An Axiomatic Treatment of Exception Handling in an Expression-Oriented Language

SHAULA YEMINI
IBM T. J. Watson Research Center

and

DANIEL M. BERRY
University of California at Los Angeles

An axiomatic semantic definition is given of the replacement model of exception handling in an expression-oriented language. These semantics require only two new proof rules for the most general case. An example is given of a program fragment using this model of exception handling, and these rules are used to verify the consistency of the fragment and its specification.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.3 [**Programming Languages**]: Language Constructs—*control structures*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.1 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*control primitives*;

General Terms: Design, Languages, Theory, Verification

Additional Key Words and Phrases: Algol 68, aliasing, axiomatic semantics, exception handling, expression language, program specification, replacement model, side effect

1 INTRODUCTION

This paper presents an axiomatic treatment of exception handling in an expression-oriented language, based on the replacement model [26]. The replacement model, in contrast to other exception handling proposals, supports all the handler responses of

This research was supported in part by the University of California Micro Program, SDC—A Burroughs Company, and NCR.

A preliminary report of this work appeared in the *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January, 1982.

Authors' Addresses: S. Yemini, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A; D. M. Berry, Computer Science Department, 3531 Boelter Hall, University of California, Los Angeles, CA 90024, U.S.A

resumption, termination, retry and exception propagation, within both statements and expressions, in a simple, modular and uniform fashion. The main result presented in this paper is that the semantics of all these handler responses can be captured using a simple axiomatic definition involving only two proof rules in addition to the rules defining the other aspects of the embedding programming language; these rules place no restrictions on allowable handler effects except for those resulting from the normal scope rules. While the replacement model can be exploited most in an expression-oriented language, it is suitable for any block-structured programming language. A syntactic extension and both operational and axiomatic semantic definitions for embedding the replacement model in ALGOL 68 [25] are presented in [26]. The operational semantics are also given in [27].

2 EXCEPTION HANDLING IN THE REPLACEMENT MODEL

Operations, i.e., procedures and functions, provided by a programming language, whether primitive or program-defined, can often be usefully defined only on a subset of the states in their domain of definition. This subset is defined by an assertion that is called the *normal case input assertion* of the operation. An *exception of an operation* is a state in that operation's domain that does not satisfy the normal case input assertion[1]. Examples of exceptions include an empty or ended file in a the read operation, a zero denominator in a division operation, an empty stack in a pop operation.

A module construct is supposed to encapsulate data objects and their operations. In order to enforce this encapsulation, the invoker of an operation should be *notified* of the detection of an exception, in order to allow the *invoker* to determine an appropriate action. This notification pattern also supports modular decomposition: the detection of an exception is done by the operation supplied by the module, but the response, which is application specific and therefore cannot be determined in the module, is left to the invoker. Notifying the invoker of an operation that an exception has been detected is called *signalling* the exception, and the operation is called the *signaller* of its exceptions. The program supplied by the invoker for responding to the detection of an exception is called the *handler*.

The replacement model adopts an expression-oriented view: a program is considered a composite expression; exceptions correspond to subexpressions that cannot be fully computed by their signaller. Procedures and functions are required to declare the identifier and data type of each exception they can signal. Any generalized expression, i.e., closed construct such as a block, loop or conditional, may be made a signaller by declaring its exceptions.

Exception handling in the replacement model consists of computing replacement effects and returning replacement values for either

(1) the signalling of the exception, after which the signaller may resume, or
(2) the invocation of the signaller of the exception.

[1] Since exceptions can be propagated, an exception of one operation may also be a result of another operation, used in its implementation, that does not have its normal case input assertion satisfied.

In any case, signalling is effected by calling the handler as a procedure. Therefore, Alternative 1, called *resumption*, is achieved by doing a normal return from the handler as a result of either reaching the end of the handler text or by reaching an **exit** completer. The value of the expression just preceding the end of the text or the completer is returned as the resulption value. To achieve Alternative 2, called *replacing*, one must arrange that a value is returned not to the signaller, but to the singaller's invoker. A **replace** completer is provided just for that purpose. The value of the expression just preceding the **replace** is returned to the signaller's invoker as a replacement for the value that the signaller would have returned in the normal case. Consistent with these alternatives, the type of an exception (See examples below for more details.) includes the types of the resumption and the replacing values, and the type of the replacement value must be identical to the return type of the signaller.

In many exception handling proposals, unhandled exceptions are automatically propagated along the chain of invokers. That is, given an exception *e*, the run-time system searches for the first handler named *e* along the chain of returns; the handlers are thus dynamically bound to the signallings of exceptions. This prevents the compiler from determining at least the identifier denoting the handler for any signaller and forces the keeping of identifier strings, or some lexically generated encoding thereof, in the run-time data. As discussed in [27], this binding can violate information hiding in that it cannot be assured that the statically enclosing handler will field the signalling, as it may be that a like-named handler inserted by some unknown caller for a different purpose will field the signalling. Accordingly, the replacement model opts for a static determination of at least the identifier denoting the handler for any signaller. The impact of this decision is that each signaller must declare which exceptions it can signal. The types of the parameters, the resume and the replace values must be specified as part of the type of an exception. Each invoker of any signaller must provide in some context statically enclosing the invoker, a handler for each exception declared with the signaller, and the handler must agree as to the types of the parameters and the resume and replace values. As a result of these rules, no signalled exception can go unhandled (and thus, there is even no need for a rule about what to do with unhandled signallings), at least the identifier denoting the handler (there may be handler variables) for each exception is known at compile time, the interface between all signallings and all fielding handlers can be verified at compile time via their common exception type. In addition, all propagation must be done explicitly; that is, the handler must explicitly signal an exception known to its signaller's invoker.

Observe that this scheme easily supports default exception handlers for all language-defined exceptions. It suffices to declare in the standard prelude block that is assumed to surround all programs, a handler for all language-defined exceptions. These will be used unless the programmer hides them with more locally declared handlers

The addition of **replace** suffices to support all of the various handler responses considered useful in previous exception handling proposals, e.g., [1, 10, 16, 17, 14, 19],

(1) replacing the immediate signalling of the exception amounts to resumption;
(2) replacing the signaller invocation amounts to termination;
(3) signalling an exception within the handler amounts to propagation;
(4) having a new invocation of the signaller within the handler, to replace the signalling invocation, amounts to retrying; and

(5) signalling an exception of a closed construct in the handler for the original exception, with the handler for this propagated exception replacing the invocation of the construct, amounts to termination of the construct as a result of the original exception.

None of the proposals of which the authors are aware can support all of these handler responses in as straightforward a fashion.

As an example consider the procedure `convert`, which takes an array-of-integers variable as a parameter, and returns the string formed by concatenating the characters represented by the integers in the array. Besides specifying the types of the parameters and the returned value, the heading of the routine text bound to `convert` also says that procedure signals the exception `badcode`. This exception is signalled when an integer for which there is no corresponding character is found. The procedure is written in ALGOL 68, modified to include the extensions required for the proposed exception handling mechanism.

```
proc convert=(ref[]int code)string
     signals(exc(int)(char,string)badcode):
begin
   string s:="";
   for i from lwb code to upb code
      do
         s:=s+repr code[i]
         # + is string concatenation; thus the above #
         # appends to s the char represented by code[i] #
      od
   on nochar=(char,char):
      # when repr signals nochar, #
      badcode(i) replace
      # signal badcode #
   no;
   s
end
```

`Convert` invokes the operator **repr**, which returns the character represented by an integer if one exists. **Repr** is assumed to have been modified here to signal the exception `nochar` if its argument does not represent any character. The handler in `convert` handles `nochar` by signalling `convert`'s own exception `badcode`. Note the strict level-by-level, explicit propagation required by the static binding rule. A signalling of an exception has the same syntactic form as a call. The value returned by a handler for `badcode` replaces the value that would have been returned by **repr**, had **repr** not signalled an exception.

`On` e = h **no** postfixed to a closed construct (here a loop) designates the handler expression h for all signallings of the exception e in that construct. In ALGOL 68, a routine text is headed by the list of the types and identifiers of its formal parameters, and its return type. The same is done for handler texts. In general, h can be any expression yielding a handler. In the following discussion, only handler texts are used.

Exc is the type constructor for exceptions. Exceptions are typed by the type of their parameters and their two return types. An exception and any handler designated for it have two return types, e.g., (**char, string**) for badcode, since a handler may either resume or replace the signaller invocation, and each case may require the handler to deliver a value of different type. By convention, the first type listed in the return type pair is that for resumption, the second that for replacement. For example, the type of **repr** is

```
proc(int)char
  signals(exc(char,char)nochar) .
```

The identifier and data type of badcode are declared in convert's heading, and are considered part of convert's data type. Including this information about each exception convert can signal enables a compiler to check that handlers of the proper types have been designated in the static scope of each invocation of convert, as required in the replacement model. Exceptions are not passed up the calling chain unless explicitly propagated by a handler.

The following examples demonstrate how the various handler responses are supported in the replacement model. In each of these examples, the handler is provided to a loop statically enclosing the invoker of the procedure that can raise the exception.

(1) *Resumption*: supply a "?" as a replacement for the **char** corresponding to badcode. Convert then resumes. The result is therefore a string in which the characters corresponding to unconvertible codes are "?".

```
do ... print(convert(nums)) ... od
on badcode=(int i)(char,string):"?"no
```

If there is never to be a replacement in a given handler, then there is no need for a **replace** in that handler; the replacement value type is then chosen only to match that of the declaration of the exception that the handler handles.

(2) *Termination of the signaller*: supply the empty string as a replacement for the **string** returned by convert. Note that the replace type of an exception is always the type returned by that exception's signaller.

```
do .. print(convert(nums)) ... od
on badcode=(int i)(char,string):""replace no
```

Since **replace** completes the expression whose value is intended to serve as a replacement for the value returned by the signaller, its semantics are as follows. Return the value to the instruction following the invocation of the signaller which signalled the exception for which this handler was designated. This is the instruction referred to by the return label for the signaller invocation, and therefore, its location in the activation record for the signaller can be statically determined.

(3) *Retry*: retry after changing the badcode to a zero. The **string** returned by the new invocation of convert is returned as a replacement for the value of the initial invocation.

```
do ... print(convert(nums)) ... od
on badcode=(int i)(char,string):
   begin
   nums[i]:=0;
   convert(nums) replace
   # replace the signalling invocation of convert
   with another invocation of convert #
   end
no
```

To support replacing subexpressions at any level uniformly, without coupling the effects made by a handler to the flow control required after the handling is completed [14, 17], any closed construct in the embedding programming language is allowed to become a signaller of exceptions. This is done by having the construct declare its exceptions in a **signals** clause following the opening bracket of the construct. Thus for example, a block, loop or conditional can become a signaller of exceptions. The rules for resumption and replacement apply uniformly to procedure signallers and any closed construct signallers.

The following example demonstrates two of the handler responses supported by the replacement model, termination of a closed construct containing the invocation signalling an exception [14, 17] and exception propagation.

In order to obtain termination of the loop after `badcode` has been detected, the loop is made a signaller of a parameterless exception called `finish`. This is done by attaching a **signals** clause declaring `finish` after the **do**. `Finish` is signalled in the handler for `badcode`, i.e., it is the propagation of `badcode`. When the handler for `finish` replaces the invocation of the signaller of `finish`, i.e., the loop invocation, the loop is terminated as required.

(4) *Termination of a closed construct and exception propagation*:

```
begin
   do signals((char,void)finish)
   begin ... print(convert(nums)) ... end
   on badcode=(int i)(char,string):finish no
   od
end
on finish=(char,void):skip replace no
   # when finish is raised, replace its signaller's
   invocation by the void value yielded by skip #
```

Skip in ALGOL 68 yields an undefined value of whatever type is required by the context.

For further details and other examples of the replacement model see [26] and [27].

3 OTHER AXIOMATIC SEMANTICS FOR EXCEPTION HANDLING

Other axiomatic treatments of exception handling that the authors are aware of include those of Cristian [6, 7], Cocco and Dulli [5], Levin [16], and Luckham and Polak [18]

The first three axiom systems are for exception handling schemes designed by their respective authors, which, like ours, are designed with the explicit intent of being verifiable. The last axiom system is for a previously designed language, Ada™[2] [2, 14]. There was some attempt to make Ada verifiable, but practicalities in language design prevented complete adherence to this goal. All are of exception handling schemes that are more restricted than ours. Most notably, all assume a non-expression oriented language. On top of that, in the Cocco-Dulli proposal, an exit-raised exception handler cannot resume its signaller and must replace its signaller. Also, in the Levin proposal, a handler cannot replace its signaller and must resume its signaller.

In each of these axiom systems, the logic used for the underlying language prohibits expressions with side effects and procedures with aliased parameters because either the axioms are based on Hoare's [12, 13] or use a weakest precondition formulation [8]. It is very helpful in this respect that Ada outlaws expressions with side-effects and aliased parameters by making them *erroneous*, i.e., non-portable. On the other hand, the use of a weakest precondition formulation allows Cristian a direct attack on total correctness. The other and our formulations deal only with partial correctness.

It is interesting to note that for Ada, Luckham and Polak had to severly restrict the propagation of exceptions that not handled locally. In Ada, exceptions can be propagated arbitrarily along the invocation chain. In other words, there is dynamic binding of handlers as the run-time system searches for the first occurrence of the correctly named handler along the invocation chain. This dynamic binding causes major difficulties to axiomatization. So, Luckham and Polak outlaw automatic, arbitrary propagation. Rather, exceptions must be explicitly propagated to a handler that must be provided by the invoker. Whether such a handler is provided can be checked statically. This restriction effectively makes the binding of the handler static and avoids the problems of Ada's dynamic binding of handlers. Thus, they have effectively restricted the Ada exception handling to be more like ours.

The work in algebraic semantics of exception handling, e.g. [9], tends to ignore side effects simply because dealing with side effects would require including the state as an explicit argument to every function.

4 AXIOMATIC SEMANTICS

The present axiomatic treatment of exception handling follows the axiomatic approach proposed by Schwartz for ALGOL 68 in [24]. A more accessible subset of this system is presented in [23]. This approach is one of those suitable for axiomatizing the replacement mechanism, since it contains rules of inference for procedures that allow parameters of arbitrary types including procedures, and contains no restrictions on side effects in expressions. It should be noted that side effects occur naturally in exception handling, since any effect made by a handler, including the printing of an error message, is a side effect of invoking the signaller.

Other suitable approaches are those of Pritchard [22], Kowaltowski [15], and H.-J. Boehm [3]. The approaches used by Kowaltowski, Pritchard, and Schwartz (and thus this paper) are to extend Hoare logic. Their axiom systems deal with expressions with side effects by considering them as commands that return values and generating for each

[2] Ada is a trademark of the U.S. Department of Defense.

subcommand that can return a value a variable that denotes its value. Thus for the command C, in the sentence "$P\{C\}Q \wedge value=v$", the input and output assertions P and Q express the side-effects and v is the value returned. Their axiom systems deal with aliasing by severing the direct mapping between program identifiers and values, that is prevalent in Hoare-style axioms, in favor of asserting properties of explicitly stated environment and storage maps. Thus to claim that the identifier x has the value v, one must assert that x is bound to location l and that location l has the value v. The difficulties with these axiom systems are that every subcommand has to have a value variable and the association between an identifier and its value always involves at least two conjuncts. These difficulties prove to be quite explosive in the size of assertions. Boehm's approach, on the other hand, is to build a new logic in which all constructs normally have side effects and return a value and to deal with side effects and returned values separately. For each construct, two rules are given, one stating its changes to contents of variables and the other stating the value it returns. By treating variables (locations) as objects distinct from their values and talking about each whenever necessary, aliasing can be handled. While ultimately, the two Boehm rules for a construct must state the same as a single Kowaltowski, Pritchard, or Schwartz rule for the same construct, each Boehm rule is simpler than the Kowaltowski, Pritchard, or Schwartz rule, and some would prefer the decomposition of concerns.

Sentences in the extended logic used in [24] have the form $N/P\{s\}Q \wedge \mathtt{a}=v$. Here, s is an expression or statement. $\mathtt{a}$ represents the value yielded by s, which is **empty** if s is a statement. P and $Q \wedge \mathtt{a}=v$ are the input and output assertions, respectively. N is a *NESTL*, which maps the set of all identifiers and derived types known at each point in the program to their declared types and, in general, provides the static properties of the program necessary for the proof. As such, it is reminiscent of the **NEST** metanotion of the Revised Algol 68 Report25 . The N distributes over all parts of the sentence; thus it applies both to the pre- and the postcondition. The axiomatization assumes that programs to be verified are compile-time correct, i.e., all type checking and type equivalencing have been done, and all grammar-imposed restrictions have been met.

The above sentence is to be read as "if P is true with respect to N, and if the evaluation of s halts, then Q is true with respect to N after evaluating s, and the value yielded by s is v".

In [24], a unique special variable $\mathtt{a}_j$ is associated with each occurrence of each expression layer in the program, representing the value yielded by evaluating that expression. In order to be able to maintain a normal form, $\mathtt{a}=v$ for assertions about $\mathtt{a}_j$s when pushing assertions through successive expression layers, the domain of formal values was enlarged to include *conditional values* of the form $(P \mid v)$ (intuitively, "if P then v"), and $v_1 \oplus v_2$ (intuitively, "v_1 or v_2"), where P is a predicate in the underlying logic, and v, v_1, v_2 are themselves formal values.

The axiomatic definition of exception handling requires

(1) a way to specify a signaller together with the exceptions it signals, independently of any specific choice of handlers, and a definition of correctness of the signaller with respect to the specification;
(2) a way to specify the effect of a handler; and
(3) an adaptation-style proof rule [13] that combines the specification of a signaller, together with the specifications of the handlers designated for an invocation of that signaller, in order to derive the effect of that invocation.

4.1 Specifying Signallers and Their Exception

Let s be a signaller of one exception e with a formal parameter vector $\boldsymbol{x}$. The following notation is used for an input/output specification of s.

$$N/\{P, Q \wedge \exists = v, e(\boldsymbol{x}) \langle E(\boldsymbol{x}), R(\boldsymbol{x}) \wedge \exists = u \rangle\}$$

P is the input assertion, $Q \wedge \exists = v$ is the normal case output assertion which is satisfied if no exceptions are signalled. E is the exception *condition* corresponding to the exception e, describing the state in which e is signalled. $R \wedge \exists = u$ is the *resumption condition*, which is required to be satisfied by a handler for e before resumption, in order to ensure that the normal case output assertion, $Q \wedge \exists = v$, is satisfied if and when s halts. The above specification states that, with respect to N, if the input state satisfies P, and if the execution of the specified construct halts, then either $Q \wedge \exists = v$ holds or the exception e is signalled and the state then satisfies E. $R \wedge \exists = u$ must be satisfied before resumption. This notation reduces to $\{P, Q \wedge \exists = v\}$ when there are no exceptions, but in that case, the conventional notation, $P\{s\}Q \wedge \exists = v$, is used.

One says that s is *partially correct* with respect to the specification above, or

$$s \text{ PC wrt } N/\{P, Q \wedge \exists = v, e(\boldsymbol{x}) \langle E(\boldsymbol{x}), R(\boldsymbol{x}) \wedge \exists = u \rangle\}$$

if and only if

$$\frac{\text{for any handler } h \text{ for } e,\ N/E(\boldsymbol{x})\{h(\boldsymbol{x})\}R(\boldsymbol{x}) \wedge \exists = u}{N/P\{s\}Q \wedge \exists = v}$$

that is, if and only if the assumption $N/E\{h\}R \wedge \exists = u$ for any handler h for e enables proving that $N/P\{s\}Q \wedge \exists = v$. This enables using the procedure proof rules to push assertions through a signalling in the process of verification, even though the handler is not known within the signaller body. The rule for an invocation of a signaller ensures that all handlers for the exception do indeed satisfy the resumption condition before resumption.

In the general case, a specification may include several $\{P, Q \wedge \exists = v\}$ pairs, each of which may have several associated exceptions. The extension to this case is straightforward; see [26]. In fact, it is to be able to express multiple $\{P, Q \wedge \exists = v\}$ pairs that this new notation is introduced.

4.2 Specifying Handler Effects

Since resumption is obtained by the same mechanism as procedure calls, and termination of a construct enclosing an invocation is obtained by simply generalizing the notion of a signaller, only one rule is needed in addition to the rules in [24] in order to specify handler semantics. This rule is for the completer **replace**.

4.2.1 *Rule for* **Replace**. **Replace** completes an expression in a handler body. The value of this expression is yielded at the program location to which control is transferred by **replace**. This location is that denoted by the return label of the invocation for which the handler was designated.

Replace preserves both the state and the returned value of its expression; however, control is transferred to the label determined by the invoker of the signaller. The interpretation of this label therefore has to be adapted to the context of each invocation for which the handler is designated, as can be seen in the three rules for invocations given in Section 4.3. Because control goes somewhere else, following the **replace**, anything may be assumed. Therefore, *false* is taken as **replace**'s postcondition. The rule for **replace** is

[REPLACE:]
$N/P\{e\}\,Q\wedge\beth=v$

$N/P\{e\ \textbf{replace}\}\,false\wedge(replace{:}Q\wedge\beth=v)$.

This rule is a natural restatement of the rule by Clint and Hoare for gotos [4]. The notation here is borrowed from that of temporal logic [21], which uses assertions of the form *label:predicate* to specify that *predicate* is to hold at the specified *label* in the program. The "*replace*:" in the consequent is an uninterpreted label, which is interpreted (adapted) in the rule for the site of an invocation, for each specific invocation.

4.3 Rules for Signaller Invocation

These rules combine the independent specifications of a signaller and of the handlers designated for a particular invocation of this signaller in order to derive the effect of the invocation.

4.3.1 *Procedure and Handler are Identifiers.* First, consider the special case situation in which the expressions yielding the signaller and the handler are both identifiers, p and h respectively, bound to values of the corresponding types, as opposed to arbitrary expressions. This is probably the most common case, and is most likely to be supported in programming languages that are not expression oriented. For brevity, it is assumed that p may signal only one exception *ex*.

[SIMPLE INVOCATION:]
1. $N/T\wedge INV\{\text{COLLAT}(e_1,....,e_n)\}\,P\wedge INV\wedge\beth_e=\boldsymbol{x}$
2. $p(\boldsymbol{x})$ PC wrt $N/\{P, Q\wedge\beth_0=v, ex(\boldsymbol{z})\langle E(\boldsymbol{z}), R(\boldsymbol{z})\wedge\beth_1=u\rangle\}$
3. $N/E(\boldsymbol{z})\wedge INV\{h(\boldsymbol{z})\}\,(R(\boldsymbol{z})\wedge INV\wedge\beth_1=u)\vee(replace{:}S\wedge INV\wedge\beth_2=w)$

$N/T\wedge INV\{p(e_1,....,e_n)\ \textbf{on}\ ex=h\ \textbf{no}\}\,(Q\vee S)\wedge INV\wedge\beth=(Q\mid v)\oplus(S\mid w)$

Premise 1 accounts for the effects of the collateral elaboration (evaluation in an unspecified order) of the actual parameter expressions $e_1, \ldots, e_n$. This elaboration may have side effects. Rules for collateral elaboration can be found in [24]. Basically this rule says that the $(e_1,....,e_n)$ has the value $\boldsymbol{x}$ if and only if it has that value under *any* order of evaluation of the constituent atomic subexpressions. This proviso is required because the Revised Report [25] says that the yield of any collateral construct whose value depends on the order of evaluation of constituent inseparable subexpressions is undefined. The precondition of Premise 1 is the precondition of the whole invocation

given in the conclusion of the rule.

Premise 2 is the specification of p in the form described earlier. Here, under input condition P, the call $p(x)$ satisfies the normal case output assertion $Q \wedge \mathfrak{a}_0 = v$ or it signals the exception $ex(z)$ under condition $E(z)$. It is required that all handlers for ex satisfy the resumption condition $R(z) \wedge \mathfrak{a}_1 = u$.

Premise 3 is the specification of the handler h with formal parameters vector z. The input assertion for h includes the exception condition $E(z)$ for its designated exception. The handler may cause either resumption or replacement of the signaller. In the first case, indicated by encountering an expression f followed by **end** or **no**, it must be shown at each such **end** or **no** that the handler establishes the resumption condition $R(z) \wedge \mathfrak{a}_1 = u$, i.e., that at all of these points $R(x)$ holds and that u is the value of f. In the latter case, indicated by an expression followed by **replace**, the replacement condition $S \wedge \mathfrak{a}_2 = w$ is determined (INV is explained below). There can also be more than one **replace** in the handler or a conditional choice between resumption and termination. Thus in general, the output assertion of a handler contains both an assertion for the resumption case and an assertion for each replacement case.

The conclusion of the rule states that if all the premises hold before the invocation, then after the invocation, the normal case output assertion or the handler replacement output assertion holds.

Since the signaller, and the handlers for a given invocation of it, are likely to have different accessing environments, an invocation may have side effects on objects in the environment containing the invocation, which are *not* accessible to the signaller (thus, no INV in Premise 2). Let INV be an assertion about objects in this environment which holds true before the invocation, and is preserved by the elaboration of the actual parameter expressions. If INV is preserved by all handlers designated for that invocation for all the exceptions that may be signalled by the invoked signaller, it is concluded that INV remains true after the signaller has been completed.

4.3.2 *Signaller is Closed Construct.* The syntactic rules for designating handlers allow an **on** clause designating handlers to be postfixed to a closed construct, thus designating the handlers in the **on** clause for all invocations within the construct; this is consistent with the static binding of handlers to signallings. In the proof rules, it is assumed that all handler designations have been copied to immediately postfix all of the associated invocations. The effect of this copying transformation can be determined statically. Note that this copying is only conceptual. In the actual program, there needs to be only one occurrence of the **on** clause.

The rule for a signaller which is a closed construct, e.g. a block, is a special case of the rule for a signaller invocation, in which there are no parameter expressions and INV is identically true.

[CLOSED CONSTRUCT INVOCATION:]

1. s PC wrt $N/\{P, Q \wedge \mathfrak{a}_0 = v, ex(z)\langle E(z), R(z) \wedge \mathfrak{a}_1 = u\rangle\}$
2. $N/E(z)\{h(z)\}(R(z) \wedge \mathfrak{a}_1 = u) \vee (replace : S \wedge \mathfrak{a}_2 = w)$

$N/P\{s$ **on** $ex = h$ **no**$\}(Q \vee S) \wedge \mathfrak{a} = (Q \mid v) \oplus (S \mid w)$

4.3.3 *General Case.* In the general case, in an expression-oriented language, both the invoked signaller and the designated handlers can be yielded by arbitrary expressions. In this case, the proof rule needs to consider all the signaller values that could possibly be yielded by these expressions. Each of these signallers may be partially correct with respect to different input, output, and exception specifications. However, they all have exceptions with identical identifiers and data types, since these are considered part of the data type of the signaller. In order to not complicate the rule further, it is assumed that the invoked signaller may signal only one exception *ex*.

[INVOCATION:]

1. $N/T \wedge INV\{$
   $\quad \text{COLLAT}(e_p, e_1, \ldots, e_n, e_h)$
   $\}\left(\bigvee_{i=1}^{m}(P_i \wedge \mathfrak{a}_p = r_i)\right) \wedge \left(\bigvee_{j=1}^{k}(P'_j \wedge \mathfrak{a}_h = h_j)\right) \wedge INV \wedge \mathfrak{a}_e = \boldsymbol{x}$

2. $\forall i, 1 \le i \le m, r_i(\boldsymbol{x})$ PC wrt $N/\{P_i, Q_i \wedge \mathfrak{a}_i = v_i, ex(\boldsymbol{z})\langle E(\boldsymbol{z})_i, R(\boldsymbol{z})_i \wedge \mathfrak{a}_1 = u_i\rangle\}$

3. $\forall i, j, 1 \le i \le m, 1 \le j \le k,$
   $\left(N/E(\boldsymbol{z})_i \wedge INV\{\right.$
   $\quad h_j(\boldsymbol{z})$
   $\left.\}(R(\boldsymbol{z})_i \wedge INV \wedge \mathfrak{a}_1 = u_i) \vee (replace{:}S_{ij} \wedge INV \wedge \mathfrak{a}_i = w_{ij})\right)$

$N/T \wedge INV\{$
$\quad e_p(e_1, \ldots, e_n)$ **on** $ex = e_h$ **no**
$\}INV \wedge \left(\left(\left(\bigvee_{i=1}^{m} Q_i\right) \wedge \mathfrak{a} = (Q_1 \mid v_1) \oplus \cdots \oplus (Q_m \mid v_m)\right) \vee \right.$
$\qquad \left.\left(\left(\bigvee_{i=1}^{m}\bigvee_{j=1}^{k} S_{ij}\right) \wedge \mathfrak{a} = (S_{11} \mid w_{11}) \oplus \cdots \oplus (S_{mk} \mid w_{mk})\right)\right)$

Premise 1 accounts for the side effects of the collateral evaluation of the expressions yielding the invoked procedure value, the handler. All are potentially conditional values. However, for both the invoked procedure and the designated handlers, it is necessary to distinguish each of the individual values comprising the conditional value in order to examine each individual specification. Thus, assuming that the procedure expression e_p may yield any one of the routine values r_i, instead of writing $\mathfrak{a}_p = p$ where p is a conditional value involving m different $(P_i \mid r_i)$s, p is separated into m different $(P_i \wedge \mathfrak{a} = r_i)$s. The same is done for $\mathfrak{a}_h$. $\mathfrak{a}_e = \boldsymbol{x}$ is shorthand for $\mathfrak{a}_{e_i} = x_i$, for $i = 1, \ldots, n$, the actual conditional parameter values.

Premise 2 includes the specifications of all the possible signallers that could yielded by e_p.

Premise 3 includes the specifications of all the possible handler values that could be yielded by e_h.

5 APPLICATION OF RULES

The following outlines the use of these proof rules in proving the correctness of `convert`. Since `convert` contains an invocation of a signaller, this example demonstrates both proving correctness with respect to a specification of the form introduced above and applying the rule for an invocation.

In order to keep the notation less cluttered, the NESTLs are omitted. It is assumed that **repr** (n) has been proved correct with respect to the following specifications.

$$\{P_{\mathbf{repr}(n)}, Q_{\mathbf{repr}(n)}, \mathtt{nochar}\langle E_{\mathtt{nochar}}, R_{\mathtt{nochar}}\rangle\}$$

where

(1) $P_{\mathbf{repr}(n)} \equiv \mathbf{true}$
(2) $Q_{\mathbf{repr}(n)} \equiv cl \le n \le ch \wedge \text{ɔ} = charrep(n)$
(3) $E_{\mathtt{nochar}} \equiv n > ch \vee n < cl$
(4) $R_{\mathtt{nochar}} \equiv Q_{\mathbf{repr}(n)}$

In the above, line-by-line, note that

(1) **repr** assumes nothing about its argument,
(2) **repr** returns the character represented by its argument when its argument is in the range $[cl, ch]$,
(3) when **repr** signals `nochar`, its argument is out of range, and
(4) in order to insure that **repr**(n) satisfies $Q_{\mathbf{repr}(n)}$, a handler must satisfy $R_{\mathbf{repr}(n)}$ before resuming **repr**(n).

It is assumed that the mapping *charrep* has been defined appropriately. The rules also assume that all type checking has already been performed. $R_{\mathtt{nochar}}$ in effect specifies that resumption of **repr** after `nochar` has been signalled cannot possibly lead to **repr** satisfying its normal case output assertion, since n is a by-value parameter.

The specification of `convert` makes use of the ALGOL 68 ascription relationship, which associates identifiers with the values to which they are bound. Since `code` is a variable identifier, it is bound to a location of an array of integers $d_{\mathtt{code}}$. The contents of this location are obtained by the mapping τ, and are denoted here by $v_{\mathtt{code}}$. (*Ascribed* and τ are close to the notions of environment and store in denotational semantics).

The following specification of `convert` is assumed.

$$\{P_{\mathtt{convert(code)}}, Q_{\mathtt{convert(code)}}, \mathtt{badcode(i)}\langle E_{\mathtt{badcode(i)}}, R_{\mathtt{badcode(i)}}\rangle\}$$

where

(1) $P_{\mathtt{convert(code)}} \equiv ascribed(\mathtt{code}, d_{\mathtt{code}}) \wedge \tau(d_{\mathtt{code}}) = v_{\mathtt{code}}$
(2) $Q_{\mathtt{convert(code)}} \equiv P_{\mathtt{convert(code)}} \wedge \tau(d_s) =$

$$\underset{j=\mathbf{lwb}\ \mathtt{code}}{\overset{\mathbf{upb}\ \mathtt{code}}{+}} \Big((cl \le v_{\mathtt{code}}(j) \le ch \mid charrep(v_{\mathtt{code}}(j))) \oplus (cl > v_{\mathtt{code}}(j) \vee v_{\mathtt{code}}(j) > ch \mid replchar)\Big) \wedge \text{ɔ} = d_s$$

(3) $E_{\mathtt{badcode(i)}} \equiv P_{\mathtt{convert(code)}} \wedge (cl > v_{\mathtt{code}}(\mathtt{i}) \vee v_{\mathtt{code}}(\mathtt{i}) > ch)$

(4) $R_{\texttt{badcode(i)}} \equiv E_{\texttt{badcode(i)}} \wedge \mathfrak{z} = replchar$

In the above, line-by-line, note that

(1) it is assumed that $v_{\texttt{code}}$ is the value at the location bound to `convert`'s argument,
(2) when `convert` terminates, the argument is unchanged, and the return value is the location of the result of the successive concatenation of character representations and replacement characters, corresponding to the cases of character and noncharacter codes, respectively,
(3) when `badcode(i)` is signalled an unrepresentable code, `i`, has been encountered, and
(4) the handler must provide a replacement character in order to enable `convert(code)` to satisfy $Q_{\texttt{convert(code)}}$.

It is useful to define the function $string(l,u)$ as

$$string(l,u) \equiv \overset{u}{+}_{j=l} \big((cl \le v_{\texttt{code}}(j) \le ch \mid charrep(v_{\texttt{code}}(j))) \oplus (cl > v_{\texttt{code}}(j) \vee v_{\texttt{code}}(j) > ch \mid replchar)\big).$$

That is, $string(l,u)$ is the string resulting from the successive concatenation of the corresponding character representation for character codes of `code`, and *replchar* for non-character codes of `code`.

The major step in the proof of correctness of `convert` with respect to its specification is proving

$P_{\texttt{convert(code)}} \wedge ascribed(\mathrm{s}, d_{\mathrm{s}}) \wedge \tau(d_{\mathrm{s}}) = \texttt{""}$ {

```
for i from lwb code to upb code do
    s:=s+repr code[i]
od
on nochar=(char,char):
    badcode(i) replace
no
```

} $P_{\texttt{convert(code)}} \wedge ascribed(\mathrm{s}, d_{\mathrm{s}}) \wedge \tau(d_{\mathrm{s}}) = string(\texttt{lwb code}, \texttt{upb code}) \wedge \mathfrak{z} = \texttt{empty}$.

This proof requires using the rule for a loop. The following notation for intervals on the integers is used.

$[l,u] = \{j \mid l \le j \le u, j \in INT\}$
$[l,u) = \{j \mid l \le j < u, j \in INT\}$

The needed rule for a loop is

[LOOP:]
1. $N/B \supset \Im([])$
2. $N/i' \in [l,u] \wedge \Im([l,i')) \{body\} \Im([l,i'])$

$N/B\{\textbf{for } i \textbf{ from } l \textbf{ to } u \textbf{ do } body \textbf{ od}\}\Im([l,u]) \wedge \text{ɔ} = \textbf{empty}$.

$\Im$ in the above is the loop invariant. Let

$$S \equiv P_{\texttt{convert(code)}} \wedge ascribed(\texttt{s}, d_{\texttt{s}})$$
$$B \equiv S \wedge \tau(\texttt{s}) = \texttt{""},$$

and

$$\Im([l,u]) \equiv S \wedge \tau(d_{\texttt{s}}) = string(l,u).$$

The proof of Premise 1 of ***LOOP*** is immediate. In order to show Premise 2 of ***LOOP***, the inductive step, the rule for an invocation of a signaller is needed in order to obtain the effect of **repr**, given the supplied handler.

It is desired to show that

```
([lwb code,i'){
    repr code [i]
    on nochar=(char,char):
       badcode(i) replace
    no
```

$$\}\Im([\textbf{lwb } \texttt{code},\texttt{i}']) \wedge \text{ɔ} = \overset{i'}{\underset{j=\textbf{lwb code}}{+}} \Big((cl \le v_{\texttt{code}}(j) \le ch \mid charrep(v_{\texttt{code}}(j))) \oplus (cl > v_{\texttt{code}}(j) \vee v_{\texttt{code}}(j) > ch \mid replchar) \Big)$$

The rule ***INVOCATION*** is used with $T \equiv \Im([\textbf{lwb } \texttt{code},\texttt{i}'])$. Since there are no side effects in evaluating `code[i]`, the result for Premise 1 of ***INVOCATION*** is

1. $N/\Im([\textbf{lwb } \texttt{code},\texttt{i}'])\{\texttt{code[i]}\}\Im([\textbf{lwb } \texttt{code},\texttt{i}']) \wedge \text{ɔ} = v_{\texttt{code}}(\texttt{i})$.

For Premise 2 of ***INVOCATION***, it is assumed that **repr** has been proved correct with respect to the given specification. The output assertion of Premise 1 trivially implies the input assertion of **repr**.

For Premise 3 of ***INVOCATION***, it is necessary to get the characterization of the handler for `nochar`. This handler propagates **repr**'s exception `nochar` as the exception `badcode` of `convert`. In proving the correctness of a signaller (here `convert`), the definition of partial correctness allows assuming that any handler for an exception signalled in the signaller's body is partially correct with respect to that exception's specified exception and resumption conditions. Thus, if it can be shown that just before the signalling of `badcode(i)`, the exception condition $E_{\texttt{badcode(i)}}$ holds, it may be

assumed that immediately after the signalling, the corresponding resumption condition $R_{\texttt{badcode(i)}}$ holds. Since for `code[i]`, $E_{\texttt{nochar}} \supset E_{\texttt{badcode(i)}}$, it may be assumed that $R_{\texttt{badcode(i)}}$ holds after the signalling of `badcode`. Thus,

3. $E_{\texttt{nochar}}$ {
   (char, char) : badcode(i) replace
   } *replace* : $P_{\texttt{convert(code)}} \wedge (cl > v_{\texttt{code}}(\texttt{i}) \vee v_{\texttt{code}}(\texttt{i}) > ch) \wedge \mathfrak{Z} = replchar$.

Applying the rule ***INVOCATION***, it can be concluded that

$$\Im([\textbf{lwb}\ \texttt{code}, \texttt{i}']) \wedge \mathfrak{Z} = ((cl \leq v_{\texttt{code}}(\texttt{i}) \leq ch \mid charrep(v_{\texttt{code}}(\texttt{i})) \oplus (cl > v_{\texttt{code}} \vee v_{\texttt{code}} > ch \mid replchar)) .$$

holds after the invocation of **repr**.

Therefore,

```
ℑ([lwb code,i'])⟨
    s:=s+repr code[i]
    on nochar=(char,char):
       badcode(i) replace
    no
⟩ℑ([lwb code,i'])∧ℨ=d_s ;
```

so that the proof of Premise 2 of ***LOOP*** immediately follows.

Applying the rule ***LOOP***, the required output assertion for the loop can be deduced.

$$\Im([\textbf{lwb}\ \texttt{code}, \textbf{upb}\ \texttt{code}]) \equiv P_{\texttt{convert(code)}} \wedge ascribed(\texttt{s}, d_{\texttt{s}}) \wedge \tau(d_{\texttt{s}}) = string(\textbf{lwb}\ \texttt{code}, \textbf{upb}\ \texttt{code})$$

The last expression in the body of `convert` is `s`, whose value is returned as the value of `convert`.

$$\Im([\textbf{lwb}\ \texttt{code}, \textbf{upb}\ \texttt{code}])\ \{\ \texttt{s}\ \}\ P_{\texttt{convert(code)}} \wedge ascribed(\texttt{s}, d_{\texttt{s}}) \wedge \tau(d_{\texttt{s}}) = string(\textbf{lwb}\ \texttt{code}, \textbf{upb}\ \texttt{code}) \wedge \mathfrak{Z} = d_{\texttt{s}}$$

The proof of the above immediately follows from the rule for elaborating an identifier. Since the above immediately implies $Q_{\texttt{convert(code)}}$, the proof of correctness of `convert` with respect to its specification is concluded.

6 CONCLUSION

Adopting an expression-oriented approach, and generalizing the concept of a signaller, enables supporting all the structured handler responses that were considered useful in

various proposals for exception handling, with minimal additional mechanism. The uniformity of the mechanism contributes to the uniformity of its axiomatic semantics. In contrast, the only other exception handling proposals supporting both resumption and termination of the signaller, those of [10] and [19], require a much more complex syntactic and semantic extension, though neither one supports all the handler responses supported in the replacement model. The mechanism can be adapted to any of the block-structured programming languages, modulo their specific restrictions, with little loss of expressive power, except that stemming from the expression orientation.

It is interesting to note that addressing exception handling in the context of modularity and program verification provides insights that contribute to simplifying the mechanism. Modularity requires both the exception state and the resumption state to be *consistent* or *possible* states in the sense of [20], otherwise they cannot be specified externally without compromising modular information hiding. This eliminates the problem of exceptions which must be resumed in order to restore the state to a consistent state. Exceptions which cannot or must not be resumed should be signalled just before a logical end of the signaller, after which there is nothing left to be done in the signaller even if resumption is attempted. This eliminates the need for constructs such as the SIGNAL and NOTIFY in [10], and SIGNAL and ERROR in Mesa [19], and answers the main argument of [16] for supporting only resumption.

While our rules appear messy and unwieldy, they are quite straightforward in that once the notation for dealing with the value of an expression is understood, the rules do say exactly what they are expected to say. The two facets contributing to the heavy appearance of the rules are the dealing with the multiple control flow possibilities and the dealing with the values returned by expressions. The first problem is inherent in exception handling; all of the other axiomatic treatments of exception handling have the same heaviness. The second source of heaviness could possibly be eliminated by use of a cleaner, more compact formulation of expressions with side-effects such as that proposed by Boehm [3]. It would be interesting for the future to rewrite the present rules assuming Boehm's logic for the underlying language.

One drawback of these rules, stemming from their unwieldiness is that they cannot effectively be used in construction of programs as is considered desirable these days [11]. Their use is limited to *ex post facto* proofs of previously written programs. Nevertheless, the communicational ideas embodied in these rules are important for program construction. Specifically,

(1) the exception condition states what the signaller knows about an attempted application of an operation and what each handler for that exception may assume,
(2) the resumption condition states what each such handler must guarantee before resuming the signaller; if it cannot, it is obliged to replace the signaller.

Whether stated formally or not, the designer of software utilizing exception handling must decide on these conditions in order to insure that signallers and handlers behave properly with each other.

ACKNOWLEDGMENTS

The authors thank the referees of the first draft of this paper for their highly detailed comments.

REFERENCES

1. IBM OS PL/I Checkout and Optimizing Compilers: Language Reference Manual. SC33-0009-2, IBM Corp. (1973).
2. Ada Language Reference Manual. MIL-STD-1815A, U.S. Department of Defense (1983).
3. BOEHM, H.-J. Side-effects and aliasing can have simple axiomatic descriptions. *ACM Trans. Programm. Lang. Sys. 7*, 4 (Oct. 1985), 637-655.
4. CLINT, M. AND HOARE, C.A.R. Program proving: jumps and functions. *Acta Inf. 1* (1972), 214-224.
5. COCCO, N. AND DULLI, S. A mechanism for exception handling and its verification rules. *J. Comput. Lang. 7* (1982), 89-102.
6. CRISTIAN, F. Reasoning about programs with exceptions. In *Proceedings Thirteenth International Symposium on Fault-Tolerant Computing* (Milano, Italy, June 27-30 1983), IEEE, 188-195.
7. CRISTIAN, F. Correct and robust programs. *IEEE Trans. Softw. Eng. SE-10*, 2 (March 1984).
8. DIJKSTRA, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ (1976).
9. GOGUEN, J.A. Abstract errors for abstract data types. In *Formal Description of Programming Concepts* (St. Andrews, N.B., Canada, 1978), North-Holland, Amsterdam, 491-527.
10. GOODENOUGH, J.B. Exception handling: issues and a proposed notation. *Commun. ACM 18*, 2 (Dec. 1975).
11. GRIES, D. *The Science of Programming*. Springer-Verlag, New York (1985).
12. HOARE, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576-580,585.
13. HOARE, C.A.R. Procedures and parameters: an axiomatic approach. In *Symposium on Semantics of Algorithmic Languages* (Minneapolis, MN, 1971), Springer-Verlag, Berlin.
14. ICHBIAH, J.D. Rationale for the design of the Ada programming language. *SIGPLAN Not. 14*, 6 (June 1979).
15. KOWALTOWSKI, T. Axiomatic approach to side effects and general jumps. *Acta Inf. 7* (1977), 357-360.
16. LEVIN, R. Program structures for exceptional condition handling. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA (June 1977).
17. LISKOV, B.H. AND SNYDER, A. *Structured Exception Handling*. Computation Structures Group Memo 155, MIT (Dec. 1977).
18. LUCKHAM, D.C. AND POLAK, W. Ada exception handling, an axiomatic approach. *ACM Trans. Programm. Lang. Sys. 2*, 2 (April 1980), 225-233.
19. MITCHELL, J.G., MAYBURY, W., AND SWEET, R. *MESA Language Manual*. Xerox Research Center, Palo Alto, CA (March 1979).
20. PARNAS, D.L. *Response to Detected Errors in Well-Structured Programs*. Computer Science Department, Carnegie-Mellon University (1972).
21. PNUELI, A. The temporal logic of programs. In *Eighteenth Annual Symposium on the Foundations of Computer Science* (Providence, RI, Oct. 31-Nov. 2 1977), IEEE, 46-57.
22. PRITCHARD, P. Program proving — expression languages. In *Information Processing 1977* (Toronto, Canada, August 2-12 1977), North-Holland, Amsterdam, 727-731.
23. SCHWARTZ, R.L. An axiomatic treatment of ALGOL 68 routines. In *Proceedings Sixth International Conference on Automata, Languages and Programming* (Graz, Austria, July 1979), Springer-Verlag, Berlin, 530-545.
24. SCHWARTZ, R.L. An axiomatic semantic definition of ALGOL 68. Ph.D. dissertation, Computer Science Department, UCLA, Los Angeles, CA (1980).
25. WIJNGAARDEN, A. VAN, MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A, SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., AND FISKER, R.G. Revised report on the algorithmic language ALGOL 68. *Acta Inf. 5*, 1-3 (1975), 1-236.
26. YEMINI, S. The replacement model for modular verifiable exception handling. Ph.D. dissertation, Computer Science Department, UCLA, Los Angeles, CA (1980).
27. YEMINI, S. AND BERRY, D.M. A modular verifiable exception handling mechanism. *ACM Trans. Programm. Lang. Sys. 7*, 2 (April 1985), 214-243.

Received June 1985; revised August 1986; accepted October 1986