An Exceptional Programming Language

John Aycock
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4
Phone: +1 403 210 9409, Fax: +1 403 284 4707
Email: aycock@cpsc.ucalgary.ca

Mike Zastre
Department of Computer Science
University of Victoria
P.O. Box 3055
Victoria, B.C., Canada V8W 3P6
Phone: +1 250 721 7220, Fax: +1 250 721 7292
Email: zastre@cs.uvic.ca

***Abstract*— The use of exceptions in programming languages is usually reserved for exceptional conditions. This is a narrow view of exceptions, however. We demonstrate how exceptions can be used to express common programming language constructs, and thus form the basis of a new type of exception-based programming language. We also generalize exceptions so that they may be thrown into a program's *future* execution, not just its past. Implementation techniques for both generalized and traditional exceptions are presented.**

***Index Terms*— Programming languages, exceptions, control flow**

I. Introduction

Exception-handling facilities are available in many programming languages, especially recent ones, and these facilities are used for 'Separating the exceptional structure from the code associated with normal operation' [1, page 192].

Early work on exceptions suggested three possible uses for them [2]:

1) handling the failure of an operation;
2) providing extra, "out-of-band" information about an operation that successfully completed;
3) monitoring an ongoing operation.

Current usage of exceptions in practice is fixated upon the first application: failure. Exceptions are used to signal exceptional failure conditions, but this gives exceptions short shrift.

There is another possible use of exceptions which has not yet been explored, where exceptions can be used for a program's control flow. Instead of separating the exceptional structure from normal code, the exceptional code *is* the normal code.

We show how exceptions can be used to implement control flow, and thus lay the groundwork for a new type of exception-based programming language in Section II. Section III discusses exception implementation issues.

Current exception-handling facilities expect thrown exceptions to be caught by exception handlers already seen during execution of the program; in other words, the program's past. In Section IV we remove this restriction, and generalize exceptions to allow exceptions to be thrown into the program's future execution as well. We consider several possible semantics and implementations of generalized exceptions as well.

II. A Compendium of Constructs

Many common control flow constructs in programming languages can be described using exceptions. Here, we use a C-style pseudocode to illustrate how key constructs can be represented in exception form. For exceptions, our pseudocode uses the familiar `try` and `catch` keywords, and `throw` obviously throws an exception. The `retry` keyword, used in an exception handler, causes the program to re-execute the `try` block from which the exception was raised; these are *retry semantics* [3].

In the examples below, the original control flow construct (without exceptions) appears to the left; the exception-based equivalent is on the right.

A. If-Else Statements

Conditional if-else statements are implemented using exceptions by first evaluating the conditional expression inside a try block, yielding a boolean value. This boolean value is then thrown; catching a true value corresponds to the if-part code, a false value to the else-part.

```
if (x < 123) {
    if-part
} else {
    else-part
}
```

```
try {
    throw x < 123
} catch true {
    if-part
} catch false {
    else-part
}
```

Of course, an if statement without an else clause is trivially represented with an empty else-part.

B. Switch Statements

The design of exception-based switch statements was suggested by Montanaro [4] for the Python programming language, and was what initially started us considering our exception-based language.

As exceptions, switch statements are simply an extension of the if-else statement. The switch expression is evaluated inside a try block, and then thrown. The catch clauses directly correspond to the original case labels.

```
switch (x) {
case 1:
    case1
case 2:
    case2
    ...
case N:
    caseN
}
```

```
try {
    throw x
} catch 1 {
    case1
} catch 2 {
    case2
    ...
} catch N {
    caseN
}
```

Different languages have different semantics with respect to whether or not execution falls through automatically from one case to the next one. The example above does not demonstrate fall-through semantics, but the fall-through effect can be created by code duplication: if $case_1$ falls through onto $case_2$, then the catch handler would contain code for both, as shown below.

```
    ...
} catch 1 {
    case1
    case2
}   ...
```

Breaking out of control flow constructs, like switch statements and loops, is a straightforward application of exceptions. The same idea extends to multi-level break statements (e.g., the Bourne shell) and labeled break statements (e.g., Java). The general form to break out of a construct is:

```
construct {
    code_before
    break construct
    code_after
}
```

```
try {
    code_before
    throw exit_construct
    code_after
} catch exit_construct {
    // do nothing
}
```

C. Do-While Loops

Repeated execution of a loop body can be captured using retry semantics. For a do-while loop, the loop condition is evaluated immediately following the loop body's code; the resulting boolean is thrown. Throwing a false boolean value indicates the end of the loop, and a true value causes the loop body to be iterated by re-executing the try clause with `retry`.

```
do {
    loop-body
} while (x < 123)
```

```
try {
    loop-body
    throw x < 123
} catch true {
    retry
} catch false {
    // do nothing
}
```

D. While Loops

A while loop can be represented as a combination of an if statement and a repeat loop. Indeed, this transformation is done in optimizing compilers to facilitate hoisting code out of loops [5], [6]. For our purposes, this representation of while loops also means that the do-nothing catch clause can be shared between the inner repeat loop and the outer if statement.

```
while (x < 123) {
    loop-body
}
```

```
try {
    throw x < 123
} catch true {
    try {
        loop-body
        throw x < 123
    } catch true {
        retry
    }
} catch false {
    // do nothing
}
```

E. For Loops

A general form of for loop, where a programmer-specified expression is permitted for loop initialization, control, and increment, can be easily changed into a while loop. For illustration, we have also added a `continue` to the loop body; a `continue` statement, in a for loop, transfers control to the increment expression.

```
for (x = 1; x < 123; x = x + 1) {
    code_before
    continue
    code_after
}
```

becomes

```
x = 1
while (x < 123) {
    code_before
    goto increment
    code_after
increment:
    x = x + 1
}
```

In exception form, this while loop is then represented as shown below.

```
x = 1
try {
    throw x < 123
} catch true {
    try {
        try {
            code_before
```

```
            throw increment
            code_after
        } catch increment {
            // do nothing
        }
        x = x + 1
        throw x < 123
    } catch true {
        retry
    }
} catch false {
    // do nothing
}
```

Clearly, with such a heavy reliance on the exception-handling mechanism in our language, the efficiency of exceptions is of major concern.

III. Exception Implementation Issues

A significant obstacle to using exceptions as a common programming construct is their relatively high run-time expense compared to traditional control-flow mechanisms. There can be places where the run-time cost of a thrown exception, such as in a typical implementation of a Java array-loop, is less than the run-time cost of the code controlling the loop [7]. Here the exception is used as a code optimization, transferring control out of a loop, with the optimization's potential profit increasing as the number of loop iterations increases.

However, in this paper we are more interested in exploring techniques for reducing the run-time cost of an *individual* exception, especially since our language makes heavy use of them. Although there has been some effort to minimize the cost of exceptions as much as possible in languages such as C++ [8], the default position of many who implement programming languages can be summarized as:

- Exceptions are rare, and if they are not then they should be made to be rare.
- It is acceptable to decrease the cost of regular code by significantly increasing the cost of exceptions [9, p.17].

Books that provide advice to programmers even go so far as to advise their readers to avoid exceptions as much as possible because of the expense [10]. Approaches towards reducing exception cost take advantage of just-in-time techniques; for example, if an exception and its handler are both local to some method, then a JIT compiler ensures the general-purpose exception-dispatch mechanism is not used for this exception, but rather a less expensive "goto" is instead used [11]. Not all JIT implementers agree with this approach, however; we have observed that thrown exceptions – even those repeatedly thrown from the same site – take more time to be caught with Sun's HotJava JIT than with the older Classic JVM [12].

Before presenting our approach, there remains one more observation. Any technique used to reduce exception-handling time should not be at the expense of code which does *not* use exceptions. Given that exceptions are common in our language, we could relax this restriction by bounding the overhead imposed on code without exceptions. Regardless of the choice, our interest is in ensuring overhead is minimized. To accomplish this we can use a combination of two techniques [12]: *minimize table lookups during stack unwinding* and *eliminate construction of unused exception objects*.

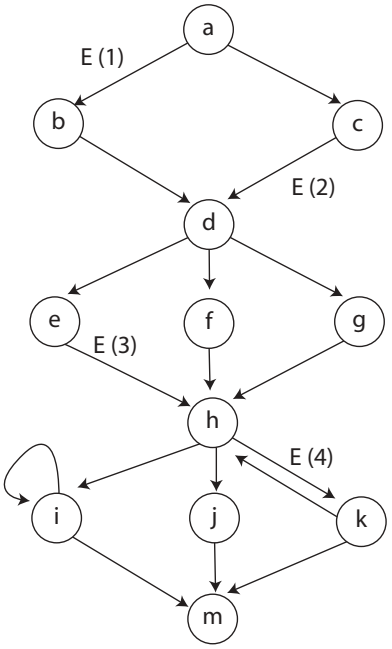

Fig. 1. Callgraph example

A. Minimize table lookups

A general-purpose exception-handling mechanism for a language such as Java is usually implemented with dynamic lookups in mind; that is, at an exception-throw site, the search for a handler proceeds by examining each activation frame on the stack and stops when this unwinding upon finding the first handler. However, we can do better than this by performing a bit of program analysis (static or dynamic) in order to answer the question: Which stack frames should be examined? Consider the callgraph in Figure III-A where labels on edges indicate handlers surrounding the callsite. For example, `a`'s call of `b` is within a handler for exception type `E`; there are four different handlers for `E` in this callgraph. If an instance of `E` is thrown when `m` is active, to which of the four handlers for `E` should control be transferred? If we know that `m` is called by `j`, then clearly we should unwind the stack to `h` and continue our search there; if `h` is called by `f`, then we can unwind to `d`; at that point, we can find the handler by determining `d`'s caller.

A more precise callgraph could result in even fewer table lookups, but there is a tradeoff here between the cost of the analysis producing such a callgraph and the run-time savings from fewer lookups.

B. Unused exception objects

In languages such as Java, throwing an exception results in some exception object being created. We can consider that some data flow exists from the throwsite to the handler because the object may be referenced by the handler. Of course, the key word is "may" as there is no requirement for a handler to use its exception object. However, the actual construction of such objects can require a significant amount of run-time effort (such as the need to construct a stack trace, for example).

If the set of handlers "reachable" from a specific throwsite do not use their exception objects, then there is no need to create such an object at the throwsite. The code emitted for the throwsite can either omit the code needed to construct the object, or equivalently be directed to call a different version of the code needed to begin the process of throwing and handling an exception.

IV. Generalized Exceptions

There is no *a priori* reason why exceptions must be limited to being thrown into a program's past execution. We define *generalized exceptions* to be exceptions which can be thrown into a program's future execution.

But what does it mean to throw an exception into the future? We look at three different semantic models for generalized exceptions:

1) subroutine calls;
2) flag setting;
3) transactions.

We examine these three models in detail in the remainder of this section.

A. Subroutine call model

Subroutines such as functions and procedures are not strictly necessary in a programming language, but do provide an important tool for abstraction and code conciseness. They do not fit well with traditional exception models, however, because a subroutine call is a statement about the future execution of the program.

Enter generalized exceptions. A subroutine call becomes a generalized exception throw, and each subroutine definition a generalized exception catch. A return from a subroutine to its call site is a `resume` statement, which continues program execution from the point where an exception was thrown – these are resumption semantics for exceptions [3].

```
call foo                  general_throw foo
...                       ...

subroutine foo() {        general_catch foo {
    subroutine_body           subroutine_body
    return                    resume
}                         }
```

There are three things to note about this semantic model. First, it implies that multiple generalized exceptions can be active simultaneously, one for each level of call depth in the original program. Second, subroutine arguments and return values, if any, would need to be passed using some separate mechanism, perhaps as attributes of the thrown exception object. Third, to express recursive subroutine calls, simply nesting generalized catch clauses is not sufficient because cycles cannot be represented.

The cycle representation problem can be avoided by making generalized try clauses implicit rather than explicit. In effect, there is an implied `general_try` clause around each generalized exception throw. Associated with this try clause are generalized catch clauses, one for each subroutine that would be visible from the call site in the original code. Overloading could also be implemented with this model: instead of throwing `foo`, for instance, we could throw `foo_int` or `foo_string`, to reflect the type of a subroutine's arguments.

B. Flag setting model

Another semantic model for generalized exceptions is the flag setting model. In this model, the raising of a generalized exception does not alter the control flow, but simply sets a flag noting the thrown exception. Program execution continues normally.

Generalized catch clauses can be attached to any block of code. When execution leaves a block with these catch clauses, and a pending generalized exception is caught, the associated catch clause is executed at that time.

For example, the code below would iterate through an array and output the last array element that matches `key`, but only if at least one matched. (The loop and conditional have been left in their original form for clarity.)

```
i = 0
while (i < length(array)) {
    if (array[i] == key) {
        last = i
        general_throw found
    }
    i = i + 1
} general_catch found {
    print last
}
```

One design issue for the flag setting model is whether or not the same generalized exception, thrown multiple times before encountering a suitable catch clause, is caught multiple times or only once. The above code would only want to see one exception caught, no matter how many times the exception had been thrown.

On the other hand, if N thrown exceptions result in N catches when a matching catch clause is found, we have the basis of a queue data structure. The code below traverses an array and queues up elements for later processing by the catch clause:

```
i = 0
while (i < length(array)) {
    if (needs_processing(array[i])) {
```

```
        general_throw array[i]
    }
    i = i + 1
} general_catch array_element {
    process(array_element)
}
```

The while loop would go through the entire array first, throwing array elements as needed; after the while loop finished, the catch clause would be executed once for each thrown element.

C. Transactional model

The transactional model of generalized exceptions is inspired by database transactions [13]. Like the flag setting model, the transactional model continues normal program execution after a generalized exception is thrown. However, no expression results are committed, and no I/O is performed, until the appropriate generalized catch handler is reached.

Intuitively, the transactional model is akin to "fast forwarding" the program's execution to the catch clause. Unfortunately, the implementation and use of such a model seems fraught with peril. It is not at all clear that a matching catch clause would ever be reached, causing the program to cease normal functioning forever. Further, there are few useful programs which can execute meaningfully without some form of I/O. While it may have intuitive appeal, this model seems mired in practical difficulties, except perhaps in restricted, well-understood contexts.

V. Conclusion

We have presented the basis of a new type of exception-based programming language, where the language primitives consist only of expression evaluation and exceptions. Control flow can be represented simply as an application of the exception mechanism; generalizing exceptions allows them to be thrown into the future as well as the past.

We have only begun to explore the possibilities of exception-based languages. Because of their high demands on exception-handling mechanisms, exception-based languages are an excellent proving ground for work in efficient exception processing, which can be applied to "normal" languages as well.

Acknowledgment

Thanks to Jan Vitek for helpful discussions about these ideas. This work was funded in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Transactions of Software Engineering and Methodology*, vol. 12, no. 2, pp. 191–221, 2003.

[2] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Communications of the ACM*, vol. 18, pp. 683–696, 1975.

[3] P. A. Buhr and W. R. Mok, "Advanced Exception Handling Mechanisms," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 820–836, September 2000.

[4] S. Montanaro, "Re: control structures (was "re: Sins")," Usenet posting to comp.lang.python, Jan. 2000.

[5] A. W. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[6] R. Morgan, *Building an Optimizing Compiler*. Digital Press, 1998.

[7] M. Zastre and R. N. Horspool, "Exploiting Exceptions," *Software: Practice and Experience*, vol. 31, no. 12, pp. 1109–1123, October 2001.

[8] D. Chase, "Implementation of exception handling, Part I," *The Journal of C Language Translation*, vol. 5, no. 4, pp. 229–240, June 1994.

[9] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kaslow, and G. Nelson, "Modula-3 report (revised)," Digital Systems Research Center, Tech. Rep. Technical Report 52, 1989. [Online]. Available: http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-052.html

[10] J. Shirazi, *Java Performance Tuning*. O'Reilly and Associates, Inc., 2000.

[11] S. Lee, B.-S. Yang, S. Kim, S. Park, S.-M. Moon, K. Ebcioğlu, and E. Altman, "Efficient Java exception handling in just-in-time compilation," in *Proceedings of the ACM 2000 conference on Java Grande*, June 2000, pp. 1–8. [Online]. Available: acm-cite/proceedings/plan/337449/p1-lee/

[12] M. Zastre, "The Case for Exception Handling," Ph.D. dissertation, Department of Computer Science, University of Victoria, 2004.

[13] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.