal{M}(l_1))$.

Hence the theorem holds and the analysis is safe. □

6. Complexity

We now consider the asymptotic running time of our analysis. The cost of computing Γ is the sum over the cost of computing *last*, $Init_F$, $\mathcal{F}$, $\mathcal{U}$, $\mathcal{M}$, $\mathcal{K}$, and $\mathcal{A}$.

The following table shows the time and space complexity of our analysis for building each of these maps, in terms of the number of program labels n.

Set	*last*	$Init_F$	$\mathcal{F}$	$\mathcal{U}$	$\mathcal{M}$	$\mathcal{K}$	$\mathcal{A}$	Γ
Time	$O(n^2)$	$O(n^2)$	$O(n^3)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^3)$
Space	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$

The constraints for *last*, $Init_F$, $\mathcal{U}$, $\mathcal{M}$, and $\mathcal{A}$ yield a complexity measure of $O(n^2)$, if the flow map is already computed. The rule for restore in the definition of the *last* function is recursive. Because the max size of the flow set is bound by program size n, which gives a bound on the number of the recursive calls, *last* takes $O(n^2)$ time. A similar argument gives $Init_F$, $\mathcal{U}$ and $\mathcal{M}$ a time complexity of $O(n^2)$. Careful observation of the algorithm for building the points-to map (Fig 10) shows that the LetApply rule (8) is recursive; this is similar to the restore rule in the algorithm for building *last*. Again, the number of such calls is limited by the size of the flow set $O(n)$. Hence we get the complexity of $O(n^2)$ for $\mathcal{A}$. The cost of building the continuations is linear in the program size. The constraints for building the flow sets are standard and lead to $O(n^3)$ complexity.

Including the cost of flow analysis, building the various maps used in the analysis requires $O(n^3)$ time. The computation of Γ takes only $O(n^2)$ time, for each `try` expression, after other helper maps are computed. Since the number of `try` expressions is bounded by n, the complexity for Γ is cubic in the program size. Since the complexity of the helper maps and Γ is bound by $O(n^3)$, the total time required by our analysis is bound by $O(n^3)$.

The space complexity shown is the space required for each individual maps. The total number of *initialized* expressions is bound by the program size n and hence we get a space complexity of $O(n)$. For each expression the number of last expressions is bound by the program size. Because the total number of expressions is also bound by $O(n)$, we get the space complexity for *last* as $O(n^2)$. Similar argument gives for each of $\mathcal{F}$, $\mathcal{U}$, $\mathcal{M}$, $\mathcal{K}$, $\mathcal{A}$, and Γ a space complexity of $O(n^2)$.

7. Related Work

There has been much work in defining expressive notions of persistency in high-level languages [4, 5] and devising algorithms to exploit persistent data structures [8, 9]. Versioning exceptions are inspired from both efforts. Peyton Jones *et al* [29, 25] present different extensions to simple exceptions in higher-order functional languages.

Our work bears obvious similarity to transaction models [26, 20, 37], but the idea of aborting modifications to the program store and environment in a higher-order language is novel. There has been recent interest in using transactional techniques for defining scalable and robust concurrent programs [18, 47]. These approaches rely on either language constructs that allow multiple threads of control to execute concurrently within a critical section [48, 49], or define hardware-supported mechanisms that track modifications to shared data [50]. When a serializability violation is detected, the changes made by the offending thread are discarded, and the thread must reexecute its changes. The rollback facility found in versioning exceptions bear obvious similarities to the logging mechanism used in transactional memory and optimistic concurrency implementations. However, the decision about when to abort a computation in a versioning exception context is determined by exceptional conditions that arise during program execution; in the case of transactional memory, aborts occur when serialization violations within a critical section are detected. In both cases, however, non-revocable actions such as I/O require alternative solutions. For example, an I/O or network operation performed within the scope of a versioning exception handler cannot be simply revoked. Realistic implementations must either require applications to provide compensation code to redress the effects of these actions, or raise an error or warning when such situations arise. Versioning exceptions also share similar goals as dynamic checkpointing schemes [1] which provide VM support to efficiently save and restore system state, although the techniques employed differ in obvious ways.

There have been other programming paradigms that provide abstractions similar to versioning exceptions. For example, goal evaluation in Prolog is expressed via backtracking that allows bindings established during clause evaluation to be revoked if unification of subsequent clauses fails. There have also been attempts to combine aspects of logic languages and other general purpose languages [44, 45]. Oz [44] is a high-level concurrent programming language that bridges aspects of logic and object oriented programming. In Curry [45] when a formal argument is evaluated with a logical variable as its actual value, a search takes place. During this search, the computation may follow different branches with different substitutions applied to the current goal. When the search operator realizes a non-deterministic solution, computation is halted and all the possible transitions are presented to the programmer. Depending on the choices provided, execution may proceed further. These techniques bear some similarity to our notion of versioning exceptions in the absence of side effects to the store. However, we are unaware of other work that exploits this notion to define new exception models in a non-logic programming context, and which present program analyses to enable efficient implementation.

Perhaps closest to our work is the tryall abstraction described by Shinnar *et. al.* [35] that integrates automated memory recovery with exception handling for C#. The goals of our two efforts are extremely similar: both address the question of how to restore program state and invariants when an exception is raised. We differ primarily in the techniques chosen. Their approach leverages runtime logging data structures to register old values of modified memory locations within a tryall block, restoring these values if an exception is thrown. In contrast, our focus has been on program analyses that achieve the same purpose. Our analysis could be used to minimize the amount of data their runtime data structures must support.

Points-to analysis is a very widely studied area. A fast context-insensitive pointer analysis is presented by Steensgaard [36] that converges more quickly, but is also more imprecise, than earlier work developed by Andersen [2]. Some optimizations that trade off between these two approaches are discussed by Shapiro and Horowitz [34]. Foster *et al* [14] discuss experimental results that explore tradeoffs between various points-to analysis frameworks. Context sensitive points-to-analyses exhibiting exponential complexity are presented by Emami *et al* [10, 43]. These analyses support first class function pointers. An alias analysis with the length of access paths being k-limited is given by Landi and Ryder [27]. While broadly similar to these other efforts, the work described here integrates points-to analyses with an inter-procedural control-flow analysis that deals with first-class functions, references and generative exceptions. Moreover, we preserve full paths in the points-to map without loss of precision due to path unification. Nevertheless, the space cost we incur is quadratic in the size of the program.

Program analysis using shape analysis is discussed by Chase *et al* in [6]. Some improvements are given by Sagiv *et al* in [32, 33]. Effect and region analysis also share similar goals to ours. To resolve questions about the scope of possible effects, Talpin and Jouvelot [39] and Talpin [38] present a type-and-effects analysis for functional programming languages. Their results (calculated in conjunction with type reconstruction) partitions the store into regions that are initialized, read or written. Region analysis helps in analyzing memory allocation, by identifying dynamically allocated variables with syntactically scoped regions of a program allowing potentially mutable variables to be allocated and deallocated in a stack discipline [41, 40, 16]. We believe the precision of the points-to analysis developed here can be improved by incorporating extensions that use region and effect inference techniques.

There has been significant work describing the hardness of the pointer analysis problem [15, 21, 42]. We give an approximate solution for the points-to-analysis applicable to our domain. Nevertheless, we can improve upon our analysis in many ways. Instead of the simple monovariant analysis, a polyvariant analysis [23, 7] would help in reducing the sizes of the abstract points-to-maps. Strong updates [6, 22], which are not incorporated into the analysis defined here, would also help in improving overall precision.

Versioning exceptions and the accompanying points-to analysis developed here have use in a number of important applications. A recent study by Weimer and Necula [51] examine a large number of Java programs, and observe that failure to provide appropriate cleanup code to restore resource invariants when an exception is raised is the cause of many runtime errors. They present a sophisticated program analysis to detect when this situation occurs. In one case study, their tool detected 800 such errors in roughly 4 million lines of code. The errors are often subtle and not immediately apparent by mere syntactic examination of the source text. Versioning exceptions obviate the need for such a tool. All effects performed within the dynamic context of the code protected by a versioning exception, including resource state changes, are *implicitly* restored to their state extant at the point the protected scope was entered. The analysis developed here is critical to the effectiveness and feasibility of these exceptions. Without it, a naive implementation would unlikely be able to support realistic programs of the kind examined in [51] where manual tracking of effects and state changes is infeasible.

8. Conclusion

This paper presents a novel exception mechanism and an associated static analysis. The mechanism allows the state of the program at the point where an exception handler is invoked to be restored to the point at which the handler is installed. The analysis developed analyzes the dynamic context of a handler-protected scope to identify those locations that need to be restored when an exception is raised.

The language we use for this paper is quite extensible as is the basis of the given analysis. Extending the language with features like complex data structures would actually make the construct of versioning exceptions much more useful. As we discuss in section 5, in the presence of multiple threads and shared variables, things need to be handled in a overly conservative manner in the current setting. We have plans to extend our current work to a setting with multiple threads and leave it as future work for this paper.

The analysis has multi-faceted applications. We show that it provides points-to and alias information. It can also be easily extended to provide escape information. For example, if in the points-to graph, a variable v in function f is only reachable by variables present in f, the references pointed by v do not escape f.

Due to its similarities with concepts found in logic programming and transactional systems, we believe the work presented here can be useful in defining efficient implementations of speculative or optimistic programming paradigms that require safe and effective revocation of state changes made to the global store.

9. Acknowledgments

We are grateful to Julian Padget, Olivier Danvy and the anonymous reviewers for their many helpful suggestions and comments.

References

1. Adnan Agbaria and Roy Friedman. Virtual-machine-based Heterogeneous Checkpointing. *Software Practice and Experience*, 32(12):1175–1192, 2002.
2. Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
3. Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
4. Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. PS-ALGOL: An Algol with a Persistent Heap. *SIGPLAN Notices*, 17(7):24–31, July 1982.
5. Malcolm P. Atkinson, Laurent Daynes, Mick J. Jordan, Tony Printezis and Susan Spence, An Orthogonally Persistent Java. *ACM SIGMOD record*, 25(4):68–75, December 1996.
6. David Chase and Mark N. Wegman and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *Proceedings of the Conference on Programming Language Design and Implementation*, 25(6):296–310, Jun 1990.
7. Charles Consel. Polyvariant Binding-time Analysis for Applicative Languages. In *Proceedings of the symposium on Partial evaluation and semantics-based program manipulation*, pages 66–77, Jun 1993.
8. James R. Driscoll, Daniel D. Sleator and Robert E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
9. James R. Driscoll, Daniel D. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of ACM*, 41(5):943–959, Sep 1994.
10. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the Conference on Programming Language Design and Implementation*, 29(6):242–256, Jun 1994.
11. Matthias Felleisen and Daniel P. Friedman. A Calculus for Assignments in Higher-order Languages. In *Proceedings of the Symposium on Principles of Programming Languages*, 314–325, Jan 1987.
12. Cormac Flanagan, Amr Sabry, Bruce F. Duba and Matthias Felleisen The Essence of Compiling with Continuations. In *Proceedings of the Conference on Programming Language Design and Implementation*, 28(6):237–247, Jun 1993.
13. Matthew Flatt, Shriram Krishnamurthi and Matthias Felleisen A Programmer's Reduction Semantics for Classes and Mixins. In *Proceedings of the Sysposium on on Formal Syntax and Semantics of Java*. Springer LNCS 1523, 241-269, 1999.
14. Jeffrey S. Foster and Manuel Fähndrich and Alexander Aiken Polymorphic versus Monomorphic Flow-Insensitive Points-to Analysis for C. In *International Symposium on Static Analysis*, pages 175–198, Apr 2000.
15. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness*. Freeman, 1979.
16. David Gay and Alexander Aiken. Language Support for Regions. In *Proceedings of the Conference on Programming Language Design and Implementation*, 36(5):70–80, May 2001.
17. Tim Harris, Simon Marlow, Simon Peyton Jones and Maurice Herlihy. Composable Memory Transactions. In *ACM Conference on Principles and Practice of Parallel Programming*. Jun 2005.

18. Tim Harris and Keir Fraser Language Support for Lightweight Transactions. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 338–402, 2003.
19. Nevin Heintze. Set-based Analysis of ML Programs. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 306–317, Jun 1994.
20. Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.
21. Susan Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Transactions on Progamming Languages and Systems*, 19(1):1–6, January 1997.
22. Suresh Jagannathan, Peter Thiemann, Stephen Weeks and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 329–341, Jan 1998.
23. Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 393–407, 1995.
24. Richard Kelsey and Paul Hudak. Realistic Compilation by Program Transformation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 281–293, 1989.
25. Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A Semantics for Imprecise Exceptions. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 25–36, 1999.
26. H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.
27. William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the Conference on Programming Language Design and Implementation* , 27(7):235–248, 1992.
28. Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic Support for Robust Distributed Programs. *ACM Transactions on Progamming Languages and Systems*, 5(3):381–404,1983.
29. Simon Marlow, Simon Peyton Jones, Andrew Moran and John Reppy Asynchronous Exceptions in Haskell. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 274–285, 2001.
30. Robin Milner, Mads Tofte, Robert Harper and David MacQueen. *The Definition of Standard ML.* MIT Press, 1997.
31. Jens Palsberg. Closure Analysis in Constraint Form. *ACM Transactions on Progamming Languages and Systems*, 17(1):47–62, January 1995.
32. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving Shape-analysis Problems in Languages with Destructive Updating. *Transactions on Progamming Languages and Systems*, 20(1):1-50, Jan 1998.
33. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-valued Logic. *ACM Transactions on Progamming Languages and Systems*, 24(3):217–298, May 2002.
34. Marc Shapiro II and Susan Horwitz Fast and Accurate Flow-insensitive Points-to Analysis. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 1–14, Jan 1997.
35. Avraham Sinnar, David Tarditi, Mark Plesko and Bjarne Steensgard. Integrating Support for Undo with Exception Handling. Microsoft Research Technical Report MSR-TR-2004-140, Dec. 2004.

36. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of 23nd Annual Symposium on Principles of Programming Languages*, pages 32–41, 1996.
37. Michael Stonebraker and Joseph Hellerstein. *Readings in Database Systems, Third Edition*. Morgan-Kaufmann, 1998.
38. Jean-Pierre Talpin. *Theoretical and Practical Aspects of Type and Effect Inference*. PhD thesis, University of Paris, 1993.
39. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
40. Mads Tofte and Lars Birkedal. A Region Inference Algorithm. *ACM Transactions on Progamming Languages and Systems*, 20(4):724–767, July 1998.
41. Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.
42. Venkatesan T. Chakaravarthy and Susan Horwitz. On The Non-Approximability of Points-to Analysis. *Acta Informatica*, 38(8):587-598, June 2001.
43. Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, 30(6):1–12, Jun 1995.
44. Gert Smolka. The Oz Programming Model. In Current Trends in Computer Science. Springer LNCS 1000, 1995.
45. M. Hanus and H. Kuchen and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In Proc. ILPS, Workshop on Visions for the Future of Logic Programming, 1995.
46. Nir Shavit and Dan Touitou Software Transactional Memory *ACM Symposium on Principles of Distributed Computing*, pages 204-213, 1995
47. Adam Welc, Suresh Jagannathan, and Antony L. Hosking Transactional Monitors for Concurrent Objects *European Conference on Object-Oriented Programming*, pages 519–542, 2004.
48. Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking A Semantic Framework for Designer Transactions *European Symposium on Programming*, pages 249–263, 2004.
49. Corman Flanagan and Shaz Qadeer A Type and Effect System for Atomicity *ACM SIGPLAN Confererence on Programming Language Design and Implementation*, 2004
50. Ravi Rajwar and James R. Goodman Transactional Lock-free Execution of Lock-Based Programs *ACM Conference on Architectural Support for Programming Languages and Systems*, 37(10):5–17, Oct 2002
51. Westley Weimer and George C. Necula Finding and Preventing Run-Time Error Handling Mistakes *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 419–431, 2004.