Statically-Scoped Exceptions: a Typed Foundation for Aspect-Oriented Error Handling

Neel Krishnaswami and Jonathan Aldrich

January 2005
CMU-ISRI-05-102

Institute for Software Research International
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

This report was originally published on the web in January 2004.

This work was supported in part by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and NSF grant CCR-0204047.

Keywords: statically-scoped exceptions, aspect-oriented programming

Abstract

Aspect-oriented programming systems such as AspectJ provide mechanisms for modularizing crosscutting error-handling concerns. However, AspectJ's advice does not integrate well with Java's checked exception mechanism. Furthermore, conventional exception-handling facilities such as AspectJ's share the problem of accidental exception capture due to the dynamic nature of exception-handling semantics.

We propose statically-scoped exceptions as an alternative mechanism for error-handling. In our system, a thrown exception is caught by the nearest lexically-enclosing exception handler, rather than the nearest handler on the call stack. Using static scoping allows us to achieve better modularization, as with AspectJ, and also precudes the problem of exception capture. We provide a static type system that tracks exceptions precisely and ensures that statically-scoped exceptions cannot be misused to cause continuation-like behavior. We hope that our system will serve as a foundation for error handling mechanisms that combine good modularization with strong static reasoning about errors.

```
public class Visitor {
  public void visitClass(Class o) {
    try {
      // method visitor code...
    } catch (ArchJavaError e) {
      e.setNode(o);
      ErrorHandler.print(e);
   }
  }

  public void visitMethod(Method o) {
    try {
      // method visitor code...
    } catch (ArchJavaError e) {
      e.setNode(o);
      ErrorHandler.print(e);
    }
  }

  public void visitLiteral(Literal o) {
    try {
      // literal visitor code...
    } catch (ArchJavaError e) {
      e.setNode(o);
      ErrorHandler.print(e);
    }
  }
  ...
}
```

Figure 1: In this visitor code, taken from the ArchJava compiler, each visit method performs a task such as typechecking on a part of the abstract syntax tree. The visitor code uses exceptions to report typechecking errors; these exceptions must be caught and reported at the end of each visit function before recovering so that typechecking can continue.

1. Introduction

Modern programming languages such as ML and Java offer *exception systems* as a structured means of handling nonlocal exits. When a function or object receives arguments that it cannot properly handle, it will raise an exception, which is propagated up the call stack until a handler is found that can process it. This is a substantial improvement over error-handling techniques such as returning status codes that must be manually checked on every call, since the programmer can separate error-handling code from ordinary code, and write programs with more flexible error-handling properties.

Although exceptions help to modularize error-handling code within a function, they often fail to effectively modularize error-handling concerns that crosscut function boundaries. For example, consider the visitor code in Figure 1. This code, taken from the compiler for the ArchJava language, divides the typechecking algorithm into visit methods for each node in the abstract syntax tree. When the typechecking code finds an error, it often cannot continue typechecking the current AST node in a meaningful way, so it throws an exception to break out back to the visitor. Each method in the visitor must be able to catch these exceptions, document the node that triggered the error, report the error to the user, and recover so that typechecking can continue in a meaningful way.

As the example shows, identical error-handling code is duplicated in each method in the visitor! Code duplication causes a number of well-understood problems, including the challenge of keeping the duplicate code in synch and the difficulty of understanding and evolving the code. Unfortunately this is unavoidable in Java, since the language does not have a good way to modularize error-handling code that crosscuts the application.

1.1 Exception Handling with Aspects

As Lippert and Lopes point out in their study of exception handling with AspectJ, aspect-oriented programming offers one solution to improving the modularity of error-handling code [3]. For example, the code in Figure 2 shows how AspectJ's `around` advice can be used to modularize the error-handling code in this example. In the error-handling aspect, the `visitors` pointcut picks out all of the visit methods, and the `around` advice wraps each call to a visit function with an exception handler that catches typechecking error exceptions.

1.2 Problems with Exception Mechanisms

Despite the fact that AspectJ is able to modularize the error-handling code in the example more effectively, this solution is

```
aspect HandleArchJavaErrors {
  pointcut visitors(Object o):
      execution(Visitor.visit*(..)) && args(o);

  void around(Object o): visitors(o) {
    try {
      proceed(o);
    } catch (ArchJavaError e) {
      e.setNode((ASTNode) o);
      ErrorHandler.print(e);
    }
  }
}
```

Figure 2: This AspectJ code modularizes the error handling code from Figure 1. The `visitors` pointcut picks out all of the visit methods, and the `around` advice wraps each call to a visit function with an exception handler that catches typechecking error exceptions.

```
let contains(tree, predicate) =
  let rec find(t) =
    match tree with
    | Empty -> false
    | Node(left, x, right) ->
       if predicate(x)
       then raise Found
       else find(left) || find(right)
  in try
    find(tree)
  with
    Found -> true
```

Figure 3: This O'Caml code checks to see if the predicate passed in is true for some element in a binary tree. The code uses exceptions to perform a non-local return when the element is found. If the predicate function happens to throw the `Found` exception, it will be captured by the try clause in contains, which will return the wrong result to the user.

unsatisfying in certain respects. First, the technique of wrapping method executions with `around` advice doesn't capture the error-handling intent well: the programmer wishes to handle all errors thrown in the `Visitor` in a uniform way, and wrapping method executions with an error handler is a crude way to accomplish this.

Second, this technique does not integrate well with exception checking. Since the aspect handles all `ArchJavaError` exceptions that are thrown from within `Visitor`, we would like our type system to document that the `visit` functions do not throw this exception. However, AspectJ's `around` advice does not change the signature of a method (including the list of exceptions it throws) so clients must be written to handle `ArchJavaError` even though the `visit` functions they call will never throw this error.

A third problem, that of *exception capture*, is common to all exception-handling mechanisms. We illustrate this problem with a function written in O'Caml to show that the issue occurs across different langauge designs. The function in Figure 3 is meant to return true if some element of a binary tree is true for a user-supplied predicate, and returns false otherwise.

The `Found` exception is used to bail out of the loop as soon as some true element is found, but if the `predicate` raises the `Found` exception, then the `contains` function will mistakenly return `true`. It would be desirable if this sort of capture were impossible.[1]

1.3 Statically-Scoped Exceptions

In this paper, we propose statically-scoped exceptions as a foundational error-handling mechanism that addresses the issues identified above. In our system, exception handlers are statically bound: a `throw` form is associated with the nearest *lexically* enclosing `catch` form. This feature allows a single block of code to handle exceptions that are thrown from any function in a module, thus modularizing error-handling code as effectively as the AspectJ solution but with a more direct mechanism.

We have designed a type system that tracks the exception handlers currently in scope, as well as the exceptions that might be thrown by each function. Our type system takes the statically enclosing exception handlers into account when inferring the

[1]The current O'Caml compiler has an extension which makes it possible to create a unique exception on every call by defining a new exception in a *local module*. However, exception capture is still possible for exceptions defined in commonly-used library code.

Variable Names	x	$\in$	`VarNames`
Exception Names	e	$\in$	`ExnNames`
Exception Names	h	$\in$	`HandlerNames`
Source Terms	t	$::=$	x
		$\mid$	$\lambda x{:}\tau.\ t$
		$\mid$	$t_1\ t_2$
		$\mid$	`()`
		$\mid$	`throw` e
		$\mid$	`exn` e `in` t
		$\mid$	$\texttt{catch}_h(t_1,\ e \Rightarrow t_2)$
Other Terms	t	$::=$	`throw` h
		$\mid$	$\texttt{handle}_h(t_1,\ e \Rightarrow t_2)$
Types	τ	$::=$	`unit` $\mid \tau_1 \stackrel{[H_s;E_d]}{\longrightarrow} \tau_2$

Figure 4: Exn Syntax

exceptions thrown by a function, so that in the `Visitor` example our system will conclude that the `visit` functions do not throw the `ArchJavaError` exception.

Just as static variable binding avoids the problem of variable capture experienced in early versions of Lisp, static binding of exception handlers lets us eliminate the exception capture problem in a principled way. Exceptions are always handled by the nearest lexically-enclosing handler, even if other handlers are on the call stack between the currently executing scope and the scope declaring the handler.

One reason static exception binding is unusual is that it permits programmers to write functions that capture the current continuation [1] (as in Scheme's `call-with-current-continuation` function). We regard the accidental introduction of first-class continuations as an undesirable feature, and have built a simple escape analysis into our type system that statically detects and forbids escaping continuations. This escape analysis complicates our type system somewhat, but we believe the cost in complexity is small in relation to the benefits of a more modular error-handling construct.

In the next section, we study statically-scoped exceptions in the foundational setting of the lambda calculus. This setting is ideal for understanding the semantic issues of our proposal and for proving that our type system is sound. A concrete language design addressing the specific problems of exception handling in Java or ML is beyond the scope of this paper, but will benefit from the understanding gained in the simpler setting here.

2. The `Exn` Language

2.1 Syntax

Figure 4 shows the abstract syntax of the `Exn` language. Variable and exceptions are assumed to be drawn from two disjoint, infinite sets of names; a third set of names is used to track the handler for each throw. Variables observe the usual convention that they may be alpha-converted freely to enforce distinctness of names bound by lambdas.

The `exn` e `in` t term introduces a new exception scope; the exception e becomes available inside the term t. Just like with lambdas, an alpha-conversion rule exists for exception names. We assume that they may be alpha-converted to ensure that all the exception names in a program are distinct.

The `throw` e term is used to throw exceptions. The handler associated with it is found by searching lexically outward in the program text, until an appropriate enclosing `catch` term is found. If no such handler is found, then `throw` e is referred to as a dynamic throw. The $\texttt{catch}_h(t_1,\ e \Rightarrow t_2)$ term catches all of the `throw` e terms within t_1, and all of the dynamic throws of e by functions used within t_1, and handles them by executing t_2. A unique tag h distinguishes this catch from all other catch expressions in the source text. There is no way to pass data along with the exception throw, but this is a convenient simplification of the language rather than an essential feature.

The $\texttt{handle}_h(t_1,\ e \Rightarrow t2)$ term is an intermediate form which cannot be used in source-level terms, and only arises during evaluation. Intuitively, it represents the mark on the stack locating where a throw should jump to. Likewise, `throw` h is another intermediate form; it is a throw whose jump target has been resolved to the handle form indicated by the label h.

The type language includes a unit type, and arrow types. Function arrows are annotated with two sets of exception names. The H_s set is the set of handler names that must be in scope for the function to be valid. The set E_d is the set of exceptions which can arise during a call to the function; they are "free", in the sense that they contain `throw` e forms that are not lexically enclosed by a `catch`.

2.2 Dynamic Semantics

Figure 5 defines the definition of values and contexts in `Exn`, and Figure 6 defines the language's dynamic semantics. The core of `Exn` is the simply-typed lambda calculus, augmented with `catch` and `throw` operations. The values of `Exn` are the unit value and functions.

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & \texttt{()} \mid \lambda x{:}\tau.\ t \\
\text{Exception Sets} & E & ::= & \epsilon \mid E, e \\
\text{Handler Sets} & H & ::= & \epsilon \mid H, h \\
\text{Handler Maps} & M & ::= & \epsilon \mid M, e \mapsto h \\
\text{Variable Contexts} & \Gamma & ::= & \epsilon \mid \Gamma, x : \tau \\
\text{Evaluation Contexts} & C & ::= & \Box \\
 & & \mid & C[\Box\ t_2] \\
 & & \mid & C[\lambda x{:}\tau.\ t\ \Box] \\
 & & \mid & C[\texttt{exn}\ e\ \texttt{in}\ \Box] \\
 & & \mid & C[\texttt{handle}_h(\Box,\ e \Rightarrow t_2)]
\end{array}
$$

Figure 5: Exn Values and Contexts

$$\frac{t \mapsto t'}{C[t] \mapsto C[t']}\ \texttt{[E-CONTEXT]}$$

$$\frac{}{\lambda x{:}\tau.\ t\ v \mapsto \{v/x\}t}\ \texttt{[E-APP]}$$

$$\frac{}{\texttt{exn}\ e\ \texttt{in}\ v \mapsto v}\ \texttt{[E-EXN]}$$

$$\frac{h'\ fresh}{\begin{array}{l}\texttt{catch}_h(t_1,\ e \Rightarrow t_2) \mapsto \\ \texttt{handle}_{h'}(\{h'/e, h'/h\}t_1,\ e \Rightarrow t_2)\end{array}}\ \texttt{[E-CATCH]}$$

$$\frac{}{\texttt{handle}_h(C[\texttt{throw}\ h],\ e \Rightarrow t_2) \mapsto t_2}\ \texttt{[E-THROW]}$$

$$\frac{}{\texttt{handle}_h(v,\ e \Rightarrow t_2) \mapsto v}\ \texttt{[E-HANDLE]}$$

Figure 6: Exn Operational Semantics

We enforce a left-to-right call-by-value semantics through the use of the evaluation contexts defined in Figure 5, and the `[E-CONTEXT]` evaluation rule. The `[E-APP]` rule is standard, but note that within an `exn` e `in` t term, evaluation proceeds until t reduces to a value v before discarding the enclosing `exn` term with the `[E-EXN]` rule. This demonstrates that `exn` e `in` t has no operational significance; we preserve it during evaluation in order to simplify the soundness proof of the language.

The `[E-CATCH]` rule defines the semantics of a `catch` forms. Reducing a catch form replaces it with a `handle` form with a fresh label, and replaces all of the visible throws with that same label.

$$\texttt{catch}_h(\texttt{throw}\ e,\ e \Rightarrow \texttt{()})$$

$$\mapsto \texttt{handle}_{h'}(\texttt{throw}\ h',\ e \Rightarrow \texttt{()})$$

However, this substitution is shadowed by intervening catch forms:

$$\texttt{catch}_{h_1}(\texttt{catch}_{h_2}(\texttt{throw}\ e,\ e \Rightarrow 12),\ e \Rightarrow 5)$$

$$\mapsto \texttt{handle}_{h'}(\texttt{catch}_{h_2}(\texttt{throw}\ e,\ e \Rightarrow 12),\ e \Rightarrow 5)$$

The `throw` h term will go up the stack to find the handler with the same label, and then invoke the handler. The unique renaming the reduction of the `catch` form did will guarantee that no intervening handlers can intercept this throw, which is how the `Exn` language avoids exception capture.

2.3 Static Semantics

`Exn`'s type system is based on the simply-typed lambda calculus, with additions to do exception tracking. The typing judgment in Figure 8 is of the form:

$$E; H; M; \Gamma \vdash t : \tau[H_s; E_d]$$

In this judgment, Γ, t and τ are as usual. E is the set of declared exceptions; new exceptions are introduced by the `exn` e `in` t

$$
\begin{array}{lcl}
\text{dyn}(\texttt{unit}) & = & \phi \\
\text{dyn}(\tau \xrightarrow{[H_s;E_d]} \tau') & = & E_d \cup \text{dyn}(\tau) \cup \text{dyn}(\tau') \\
\\
\text{stat}(\texttt{unit}) & = & \phi \\
\text{stat}(\tau \xrightarrow{[H_s;E_d]} \tau') & = & H_s \cup \text{dyn}(\tau) \cup \text{dyn}(\tau')
\end{array}
$$

Figure 7: Exn Auxilliary functions

form. H is the set of lexically-available handlers, and are introduced by `handler` forms. M is a mapping between the exceptions a `catch` form traps and the handler it defines. E_d are the exceptions that the expression t might throw – they are introduced by `throw` e forms that are not enclosed by a `catch`. Likewise, H_s lists the handlers that must be live for t to typecheck.

Functions types contain both the dynamic exceptions the function may throw, and the static exception handlers that the function relies on. For example, the expression

$$\texttt{exn}\ e\ \texttt{in}\ \texttt{catch}_h(\lambda x{:}\texttt{unit}.\ \texttt{throw}\ e,\ e \Rightarrow \texttt{()})$$

will not typecheck, because the lambda will have a type of $\texttt{unit} \xrightarrow{[e;\epsilon]} \texttt{unit}$, and the `[T-CATCH]` rule requires that $e \notin \text{stat}(\tau)$. We track static handlers such as h in the function type so that a function can't escape the scope of a handler if it uses that handler. This restriction implies that a function which relies on an enclosing handler may be passed to other functions as an argument within the `catch` block, but that it can't leave the block as a value. Thus, `catch` and `throw` cannot be used to create first-class continuations with dynamic extent.

3. Type Soundness

The typing rules for `Exn` are found in Figure 8, and we prove their soundness using a conventional pair of progress and preservation theorems.

Lemma 1 (Progress)
If $E; H; \epsilon; \epsilon \vdash t : \tau[H_s; \epsilon]$, then either

- *t is a value, or*
- *there exists a t' such that $t \mapsto t'$, or*
- *there exists a C such that $t = C[\texttt{throw}\ h]$, with $h \in H$.*

Proof: This is proved by a straightforward induction over the typing derivation. ■

Lemma 2 (Type Preservation)
If $E; H; \epsilon; \epsilon \vdash t : \tau[H_s; \epsilon]$, and $t \mapsto t'$, then there exists $H'_s \subseteq H_s$ such that $E; H; \epsilon; \epsilon \vdash t' : \tau[H'_s; \epsilon]$.

Proof: This is proved by a straightforward induction over the reduction. ■

Together, these permit us to prove a soundness theorem.

Theorem 3 (Soundness)
If $E; \epsilon; \epsilon; \epsilon \vdash t : \tau[\epsilon; \epsilon]$, then t will diverge or reduce to a value of type τ.

Proof: Observe that when $H = \epsilon$, the progress lemma requires that the term either be a value or step. Soundness then follows immediately. ■

4. Related Work

Foundations. Nanevski[4] has designed a system closest to `Exn` in spirit – he designed a core calculus based on constructive modal logic extended with names, and modelled named control effects (such as exceptions and composable continuations) using the necessity and possibility operators of modal logic. The primary difference between the two systems is that `Exn` exceptions are statically bound, rather than dynamically bound. We hope to add polymorphism to our system in the future by building on Nanevski's notion of "support polymorphism."

Pessaux and Leroy[5] designed an exception analysis for Ocaml. Their system differs quite substantially from ours; Ocaml's exceptions handlers are dynamically scoped, and are first-class values. Pessaux and Leroy designed a dual type-based and interprocedural flow analysis to track specific exception values. The flow analysis is not directly relevant to our work with `Exn`,

$$\frac{}{E; H; M; \Gamma \vdash \texttt{()} : \texttt{unit}[\epsilon; \epsilon]} \ \texttt{[T-UNIT]}$$

$$\frac{x : \tau \in \Gamma}{E; H; M; \Gamma \vdash x : \tau[\epsilon; \epsilon]} \ \texttt{[T-VAR]}$$

$$\frac{E, e; H; M; \Gamma \vdash t : \tau[H_s; E_d] \quad e \notin \text{dyn}(\tau) \quad e \notin \text{stat}(\tau) \quad e \notin E_d}{E; H; M; \Gamma \vdash \texttt{exn}\ e\ \texttt{in}\ t : \tau[H_s; E_d]} \ \texttt{[T-EXN]}$$

$$\frac{\begin{array}{r} E; H; M, e \mapsto h; \Gamma \vdash t_1 : \tau[H_s; E_d] \\ E; H; M; \Gamma \vdash t_2 : \tau[H'_s; E'_d] \end{array} \quad e \in E \quad e \notin \text{stat}(\tau)}{E; H; M; \Gamma \vdash \texttt{catch}_h(t_1,\ e \Rightarrow t_2) : \tau[(H_s - \{h\}) \cup H'_s; E_d \cup E'_d]} \ \texttt{[T-CATCH]}$$

$$\frac{\begin{array}{r} E; H, h; M; \Gamma \vdash t_1 : \tau[H_s; E_d] \\ E; H; M; \Gamma \vdash t_2 : \tau[H'_s; E'_d] \end{array} \quad e \in E \quad e \notin \text{stat}(\tau)}{E; H; M; \Gamma \vdash \texttt{handle}_h(t_1,\ e \Rightarrow t_2) : \tau[(H_s - \{h\}) \cup H'_s; E_d \cup E'_d]} \ \texttt{[T-HANDLE]}$$

$$\frac{e \in E \quad e \mapsto h \notin M \quad H \cap \text{dyn}(\tau) = \phi}{E; H; \Gamma; \texttt{throw}\ e \vdash \tau : \epsilon[e;]} \ \texttt{[T-THROW-D]}$$

$$\frac{e \in E \quad e \mapsto h \in H \quad H \cap \text{dyn}(\tau) = \phi}{E; H; M; \Gamma \vdash \texttt{throw}\ e : \tau[h; \epsilon]} \ \texttt{[T-THROW-S]}$$

$$\frac{e \in E \quad h \in H \quad H \cap \text{dyn}(\tau) = \phi}{E; H; M; \Gamma \vdash \texttt{throw}\ h : \tau[h; \epsilon]} \ \texttt{[T-THROW-H]}$$

$$\frac{\begin{array}{l} E; H; M; \Gamma \vdash t_1 : \tau_2 \xrightarrow{[H_s; E_d]} \tau[H^1_s; E^1_d] \\ E; H; M; \Gamma \vdash t_2 : \tau_2[H^2_s; E^2_d] \end{array}}{E; H; M; \Gamma \vdash t_1\ t_2 : \tau[H^1_s \cup H^2_s \cup H_S; E^1_d \cup E^2_d \cup (E_d - H)]} \ \texttt{[T-APP]}$$

$$\frac{E; H; M; \Gamma, x : \tau' \vdash t : \tau[H_s; E_d]}{E; H; M; \Gamma \vdash \lambda x{:}\tau'.\ t : \tau' \xrightarrow{[H_s; E_d]} \tau[\epsilon; \epsilon]} \ \texttt{[T-FUN]}$$

Figure 8: Exn Typechecking

but their type system does feature a use of row polymorphism to track exception constructors, which could possibly be adapted for a future polymorphic version of `Exn`.

Hayo Thielecke [6] investigated the logical and control properties of a throw/catch calculus by using a double-cps conversion on a single source language and systematically varying how the exception continuation was bound.

Practical Systems. There are two significant full-fledged languages which check exception declarations in function and method bodies – Java and Modula-3. Both of these languages also permit the use of unchecked exceptions; Java, through the use of exceptions which subclass `RuntimeException`, and Modula-3, through the use of a `RAISES ANY` declaration in a procedure definition. Java's use of class-based exceptions also allows exception handlers to catch exceptions of subclasses, which is not directly supported by our system.

As discussed in the introduction, AspectJ's around advice provides a way to wrap all the methods in a scope with exception handlers, modularizing error-handling code much as statically-scoped exceptions do [2]. Lippert and Lopez's study was one of the first to document the advantages of modularizing error-handling code using AspectJ [3]. Because AspectJ has a powerful pointcut mechanism, it is more flexible than statically-scoped exceptions, because a pointcut is not limited to a particular static scope. On the other hand, AspectJ's around advice does not modify the static type of the methods it wraps.

5. Conclusion

This paper proposed statically scoped exceptions, a new construct for error handling that supports better modularization of error-handling concerns. We have given a static type system for our exception construct, guaranteeing that a well-typed program will not terminate with uncaught exceptions, and ensuring that static scoping for exceptions does not lead to continuation-like behavior. We hope this system will serve as a foundation for practical error-handling systems that combine good modularization of error-handling code with strong reasoning about program errors.

In future work, we intend to extend the system to allow functions to be polymorphic in the exceptions they throw. We also plan an empirical study of uses of exceptions in the ML standard libraries, to better understand both the usability and the benefits of our proposal in practice.

6. Acknowledgements

We thank John Reynolds, Bob Harper, and other members of the POP group for their feedback on our proposal.

7. REFERENCES

[1] Y. Kameyama. Towards logical understanding of delimited continuations. In *Continuations Workshop*, January 2001.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, June 2001.

[3] M. Lippert and C. V. Lopes. A Study on Exception Detection and Handling using Aspect-Oriented Programming. In *Interntional Conference on Software Engineering*, June 2000.

[4] A. Nanevski. A modal calculus for control effects. Submitted, 2004.

[5] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[6] H. Thielecke. Comparing control constructs by typing double-barrelled cps transforms. In *Continuations Workshop*, January 2001.