A NEW APPROACH TO SPECIFYING AND HANDLING EXCEPTIONS

BY

P. A. SUBRAHMANYAM

UUCS-80-107

JANUARY 1980

A New Approach to Specifying and Handling Exceptions

P. A. Subrahmanyam
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

January 1980

Abstract

An operation generally exhibits different patterns of behavior over different parts of its domain. Depending upon the context, such behavior may either be conceived of as "normal" or as an "exception." Thus, the behavior of an operation is quite naturally characterized by the set of partial operations that characterize its (different) behavior on its subdomains, and exceptions essentially serve to extend (modify) the normal behavior of an operation.

In this milieu, we consider the issues of specifying and "handling" exceptional conditions that might occur during the execution of an operation. We argue that one of the important features that an exception handling mechanism need possess is to enable the behavior of an operation to be altered over *part* of its domain i.e. that of being able to incrementally modify the semantics of the partial operations that serve to characterize an operation. Surprisingly, this requirement arises out of a pragmatic consideration -- that of providing for the existence of a library of subroutines. We propose a general mechanism for specifying exceptions and their handlers that does not compromise the possibility of efficient implementations. Examples of application of the method are presented in the context of abstract (algebraic) data type specifications, using skeletal specifications of a Stack and an error-correcting parser for a context free grammar. The major advantage of the proposed mechanism over conventional approaches is that of completely avoiding the problem of "dynamic context propagation."

Keywords: exceptions, exception handling, specification, abstract data types, verification, modules.

Table of Contents

A New Approach to Specifying and Handling Exceptions

1. On Exceptions and Exception Handling

Programming involves representing the primitive operations and objects relevant to a problem domain using other primitives operations and objects that are presumed to be available; ultimately, such primitives are those provided by the underlying hardware. Each layer of such a representation is commonly conceived of as a "module" that provides a consistent ("correct") implementation of the specifications at the preceding level, the topmost level being the initial problem specification. The collection of objects and operations at any such level can, however, be specified in a representation independent manner -- such a specification is termed an abstract (data type) specification.

Because abstract data types provide a linguistic vehicle to embody the important notions of modularization and abstraction, they aid in the formulation of a piecewise design of a program. For this reason, they are fast becoming a standard tool in programming, and have been incorporated in several recent programming languages in one of various forms [15, 22] (many of these not so "abstract" e.g. ADA [10].) The modularization that is thus achieved is commonly associated with the fact that the operations defined on such an abstract data type specify *all* of the allowed manipulations on the "objects" of the type, and therefore enable a single, well defined channel of access to such objects.

The notion of modularization, however, can be further exploited in specifying the semantics of operations defined on a data type. This can be done by making piecewise assumptions that certain conditions hold in a particular "computing environment," and specifying the semantics of the operation under this assumption. Thus, for example, it is convenient to define the semantics of a parser first under the assumption that the input program is a legitimate string in the language generated by the grammar, and then in the case when the input string is not generated by the grammar. Quite often, the second condition entails some form of error correction.

Thus, it is perfectly natural to view the overall behavior of an operation f as being characterized by the piecewise behavior of "partial" operations f_1, ..., f_n that are

defined only when some associated conditions P_1, ..., P_n on the computing environment are satisfied. If, in the course of performing the operation f, condition P_i is detected, then the semantics of f are those of f_i.

Of course, in a specific context, some of the conditions in $\{P_1, ..., P_n\}$ might arise more or less frequently than others. In such cases it is usual to think of the more frequently occurring case as the "normal" case, and of the others as "exceptions." In the latter case, an *"exception (condition)"* is said to have been *"raised"* (i.e. some associated predicate P_i evaluates to true;) the execution of f_i is called "*handling*" the exception. The distinction between what is normal, and what is an exception, however, is often quite nebulous: it usually depends purely on the programmer's desire to play down one case, or to share costs differently in an implementation, etc. [13]. It is imperative, however, that any implementation be consistent with the specified semantics.

It therefore follows that exceptions essentially serve to augment an operation's behavior in the "normal" case by extending the operation's domain and/or range: this important observation was made by Goodenough in [7]. Paradoxically, however, exception handling has always been treated as a "control flow" issue. As we shall see, the unfortunate consequence of adopting such a perspective has been the proliferation of a number of increasingly complicated proposals for exception handling mechanisms [13]. We will argue here that exception handling is basically *not* a control flow issue, and that treating it as such gives rise to needless problems.

Divesting the issue of control flow from that of exception handling has a second non-trivial consequence: our treatment of exception handling no longer centers around the von Neumann architecture with its emphasis on the distinction between control and data, but is applicable more generally. In particular, it is of relevance in the context of applicative programming and algebraic data type specifications.

The approach to exception handling proposed in this paper has the desirable (and, we believe, important) feature that the formal semantics of operations, the notion of an implementation, and the "correctness" of an implementation are relatively simple extensions of the corresponding notions in the "normal" case. We will, however, only outline the nature of these extensions here; the mathematical details can be found in a companion report. Also, we do not discuss here the problem of applying these concepts

to the realm of concurrent programming.

2. Summary of the Paper

We continue this paper in section 3 by focussing on the essential features that are needed in an exception handling mechanism, as opposed to the idiosyncrasies of specific proposals. In section 4, we then propose a new method for exception handling that attempts to provide the necessary features while eliminating the complications introduced by extant techniques; in particular, the problem of "dynamic context propagation" is completely avoided. The proposed mechanism is illustrated in section 5 by its application to two examples. The examples presented revolve around skeletal specifications for a Stack, and an error-correcting parser (which might form a part of a compiler.) Some tentative conclusions are contained in section 8.

3. Relevant Issues in Exception Handling

Most of the existing mechanisms for exception handling have treated the problem as a control flow issue. An excellent summary of several of these methods may be found in [7], [13]. The attempts since then have not diverted in any major sense from this perspective e.g. [10, 11], [18], to name only a few. We attempt to distill those features that seem essential to an exception handling mechanism (as mandated by its *desired* behavior,) as opposed to details particular to existing proposals.

To recapitulate, exceptions essentially serve to extend (or piecewise modify) the "normal" domain (or behavior) of an operation. Any exception handling mechanism should therefore provide a way to specify exceptions that reflects this fact in a natural manner, while preserving the other desirable features of the environment in which such a mechanism is embedded; these features include:

- The existing boundaries of abstraction and modularization should be preserved. This fact is particularly important to stress, since, in the model adopted conventionally, the contexts of signalling and the handling of an exception are distinct and as a consequence information flow occurs across abstraction boundaries (i.e. across the boundaries of "modules" that constitute an implementation.) It is therefore necessary to preserve the abstractions when such a transition occurs. Unfortunately, however, this important fact is often overlooked in most exception handling mechanisms with the result that the programmer is often confronted with cryptic error messages like "addressing exception at location 05839"

(s/he never was consciously trying to access such a location) or "index into array exceeds bounds" (s/he was only trying to push items onto a Stack.)

- The formal semantics of operations, the notion of an implementation and its "correctness," as well as the verifiability of the correctness of implementations should be relatively easily extensible from the normal case to the exceptional case.

- The possibility of obtaining efficient implementations should not be compromised.

In other words, since exceptions are as natural a part of an operation as its "normal" behavior, any facility for specifying and handling them should not, ideally speaking, unduly disrupt the existing state of affairs. Despite this, most of the mechanisms that have been proposed for exception handling seem to have gone against this precept. We therefore attempt to isolate the reasons for this disparity, with a view to remedying the situation.

One of the primary reasons for the existence of non-uniform exceptional handling mechanisms is the fact that, in conventional block-structured programming languages, the program context in which an unusual event or condition -- the exception -- is detected may not be the best context in which to process it. It is therefore necessary to *propagate* exceptions to the appropriate "context" wherein it can be processed. To provide for adequate flexibility in handling exceptions, it is often necessary to violate the scope rules implied by block structuring, and most mechanisms do so to a greater or less extent. An *immediate* consequence of such a decision is to compromise the desire to preserve the benefits of block-structuring, thereby making both the comprehension and the verification of such programs much more difficult. Even more tragic is the fact that most proposed mechanisms only half-heartedly make this compromise, thereby sacrificing some amount of flexibility in addition to incurring the other disadvantages.

3.1. A new paradigm for exception handling

We now argue that the facility that is most essential in an exception handling mechanism is one of enabling "incremental changeability" of an operation's behavior on *parts* of its domain, and then propose an approach to exception handling which achieves the desired goals *without* having to dynamically propagate contexts, etc. Both our

rationale as well as the proposed mechanism are based on the following assumptions:

1. A formal statement of the problem specification is an *essential* pre-requisite for the development of reliable software.

2. Such a formal specification should explicitly specify all the relevant conditions on its input values, and the corresponding action to be taken i.e. all "exceptional" conditions *that are relevant to the operation's semantics* must be explicitly specified. (It is crucial to note that this does not imply specifying the exceptions that any of the implementing modules might raise.) Any unspecified condition is treated as causing a globally visible error.

3. The implementation must be designed so as to ensure its adherence to the mandated specifications. Thus, for example, if we consider the figure 3-1, the implementation "module" at level I should ensure consistency with the specifications of level I+1.

As a natural consequence, any conditions that are irrelevant to the specifications at some level, say I, should not be "propagated" to that level. Not only does such propagation introduce needless machinery, it is quite meaningless: since level I is not even aware of the details at level I-1 that caused the exception in question, and what is more, does not, and in fact should not, even care!

Our basic assumption that level I is implemented correctly by level I-1 avoids any problems inherent in such "dynamic" propagations. At first sight, then, it even appears that there is no need for *any* exception handling mechanism. Pragmatically, however, this is somewhat of an oversimplification. For, if each implementation must be tailored to the requirements at level I, need this imply that for even very slight differences in specifications, (that might arise due to the need for slightly different "handlers",) the routines at level I-1 be re-implemented? Hopefully not. What is therefore needed of an exception handling mechanism is the facility of being able to view the same operation from slightly different perspectives -- that is, the facility of enabling incremental changeability of an operation's semantics on part of its domain. In its absence, it is no longer possible to have standard library routines -- which is clearly an undesirable consequence. An exception handling mechanism in fact need just address this question, for the original existing method of solving this problem by "propagating" the exception to the invoking "context" clearly leads to clumsy ways of handling the (wrong!) issues.

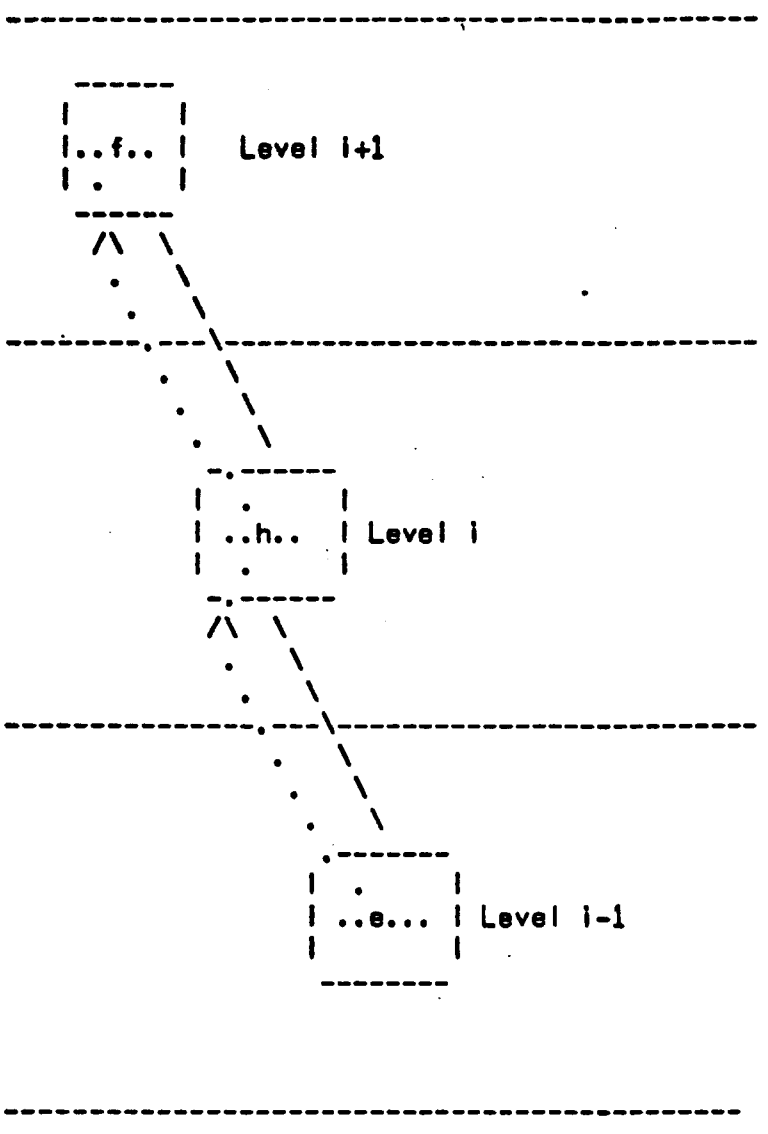

Figure 3-1: Hierarchies of Modules.
An exception at level i-1 should *not* be propagated to level i

4. Details of tho Proposed Exception Handling Mechanism

We view the overall behavior of an operation f as being characterized by a set of partial operations (functions,)[1] each of which characterizes the piecewise behavior of f when some associated predicate holds over its domain. The range of a function may be a disjoint union of types T_1, T_2, ..., T_n, denoted by $T_1 + T_2 + \ldots + T_n$.

We denote the n partial functions characterizing the function f, $f : T \rightarrow T_1 + T_2 + \ldots + T_n$, by $f.T_1$, ..., $f.T_n$. We refer to the tuple $T,T_1+\ldots+T_n$ as the arity of the function f.[2] We note that T can in general represent any arbitrary tuple of types. The partial function $f.T_i$ has arity

$$f.T_i : T \rightarrow T_i$$

Associated with each partial function $f.T_i$ is a predicate $P.f.T_i$ of arity

$$Pf.T_i : T \rightarrow \text{Boolean.}$$

This predicate $P.f.T_i$ embodies the condition on the input domain T when the behavior of f is characterized by the partial function $f.T_i$. Thus, the semantics of the function f can be expressed as follows:

f(t) = if $P.f.T_1(t)$ then $f.T_1(t)$
else if $P.f.T_2(t)$ then $f.T_2(t)$
...
else if $P.f.T_i(t)$ then $f.T_i(t)$
...
else if $P.f.T_n(t)$ then $f.T_n(t)$

Consequently, the specification of the semantics of f involves specifying the semantics for $\{P.f.T_i\}$ and $\{f.T_i\}$. Since an exception merely serves to extend the domain and/or range of a function, the above framework needs no additional embellishments to allow for the specification of "exceptional conditions." The intuition behind this is that if a predicate $P.f.T_i$ on the input corresponds to an exceptional

---

[1] Throughout this paper, we use function and operation synonymously. Strictly speaking, however, we wish to allow non-determinism as an intrinsic characteristic of an operation, and as a consequence we use the term "function" quite loosely to allow such an interpretation.

[2] Although the term arity is sometimes confused with functionality, there is a technical difference: the former alludes only to the syntactic strings *naming* the domain(s) and range(s) of a function, whereas the latter refers to the domain(s) and range(s) themselves.

condition, then the partial function $f.T_i$ returns an "exception type" and the "handling" of this exception is accordingly embodied in the semantics of $f.T_i$ *as defined on the exception type* (c.f. section 4.1.)

In order to be able to provide for tailored handlers for exceptions (c.f. Section 3,) it is necessary to permit incremental modification of the semantics of a function f defined on the type T. For this purpose, we adopt the following convention: if the predicate $P.f.T_i$ characterizes an exception condition, then

1. The "modifiability" of the handler for this condition is syntactically indicated by prefixing T_i with an asterisk in the syntactic specification of f thus:

$$f : T \rightarrow T_1 + \ldots + {}^*T_i + \ldots + T_n$$

2. The semantics of f take the form

$f(t) = ...$
else if $P.f.T_i(t)$ then $f.T_i(f.T_i(t))$
....

(The point to be noted here is that the function $f.T_i$ is seemingly applied twice! We will elaborate on the significance of this shortly.)

Usually, all but one of the predicates $P.f.T_1, \ldots, P.f.T_n$ correspond to an "exceptional" condition. If we assume that $P.f.T_1$ corresponds to the normal case, then the semantic specification of f takes the form:

$f(t) =$ if $P.f.T_1(t)$ then $f.T_1(t)$
else if $P.f.T_2(t)$ then $f.T_1(f.T_1(t))$
....
else if $P.f.T_n(t)$ then $f.T_n(f.T_n(t))$

4.1. On the "Handling" of Exceptions

If $P.f.T_i$ corresponds to an "exceptional condition" in the domain of f, then the type T_i represents the "handler type" for this condition. Obviously, T_i is no different from any other type, and can be arbitrarily complex. Also, since an instance of T_i is spawned by the function $f.T_i$, T_i is a "parameterized exception handler" and the function $f.T_i$ is one of

the *base constructors*[3] of type T_i. As a result, the type T_i must, of necessity, have the following two functions defined on it:

1. the base constructor $f.T_i : T \rightarrow T_i$, and

2. a distinct function $f.T_i : T_i \rightarrow T_i'$ which differs from the base constructor $f.T_i$ by having a different arity.

If the exception handler for the condition $P.f.T_i$ is modifiable, (indicated by the asterisk in $f : T \rightarrow \ldots + {}^*T_i + \ldots$,) then the type returned by f is T_i', with its semantics given by the semantics of $f.T_i : T_i \rightarrow T_i'$. Thus, both the syntax and the semantics of f are determinable only at "execution time." This explains the reason for the two applications of $f.T_i$ present in the semantics of f: they are applications of two *different* functions that have the same name -- the first application is that of the base constructor $f.T_i:T\rightarrow T_i$ which serves to establish the proper "environment" for the exception handler, whereas the second application of $f.T_i: T_i \rightarrow T_i'$ serves to initiate the handling itself.

4.2. Chained Exceptions

Using the method detailed above, it is possible to deal with arbitrary levels of exceptions (i.e. exceptions within an exception handler) in a uniform way, since the functions at each level are incrementally modifiable. Thus, if

$$f : T \rightarrow \ldots + {}^*T_i + \ldots$$

and

$$f.T_i : T_i \rightarrow \ldots + {}^*T_i' + \ldots$$

then $P.f.T_i.T_i'$ characterizes an exception condition (which may in turn have another parameterized exception handler.)

4.3. Globally Visible Errors

The only "exception" to the description above occurs when the mandated specification (and/or performance) at level i+1 *cannot* be achieved by the implementation

---

[3] A base constructor of a type is a function that serves to spawn new instances of a type. Quite often, $f.T_i$ is the *only* base constructor of the type T_i.

module at level I. Examples of situations of this kind are: "unrecoverable disk failure," "power failure," etc. In such a case, a globally visible error occurs, which might uniformly have the effect of "collapsing" all terms to a single "error" type. This situation may also be treated as in [5].

4.3.1. Backward Error Recovery

In order to have robust "programmed" exception handling (sometimes termed forward error recovery,) the disjunction of the predicates $\{P.f.T_i\}$ characterizing the subdomains of the partial functions $\{f.T_i\}$ should always be TRUE; that is, all anticipated cases $\{P.f.T_i\}$ should serve to exhaust the domain of a function. The notion of globally visible errors may also be interpreted so as to account for unanticipated exceptions; usually, the only way to recover from such exceptions is to "rollback" to some previous state of the system that is known to be consistent. We will not, however, elaborate on this aspect of exception handling, as it relate more to issues of fault-tolerant systems, and is outside the scope of this paper [8], [17].

5. Some Examples

We now illustrate the application of the proposed method for specifying exception conditions and associated "handlers" by its use in two examples: a bounded stack, an error correcting parser for a context free language.

5.1. A Bounded Stack

The specifications for the type Bounded-Stack are given in figure 5-1. There are three exceptional conditions that may arise: StackOverflow (while applying PUSH to an already "full" stack,) StackUnderflow (while applying POP to an empty stack,) and ItemUndefinedError (while applying TOP to an empty stack). These three conditions are embodied in the predicates P.PUSH.Stack, P.PUSH.StackOverflowHandler, P.POP.Stack, P.POP.StackUnderflowHandler, P.TOP.Item, and P.TOP.ItemUndefinedError defined over the domains of the operations PUSH, POP, and TOP. The complete definitions of these functions is given in figure 5-1. We can now specify the "error handler types."

5.1.1. Three Handlers for the Exception StackUnderflow

For the sake of illustration, we list three distinct handlers for the case of Stack underflow (see figure 5-2, 5-3, 5-4). The first simply outputs the string "STACK UNDERFLOW". The second outputs the string "STACK UNDERFLOW:" concatenated with the present Stack configuration (converted to the form of a printable string by the function CONVERT: Stack -> String.[4]) The third "resumes" normal execution by returning the Stack NEWSTACK.

5.1.2. Handlers for the exception StackOverflow

Two distinct handlers for the exception StackOverflow are given. The first handler "corrects" the overflow condition that has occurred by creating a new Stack of size one greater than the one in which the overflow has occurred, and initializing it with the contents of the previous Stack. The second handler for Overflow merely returns the String "STACK OVERFLOW".

5.2. An Error-Correcting Parser

We consider here the problem of parsing a sentence generated by a context-free grammar e.g. LL(1) as might be required in course of the compilation of say, a PASCAL program. The output of the parser in such a case is a parse tree (Parse-Tree,) and a set of messages (Parser-Msgs) generated during the parsing phase. The input to the parser consists of the grammar for the language (perhaps in some suitable tabular form,) the input program string, and an initially empty parse stack. If and when an error is detected during a parse, it is often desirable to attempt some form of error-recovery (see, for example, [19]). The detection of a syntactic error in the input string can be viewed as an exceptional condition where the associated handler performs the syntactic error recovery functions. We merely delineate below (figure 5-6) the structure of such a parser to illustrate how such an exception handler may be structured, without detailing the semantics of any of the functions involved.

---

[4] We do detail any semantics for CONVERT here, but it should be quite obvious how this may be done.

Type Bounded-Stack

Syntax

```
NEWSTACK: Integer -> Stack

PUSH: Stack, Item -> Stack + *StackOverflowHandler
    PUSH.Stack : Stack, Item -> Stack
    PUSH.StackOverflowHandler : Stack, Item ->
                                        *StackOverflowHandler
POP: Stack -> Stack + *StackUnderflowHandler
    POP.Stack : Stack -> Stack
    POP.StackUnderflowHandler : Stack -> *StackUnderflowHandler

TOP: Stack -> Item + *ItemUndefinedError
    TOP.Item : Stack -> Item
    TOP.ItemUndefinedError : Stack -> *ItemUndefinedError

ISEMPTY: Stack -> Boolean
BOUND: Stack -> Integer
SIZE: Stack -> Integer
```

{The following are the various "characteristic" predicates for relevant subdomains}

```
P.PUSH.Stack : Stack, Item -> Boolean
P.PUSH.StackOverflowHandler : Stack, Item -> Boolean
P.POP.Stack : Stack -> Boolean
P.POP.StackUnderflowHandler : Stack -> Boolean
P.TOP.Item : Stack -> Boolean
P.TOP.ItemUndefinedError : Stack -> Boolean
```

Semantics

{Note that the functions NEWSTACK and PUSH.Stack serve to generate all instances of the type Stack.}

```
ISEMPTY(NEWSTACK(n)) = TRUE
ISEMPTY(PUSH.Stack(s,x)) = FALSE

SIZE(NEWSTACK(n)) = ZERO
SIZE(PUSH.Stack(s,x)) = 1 + SIZE(s)

BOUND(NEWSTACK(n)) = n
BOUND(PUSH.Stack(s,x)) = BOUND(s)

P.PUSH.Stack(NEWSTACK) = TRUE
P.PUSH.Stack(PUSH.Stack(s,x)) = if SIZE(s) < BOUND(s)
                                then TRUE
                                else FALSE
```

{Note that P.PUSH.StackOverflowHandler(s) = NOT(P.PUSH.Stack(s))}

```
P.PUSH.StackOverflowHandler(NEWSTACK,x) = FALSE
P.PUSH.StackOverflowHandler(PUSH.Stack(s,x),x) =
          if SIZE(s) < BOUND(s) then FALSE else TRUE

P.POP.Stack(NEWSTACK) = FALSE
P.POP.Stack(PUSH.Stack(s,x)) = TRUE
```

```
{Note that P.POP.STACKUNDERFLOWHANDLER(s) = NOT(P.POP.Stack(s))}

P.POP.StackUnderflowHandler(NEWSTACK) = TRUE
P.POP.StackUnderflowHandler(PUSH.Stack(s,x)) = FALSE

P.TOP.Item(NEWSTACK) = FALSE
P.TOP.Item(PUSH.Stack(s,x)) = TRUE

P.TOP.ItemUndefinedError(NEWSTACK) = TRUE
P.TOP.ItemUndefinedError(PUSH.Stack(s,x)) = FALSE


{Now the semantics of the partial functions characterizing POP and TOP}


POP.Stack(PUSH.Stack(s,x)) = s
POP.StackUnderflowHandler(NEWSTACK(n)) =
   StackUnderflowHandler(StackUnderflowHandler(NEWSTACK(n)))

TOP.Item(PUSH.Stack(s,x)) = x
TOP.ItemUndefinedError(NEWSTACK(n)) =
        ItemUndefinedError(ItemUndefinedError(NEWSTACK(n)))


{Now the semantics of the functions PUSH, POP, and TOP in terms of the
partial functions that characterize them. Note that this definition
follows automatically from the definitions given above -- we give this
explicitly only for the sake of completeness.}


PUSH(s,x) = if P.PUSH.Stack(s,x) then PUSH.Stack(s,x)
            else if P.PUSH.StackOverflowHandler(s,x)
                 then PUSH.StackOverflowHandler(s,x)

POP(s) = if P.POP.Stack(s) then POP.Stack(s)
         else if P.POP.StackUnderflowHandler(s)
              then POP.StackUnderflowHandler(s)

TOP(s) = if P.TOP.Item(s) then TOP.Item(s)
         else if P.TOP.ItemUndefinedError(s)
              then TOP.ItemUndefinedError(s)

End Stack
```

Figure 5-1: Bounded-stack

Type UnderflowHandler[Stack]

Syntax

```
POP.UnderflowHandler : Stack -> UnderflowHandler
POP.UnderflowHandler : UnderflowHandler -> String
```

Semantics

for all s *in* Stack:

```
POP.UnderflowHandler(POP.UnderflowHandler(s)) = "STACK UNDERFLOW"
```

End UnderflowHandler

Figure 5-2: UnderflowHandler - 1

Type UnderflowHandler[Stack]

Syntax

```
POP.UnderflowHandler : Stack -> UnderflowHandler
POP.UnderflowHandler : UnderflowHandler -> String
```

Semantics

for all s *in* Stack:

```
POP.UnderflowHandler(POP.UnderflowHandler(s)) =
                CONCATENATE("STACK UNDERFLOW:", CONVERT(s))
```

End UnderflowHandler

Figure 5-3: Underflowhandler - 2

Type UnderflowHandler[Stack]

Syntax

```
POP.UnderflowHandler : Stack -> UnderflowHandler
POP.UnderflowHandler : UnderflowHandler -> Stack
```

Semantics

for all s *in* Stack:

```
POP.UnderflowHandler(POP.UnderflowHandler(s)) = NEWSTACK
```

End UnderflowHandler

Figure 5-4: UnderflowHandler - 3

Type OverflowHandler[Stack]

Syntax
PUSH.OverflowHandler : Stack, Item -> OverflowHandler
PUSH.OverflowHandler : OverflowHandler -> Stack

{Auxiliary functions used by the OverflowHandler}

COPY : Stack, Stack -> Stack

Semantics

for all s1,s2 *in* Stack

PUSH.OverflowHandler(PUSH.OverflowHandler(s)) = COPY(s,NEWSTACK(BOUND(s)+1))

COPY(s1,s2) = if ISEMPTY(s1)
then s2
else COPY(POP(s1),PUSH(s2,TOP(s1))

End OverflowHandler

Type OverflowHandler[Stack]

Syntax

PUSH.OverflowHandler : Stack -> OverflowHandler
PUSH.OverflowHandler : OverflowHandler -> String

Semantics

for all s *in* Stack

PUSH.OverflowHandler(PUSH.OverflowHandler(s)) = "STACK OVERFLOW"

End OverflowHandler

Figure 5-5: StackOverflow Handlers

Type Parser
Syntax
.....

```
PARSE : Grammar, Parse-Stack, String ->
                                        <Parse-Tree, Parser-Msgs>
                                      + *Syntax-Error
```

.....

Semantics

for all g *in* Grammar, ps *in* Parse-Stack, s *in* String
....
PARSE(g,ps,s) = .. if (s is not well formed) then RECOVER(g,ps,s)
....

End Parser

Type Syntax-Error*(Parser-Stack)*

Syntax

```
RECOVER : Grammar, Parse-Stack, String -> Syntax-Error
RECOVER : Syntax-Error -> <Parse-Tree, Parser-Msgs>
```

{This function "recovers" from the error as best as possible, using the existing states of the Parse-Stack, the Input String, and a knowledge of the grammar}

```
EMIT-ERROR-MESSAGE : Syntax-Error -> String
```

{An appropriate error message is emited}

```
PROGRAM-SUFFIX : Syntax-Error -> String
```

{The remainder of the input string is "reset" according to some appropriate error recovery Scheme (see [19] for an example.}

```
PARSE-STACK : Syntax-Error -> Parse-Stack
```

{The parse Stack is re-configured, again in accordance with some error recovery algorithm.}

Semantics

....

```
RECOVER(RECOVER(g,ps,s)) =
 <PROJ(1,PARSE(g,RECOVER(RECOVER(g,ps,s)),PROGRAM-SUFFIX(RECOVER(g,ps,s)))),
  CONCATENATE(EMIT-ERROR-MESSAGE(...kind of error...),
    PROJ(2,PARSE(g,RECOVER(RECOVER(g,ps,s)),PROGRAM-SUFFIX(RECOVER(g,ps,s))))>
```

{PROJ(1,...) and PROJ(2,...) respectively project out the first and second components of a pair.}

Figure 5-6: Skeletal Specifications of an Error Correcting Parser

6. Implementation Considerations

The mechanism described above enables full generality in the specification of exceptions as well as their handling. Goodenough [7] distinguishes between three possible types of exceptions:

1. ESCAPE exceptions, which *require* termination of the operation raising the exception;

2. NOTIFY exceptions, which *forbid* termination of the operation raising the exception;

3. SIGNAL exceptions, which permit the handler to decide whether to resume or terminate the operation raising the exception.

It is a trivial matter to enforce any of these restrictions in a specific implementation, if it is so desired. It is also equally simple to conjure up syntactic extensions for,say, a compiler to check such a restriction.

In addition, it is possible to have standard "default" handlers associated with every type (which are modularly replaceable,) or even "fixed" handlers. The implementation overhead paid to achieve this generality is quite negligible; further, the implementation itself is quite straightforward.

6.1. On Parallelism in Implementations

Although we have mainly been emphasizing the aspects essential to the specification of exceptions and their handlers, the proposed method lends itself to implementation using parallelism that is inherent in the way of specification. It should first be noted that at most one of the predicates that identify the subdomain in which the arguments of a function lie can evaluate to true. As a consequence, there are two immediate ways that the evaluation of a function can be speeded up:

1. In case of a purely sequential evaluation, the predicates can be evaluated in the order of descending probabilities of their being true. (Alternatively, the user may explicitly order the if-then-else's to reflect this order.)

2. All the predicates can be evaluated in parallel: the first one evaluating to true then serves to indicate which computation should proceed next. If the available processing elements are dynamically reconfigurable, then this solution can be quite efficient.

We have observed that the complexity of the "discriminant" predicates $\{P.f.T_i\}$ is usually quite small (compared to the complexity of $f.T_i$. Moreover, there is a lot of commonality that is present in the computation of these predicates: this fact can also be exploited to yield efficient parallel evaluation strategies.

7. Some Comparisons

Since the exception handling mechanism described herein is particularly relevant in the contexts of functional programming and algebraic data type specifications, we attempt a brief comparison with existing proposals for dealing with "errors" in abstract data types.

In [6] errors are handled by having a global error type Error; whenever any function of an arbitrary type has an Error in its argument, the entire term collapses to the single instance of Error of this type. This approach is theoretically sound, but is usually too restrictive (and perhaps unnatural at times) in practise -- it does not provide any practical mechanism for error *recovery* as such. Goguen [5] has developed a theory of "error algebras" wherein operations (and their semantic specifications) are classified as being "ok-operations" ("ok-specifications") or "error-operations" ("error-specifications"). Again, this proposal is theoretically sound, but lacks the flexibility that is required in a general error handling mechanism; in particular, it does not provide adequate mechanisms for actual recovery. Majster [16] discusses a method for specifying errors in algebraic data types that is in a sense similar to ours; however, we believe that allowing functions to return a disjoint union of types provides a convenient way of thinking about exceptions, whereas Majster does not allow for functions to return disjoint union of types. More importantly, she does not address the practical problems inherent in error handling and the issue of how such a mechanism fits into a general programming paradigm.

We summarize below some of the advantages of the proposed exception handling mechanism.

- There is a precisely defined notion of abstraction, and of an implementation of an abstraction.

- The different exceptions, as well as the associated "invocation" conditions and their handlers are clearly visible in the specifications.

- There is a provision for standard "default" exception handlers, (which are *replaceable*) and "fixed" exception handlers.

- It is possible to pass parameters to the exception handling type, thus allowing full generality in the recovery mechanism (allowing an operation to be either resumed or terminated.)

- There is no longer any need for any "dynamic propagation of contexts."

- Disjoint unions of types are allowed as ranges of operations.

8. Conclusions

Most of the existing mechanisms for exception handling have treated the problem as a control flow issue. Because of the fact that exceptions usually arise in a context that is different from the one where they can be most appropriately handled, this has had the unfortunate consequence of creating needless complications due to necessitating the propagation of dynamic execution contexts. As a result, several of the desirable features of the environment in which such exception handling mechanisms are embedded (ease of understanding, locality, block-structuredness of the scope rules, modularity, etc.) are often compromised and flexibility is lost. The paradigm for exception handling that is proposed here differs radically from existing mechanisms in the philosophy it espouses: it views the problem of exception handling as one of satisfactorily allowing for an operation to be specified in terms of the partial functions that characterize it, while providing for *incremental* modifiability of it semantics. A major advantage of this is that the problem of dynamic propagation of contexts becomes non-existent. The proposed mechanism is of particular relevance in the context of abstract (algebraic) data types and functional programming.

References

1. C.Bron et. al. A Proposal for Dealing with Abnormal Termination of Programs. Twente University of Tehnology, 1976.

2. Digital Equipment Corporation. BLISS-11 Programmer's Manual. . Maynard, Mass., 1974.

3. C.M.Geschke et. al. Early Experience with MESA. Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN, March, 1977.

4. J.A.Goguen and J.J.Tardo. An Introduction to OBJ: A Language for Writing and Testing I ormal Algebraic Program Specifications. Proceedings Of A Conference On Specifications Of Reliable Software, IEEE, April, 1979, pp. 170-189.

5. J.Goguen. Abstract Errors for Abstract Data Types. Formal Descriptions of Programming Concepts, 1978, pp. 491-526.

6. J.Goguen, J.Thatcher,E.Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In R.Yeh, Ed., *Current Trends in Programming Methodology, Vol IV*, Prentice-Hall, N.J, 1979, pp. 80-149.

7. J.B.Goodenough. Exception Handling: Issues and a Proposed Notation. *CACM 18*, 12 (December 1975), 683-696.

8. J.J.Horning. Program Structure for Error Detection and Recovery. Proc. Conf. on Operating Systems: Theoretical and Practical Aspects, IRIA, 1974.

9. IBM Corporation. PI/I (F) Language Reference Manual, Form GC28-8201. IBM Corporation, 1970.

10. J.D.Icbiah, J.G.P.Barnes, J.C.Heliard, B.Krieg-Bruckner, O.Roubine, B.A.Wichmann. ADA - Language Reference Manual. *SIGPLAN Notices 14*, 6, PART A (June 1979), 1-1 - 15-12.

11. J.D.Ichbiah, J.G.P.Barnes, J.C.Heliard, B.Krieg-Bruckner, O.Roubine, B.A.Wichmann. Rationale for the Design of the ADA Programming Language. *SIGPLAN Notices 14*, 6 PART B (June 1979).

12. B.W.Lampson, J.G.Mitchell, E.H.Satterthwaite. On the Transfer of Control Between Contexts. Lecture Notes in Computer Science, 1974, pp. 181-203. Springer-Verlag, N.Y. 1974

13. R.Levin. *Program Structures for Exception Handling*. Ph.D. Th., Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1977.

14. B.H.Liskov and A.Snyder. Exception Handling in CLU. *IEEE Trans. on Software Engg. SE-5*, 6 (November 1979), 546-558.

15. B.H.Liskov, A.Snyder, R.Atkinson, C.Schaffert. Abstraction mechanisms in CLU. Tech. Rept. Computation Structures Group Memo 144-1, MIT-LCS, Jan, 1977.

16. M.E.Majster. Treatment of Partial Operations in the Algebraic Specification Technique. Proceedings of a Conference on the Specifications of Reliable Software, IEEE, April, 1979, pp. 190-197.

17. P.M.Melliar-Smith and B.Randell. Software Reliability -- The Role of Programmed Exception Handling. *SIGPLAN Notices 12*, 3 (March 1977).

18. H.J.Kugler, N.Lehmann, P.Putfarken, C.Unger. Project NICOLA, Project Report #3, March 1979, sections 2.6.9, 2.7. . Universitat Dortmund, Abteilung Informatik, Postfach 500 500, D-4600, Dortmund 50, FDG

19. A.B.Pal and R.B.Kieburtz. Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table Driven Parsers. *ACM Transactions on Programming Languages and Systems 2*, 1 (January 1980), 18-41.

20. D.L.Parnas and H.Wurges. Response to Undesired Events in Software Systems. T.H.Darmstadt, 1976.

21. B.Randell. System Structure for Fault Tolerance. *Proc. Intl. Conf. on Reliable Software 10*, 6 (June 1975), 437-449.

22. W.A.Wulf, R.L.London, M.Shaw. Abstraction and Verification in ALPHARD. CMU, ISI, August, 1976.