AN AXIOMATIC TREATMENT OF EXCEPTION HANDLING

Shaula Yemini
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

1. INTRODUCTION

This paper presents an axiomatic treatment of exception handling, based on the replacement model [16]. The replacement model, in contrast to other exception handling proposals, supports all the handler responses of resumption, termination, retry and exception propagation, within both statements and expressions, in a modular, simple 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 scope rules. The model 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 ([15]) are presented in [16].

2. EXCEPTION HANDLING IN THE REPLACEMENT MODEL

Operations provided by a programming language, whether primitive or program-defined (functions and procedures), 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 we call the normal case input assertion of the operation. We call an exception of an operation 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 for the read operation, a zero denominator in division, an empty stack in pop.

For maximal usefulness of a module construct that encapsulates objects and related operations, an invoker of an operation should be notified of the detection of an exception, to allow it to determine an appropriate action.
This 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 cannot therefore 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 a 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 in their signaller. Operations (procedures and functions), are required to declare the identifier and data type of each exception they signal. Any generalized expression (i.e., closed construct such as a block, loop or conditional), may become a signaller, by declaring its exceptions.

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

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

Since in 1. signalling has the effect of a normal procedure call to the designated handler, no new keyword is required for supporting resumption; an end terminating a handler has exactly the intended semantics. Thus our only

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

syntactic extension, apart from extensions to the type system, is the completer (a piece of punctuation similar to a closing bracket) replace, allowed within handler bodies. Replace completes an expression in a handler. The value of this expression will serve as a replacement for the value that the signaller would have returned in the normal case.

The addition of replace suffices to support all of the structured handler responses considered useful in previous exception handling proposals (e.g. [3], [1], [5], [7], [4] and [9]): replacing the immediate signalling of the exception yields resumption; replacing the signaller invocation yields termination; signalling an exception within a handler yields propagation; having a new invocation of a signaller within a handler, to replace a signalling invocation, yields retry; 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, yields termination of the construct as a result of the original exception. None of the proposals we are aware of, can support all of these handler responses in a structured fashion.

As an example consider the procedure "convert", which takes an array-of-integers variable as a parameter, and returns the string of the integers' character representations, signalling an exception "badcode" 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 our exception handling mechanism.

```
proc convert = (ref [] int code) string
       signals (exc (int) (char, string) badcode):
begin
  string s := "";
  for i from lb code to ub code
    do
    s := s + repr code[i]
    # append 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 has been modified here to signal the exception "nochar" if its argument does not represent any character. Convert handles nochar of repr by signalling convert's own exception badcode, thus propagating nochar. A signalling of an exception has the same syntactic form as a call. The value returned by a handler for badcode will replace 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 procedure denotation is headed by the list of the types and identifiers of its formal parameters, and its return type. The same is done here for handler denotations. In general, h can be any expression yielding a handler. In the following, we use only handler denotations.

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), has 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 a 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, in order to enable 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.

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 correponding to unconvertable codes are "?".

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

2.i Termination of the signaller: supply the 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: return the value to the point (label) following the invocation which signalled the exception this handler was designated for. This is the return label for the signaller invocation, and can therefore be statically determined.

2.ii Retry: retry after changing the badcode to a zero. The string returned by the new invocation of convert will be 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
  # call convert again, to replace the result of
    the previous call #
  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 (as in [4, 7]), any closed construct in the embedding programming language is allowed to become a signaller of exceptions. This is done by the construct declaring 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 will demonstrate two of the handler responses supported by the replacement model: termination of a closed construct containing the invocation signalling an exception (as in [4, 7]), 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 after the do, declaring finish. The two types appearing in finish's heading are its resume and replace types respectively. Finish is signalled in the handler for badcode (i.e., it is the propagation of badcode). When the handler for finish replaces finish's signaller's invocation - the loop invocation, the loop will be terminated, as required:

```
3. Termination of a closed construct and
exception propagation:
begin
  do signals (exc (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 value yielded by skip #
```

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

For further details and examples of the replacement model see [16].

3 AXIOMATIC SEMANTICS

Our axiomatic treatment of exception handling follows the axiomatic approach proposed by Schwartz for Algol 68 in [13]. We find this approach most 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.

Sentences in the extended logic used in [13] have the form N / P {s} Q ⌈=v. S is an expression or statement. ⌈ represents the value yielded by s (empty if s is a statement). P and Q∧⌈=v are the input and output assertions respectively. N is a NESTL, which maps the set of all identifiers and derived modes known at each point in the program, into their declared modes. The NESTL provides the static properties of the program necessary for the proof. The axiomatization assumes that programs to be verified are compile-time correct, i.e., all type checking and mode 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".

The axiomatic definition of exception handling requires:

1. a form for specifying a signaller together with the exceptions it signals, independently of any specific choice of handlers,
2. a definition of correctness of a signaller with respect to a specification of the above form,
3. a form for specifying the independent effect of a handler, and
4. an adaptation style proof rule 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.

3.1 Specifying Signallers and their Exceptions

Let s be a signaller of one exception e with formal parameters x. We use the following notation for an input/output specification of s:

(1) N / {P, Q∧⌈=v, e(x) <E(x), R(x)∧⌈=u>}

P is the input assertion. Q∧⌈=v is the normal case output assertion which will be satisfied if no exceptions are signalled. E is the exception condition corresponding to the exception e, describing the state when e is

signalled. R∧1=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∧1=v, will be satisfied if and when s halts. The above specification states that if the input state satisfies P, and if the execution of the specified construct halts, then either Q∧1=v holds, or the exception e is signalled and the state then satisfies E. R∧1=u must be satisfied before resumption. This notation reduces to {P, Q∧1=v} when there are no exceptions, but in that case we will use the conventional notation: P {s} Q∧1=v.

We say s is partially correct with respect to the specification above, or

s pc wrt N/ {P, Q∧1=v, e(x) <E(x), R(x)∧1=u>}

iff

$$\frac{\text{for any handler h for e } N/\ E(\underline{x})\ \{h(\underline{x})\}\ R(\underline{x})\wedge 1=u}{N/\ P\ \{s\}\ Q\wedge 1=v}$$

i.e., iff the assumption N/ E {h} R∧1=u, enables proving that N/ P {s} Q∧1=v. This enables us to use 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 will ensure that a handler does indeed satisfy the resumption condition before resumption.

In the general case, a specification may include several (P, Q∧1=v) pairs, which may each have several associated exceptions. The extension to this case is straightforward (see [16]).

3.2 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, we need only one additional rule (to the rules in [13]), in order to specify handler semantics. This is the rule for the site of the completer replace.

3.2.1 Rule for Replace

Replace completes an expression in a handler body. The value of this expression is yielded at the program location (label) to which control is transfered by replace. This label is the return label of the invocation the handler was designated for, and is statically determinable.

Replace has the effect of preserving both the state and the returned value, except that control is transferrd to the label determined by the invocation. The interpretation of this label therefore has to be adapted to the context of each invocation for which the handler is designated, as will be seen in the rule for the site of an invocation given below. The rule for replace is:

$$\frac{N\ /\ P\ \{e\}\ Q\ \wedge\ 1=v}{N\ /\ P\ \{e\ \underline{\text{replace}}\}\ \text{replace: } Q\wedge 1=v}$$

The notation here is borrowed from temporal logic ([12]), which uses assertions of the form <label>: <predicate> to specify that the predicate is to hold at the specified label in the program. The "replace:" in the consequent is an uninterpreted label, which will be interpreted (adapted) in the rule for the site of an invocation, for each specific invocation.

3.2.2 Rule for a Signaller Invocation

This rule combines the independent specification of a signaller, and the independent specification of the handlers designated for a particular invocation of this signaller, to derive the effect of the invocation.

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. 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), we will conclude that INV remains true after the signaller has completed.

In [13], a unique special variable 1_i 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: "1=v" for assertions about 1_is when pushing assertions through successive expression layers, the domain of formal values was enlarged to include conditional values of the form (P|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.

We first give a special case of the rule for the case where 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, we assume that p may signal only one exception ex.

The rule is:

1. N/ T∧INV {COLLAT(e_1,...,e_n)} P∧INV∧1_e=$\underline{x}$
2. N_p/ p($\underline{x}$) pc wrt {P, Q∧1_0=v, ex($\underline{z}$)<E, R∧1_1=u>}
3. N/ E∧INV {h($\underline{z}$)} R∧INV∧1_1=u V replace:S∧1_2=y

---

N/T ∧ INV {p(e_1,..,e_n) $\underline{on}$ ex=h $\underline{no}$}
(Q V S) ∧ INV ∧ 1=(Q|v)⊕(S|y)

Premise 1 accounts for the effects of the collateral elaboration (evaluation in an unspecified order) of the actual parameter expressions e1,...en. This elaboration may have side effects. (Rules for collateral elaboration can be found in [13]).

Premise 2 is the specification of p, in the form described earlier.

Premise 3 is the specification of the handler h. The input assertion for the handler naturally involves the exception condition for the exception the handler is designated for. A handler may either cause resumption of the signaller after handling (indicated by terminating the handler by $\underline{end}$, without any preceding $\underline{replaces}$); in this case it must be shown that the handler establishes the resumption condition, in order to be able to conclude that the normal case output assertion holds after p terminates; alternatively, it may cause replacement of the signaller (indicated by an expression followed by $\underline{replace}$); in this case the result of the invocation will be the result of evaluating the replacement expression. There can also be more than one $\underline{replace}$ 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 the replacement case.

The consequent states that if all the premises hold before the invocation, then after the invocation, either the normal case output assertion or the handler's replacement output assertion hold.

The syntactic rules for designating handlers allow an $\underline{on}$ clause designating handlers, to be postfixed to a closed construct, thus designating the handlers in the $\underline{on}$ clause for all invocations within the construct. In the proof rules it is assumed that all handler designations have been copied to immediately postfix the associated invocations. This copying transformation can be determined statically.

The rule for a signaller which is a closed construct (e.g. block) is a special case of the rule for a signaller invocation, where there are no parameter expressions and INV=true:

1. N/ s pc wrt {P, Q∧1_0=v, ex($\underline{x}$) <E, R∧1_1=w>}
2. N/ E {h($\underline{z}$)} R∧1_1=w V replace: S∧1_2=y

N/ P {s $\underline{on}$ ex=h $\underline{no}$} (Q V S) ∧ 1=(Q|v)⊕(S|y)

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 (though they will 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, we assume that the invoked signaller may signal only one exception ex. The rule is thus:

1. N/T∧INV{COLLAT(e_p,e_1,...,e_n,e_h)}
$[\bigvee_{i=1}^{m} (P_i \wedge 1_p = w_i)] \wedge [\bigvee_{j=1}^{k} (P'_j \wedge 1_h = h_j)] \wedge INV \wedge 1_{\underline{e}} = \underline{x}$

2. ∀ i, 1≤i≤m N/ w_i($\underline{x}$) pc wrt
{P_i, Q_i∧1_1=v_i, ex($\underline{y}$) <E_i, R_i∧1_2=z_i>}

3. ∀ i,j, 1≤i≤m, 1≤j≤k
[N/ E_i∧INV {h_j($\underline{y}$)}
R_i∧INV∧1_2=z_i V replace:S_{ij}∧INV∧1_1=u_{ij}]

N/T∧INV {e_p(e_1,...,e_n) $\underline{on}$ ex=e_h $\underline{no}$} INV ∧
$[((\bigvee_{i=1}^{m} Q_i) \wedge 1_1 = (Q_1|v_1)\oplus\ldots\oplus(Q_m|v_m)) \vee$
$((\bigvee_{i=1}^{m}\bigvee_{j=1}^{k} S_{ij}) \wedge 1_1 = (S_{11}|u_{11})\oplus\ldots\oplus(S_{mk}|u_{mk}))]$

Premise 1. accounts for the side effects of the (collateral) evaluation of the expressions yielding the invoked procedure value, the parameters to the invocation, and the designated handler. All are potentially conditional values. However, for both the invoked procedure and the designated handlers, it is necessary to distinguish all the individual values comprising the conditional value in order to examine all the individual specifications. Thus, assuming that the procedure expression e_p may yield any one of the routine values w_i, instead of writing 1_p=p where p is a conditional value involving m $(P_i|w_i)$s, p is separated into the m $(P_i \wedge 1=w_i)$s. The same is done for 1_h. $1_{\underline{e}}$=$\underline{v}$ is shorthand for 1_{e_i}=v_i, i=1,...,n the actual (conditional) parameter values.

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

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

In the following we outline the use of our proof rules in proving the correctness of convert. Since convert contains an invocation of a signaller, this will demonstrate both proving correctness with respect to a specification of the form we have introduced, and applying the rule for an invocation.

In order to keep the notation less cluttered, the NESTLs are omitted. We assume repr(n) has been proven correct with respect to the following specification:

$\{P_{repr}(n) \equiv$ true

repr assumes nothing about its argument

$Q_{repr}(n) \equiv$ c1$\leq$n$\leq$ch $\wedge$ l= char rep(n),

repr returns the character represented by its argument when its argument is in the range (c1,ch).

nochar $\langle E_{nochar} \equiv$ n>ch $\vee$ n<c1,

when repr signals nochar, its argument is out of range

$R_{nochar} \equiv Q_{repr}(n)\rangle\}$

in order to ensure that repr satisfies $Q_{repr}(n)$, a handler must satisfy $Q_{repr}(n)$ before resuming repr

It is assumed that the mapping char rep has been defined appropriatly. The rules also assume that that all type checking has already been performed. R_{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 "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_{code}. The contents of this location are obtained by the mapping τ, and are denoted here by v_{code}. (Ascribed and τ are close to the notions of environment and store in denotational semantics).

We assume the following specification of convert:

$\{P_{convert} \equiv$ ascribed(code,d_{code}) $\wedge \tau(d_{code}) = v_{code}$

assuming that v_{code} is the value at the location bound to convert's argument,

$Q_{convert}$(code) $\equiv P_{convert} \wedge \tau(d_s) =$

$$\underline{ub}\ \text{code} \quad \underset{i=\underline{lb}\ \text{code}}{+} [(c1 \leq v_{code}(i) \leq ch \mid \text{char rep}(v_{code}(i))) \oplus (c1 > v_{code}(i) \vee v_{code}(i) > ch \mid \text{replchar})]$$

$\wedge\ l = d_s$

when convert terminates, the argument will be unchanged, and the returned value will be 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

$\langle E_{badcode}(i) \equiv P_{convert} \wedge (c1 > v_{code}(i) \vee v_{code}(i) > ch)$

when badcode is signalled an unrepresentable code has been encountered,

$R_{badcode}(i) \equiv E_{badcode} \wedge l =$ replchar$\rangle\}$

and the handler must provide a replacement character in order to enable convert to satisfy $Q_{convert}$

We define

$$\text{string}(l,u) \equiv \underset{i=l}{\overset{u}{+}} [(c1 \leq v_{code}(i) \leq ch \mid \text{char rep}(v_{code}(i))) \oplus (c1 > v_{code}(i) \vee v_{code}(i) > ch \mid \text{repl char})]$$

string(l,u) is the string resulting from the successive concatenation of the corresponding character representation for character codes of code, and repl char for non character codes of code.

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

$P_{convert} \wedge$ ascribed(s,d_s) $\wedge\ \tau(d_s)$=""

{for i from lb code to ub code
do
s := s + repr code[i]
od
on nochar = (char,char):
badcode(i) replace
no }

$P_{convert} \wedge$ ascribed(s,d_s)

$\wedge\ \tau(d_s)$=string(lb code,ub code) $\wedge$ l=empty

This requires using the rule for a loop. We use the following notation for intervals on the integers:

[l,u]={j| l$\leq$j$\leq$u, jεINT}

[l,u)={j| l$\leq$j<u, jεINT}

The rule for a loop which we need to apply is:

1. N/P => I([])

2. $N'/i' \varepsilon [l,u] \wedge I([l,i'))$ {body } $I([l,i'])$

N/P{for i from l to u do body od}

$I([l,u]) \wedge l$=empty

I in the above is the loop invariant. Let:

$S \equiv P_{convert} \wedge ascribed(s,d_s)$

$P \equiv S \wedge \tau(s)=""$

$I([1,u]) \equiv S \wedge \tau(d_s)=string(1,u)$

The proof of premise 1. is immediate.

In order to show 2., (the inductive step), we need to use the rule for an invocation of a signaller, in order to get the effect of repr given the supplied handler.

We want to show that

I([lb code,i'))
{repr code[i]
  on nochar = (char,char):
    badcode(i) replace
  no }
I([lb code,i']) ∧ ⌐=
 + [(cl≤v_{code}(i)≤ch| char rep(v_{code}(i)))
  ⊕ (cl>v_{code}(i) ∨ v_{code}(i)>ch|replchar)]

We use the rule for an invocation with T≡I([lb code,i')). Since there are no side effects in evaluating code[i], we have for premise 1. of the rule for an invocation:

1. N/ I([lb code,i')) {code[i]}
  I([lb code,i')) ∧ ⌐=v_{code}(i)

For premise 2. of the rule for an invocation, we assume repr has been proven correct with respect to the given specification. The output assertion of 1. trivially impiles the input assertion of repr.

For premise 3. of the rule for an invocation, we need to get the characterization of the handler for nochar. This handler propagates repr's exception nochar as the exception badcode of convert. Our definition of partial correctness allows to assume in proving the correctness of a signaller (here convert), 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 $E_{badcode}(i)$ holds, it may be assumed that immediately after the signalling, the corresponding resumption condition $R_{badcode}(i)$ holds. Since for code[i]

$E_{nochar} \Rightarrow E_{badcode}(i)$,

we may assume $R_{badcode}(i)$ holds after the signalling of badcode. Thus:

3. E_{nochar}
{(char, char): badcode(i) replace }
replace: $P_{convert} \wedge$ (cl>v_{code}(i) ∨ v_{code}(i)>ch)
  ∧ ⌐= replchar

Applying the rule for an invocation, we can conclude that:

I([lb code,i']) ∧
⌐= ((cl≤v_{code}(i)≤ch| char rep(v_{code}(i)) ⊕
  (cl>v_{code} ∨ v_{code}>ch| replchar))

holds after the invocation of repr.

Therefore

I([lb code,i'))
{s := s + repr code[i]
 on nochar = (char,char):
   badcode(i) replace
 no }
I([lb code,i']) ∧ ⊥=d_s

so that the proof of premise 2. of the rule for a loop immediately follows.

Applying the rule for a loop, we can now conclude the required output assertion for the loop:

I([lb code,ub code]) ≡
$P_{convert} \wedge ascribed(s,d_s)$
  ∧ $\tau(d_s)$=string(lb code,ub code)

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

I[lb code,ub code])
{s}
$P_{convert} \wedge ascribed(s,d_s)$
  ∧ $\tau(d_s)$=string(lb code,ub code,code,replchar)
  ∧ ⊥=d_s

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

4. CONCLUSIONS

Adopting an expression oriented approach, and generalizing the concept of a signaller, enabled us to support 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 simplicity of its axiomatic semantics. In contrast, the only other exception handling proposals supporting both resumption and termination of the signaller, [1] and [9], 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.

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 be "consistent" or "possible" states in the sense of [11], 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 [1], and SIGNAL and ERROR in Mesa ([9]), and the main argument of [5] for supporting only resumption.

References

1. Goodenough, J. B. "Exception Handling: Issues and a Proposed Notation", Comm. ACM, 18,2 December 1975.
2. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming", Comm. ACM 12,10 October 1969.
3. IBM OS PL/I Checkout and Optimizing Compilers: Language Reference Manual SC33-0009-2 IBM Corp. 1973.
4. Ichbia, J. et al "Rationale for the Design of the ADA Programming Language" SIGPLAN Notices 14,6 June 1979.
5. Levin, R. "Program Structures for Exceptional Condition Handling". PhD Thesis, Carnegie Mellon University, June 1977.
6. Liskov, B. H. and S. N. Zilles, "Specification Techniques for Data Abstractions", IEEE Trans. Software Eng. 1,1 March 1975.
7. Liskov, B. H. and A. Snyder, "Structured Exception Handling" Computation Structures Group Memo 155, MIT December 1977.
8. Luckham, D. C., and W. Polak "ADA Exception Handling: An Axiomatic Approach", ACM TOPLAS 2,2 April 1980.
9. Mitchell, J. G., W. Maybury and R. Sweet "MESA Language Manual", Xerox Research Center, Palo Alto March 1979.
10. Parnas D. L., "A Technique for the Specification of Software Modules", Comm. ACM 15,5 May 1972.
11. Parnas D. L., "Response to Detected Errors in Well-Structured Programs", Computer Science Dept., Carnegie-Mellon University 1972.
12. Pnueli, A. "The Temporal Logic of Programs", 19th Annual Symp. on Foundations of Computer Science, Providence R.I. November 1977.
13. Schwartz, R. L., "An Axiomatic Semantic Definition of ALGOL 68", PhD Thesis, UCLA 1978.
14. Schwartz, R. L., "An Axiomatic Treatment of Algol 68 Routines", Proc. 6th International Conf. on Automata, Languages and Programming, Gratz Austria July 1979.
15. van Wijngaarden, A. et al "Revised Report on the Algorithmic Language Algol 68", Acta Informatica 5 1975.
16. Yemini, S. "The Replacement Model for Modular Verifiable Exception Handling", PhD Thesis, UCLA 1980.