Thesis Dissertation

"Exception Handling: The Case Against"

Andrew P. Black
(Balliol College)
1982

ABSTRACT

It has recently become apparent that only by striving for the utmost simplicity in programming languages can one hope to produce programs which express their semantics clearly. Thus it is essential to examine any new language feature to see if its contribution to ease of expression outweighs its cost in terms of language complexity.

This thesis examines exception handling mechanisms in this light. Such mechanisms are included in the programming languages PL/I, CLU and Ada, and extensive proposals have been made by Levin. All the mechanisms are "high-level" in the sense that they can be simulated by conventional language features. Their designers offer only the vaguest indication of their range of applicability, and when the motivating examples are re-written without exception handling there is often an improvement in clarity. This is partly because of the reduced weight of notation, and partly because the exception handling mechanisms obscure what is really happening.

The same techniques which produce these improvements in programs can be applied to axiomatic definitions of data types. This finding contradicts the claim of some other workers in data abstraction that exceptions are essential for the description of data types like stack.

The role of exception handling mechanisms in the construction of fault-tolerant computer systems is also examined. It has been suggested that exception handling and fault-tolerance are really different facets of the same problem. However, careful examination of these facilities leads to the conclusion that exception handling is only relevant to anticipated events, whereas fault-tolerance addresses the problem of residual design errors which are of their nature totally unexpected. The very real problems of survival after a component violates its specification is not addressed by exception handling.

ACKNOWLEDGEMENTS

This thesis, both as an idea and in its present form, is the result of varied influences. My debts are many:

To the authors of the Algol 60 Report, for kindling my interest in programming languages.

To Brian Shearing, for proving that both syntax and semantics do matter in the world of commercial software.

To Les Belady and the staff of the Software Technology Project at the IBM T.J. Watson Research Center, who taught me a lot about the problems of modifying real software and provided support for the work on exception handling and data abstraction [10] out of which this thesis grew.

To Tony Hoare, for convincing me that simplicity is a more worthy goal than complexity, and more of a challenge, too.

To Jim Horning, Jim Mitchel and others at Xerox PARC for answering my questions on Mesa and its use.

To the Science Research Council, for providing the studentship which enabled me to undertake this research.

To the members of the Programming Research Group, Oxford, and the members of the Computer Science Department at the University of Washington, for providing constructive criticism, for tolerating my irregular hours and moods, and for providing computing resources.

To my wife Gloria, for typing this thesis (twice), for giving me moral support in its writing, and most of all for being there when I needed you.

Thank you.

CONTENTS

Acknowledgements i

Introduction 1
0.1 The Quest for Simplicity 1
0.2 What is a Simple Programming Language? 2
0.3 Exceptions and Exception Handling 4
0.4 Structure of this Thesis 7

Chapter 1: What are Exceptions? 10
1.1 Cristian's View of Exceptions 10
1.2 Exceptions in CLU 12
1.3 Exceptions as Apology Messages 15
1.4 Levin's View of Exceptions 19
1.5 Exceptions in Ada 21
1.6 Summary 25

Chapter 2: A Historical View of Exception Handling 26
2.1 Exceptions in Fortran and Algol 60 26
2.2 Clearing Up 29
2.3 Exception Mechanisms in PL/I 31
2.4 Exception Handling in Multics 39
2.5 Exception Handling in Algol 68 43
2.6 Conclusion 47

Chapter 3: "Structured" Exception Handling 49
3.1 A Taxonomy for Exception Handling 49
3.2 Exception Handling in CLU 51
3.3 Exception Handling in Mesa 57
3.4 Levin's Proposals for Exception Handling 66
3.5 Exception Handling in Ada 77
3.6 Summary 83

Chapter 4: Exception Handling in Action 84
4.1 The CLU Sum_Stream Example 88
4.2 Levin's Examples 98
4.3 Experience with Mesa Signals 137
4.4 Implementation Efficiency 143
4.5 Conclusion 150

Chapter 5: Exception Handling and Abstract Data Types 152
5.1 Errors and Exceptions in Axiomatic Type Definitions 152
5.2 Defining Partial Operations 153
5.3 The Problem with Error 157
5.4 A Rigorous Definition of Data Type Union 160
5.5 Formal Specification of Two Data Types 164
5.6 Conclusion 167

Chapter 6: Programming without Exception Handling 168
6.1 Programming with Discriminated Unions 168
6.2 Manipulating Procedures in Programming Languages 181
6.3 Conclusion 190

Chapter 7: Catastrophe Handling 191
7.1 Catastrophes, Exceptions, Errors and Faults 191
7.2 Fault-tolerant Computing 193
7.3 Using a Recovery Mechanism for Exception Handling 213

Chapter 8: Conclusion 217
8.1 Exception Handling is Unnecessary 217
8.2 Exception Handling is Undesirable 220
8.3 Exceptions and Catastrophes 227
8.4 On Subjectivity and Proof 229
8.5 Suggestions for Further Work 230

References 232

INTRODUCTION

Much recent research in the areas of programming language design and programming methodology has been devoted to proposing and defining new language features, and indeed whole new languages. This thesis represents a departure from that trend. It *opposes* a language feature, and one which has recently become fashionable: exception handling.

0.1 The Quest for Simplicity

Until the early 1970's language design was considered to be a process of innovation, of creating new and varied features which would make programming easy for a growing user population. The task has now changed. We are coming to realize that "powerful" programming languages can compound problems rather than contribute to their solution. The availability of a large number of features can be a burden instead of a convenience.

It has been convincingly argued that building large programs is intrinsically difficult because of "our inability to do much" [21]: our ability to master complexity is strictly limited. We can hope to solve large and complicated programming problems only by dividing them into manageable parts and by keeping each part as simple as possible. For this effort to be successful our most basic tool, the programming language itself, must also be kept as simple as possible. An unnecessarily complex language bristling with ornamental "features" is an added burden which cannot be tolerated. There is a pressing need for consolidation, not innovation, in programming languages. Simplicity should be pursued not as one goal amongst 118 [95] but as the principal objective of language design.

The need for simplicity has been eloquently argued in [45], [79], [32] and [39]. It is not necessary to repeat these arguments here. Instead I wish to consider their impact on the design of languages and language features.

0.2 What is a Simple Programming Language?

A simple language is not obtained just by minimizing the number of features it contains. It is also necessary to ensure that the included features reflect natural concepts in the mind of the programmer, and to minimize the interdependence of the features. A universal Turing machine provides a lower bound on the number of features necessary to perform computation. Howerver, it is clear that primitives which reflect our thought processes are needed if programs are to be comprehensible.

In designing Algol 68 an attempt was made to keep the number of concepts included in the language small. This was accomplished by generalizing each feature as much as possible, subsuming other features in the process. The trouble with this approach is that the generalized features are foreign to the way most programmers think about problems.

One example will serve to illustrate this point. The concepts of "variable" and "constant" were well understood by programmers in Algol 60 and its successors. Pointers were not so well understood, but they were considered necessary to provide an adequate data structuring capability. However, instead of including a restricted form of pointer specialized for this purpose, "references" were invented and generalized until they subsumed the notions of variable and constant. Instead of

```
const i:complex = c;
var v:complex;
var pc:pointer to complex;
```

Algol 68 uses the declarative forms

```
complex i = c;
ref complex v = loc complex;
ref ref complex pc = loc ref complex; .
```

Three* concepts have been replaced by one: unfortunately this is not a simplification. References are difficult to reason about; making the programmer use them when ordinary variables are adequate complicates his task. It is not even possible to produce a subset of the language (say, for teaching purposes) which does not include references.

The designers of Algol 68 were aware of the dangers of complexity. In order to make the interaction of the various features as tractable as possible, one of their design goals was "orthogonality". Features were intended to be independent, and to combine in regular ways. The mode concept interacts with control structures very regularly: the ordinary *if ... then ... else* is a "void choice clause using boolean"; an integer valued conditional expression is an "integer choice clause using boolean", and so on. Orthogonality is certainly a desirable property for the composition of semantics in a language, and Algol 68 would be even more difficult to understand if it were not so regular. However,

---

* In fact, at least one more important concept, recursive data structures, was avoided too.

regularity is not a substitute for simplicity: both are essential properties of a well-designed language.

The need for simplicity means that any new feature must be examined very critically to determine whether or not it should be included in a language. For example, a choice construct is desirable because it permits enumeration of cases - but guarded commands [22] are preferable to if then else and unlabelled case statements because they make explicit the condition to which each statement sequence corresponds. Procedures and abstract data types are amongst the most powerful tools available for dealing with large problems - precisely because abstraction is one of our most significant mental aids [21].

0.3 Exceptions and Exception Handling

The problem of programs that fail to perform their allotted tasks has always existed. Over the last five or six years a belief has arisen that this problem can be eliminated, or at least significantly eased, by the introduction of language constructs for "exception handling". Such features have been proposed by Goodenough and Levin, and are present in the programming languages PL/I, Bliss, CLU, Mesa and Ada.

I do not believe that exception handling mechanisms are necessary. Primarily this is because there is no single mental process for them to capture. Rather, failures occur because of three problems:

(i) deciding exactly what a program should do;

(ii) writing code to do it correctly;

(iii) relying on an underlying machine which may fail.

There are techniques for dealing with each of these difficulties. Formal specifications and structured programming help us to cope with (i)

and (ii) respectively. By recursive application to the underlying machine they can also reduce (iii), although hardware and software fault-tolerance techniques may be necessary in addition.

I shall argue that attempting to bring these diverse concerns under the common heading of "exception handling techniques" has resulted in the all-embracing definition of exceptions as those conditions brought to the attention of a program's invoker [30], and the invention of exception handling mechanisms powerful enough to subsume the role not only of subroutines but of coroutines and jumps too. It seems likely that such mechanisms will reduce reliability rather than increase it. Not only do they complicate the programming language, they do so in the most critical places. Exception handling mechanisms have been engineered so that their use is economic only when they will be rarely exercised - the very circumstance in which an error of logic is likely to remain undetected. Their use has been urged when there are many cases needing different treatment, but instead of making the handling of each case explicit at the place where it may occur, exception handling mechanisms aim to hide from scrutiny both the existence of the difference cases and the program's response.

This thesis aims to show that unusual and undesired events, if anticipated, can be adequately dealt with in exactly the same way as common and desirable events: by the application of the ordinary constructs of structured programming. On the other hand, the occurrence of an unanticipated event cannot be dealt with at all. That is why one of the programmer's major responsibilities is to anticipate all possible events:

a language feature which pretends to take over that responsibility is both misguided and dangerous.

On practical grounds, too, exception handling seems to be poorly motivated. Many strange and wonderful language features have been proposed over the last twenty-five years. It seems likely that for even the most baroque there is one example which shows them to be useful: the very example which was responsible for their invention. Nevertheless, I have failed to find an example of a programming problem amenable to simple and natural solution by an exception handling mechanism which cannot be so solved without one.

Apart from the lack of real need, there are other grounds for objecting to exception handling mechanisms. Most of the mechanisms proposed to date are badly designed. They impair readability and program proof, introduce problems such as clearing-up, violate modularity assumptions, and in some formulations even allow one process to interfere with another.

Implementation is another problem. The increased cost of a compiler which implements an exception handling mechanism may be small, and may not be significant to the employers of the compiler writers. This is probably true of the compilers for PL/I, the only widely available language to incorporate exception handling. Unfortunately, the indirect costs are much larger, and must be born by the users of such a compiler. It is likely to be larger, slower, more prone to bugs, and less explicit with its error messages. And some of these costs are inflicted on the user every time a program is compiled, even if the exception mechanism isnever used.

My rejection of exception handling mechanisms does not mean that I am entirely satisfied with existing programming languages. Many strongly typed languages do not provide a convenient way for a procedure to return results of different types, and exception handling mechanisms have been used as a way of simulating such results. I will use a type constructor called oneof for producing discriminated unions of types; this enables "exceptional" results to be communicated in the same way as ordinary ones. Human fallibility also forces me to recognize that the totally unexpected will sometimes occur, and that some way must be found of minimizing the consequences of such a catastrophe. There have indeed been proposals for doing so; they have in common the division of the system into compartments which attempt to contain the catastrophe and its consequences. Each nested recovery compartment represents a frame of reference in which progressively more disastrous events are anticipated, but progressively less comprehensive recovery is attempted. Catastrophes are survived rather than handled.

0.4 Structure of this Thesis

Chapter 1 attempts to define the term "exception". The definitions used by the designers of various exception handling mechanisms are compared, but none is found to be satisfactory.

Chapter 2 examines some early methods of dealing with exceptions. The treatment is chronological; it includes methods applicable to Algol 60 and Fortran, the primitive exception handling of PL/I, and exceptions in Algol 68.

Chapter 3 considers the more recent proposals for exception handling mechanisms. They generally provide more structure than do PL/I ON

conditions, but there is great variation between different languages. CLU, Mesa and Ada are studied in detail, as are the proposals of Levin.

Chapter 4 is entitled "Exception Handling in Action". It contains examples which have been used by the designers of various exception mechanisms in an attempt to justify them. I present the original form of the example and a revised version which achieves a similar effect without special purpose language features.

Chapter 5 examines exceptions in specifications of data types. The axiomatic method of type definition is briefly introduced, and used to rigorously define the oneof type constructor used in Chapter 4. 'Stack' and 'SymbolTable' are defined using oneof, illustrating that exceptions and "abstract errors" [27] unnecessarily complicate the problem of data type definition.

Chapter 6 looks at the language features assumed by my examples in Chapter 4. Specifically, various languages are examined to see if the oneof constructor can be conveniently provided. The treatment of procedures as manipulable values is motivated, and the complexity of these features is compared to that of exception handling.

Chapter 7 investigates techniques for dealing with catastrophes, i.e. the totally unexpected. The key idea is found to be containment, and various containment strategies are surveyed. Operating systems and the Newcastle Recovery Block scheme are considered, and the way in which they differ from exception handling mechanisms is brought out. Because the possibility of hardware failure is ever-present, catastrophe handling seems to be an essential part of a reliable system. Exception handling has no such fundamental role.

Chapter 8 summarizes my main arguments and draws the conclusion that exception handling mechanisms are not a desirable feature of a modern programming language.

Chapter 1

WHAT ARE EXCEPTIONS?

This chapter will attempt to answer the philosophical question "what is an exception" and to examine the objectives of recent proposals for exception handling mechanisms. The details of individual mechanisms are deferred to later chapters: here I attempt to isolate the perceived need which they were designed to satisfy.

The starting point for most of the recent developments in exception handling is the 1975 papers of Goodenough [29] [30]. He defines exceptions as those conditions which an operation brings to the attention of its invoker. He claims that it is characteristic of an exception that its significance is known only outside the operation which detects it. Exceptions are not necessarily rarely activated: a procedure may generate exceptions many times in the course of a single call. The first part of Goodenough's definition is of course quite universal: it includes any results or observable effects of the operation. His proposals for exception handling are equally general, and have been criticised on those grounds (see, for example, [99],[68] and [16]). As far as I know there has never been a language which planned to incorporate a mechanism of such power, and it would be unfair of me to launch the case against exception handling by repeating the criticisms of others. It is more appropriate to search for a less general definition.

1.1 Cristian's view of Exceptions

In [16] Cristian attempts to set out rigorous definitions of the concepts involved in exception handling. He states that an exception occurs if the standard precondition of an operation is false when that operation is invoked.

The "standard precondition" of an operation is defined as the weakest precondition guaranteeing the termination of the operation in a state which satisfies its postcondition. A later paper concerned with the automatic detection of exceptions [8] echoes this definition.

An operation may be invoked with its preconditions false for two reasons. First, it is possible that the designer or implementor of the operation documented it incorrectly. It can then happen that although the invoker meets the published precondition, the operation cannot possibly establish its postcondition because the real precondition is stronger than the published one.

The second possibility is that there is a mistake in the code which invokes the operation. It is the duty of the programmer who uses an operation to ensure that it is invoked only from those states which *do* satisfy that operation's precondition. If the programmer is negligent in that duty a programming error occurs.

It is a truism that programming errors should be corrected, not "handled". If the programmer is aware of the mistake then of course he should correct it. If, as too often happens, he is unaware of the error then he cannot correct it, but neither can he handle its effect. Before he can even consider writing an exception handler he must be aware that the exception can occur, which implies that the generation of an exception signal must be part of the specified behaviour of the operation. But, by definition, no exception occurs when the operation is used according to its specification!

1.2 Exceptions in CLU

The language CLU (see Section 3.2) incorporates an exception handling mechanism. The reference manual [61] states that an exception occurs when a routine cannot complete the task it is designed to perform. Superficially this definition is very like that of Cristian; provided that the precondition of a routine is satisfied it should always be able to complete its task. However, the examples in the manual and the informal nature of the CLU definition make a more general interpretation possible.

It is the duty of the designer of an operation to ensure that it is not impractical or impossible. One might imagine a client going to a software designer and asking for the provision of a 'table' with two operations; 'table := Insert(name, data, table)' and 'data := LookUp(name, table)'. As a responsible professional the software engineer ought to tell the client that the 'LookUp' operation cannot be implemented in its full generality: 'data' can only be returned if the 'name' has been inserted into the 'table'. 'LookUp' must be guarded by a precondition to this effect. The client may therefore decide that he needs an operation 'bool := IsIn(name, table)' so that this precondition can be checked. The engineer should then advise the client that in the interests of efficiency he should prefer an operation 'LookUpIfPossible' which returns either the 'data' or an indication that the name was not in the table.

Now let us examine again the statement from [61]. "An exception occurs when a routine cannot complete the task it is designed to perform." The task of 'LookUpIfPossible' is always possible: a professional was employed to ensure this is so. However, it is not what the client wanted; he wanted 'LookUp'. From *his* point of view an exception occurs when 'name' is not in 'table'. From the subjective viewpoint of the client we can indeed find situations where a routine cannot complete its task. From the objective viewpoint of the software engineer there is no such situation, or at least he has tried to ensure that there is none.

There are many examples of such "subjective" exceptions: taking the top of an empty stack, reading from an empty file and dividing by zero immediately come to mind. Each such operation can be realized in two ways. The domain of the operation can include "holes", i.e. there can be values for which the operation is not defined. This is the way division is usually treated: it is not defined if the divisor is zero. Alternatively, the range can contain a "bump", an extra value of a distinct type. It is possible to define a (different) division operation on all pairs of integers whose range is the union of the rational numbers and the distinguished value 'ZERO DIVIDE'. Figure 1.1 illustrates the examples. It is worth emphasizing that there are no *errors* involved. The versions with the holes in the domain are perfectly acceptable from a theoretical point of view; they are also eminently practical providing that the domain can be checked cheaply. This is so with division, for example. They become impractical (but not wrong) when the domain is expensive to check, as with 'LookUp'.

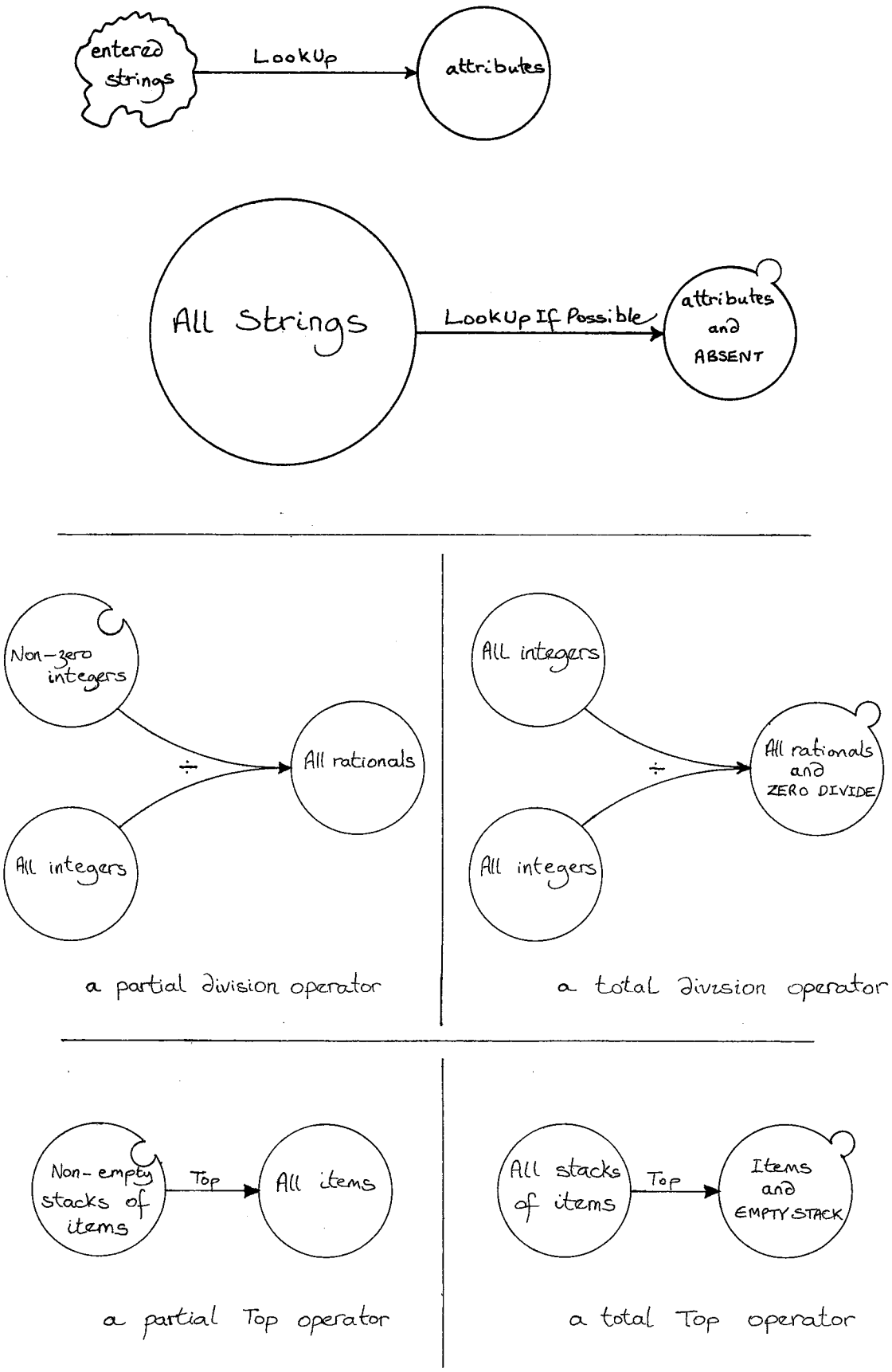

Figure 1.1: Partial and Total Versions of Three Functions

This, then, is one definition of exception: the "inconvenient" result of an operation. It should be remembered that it is totally subjective; it may nevertheless be quite useful.

1.3 Exceptions as Apology Messages

Another possible interpretation of exception was hinted at above. There may be situations where a routine *cannot* complete its task because there are insufficient resources available. These situations will arise despite the best efforts of the software engineer: they are the inevitable consequences of attempting to implement an infinte abstraction on a finite machine.

The infinite set of integers, easy to define and familiar in use, cannot be represented on a computer which can assume only a finite number of states. The same applies to many convenient abstractions: unbounded tables, queues and stacks cannot be implemented on finite computers.

What should the conscientious software engineer do when his client asks for an implementation of an unbounded table? Unlike the LookUp of a not-yet-inserted item, there is nothing wrong with the abstraction. Nevertheless, he must tell his client that this clean simple and desirable abstraction is impossible to implement, and offer one of two unattractive alternatives. The first is a different abstraction, such as a bounded stack or a finite set of integers, which is not only not what the client wants but is also clumsier and more complicated to use. The second alternative is a partial implementation of the desired abstraction, an implementation which will work often enough to satisfy the client but which will sometimes fail and emit an appropriate apology message. The generation of an exception can be viewed as just such an apology message.

The earliest reference I have been able to trace which considers exceptions in this way is [74]. This paper commences "During the execution of a program a number of exceptional conditions can arise, not from program defined action, but as a result of exceeding some computer limitation". This is curious because the paper goes on to consider exception handling in PL/I; of the eighteen exception conditions listed only one arises from a computer limitation.

Here, then, is another definition: an exception is an admission by the implementation of an abstraction that it cannot comply with its specification. This definition is objective provided that one has a specification describing the required behaviour. A particular implementation may satisfy a restricted set of axioms but fail to satisfy a more demanding specification.

An implementation of any programming language incorporating infinite abstractions may benefit from a mechanism for saying what to do when the implementation is insufficient. An exception handling mechanism designed for and restricted to this purpose might be a useful adjunct to an infinite language. Unfortunately, except for the short paragraph in [74] cited above, exception handling mechanisms have been neither designed nor advertised on the basis of this usage. Even Goodenough's elaborate scenarios [30] do not include the use of exceptions for signalling implementation insufficiencies. More recently Levin and others at Xerox Palo Alto Research Center have classified the use of the Mesa signalling mechanism and have noted that one use is to report failure of an implementation.*

---

* Private communication.

Nevertheless, it is in general fair to say that no mechanism has yet been designed to deal specifically with the problem of implementing infinite abstractions on finite machines. The exception handling mechanisms that will be examined in this thesis do not address the problem. Because they are part of the programming language and may therefore invoke unbounded abstractions, they are themselves unimplementable in general. For example, an exception handling mechanism would not be useful for dealing with overflow of the run-time stack used for procedure calls. Where could one put the linkage to the exception handler and the routines it calls? Moreover, while exception handling mechanisms are designed for dealing with results that can only be interpreted *outside* of the operation that detects them (see Goodenough's remark quoted above), it is characteristic of an implementation insufficiency that its details are meaningful only *inside* the implementation of the abstraction. Effective recovery from an implementation insufficiency is possible only if information about the implementation is available. Yet it is the very purpose of the abstraction to hide that information.

To illustrate this point, consider the abstraction "integer". Given the external information that the implementation of integers broke when adding 32 760 to 10, there is nothing that the user can do. Only if he is aware that the integers are represented as a row of sixteen binary digits according to the two's complement convention will he be able to determine if useful computation can be continued with -32 766. A knowledge of the external specification (Peano's axioms) alone is insufficient.

The problem is essentially the same for programmer defined abstractions. It is obviously incumbent on an implementation to apologise

when it cannot append a value to a list because it has run out of store. However, it is not clear that the user module can do anything useful when the apology is received. The lists module has failed to comply with its specification, which is the only basis on which the user module can act. Perhaps then a mechanism for dealing with implementation insufficiencies should allow for the *generation* of exceptions (i.e. apology messages) but should not provide, within the programming language, any mechanism for *handling* these exceptions and continuing execution. This is consistent with conditional correctness. Providing that the program terminates normally its results will be in accord with its specifications; however, should the program fail to terminate normally, either by looping indefinitely or by generating an apology message, then nothing may be assumed about any result it may produce.

This view of a program as an object which possibly may fail in an unpredictable way is one that has traditionally been adopted by operating systems. Various techniques for continuing meaningful computation have been developed; this topic is given further consideration in Chapter 7. As was indicated in the introduction, these techniques are essentially different from those of programmed exception handling, which deal with anticipated failures.

E.C. Hehner* has suggested another approach to the problem of implementation insufficiency. Given a particular program and a particular implementation, it is possible to supply implementation dependent annotations which instruct the implementation how to proceed when it cannot comply with the axioms.

---

* Private communication.

For example, if a program manipulates rational numbers which are implemented with binary floating point binary, the annotations could tell the implementation to use an approximation when it cannot represent the result of dividing one by three. This particular example shows that abandoning the computation is not always the appropriate response to an implementation insufficiency. Perhaps implementation dependent instructions on how to continue cannot be avoided.

Nevertheless, the notion of extensive annotations is intellectually unappealing. The problem of dealing with approximate arithmetic is sufficiently important for numerical analysts to have developed techniques for minimizing and estimating rounding errors. It is quite reasonable for a programming language to provide as an abstraction numbers with a finite accuracy and bounded range; the numerical analyst can then study the propagation of rounding errors within the programming language. This argument can be generalized to the claim that whenever one would require continuation after the detection of an implementation insufficiency one also needs a theory in which to reason about the validity of the results. The best way of obtaining this is to alter the specification so that the "implementation insufficiency" is now part of the expected behaviour. In other words, the unimplementable abstraction must be given up in favour of a more modest implementable one, even if it is less convenient.

1.4 Levin's View of Exceptions

Levin's thesis [59] deals solely with the subject of exception handling, and proposes a new and powerful mechanism for incorporation into programming languages. Nevertheless, he is forced to admit that he failed to find a definition of "exception". He rejects the common guideline that an

exception is a rarely occurring event, justifying this stand with the example of looking up a name in the compiler's symbol table. There are two possible results: name absent and name present. Which of these is more frequent depends on context. Why should one result be considered as an exception when the other is not? Levin's solution to this dilemma is to treat both results as exceptions.

Levin also rejects the identification of exceptions with errors, although he includes errors as a proper subset of exceptions. However, he does not define the term *error*. He is also willing to classify both input/output interrupts and other interprocess communication as exceptions should this be convenient.

How then does Levin distinguish exception handling from other techniques of program construction? He offers the guideline that exception handling is to be preferred when a programmer wishes to "play down" the processing of a particular case in order that another case may be emphasised. Of course, in the end this comes down to a matter of taste. Levin considers that this lack of a firm boundary increases the applicability of his mechanism and is therefore desirable.

It is clear even from the title of this thesis that my views differ from those of Levin. As far as taste is concerned, mine is to prefer to see explicitly where different cases may occur; as Levin points out, guarded commands [22] express this adequately. (So do ordinary conditional statements.) In my view it is not surprising that Levin failed to formulate a satisfactory definition of exception: exceptions cannot be defined because they do not exist as an abstract concept. I am not claiming here that Levin's idea of separating the treatment of certain

rare cases from that of other, more common cases is never helpful. However, some works in this field elevate errors and exceptions to the status of an abstract and fundamental concept. In particular, *Abstract Errors for Abstract Data Types* [27] defines a whole mathematical system called "Error Algebras" for dealing with "exceptions" such as looking for a name which is not in a symbol table. Although the theory is complicated it does not permit recovery from exceptions. A simplified way of dealing with such situations is described in Chapter 5.

Another paper which suggests that "exception" is an abstract notion is [67], which provides an axiomatic description of some uses of the Ada exception handling mechanism. The authors find it necessary to state that "Our specifications and proof rules apply to programs with exceptions regardless of whether exceptions are used only for error situations or as a method of programming normal program behaviour." It is as if they believe that it is possible to write axioms which distinguish the two cases!

1.5 Exceptions in Ada

In view of the probable economic importance of the Ada programming language it is pertinent to ask how its designers interpreted the term "exception". The details of the particular mechanism they adopted are discussed in Section 3.5; we are here concerned with the more philosophical question "What *is* an Ada exception?".

The rationale for the design of Ada is described in [55]. Section 12.1 states that exception handling "provides a facility for local termination upon detection of errors". Exception handling should be restricted "to events that can be considered (in some sense) as errors, or at least as

terminating conditions". However, the term "error" is not defined, and it is clear from the context that the last phrase means only that an exception will lead to the termination of the current invocation.

Section 7.2, which discusses the implementation of parameter passing, gives another clue as to the interpretation of an exception. Parameters in Ada can be implemented either by reference or by copying: in *normal* situations the semantics are identical. The case of a subprogram terminated by an exception is classified as *abnormal*: the fact that out parameters may or may not be updated is stated to be of no importance. The revised definition of Ada [96] retains this rule (Section 6.2) and adds more concerning optimization in the presence of functions which may generate exceptions (Section 11.8). Specifically, operators whose result values depend only upon their arguments may be invoked as soon as those arguments are known, "even if this invocation may cause an exception to be propagated". "The operation need not be invoked at all if its value is not needed, even if the invocation would raise an exception."

When promulgating these rules the designers of Ada seemed to have in mind expressions such as 'a or fun(b)'; they wished to permit the invocation of 'fun(b)' to be omitted when 'a' is true even if it could raise an exception. This is consistent with an interpretation of exceptions as indicating an implementation insufficiency. Suppose that the invocation of 'fun(b)' would require more resources than are available and would thus generate an exception indicating this fact. Clearly an implementation which avoids invoking 'fun(b)' and thus produces the correct result instead of an apology message is to be applauded. Omitting the invocation is *not* sensible if exceptions represent essential results. It

therefore seems that exceptions in Ada are intended as a means of indicating implementation insufficiencies.

Unfortunately the examples given in the Report [96] do not observe this intention. Section 12.4 presents a stack package with procedures 'Push' and 'Pop' but no predicate 'IsEmpty'. Instead the 'Pop' procedure may raise the exception 'Underflow' if it is applied to an empty stack. 'Underflow' is thus an essential part of the result of 'Pop'. On the other hand, the 'Push' procedure may raise the exception 'Overflow', which presumably indicates an implementation insufficiency. (One cannot be sure because the semantics of the package are not specified.) These two exceptions representing such different concepts are generated with the same syntax and are declared on the same line.

Section 1.6 of the Ada reference manual also provides some clues about the meaning of "exception". It states that Ada recognizes three categories of error. The first are errors which must be detected at compile time; it seems clear that this category includes syntactic errors, such as real numbers with multiple exponents and constant objects on the left-hand side of assignments. Of course, sequences of characters which violate the Ada syntax in this way are not Ada programs at all. However, such "non-programs" are often informally referred to as "programs with errors", and it would be churlish to object to this usage. Indeed, it is the only guide to the meaning of "error" offered by the manual, so I will assume that an error is that which divides programs from non-programs.

The third category of error is consistent with this interpretation. "The language specifies certain rules that must be obeyed by Ada programs, although Ada compilers are not required to check that such rules are not

violated." Programs which violate the rules of Ada in this way are said to be "erroneous", and the effect of executing them is not predicted by the reference manual.

The remaining category contains "errors that must be detected at run time. These are called exceptions". These phrases lead very rapidly to a contradiction. If the interpretation of "error" derived above is indeed correct, then exceptions can only occur in non-programs. And yet exception handling certainly seems to be a part of the Ada language: a whole chapter of the reference manual is devoted to it! But if exceptions can be generated by valid Ada programs, what is an error?

The purpose of a programming language definition is to determine unambiguously both the set of character strings which represent programs in the language and the meaning of those programs. With its references to "erroneous programs" and "exceptions" the Ada reference manual fails to do these things. In the scope of the declaration

```
i : integer range 1..10
```

is 'i := 15' an Ada assignment? No, because '15' does not satisfy the range constraint of 'i', which the manual says it *must*. Or yes, because the effect of this assignment is to raise the exception 'constraint_error'.

I am thus unable to answer the question which introduced this section. It seems that the designers of Ada did not have a very clear idea of what an exception ought to be. None of the possible interpretations are consistent with the text and examples of the reference manual. The result of this is not just ambiguity over whether certain texts are Ada programs; as will be demonstrated in Section 3.5, the very semantics of legal programs can sometimes be surprising.

1.6 Summary

It is clear that exceptions have several unrelated uses: announcing programming errors, apologising for implementation insufficiencies, delivering "unusual" results and achieving interprocess communication are the examples mentioned in this chapter. Perhaps a mechanism which could serve all of these purposes would be useful. Very general mechanisms have appeared in the history of programming languages; the goto statement is an obvious example. Gotos, like exceptions, can be used to simulate many language features. There is another similarity, too: both exceptions and gotos are provided by the hardware (as interrupts and jumps).

It is no discredit to the pioneers of programming to have investigated the properties of such universal mechanisms. But the goto belongs to an earlier age. We are no longer seeking mechanisms which do as many different things as possible; rather we are trying first to decide what we want to do, and then looking for ways to do exactly that.

The designers of exception mechanisms have not been very successful in their quest for the all-embracing. A run-time exception handling mechanism is clearly inappropriate for dealing with mistakes which occurred when the program was written. Insufficiencies of implementation cannot of their nature be "handled" by a mechanism which is part of the programming language. Exception mechanisms have been devised which deal with unusual results and interprocess communication, but such mechanisms can only be justified on the basis of a useful commonality between these things, and then only if they are simple to understand and implement. Exceptions may be a useful design notion, but such utility must be demonstrated, not assumed.

Chapter 2

A HISTORICAL VIEW OF EXCEPTION HANDLING

This chapter attempts to put the recent flurry of interest in exception handling in perspective. Inconvenient results and undesired events are not new phenomena; it is interesting to see how they were dealt with before the invention of special purpose exception handling mechanisms. Algol 60 and Fortran will be used to illustrate the techniques available.

One of the first languages to provide a specialized exception handling mechanism was PL/I. It was developed by the IBM Corporation in the 1960's, although it was not standardized until 1976. The PL/I exception mechanism was clearly conceived as a means of trapping a varied collection of language-defined run-time errors. The deficiencies in the PL/I mechanism are noted and the extensions provided by the implementors of the Multics operating system are discussed.

The other major language of the 1960's was Algol 68. Whereas PL/I provides a large number of specialized constructs, each intended for a specific application, Algol 68 offers a small core of very general facilities, with no mechanism specifically designed for exception handling. The methods used to handle undesired events are examined.

2.1 Exceptions in Fortran and Algol 60

Algol 60 and Fortran do not include any mechanism specifically designed for exception handling. Hill [44] considers the problem in the context of a function of 'A' with an integer argument 'n'. He assumes that 'A' is only defined for 'n ≥ 0', and that the legality of 'n' is checked in the body of

the function. If the check fails, what action should be taken? This problem is quite fundamental: the idea that certain things are not defined cannot be expressed in an algorithmic language. Undefinedness can only be interpreted as freedom to do anything the implementor sees fit. Thus, if Hill's question is understood in its strictest terms, any answer is valid. Having stated that 'A' has a limited domain, if a user is foolish enough to invoke it outside that domain he cannot blame anyone but himself for the consequences. However, since programmers are human and mistakes do happen, the reasonableness of the action should 'n' be negative is a legitimate engineering concern.

One possible action is to complete the function 'A' by decreeing that in the case 'n < 0' the result is 'x'. Then the function implemented is not 'A' at all but a new function 'A'', where 'A'(n)' = 'A(n)' if 'n ε Dom(A)', and 'x' otherwise. Nevertheless, all correct calls of 'A' can be replaced by calls of 'A'' without any change in the action of a program containing them. But what of the incorrect calls? If 'x' is distinct from all the valid results of 'A' (i.e. 'x ∉ Ran(A)') then the result of 'A'' can be tested. This has the disadvantage that the programmer must remember to perform the test; he is no more likely to do that than to remember to test the precondition 'n ≥ 0'! Hill also points out that if 'A'' is called within a complex expression the result may be impossible to test. Furthermore, if 'Ran(A)' is the whole of the result type, no suitable 'x' is available.

If the requirement of functionality is dropped when the value of 'n' is outside the permitted range, many more courses of action become available. Hill lists the following possibilities:

(i) Terminate the program after printing a message, as is usually done for standard functions like square root.

(ii) Add an extra output parameter. A Boolean result can be used to indicate the success of the call. If several errors are possible an extra integer result can be used to indicate which error occurred. However, some value must still be returned as the result of the function.

(iii) Use a Boolean function which has as its result the validity of 'n', and sets an output parameter to the value of the required function 'A(n)' only if 'n ε Dom(A)'.

(iv) Add a label parameter to 'A', and jump to that label if 'n' is invalid. This is allowed in Algol 60 [73] [19] but not in ANSI Fortran [3].

(v) Add a procedure parameter to 'A', and call that procedure if 'n' is invalid. Of course, control returns to 'A' when the procedure completes, and one of the other possibilities listed above must then be adopted. However, the choice of what to do can be influenced by results returned by the procedure. It is also possible, in Algol, to include in the procedure a non-local goto and thus not to return at all.

Hill prefers method (iv), even though he is opposed to the widespread use of goto statements: "The shock of a goto, jumping away from the scene of action to start picking up the pieces elsewhere, seems to me to be just what is required..." My view depends on whether it is easy to test if the parameters are in the right domains. If they are, then (i) seems to be most appropriate. The objection that this precludes recovery is not well

founded, for it is easy to test the parameter before calling the function and to program an appropriate recovery action at that point. If it is not easy to test the precondition then the function should be generalized until it is - using "impossible results", or methods (ii) or (iii).

A variant of method (v) was used extensively in the AED Free-Storage Package [81]. In this system is was possible to partition the available free store into several *zones*, which might themselves be partitioned into sub-zones and so on recursively. Each zone obtained a cache of free store from its parent, which it then distributed to its users. Sometimes a zone might not contain enough storage to satisfy a request, a typical "exceptional" situation. This and similar eventualities were dealt with by calling a so-called *Help Procedure*, a procedure supplied by the user, the integer result of which told the storage system how to proceed. Various primitive routines were available so that the Help Procedure could, as a side effect, manipulate the zone and attempt to cure the problem.

One of the significant things about a Help Procedure is that it was not associated with a particular call on the package but with a zone of storage. A zone was established by calling a procedure whose parameters defined the attributes of the zone, including the Help Procedure. Help procedures were associated with the *data* abstraction of storage zone, not with the *control* abstraction of requesting a piece of store. This distinction appears again in connexion with Algol 68 transput (see Section 2.5), and in the proposals of Levin [59] (see Section 3.4).

2.2 Clearing Up

Before going on to describe the exception handling mechanism of PL/I, it is necessary to digress a little and examine one of the problems that

mechanism fails to solve. It often happens that a program initiates an activity which requires completion even if the program fails. Indeed, it is a major function of an operating system to restore itself to some standard state after a user program has run, regardless of whether that program worked correctly. This involves *clearing up* the arbitrarily complex mess the user program left behind.

Clearing up is also necessary on a smaller scale. Any routine which allocates storage,initiates a peripheral transfer or indicates in some global fashion that it is active, must ensure that the storage is returned, the transfer completed or cancelled, and the global indication reset before it terminates. There is normally no difficulty in achieving this provided that termination is always voluntary, i.e. due to completion of the body of the routine or execution of a normal return. Problems arise, however, if it is possible for a routine to be terminated *abnormally*. This can occur if a call is made to a procedure at a lower level which, on detecting some undesirable event, unilaterally decides not to return. Even if the calling routine is aware that this may happen, it is inconvenient, expensive and sometimes impossible for order to be restored before calling every such procedure.

Abnormal termination can only occur if the programming language allows some form of non-local goto statement. In Fortran [3] a sub-program or function must return to its caller (unless it executes a STOP statement, which causes execution of the entire program to be concluded). In Algol 60 it is possible to jump to a label in any statically enclosing block; it is also possible to pass labels as parameters through an arbitrary number of procedure calls.

Of Hill's proposals for dealing with faults in functions, both passing labels and procedures as parameters (iv and v) may give rise to the clear-up problem if the overall effect is to jump from a routine body to a scope outside that which called the offending function. The language PL/I provides a mechanism for trapping exceptions which all but compels the programmer to use such non-local jumps.

2.3 Exception Mechanisms in PL/I

PL/I was the first general purpose programming language to include specific facilities for handling exceptions, known in PL/I as *conditions*. It is possible for programmers to define their own conditions, and there are twenty language-defined conditions which are detected either by the hardware of the computers for which PL/I was designed, the software of the language implementation or the library of built-in functions. This description is based on [68] and [4].

A programmer designates what action is to be taken on the occurrence of an exception condition by dynamically establishing a so-called *on-unit* as a handler for that condition. When a condition occurs during the performance of some operation, the operation is interrupted and the most recently established on-unit is invoked; if no on-unit has been established a default action is taken.

Exception handlers are called on-units because they are established by the use of the ON statement. The handler has the syntactic form of a begin block or a single statement (in practice usually a call statement). Examples (after [68]) are

```
ON UNDERFLOW  PUT LIST('Underflow in Tabulate');

ON OVERFLOW   BEGIN;
                  CALL P;
                  GO TO L;
              END;
```

The effect of the ON statement is constrained by the block structure of the language (where block means begin block or procedure block). An ON statement establishes the given handler for the named condition until the current block is left; at that time the handler for the dynamically enclosing block again becomes current. The enclosing handler can also be re-established explicitly: the REVERT statement is used for this purpose. If more than one ON statement for the same condition occurs in any block, the previously established handler becomes inaccessible; the REVERT statement cannot be used to reinstate an old handler established in the current block. Thus the ON statement has some of the properties of a declaraction (such as lasting only until the end of the block) and some of the properties of assignment (such as being able to completely obliterate a previous value).

The distinguishing feature of PL/I exception handling is captured in the phrase "dynamically enclosing block", i.e. the block which called the current block. For a begin block this is, of course, its textually enclosing block. However, a procedure block may be called from many different places and may thus be provided with different handlers for its conditions on each invocation. Moreover, those handlers cannot be determined from a static scan of the program, but only from its simulated execution. PL/I condition handlers thus behave rather like free variables in Lisp. This is remarkable because such dynamic inheritance is not used in any other part

of the language. PL/I free variables follow normal Algol scope rules, and even condition prefixes (discussed below) have static scope.

Conditions in PL/I are divided into various groups. Computational conditions occur during evaluation of an expression or assignment. Examples are SUBSCRIPTRANGE, which occurs if an attempt is made to access a non-existent array element, ZERODIVIDE, and STRINGSIZE, which occurs if an attempt is made to assign a string to a variable of insufficient length.

Input-output conditions are always associated with a particular file as well as with an action. Examples of input-output conditions are ENDFILE (an attempt to read beyond the end of a sequential file) and UNDEFINEDFILE, which occurs when an external file satisfying the requirements of the program cannot be found. When on-units for these conditions are established a file must be named, i.e. one must write

```
ON ENDFILE (file) Action;
```

and not

```
ON ENDFILE Action;
```

Incidentally, the ENDFILE condition is the *only* way of detecting the end of a sequential file in PL/I.

There are some other conditions mentioned in the language definition, such as AREA, signalled when a storage area is too small to accommodate a demand for based storage, and ERROR, which the implementation may signal when an illegal program is detected. However, when an implementation-defined limit is exceeded the PL/I standard guarantees only that one or more of a list of ten conditions will be raised.

Each language defined condition has a default handler, which is invoked

if the programmer does not establish his own; the default handler can also be established explicitly be writing

```
ON condition SYSTEM;
```

The handlers for some conditions are allowed to return control to the interrupted operation while others are not. For example, the UNDERFLOW condition arises if the result of a floating point operation is too small to represent. If the handler for this condition returns, evaluation is continued with zero as the result. The default handler for UNDERFLOW prints a message and then returns. Conversely, the handler for FIXEDOVERFLOW is not allowed to return.

The programmer may define his own conditions and then specify handlers for them with ON statements. Such conditions must be explicitly raised with the SIGNAL statement. It is not possible to associate programmer-defined conditions with data structures in the way the ENDFILE condition is associated with files. MacLaren [68] comments as follows on this aspect of PL/I:

> Allowing programmer defined conditions is a natural way to unify the treatment of exceptions. However a programmer defined condition is really nothing more than a procedure variable with the following peculiar properties:
>
> Assignment to the variable in one block activation has no effect on assignments in previous block activations. The values assigned to the variable must be procedures with no parameters. Assignments are made by ON and REVERT statements. The only way to access the value of the variable is by a SIGNAL statement.
>
> Clearly a programmer can handle his own exceptions more flexibly by using the normal procedure variables provided in PL/I.

Various *ad hoc* facilities complete the PL/I exception handling machinery. There are a few global variables which the programmer can test within a condition handler to discover more about the cause of the exception. For example, the CONVERSION condition occurs during string to numeric

conversion when the string does not have the syntax of a numeral. The PL/I pseudo-variable ONSOURCE contains the offending string; moreover, its value may be changed in an attempt to repair the fault. However, the exact effect of such an assignment is an area in which implementations tend to depart from the PL/I standard (see [68]).

Condition prefixes provide a way of "disabling" certain conditions, generally the computational conditions. If a condition occurs and it is enabled, the appropriate on-unit is invoked. Disabling a condition is equivalent to asserting that it will not occur. It follows that any such occurrence is a failure and that further execution is undefined. The point of this rule is to enable expensive checking (such as that of array subscripts) to be turned off: if the programmer has asserted that a condition will not occur, the implementation is relieved of any responsibility if it does. OVERFLOW is enabled by default because most hardware will trap it automatically; SUBSCRIPTRANGE is disabled by default because it is usually detectable only by software checks.

Condition prefixes are nothing more than compiler directives which are affixed to individual statements or blocks and control the generation of checks. Their scope is thus static, contrasting with that of ON statements. For example, in

```
(NOOVERFLOW): BEGIN;
                  ON OVERFLOW CALL overerror;
                       .
                       .
                       .
                  x = x + v
  (OVERFLOW):     a = b * c;
                  CALL P(x + a)
                       .
                       .
                       .
              END;
```

the overflow condition is disabled throughout the begin block except in the statement labelled (OVERFLOW): occurrence of overflow during the execution of 'a = b * c' will result in the invocation of the procedure 'overerror', as directed by the ON statement. Whether overflow is enabled during the execution of 'P' depends on the condition prefix attached to the body of 'P', which may be separately compiled. If 'P' neither disables OVERFLOW nor establishes its own condition handler, occurrence of overflow in'P'will also cause 'overerror' to be invoked.

A common misconception is that PL/I on-units provide the ability to trap hardware interrupts. They do not. It is essential to the language that conditions occur only at discrete points. At all such points the "machine" of the PL/I language definition must be in a well-defined state, so the condition handler can manipulate both program variables and the hidden variables of the implementation. If an interrupt occurred while the system was performing an "indivisible" action, such as altering the state of the free store system, and arbitrary user code were to be executed in response, the effect could not be predicted.

As one of the first languages to include a specialized exception handling mechanism, it would be surprising if PL/I was entirely satisfactory in this respect. Notions of what are and what are not good programming practices have changed since PL/I was designed. The result is that the exception handling facilities of PL/I are at variance with current ideas of program structuring.

Any PL/I procedure inherits from its caller a nest of handlers for various conditions. The consequences of the occurrence of a given condition in a procedure cannot be determined by looking at the text of that

procedure and its actual parameters. They depend on statements executed in other, possibly separately compiled, procedures, and may vary from one call to the next. The only way to avoid this dependence is to include an on-unit for every conceivable condition in every procedure: even then it is possible that the inconceivable may occur. The current emphasis on reliability requires that occurrence of the totally unexpected should cause an alarm (such as an error halt); this cannot be achieved in PL/I.

The dynamic nature of on-units makes each condition behave like a global variable, or more precisely like a global stack. The argument usually put forward in defense of this mechanism is that the invoker of a library procedure should be able to treat a condition arising in that procedure in the same way as when it arises in his own code. This may indeed be desirable provided the condition has one specific meaning, as is the case for a few of the language defined conditions such as ENDPAGE. But for the rest, and for programmer defined conditions, the meaning of a condition raised by an alien subroutine may be completely different from the meaning of a condition of the same name raised by the rest of the program. I cannot agree that the provision of a single handler is desirable *a priori*.

Trying to establish different handlers for different occurrences of the same condition is very cumbersome. MacLaren gives the example of a program with several GET statements which should take different actions if they encounter the end of the input file. Each one must be prefixed by an 'ON ENDFILE (f) ...' statement which sets up the appropriate handler, and probably postfixed by a 'REVERT ENDFILE (f)' statement too. This is expensive to implement and obscures the function of the program.

These problems may be avoided by establishing one condition handler

for 'ENDFILE (f)' which sets a Boolean variable 'end_of_file_F', and testing this after each GET. Assuming that the correct handler is always invoked, and that the programmer remembers to check the state of 'end_of_file_F', all will be well; but this is a cumbersome way of implementing a predicate that could have been provided by the input system at little cost.

Another problem with PL/I on-units is finding a sensible action for the condition handler to take. First, there is a list of *ad hoc* restrictions on what a condition-handler can do. The restrictions vary from one condition to another; they are intended to permit the generation of efficient code. (This is discussed in detail in [68].)

Second, one should note that, unless special precautions are taken, a condition handler is inside its own scope. Should the same condition occur inside the on-unit, the on-unit is invoked again recursively. "Taking special precautions" is quite inconvenient. The following solution is after MacLaren.

```
BEGIN;
    ON OVERFLOW
    BEGIN;
        ON OVERFLOW GOTO recursive_overflow_handler;
             /* Remainder of the overflow on-unit */
    END;

    GOTO start;

recursive_overflow_handler:
    REVERT OVERFLOW;
    SIGNAL OVERFLOW;
start:
        /* Main Part of Block */
END;
```

Such contortions are very difficult to follow.

Lastly, some method must be found to terminate the handler. One way of doing this is by the execution of a STOP statement, which terminates

the entire program. This could be appropriate if the condition were catastrophic. Another possibility is executing the END statement which terminates the handler; this implies a return to the context which raised the condition. While it may be reasonable to return after some conditions (such as an ENDFILE which has been handled by setting a variable), it is sometimes desirable to abandon the operation which raised the condition; moreover, a return is prohibited after eight of the language-defined conditions. In these cases, assuming that one does not want to STOP, it is necessary to execute a GOTO statement and to transfer control to the block activation that established the on-unit, or to a scope enclosing it.

Such a transfer of control is potentially non-local, and may therefore cause an arbitrary number of procedure and block invocations to be abandoned. This obviously gives rise to the clear-up problem described in Section 2.2. The implementation will presumably clear up established on-units and stack frames, but PL/I does not provide a mechanism which permits a programmer to do his own clearing up of files or off-stack storage.

2.4 Exception Handling in Multics

The Multics system was developed at M.I.T. as a research project intended to examine how a reliable operating system with a multitude of user interfaces could best be constructed. It was determined at the start that the system should be written in a high-level language and be published [14]. A machine independent high level language was therefore required, and at the time of the inception of the project, PL/I was considered to fit this description.

Multics provides an exception handling mechanism derived from PL/I ON conditions but including several extensions. The first extension is the

provision by the system of the procedures 'condition_', 'reversion_' and 'signal_' which correspond to the PL/I ON, REVERT and SIGNAL statements, but are more general. These procedures enable programmers in languages other than PL/I to use the Multics exception handling mechanism. This is possible because, as we have seen, the effect of the PL/I ON, REVERT and SIGNAL statements is dynamic.

The major extension to PL/I condition handling was an attempt to solve the clear-up problem. Any block which performs an action which may need to be cleared up, such as opening a file or allocating off-stack storage, should group the necessary 'clearing up' actions into the handler for a special condition called 'cleanup'. If the block completes normally then this handler becomes disestablished on block exit in the usual way. However, if the block is terminated by a non-local GOTO, a system routine called the unwinder is given control. The unwinder goes down the chain of activation records and finds each routine which is about to be aborted. Before releasing its local variables and carrying out any other clearing up that the system realises is necessary, the unwinder searches for a handler for 'cleanup'. If one is found, it is invoked. It is important to realise that a special mechanism is used to invoke the cleanup handlers; the unwinder cannot raise the 'cleanup' condition in the normal way. This is because the handler for each about-to-be-aborted invocation must be executed exactly once; if there is no handler then no clearing up is necessary. An ordinary signal raised at the source of the transfer would cause only the most recently established handler to be executed. Thus, the use of the ON mechanism for establishing cleanup actions, while convenient syntactically, does not greatly influence the cleanup mechanism itself. The

condition mechanism does ensure that cleanup handlers are automatically reverted when the context which established them ceases to exist. Nevertheless, the manner in which they are invoked must be quite separate from the normal method of invoking a handler.

Another extension provided by the Multics implementors was the use of the condition mechanism to trap asynchronous interrupts - primarily attention interrupts from a user's terminal. Similar facilities are provided by many IBM implementations. Such interrupts can occur at any time and can invalidate the assumption made by the handler that the implementation is in a consistent state. Of the Multics extension MacLaren writes

> it almost always works, but there do appear to be a few obscure windows that cause occasional failure. This situation is satisfactory for interrupts from the terminal. It would probably not be acceptable for process control interrupts.

When writing this he was obviously unaware of the frustration caused to a user by a system which randomly ignores some of the things typed at a terminal.

Given PL/I and the task of designing a reliable operating system, the Multics team had options other than adopting PL/I-like condition handling. One course of action was not to use a special exception handling mechanism at all. Organick [75, p. 191] discusses this possibility.

> Signalling via the SIGNAL statement offers the programmer an attractive way to invoke a subroutine without actually having to specify its name, letting its designation be determined dynamically, as determined by the ON statement most recently executed for the same condition. ...
>
> You may be wondering if signalling is merely a fancy programmer's convenience for avoiding the need to set and return proper status arguments. Isn't it the case that a condition can always be reflected backward via possibly a chain of such status returns until it reaches the procedure that knows what to do about it (i.e. what handler to invoke)? Certainly, but even this approach has its shortcomings,

> especially if the chain of returns is long and antecedents on this chain are unable to add any new contextual information (i.e. intelligence) that will clarify or qualify the (recovery) action that should be taken.
>
> As a rule of thumb, therefore, recognized errors and other conditions should be signalled rather than relayed as status arguments whenever the immediate callers are not expected to be 'smart' enough to improve the quality of the 'recovery' from the given condition.

The claimed advantage of condition handling is that notification of an exceptional event can be passed up the call chain until a procedure willing and able to handle the event is found, without the intervening procedures having to know anything about it. If status arguments are used then every procedure in the call chain must be aware of their significance.

However, this advantage is largely fictional. In a hierarchal system, each layer of procedures represents a level of abstraction. Dealing with an exception means either acting on it at the correct level of abstraction or passing it up the hierarchy. But passing it up involves changing the meaning to that appropriate at the higher level. There is thus very little difference between setting status returns at each call and relaying an appropriate exception. In both cases the problem must be analysed and interpreted in the light of all the current information. Even if the correct response is to pass the problem up a level, this cannot be determined without examining it first.

To make this concrete, let us consider the example of a function 'invert' which either delivers the inverse of a matrix or raises the 'zerodivide' exception in the case that the matrix is singular. Suppose that this function is called from a procedure 'Solve' which is attempting to solve a set of simultaneous equations. Organick's argument is that the caller of 'Solve', say 'Main', having been informed of the singularity, can then attempt to re-solve in a space of reduced dimensionality. But for

'Main' to interpret the 'zerodivide' exception as notification of singularity is quite against the spirit of structured, modular programming. The caller of 'Solve' has no business knowing that 'invert' is ever called, far less that the 'zerodivide' exception implies a singularity. Indeed it may not - for 'Solve' may do many divisions for its own purposes, any one of which may signal 'zerodivide', because of a programming error if for no better reason. It is really for 'Solve' to determine what to do if 'Invert' fails, for only 'Solve' knows enough about what is being inverted to take sensible action. If all it can do is to signal to its caller "solutions may not exist", then that is fair enough. It is a very different piece of information from "zerodivide".

Organick's "rule of thumb" also ignores an essential property of a hierarchal system. A given procedure must be callable from a variety of higher level procedures, and cannot therefore use any knowledge about those higher levels. It follows that it is impossible to decide whether the immediate caller is "smart" enough to improve the quality of the recovery: it is not even possible to know who the immediate caller is.

2.5 Exception Handling in Algol 68

In contrast with PL/I, Algol 68 contains no special purpose exception handling machinery. It is interesting to see how Algol 68 deals with those situations in which PL/I resorts to raising a condition.

Many of the PL/I computational conditions arise because of the finite nature of the computer on which the language is implemented. Such failures of representation include normal arithmetic underflow and overflow. In these cases the Algol 68 Report [97] is specific in leaving subsequent actions undefined. Similarly, the result of division by zero is an

undefined value. The implementer is given complete freedom to do as he likes, and the programmer must take care to avoid the consequences.

This situation is not altogether satisfactory, but the designers of Algol 68 were forced to make a compromise. They were an international group designing a language for a varied collection of machines. For example, on ICL 1900 series computers overflow in a fixed point operation sets a special register but does not interrupt the normal execution sequence. The overflow can only be detected by explicitly testing the register. If Algol 68 were to insist that some specific action be taken on encountering overflow, an implementation for such a machine would be prohibitively expensive. (When IBM designed PL/I they were in a better position: on conditions could correspond to exactly those eventualities which the hardware of the System/360 was able to trap.)

What Algol 68 does provide are aids to help in avoiding overflow in the first place. There are a number of "environment enquiries", whereby a program can determine the maximum and minimum values of a given type, the number of different precisions of arithmetic available, and so on [97, §10.2.1]. If conditions like overflow are liable to arise in only a few places they can be avoided by careful programming, but if they may occur almost anywhere the language is of no real help. Arguably, of course, PL/I is of little more help; all one can do in the latter case is to establish a global handler which prints a message and stops.

Environment enquiries, invented by Naur, are undoubtedly one of the most successful aspects of Algol 68; the idea has been adopted by most of its successors, including Modified Algol 60 [19]. The relationship with exceptions should be clear. When asked to include a function with a

restricted domain, there are two things that a language designer might reasonably do. The first is to provide a means whereby membership of the domain may be tested: that is the role of environment enquiries. The second is to extend the domain and make the function total; in this case the range must also be extended by the addition of "exceptional" results.

A different approach is used by the Algol 68 transput (i.e. input and output) mechanism. Information in the system resides in "books" containing text, some identification, the position of the logical end of the text, and so on. An Algol 68 file is the means of communication between a program and a book; a book may be linked to several files simultaneously (via one or more channels, which correspond to types of devices).* There are enquiries to determine whether a book opened on a file may be accessed in particular ways, such as 'get possible', which is true if the file may be used for input, and 'reidf possible', which is true if the identification of the book can be changed. More interestingly, it is possible to associate "event routines" with a file [97, §10.3.1.3.cc]. There are seven such routines, each associated with a particular transput condition: physical and logical end of file, end of line and page, exhaustion of a format and conversion errors. When one of these conditions occurs the appropriate routine is called. If that routine mends the condition it should return true; transput will then continue. Alternatively, it could return false; in this case the transput routine will take a default action. Not all of the conditions are errors, and the default actions are chosen

---

* More conventional terminology would be to call an Algol 68 book a file. Algol 68 files are more mnemonically known as streams [91].

appropriately. For example, if the routine associated with the "end of format" condition returns false, the default action of the transput routine is to re-use the format from the beginning. On the other hand, if the "physical file end" condition is not mended (i.e. the event routine returns false), the default action is left undefined by the report. End of line and page can be detected by "position enquiries" as well as with event routines, but the latter are obviously more convenient if one simply wishes to add page numbers to a listing.

Procedures in Algol 68 are data objects which can be freely stored in data structures. A library written by a user in ordinary Algol 68 could employ an event mechanism modelled on that of the transput system. The only facility used by the transput mechanism not available to an ordinary programmer is that of "hiding" the names of the field selectors. One cannot assign to the 'page mended' field of a file, but must use the 'on page end' routine to assign a new event routine.*

One of the consequences of procedures being data objects is that they have the scope of the block in which they are declared. Algol 68 prohibits the assignation of a value to a name which has an older scope [97, §5.2.1.2.b]. The following (illegal) example is given in the Report, and assumes that the file 'intape' is opened in a surrounding block. The intention is to count the number of integers on the input tape.

---

* Although this corresponds to modern ideas on hiding the representation of an abstract object, Algol 68 introduced it for quite a different reason; see [97, §0.3.7].

```
begin    int n := 0
;        on logical file end( intape
                            , (ref file file)bool:goto f
                            )
;        do  get(intape, loc int); n plusab 1 od
;     f: print(n)
end
```

This is illegal because the scope of the procedure

```
(ref file file)bool:goto f
```

is the block of the example, and it cannot be assigned to a field of 'intape', which has an older scope (or else it would be possible to jump into a block). The solution is to write

```
begin   int n := 0
;       file auxin := intape
;       on logical file end( auxin
                           , (ref file file)bool:goto f )
;       do   get(auxin, loc int); n plusab 1 od
;    f: print(n)
end
```

so that both the event procedure and the field to which it is assigned go out of scope simultaneously. The effect is the same as PL/I's automatic reversion of handlers on block exit, although the syntax is clumsier.

It is important to notice that the event procedures are associated with files (like 'intape') rather than with transput operations (like 'get'). This ensures that all occurrences of a condition on a given file will be dealt with uniformly: what to do at the end of a line is a property of the file abstraction. It does have the disadvantage that the setting of the handler may be lexically remote from the operation which invokes it.

2.6 Conclusion

All the exception handling techniques discussed above fail to restrict the flow of control. This is partly because all the languages discussed provide

an unrestricted goto statement; more precisely, it is because a non-local transfer of control is often the only action a handler can reasonably take.

PL/I ON conditions bear the same relationship to machine interrupts as do goto statements to jump instructions. Both interrupts and jumps are very powerful facilities, but it is now widely recognized that it is a mistake to include them in high level languages. To paraphrase a famous remark, PL/I ON conditions are too much of an invitation to make a mess of one's program [20].

Chapter 3

"STRUCTURED" EXCEPTION HANDLING

The realization of the deficiencies of PL/I ON conditions (see Section 2.3) has provided the impetus for the development of language features for exception handling which help to retain structure and constrain the flow of control. Although such features are not present in any widely used language, many proposals have been made and incorporated into experimental languages. The Ada programming language sponsored by the U.S. Department of Defense also contains an exception handling facility and seems likely to become widely used in the future.

This chapter reviews the exception handling proposals of various authors and examines the facilities available in CLU, Mesa and Ada. A taxonomy of exception handling mechanisms is first presented, and then the various proposals are evaluated within this framework.

3.1 A Taxonomy for Exception Handling

Various authors use the same words to denote different ideas, so this study must begin by defining its terms.

An *operation* is an abstract entity, a mapping from arguments to results. A standard operation like addition or concatenation may be implemented in firmware or machine code, and a programmer defined operation may be implemented as a piece of high level language text. In either case the term *routine* will be used to mean a piece of program text implementing an abstract operation.

A routine text may contain *invocations* of operations; it is said to be

the *caller* of the routines which implement them. An invocation is a piece of text, and may give rise to zero, one or many activations of the routine which it names. (For example, the invocation may be controlled by a while statement.) An activation may *generate* an exception; this results in the *raising* of the exception in some (other) routine (such as the one which caused that activation). The *handler* is the program text which is executed in response to the raising of an exception.

An early version [29] of Goodenough's widely cited survey paper [30] included a useful classification of "exception association methods", that is, methods of linking the raising of an exception and a handler. *Local* methods make a reference to a handler part of the text of the invocation; *global* methods associate a handler implicitly. Passing an exception handling routine as a parameter to an invocation is a local method of associating a handler; PL/I on conditions are a global method.

Two further classifications are used by Liskov and Snyder in [62]. Exceptions are essentially a communication mechanism between the activation which detects an unusual event (and generates an exception to indicate this fact) and the activation which deals with the event (by handling the exception appropriately). Liskov and Snyder state that the "obvious candidates" for handling an exception generated by an activation are the set of activations in existence when the exception occurs. They exclude the generator (because if it could deal with the event there would be no need to raise an exception), and then classify techniques according to whether activations other than the immediate caller of the generator are allowed to handle the exception. They use the adjectives single-level and multi-level to describe these two options. Levin [59] bases his proposals on the

premise that these "obvious candidates" are not the right ones. In common with Parnas [76] he feels that the candidates should be the members of the "uses" hierarchy, whereas the activations in existence when an exception is detected form the "calls" hierarchy. The two hierarchies are in general distinct. Nevertheless, the single or multi-level classification can be applied to the uses hierarchy too, i.e. it is conceivable that the users of a user might be able to handle an exception.

Liskov and Snyder's second classification is according to whether an activation continues to exist after it has generated the exception. The generator can be considered either to call the handler itself, or to return to its invoker in such a way that the handler is activated. In the first case it is possible for the handler to return to the generator, causing its *resumption;* in the second case this is not possible because generating an exception is considered to *terminate* the generator.

Many more classifications could be used. Possibilities are whether a technique is designed to deal only with rarely occurring events or with common ones, whether exceptions can be raised only by statements or also by expressions, and whether parameters are associated with exceptions. Nevertheless, the framework I have outlined should be sufficient for the reader to appreciate the differences between the mechanisms described in the remainder of this chapter.

3.2 Exception Handling in CLU

The CLU language was developed by the Computation Structures Group at M.I.T. between 1974 and 1978. Many excellent papers have been published which not only describe the language but also explain why its features have a particular form. An overview of CLU is available in [64]. The description

below is based on [60], [62], [61] and [7].

CLU takes the view that a routine may terminate in one of several ways. A normal return indicates that the operation it implements has been successfully completed. However, if such completion proves to be impossible the routine executes a signal statement instead. Results may be delivered by both return and signal; the heading of a routine must specify the various possibilities. The library routine for exponentiation has the type:

```
power: proctype (real, real) returns (real)
       signals ( zero_divide, complex_result,
                 overflow, underflow )
```

'zero_divide' occurs if 'arg1' = '0' and 'arg2' < '0'. 'complex_result' occurs if 'arg1' < '0' and 'arg2' is non-integral. 'overflow' and 'underflow' occur if the magnitude of the result is too large or too small to represent. Few of the library routines actually use the facility of passing arguments with a signal, although some of the file manipulation routines pass strings which more precisely define the error.

An exception is generated by means of a signal statement such as 'signal complex_result'. The name of the exception can be followed by a list of parameters if appropriate. The exception must either be listed in the routine heading or be the special exception 'failure'. The signal statement, like the return statement, terminates the current routine activation; execution continues in the caller, i.e. the routine containing the invocation which activated the generator. That invocation raises the exception, and somewhere in the caller there must be a handler for it.

Handlers are specified by an except statement, which associates a list of exception handlers with a statement. The statement in question can be

a single invocation or a large block. For example, to handle the exceptions raised by calls of 'power' in a block, one could write

```
begin
    ... body of block containing invocations of power...
end except when zero_divide: h1
           when overflow, underflow: h2
           others: h3
    end
```

If the 'zero_divide' exception is raised by an invocation in the block, handler 'h1' will be executed. If 'underflow' or 'overflow' occur, 'h2' will be executed. If *any* other exception is raised, 'h3' will be executed. The handlers 'hi' will themselves usually be statement sequences, and may contain other except statements.

If the execution of the statement enclosed by an except statement completes without raising an exception, control passes to the statement following the except statement without executing any of the handlers. However, if an exception is raised during the execution of the statement, control passes immediately to the textually closest handler for that exception. (It is necessary to say "closest" because except statements can be nested.) When execution of the handler completes, control passes to the statement following the except statement containing that handler. A handler is outside its own scope: an exception occurring in a handler must be caught either by an except statement within the handler body or by another, enclosing, except statement.

If a list of exception names in an except statement is followed by a parameter list, all the named exceptions must have the same number of arguments as there are parameters in the list, and the types must correspond. The parameter list may be represented by an asterisk: this discards the actual

arguments, and can be used for any exception. An others handler may appear once or not at all; it handles any exception not listed by name in the except statement, including 'failure' (see next paragraph). others may be provided with a string parameter, which will be set to the exception name; any arguments of the exception are inaccessible.

CLU does not syntactically enforce the restriction that there must be a handler for every exception an invocation can raise. Instead, any unhandled exception is converted to the special exception 'failure("unhandled exception: name")'; however, if the 'failure' exception itself is not handled it is passed on unchanged. Every invocation is considered to be able to raise the 'failure' exception even though it appears in no routine heading. This policy was adopted because the programmer may be able to prove that a given exception will not arise, and should therefore not be forced to provide a handler for it.

From the above it will be seen that exception handlers in CLU are static and can be attached only to statements, not expressions. Exceptions are propagated up the call hierarchy by exactly one level; they cannot be caught within the routine that generates them, but rather cause its activation to be terminated.

The decision to allow exceptions to propagate by only a single-level was derived from the hierarchal program design methodology which the language supports. Each CLU routine implements an abstract operation; the caller of a routine need know only the specification of that operation, and not how it is implemented. The exceptions generated by that routine form part of the specification of the operation, and it is appropriate for the caller to be aware of them. This is why they are listed in the routine heading. However, it is *not* appropriate for the caller to know about the exceptions generated

by routines used in the implementation of the operation. Such exceptions must be handled by the routine containing the invocation which raised them. Of course, if that routine decides that the appropriate action is to itself raise an exception, then it is free to do so.

The CLU except statement has one serious deficiency. The placement of a handler is governed by two constraints.

(i) The handler must be suffixed to the statement whose execution is to be terminated should the exception arise. The statement following the handler must be an appropriate continuation, assuming the handler does not execute a return or signal. (CLU does not have a goto statement.)

(ii) The handler must be appropriate for all occurrences of the named exception. If two invocations raise the same exception but require different handlers, then they must be caught by different except statements, one of which must be situated so that only one of the invocations is within its scope.

These two constraints may conflict. [62] gives as an example a statement 'S' which invokes a routine 'sign' at two different points. 'sign' may generate the 'neg' exception, which should be handled differently for each invocation; however, execution should continue with the statement following 'S' in both cases. If the except statement is used to attach separate handlers to each invocation then some other mechanism is needed to achieve the desired flow of control. One possibility is to set and test variables, as in the following code.

```
begin
        ... statements 1 ...
    xneg := false
    a := sign(x) except when neg (i:int): s1;
                                          xneg := true
                 end
    if not xneg
    then
            ... statements 2 ...
        yneg := false
        b := sign(y) except when neg (j:int): s2;
                                              yneg := true
                     end
        if not yneg
        then
                ... statements 3 ...
        end
    end
end
```

However, if one needs to do that there is little point in having an exception handling mechanism: the code would be simplified if the result of 'sign' were tested with the if statement directly. So CLU includes an exit statement to perform local transfers of control. Using it the example can be rewritten as follows.

```
begin
        ... statements 1 ...
    a := sign(x) except when neg (i:int): s1;
                                          exit done
                 end
        ... statements 2 ...
    b := sign(y) except when neg (j:int): s2;
                                          exit done
                 end
        ... statements 3 ...
end except when done:
    end
```

Exits provide for local control transfers in a similar way to the situation-case statement of Zahn [104]. An exit statement behaves like an invocation which raises an exception. Whereas a signal statement raises an exception in the

calling routine, the exit statement raises the exception directly in the *current* routine. An exception raised by an exit statement must be handled *explicitly* by an except statement; it is not sufficient for the except statement to contain an others arm. Exits can, of course, be used to prematurely terminate any block, not just an exception handler.

3.3 Exception Handling in Mesa

The Mesa language has been developing at Xerox Palo Alto Research Center since 1974. It is used for systems implementation by research staff. Prior to the introduction of Mesa, these users had developed a wide range of programming styles; as far as is possible, Mesa attempts to accommodate them all. It contains the most extensive implemented exception handling mechanism of which I am aware.

Information on Mesa is sometimes difficult to obtain. An overview of the language was published [25], and the Manual is fairly readily obtainable [72]. An evaluation of the Mesa mechanism has been made by Horning [51]. This section has also benefitted from discussion with Dr. J.G. Mitchell. The Mesa exception mechanism is very general and powerful. Its description occupies ten pages of the Mesa Manual, and only a summary is attempted here.

An exception in Mesa is called a *signal*, and has a data type rather like that of a procedure: signals may have both parameters and results. Signals differ from procedures in the way they are associated with a body. The value of a procedure represents its body more or less directly. In contrast, the value of a signal is a unique *code* on which the only operation is test for equality. The appropriate routine body is located if and when the exception is raised.

A signal is generated by a signal statement, which has the same syntax as a procedure call except that it is prefixed by the symbols SIGNAL, ERROR or RETURN WITH ERROR.

Generation of a signal will eventually cause a handler to be invoked; its selection is described below. If the handler completes without performing a non-local transfer of control, the effect is to return to the statement following the signal call exactly as with a procedure call. The symbol ERROR provides a way of prohibiting such a return; any attempt to return after an error call is illegal (and causes another ERROR called 'ResumeError'). If a signal is declared as an ERROR then it is a syntax error to use it in a signal call; if it is declared as a SIGNAL, it may be used in any signal call, including an error call. Thus errors are a sort of sub-type of signals.

Signal calls may return results, and may therefore appear in expressions; this is also true of error calls, although since they never return at all this is purely for syntactic convenience. (A similar feature is found in Algol 68, where a goto statement may be treated as an expression of any type.)

A handler is associated with a signal by means of a so-called *catch phrase*. A catch phrase has as its scope a piece of program text. It may appear after the BEGIN of a block or the DO of a loop, in which case its scope is the whole of that block or loop. It may also appear within the brackets of a call, separated from the arguments by an exclamation mark; the scope of such a catch phrase is the routine activated by that call, but not the argument list itself. A catch phrase is said to be *enabled* when control is within its scope in a *dynamic* sense. All this may be

clarified by an example (← is the assignment symbol).

```
Report  :  PROCEDURE[message : TEXT] =
BEGIN     ENABLE catch phrase 1
;             ... declarations ...
;         IF ...   THEN SIGNAL ImpossibleError
;             ...
;         InputFileName ← ConstructCurrentFileName[]
;         LineNr ← LineNr + 1
END;
ConstructCurrentFileName : PROCEDURE RETURNS[FName : TEXT ] =
BEGIN       ...initialization...
;       UNTIL NameElement = Nil
        DO  ENABLE catch phrase 3
        ;
             ...
        ;   FName ← AppendText[ FName
                              , FileName[NameElement] ]
        ;   NameElement ← Next[NameElement]
        ENDLOOP
END;

AppendText : PROCEDURE[Head : Text, Tail: TEXT]
                                        RETURNS[t:TEXT] =
BEGIN       ...
;       t ← ConcatText[Head, Tail ! catch phrase 4]
;       ReturnText[Head]
END
```

The routine 'Report' enables 'catch phrase 1' for the whole of its body. Signals generated by the statements of that body, such as the assignment to 'LineNr', will be offered to 'catch phrase 1'. This is true even if the 'SIGNAL ImpossibleError' statement is executed.

'Report' invokes 'ConstructCurrentFileName', which does not have a catch phrase attached to the whole of its body. If a signal is generated while executing the initialization it will be offered to 'catch phrase 1' in 'Report', as this is the dynamically enclosing context. However, the UNTIL ... ENDLOOP statement within 'ConstructCurrentFileName' has its own catch phrase, and signals generated by the body of the loop will be

offered to 'catch phrase 3' first.

'AppendText' illustrates a catch phrase attached to a call. Signals raised (but not caught) within 'ConcatText' will be offered to 'catch phrase 4'.

A catch phrase may *reject* a signal, in which case it passes up to the dynamically enclosing catch phrase. So if 'AppendText' is called from within the loop of 'ConstructCurrentFileName' and generates a signal which 'catch phrase 4' rejects, it will be offered to 'catch phrase 3'. The Mesa run time system guarantees that all otherwise uncaught signals will be caught at the highest level by the debugger.

It should now be apparent that the Mesa signal statement both *generates* an exception and *raises* it. This is also true of the error statement. Thus in Mesa an exception may be handled by the routine which itself generated the signal or error. The RETURN WITH ERROR statement is used to raise an exception in the environment of the calling routine. In the example, if the UNTIL ... ENDLOOP construct contained such a statement, e.g. 'RETURN WITH ERROR FileNameStructureInconsistent', the signal would first be offered to 'catch phrase 1'. Resumption is not possible after a RETURN WITH ERROR statement.

The catch phrase itself is rather like a case statement; it takes the following form.

```
BEGIN
    sig 1 => body A;
    sig 2, sig 3 => body B;
    sig 4 => body C
END
```

The 'sig i' are variables which must evaluate to signals. If 'sig 1' is declared to be of type

SIGNAL [param : p] RETURNS [result : r]

then 'param' and 'result' are implicitly available within 'body A' as the names of the parameter and result of the signal call. If 'sig 2' and 'sig 3' do not have identical types their parameters and results are not available within 'body B'. Clearly the scope of the signal declaration must encompass all the signal calls and catch phrases which refer to it.

The last (or only) clause in a catch phrase may have the form 'ANY => body'. When a signal is offered to a catch phrase its value is compared, in order, with each signal value in the variable lists preceding the => symbol. If a match is found control passes to the appropriate body. ANY matches any signal code, and is primarily intended for use in the debugger.

If none of the variables in the catch phrase matches the signal, the signal is "rejected" and propagated to the next catch phrase in the call hierarchy. The body of a handler that has caught a signal may also explicitly reject it; thus partial recovery from an exception may be effected, and then the signal passed on up the hierarchy.

The body of a handler behaves exactly like a routine called by the generator. Specifically, it may return to the generator (provided the signal was not generated by an error call). This is done by means of the RESUME statement, which may also return values to the generator, according to the type of the signal.

If resumption is inappropriate, the handler must execute some form of non-local transfer of control. In addition to the ordinary methods of control transfer, two special statements, RETRY and CONTINUE, may be used within catch phrases. RETRY activates a transfer to the *beginning* of the statement to which the catch phrase belongs; CONTINUE initiates a transfer to the statement *following* the one to which the catch phrase belongs. In the case of a catch phrase attached to a loop body, CONTINUE means "go around the loop again"; the same effect can be achieved with Mesa's LOOP statement.

Mesa also has a restricted GOTO (rather like a CLU exit), and an EXIT statement for terminating loops. Before any of these non-local transfers are completed each about-to-be-aborted activation is given an opportunity to do some clearing-up (see Section 2.2). Syntactically, the clear-up code appears as a handler for the special signal 'unwind'. However, 'unwind' is not treated like an ordinary signal. The implementation of non-local control transfers ensures that 'unwind' is offered exactly once to each catch phrase of each activation which is about to be aborted. If a given catch phrase has no handler for 'unwind' the signal is *not* propagated as an ordinary signal would be. There are no constraints on the action which an unwind handler can take. In particular, it may itself perform a non-local transfer of control and initiate a second 'unwind'. No further clearing-up is then done for either transfer, and the first transfer is abandoned in favour of the second one.

It is a common misconception that exception handling mechanisms like that of Mesa "solve" the clear-up problem. They do not. The "unwind"

signal in Mesa (and 'cleanup' in Multics, see Section 2.4) cannot be treated as ordinary signals. The inclusion of 'unwind' amongst the signal values in a catch phrase is syntactically convenient but semantically irrelevant (and expensive at execution time). It would be closer to the truth to say that Mesa exception handling *causes* the clearup problem, because all non-local control transfers are associated with signals. (This is a slight oversimplification because most systems need some provision for clearing-up after a catastrophic failure. This is considered in Chapter 7.)

It should be clear that Mesa's exception handling mechanism is much more powerful than that of CLU. Both mechanisms are confined to the call hierarchy and are engineered so as to be suitable for rarely occurring events. However, by allowing signals to span more than one level of call, and by not requiring a list of signals to appear in the heading of a routine, the Mesa mechanism achieves greater power and convenience. Horning [51] observes that there are responsibilities associated with the exercise of this power.

> It it necessary to document not only the names and meanings of the signals that the components may use directly or indirectly, but also the names and meanings of any parameters supplied with the signal, whether the signal may be resumed, and if so, what repair is expected and what result is to be returned. Unless all this information is provided, it will be difficult for users to respond correctly to signals. Each programmer must decide which signals to handle via catch phrases, and which to reject (i.e. to incorporate into the interface of his component).
>
> ... The more levels through which a signal passes before being handled, the greater the conceptual distance is likely to be between the signaller and the handler, the greater the care necessary to ensure correct handling, and the greater the likelihood of some intermediate level omitting a necessary catch phrase for 'unwind'.

The ability to resume after a signal is stressed is one of the great advantages of the Mesa mechanism. This is not because the generating routine is often resumed but because when a signal is not caught control passes to the debugger, and all the control context which existed when the signal was generated is then available for inspection. In discussion Jim Mitchell has claimed that ninety-nine per cent of signals are intended for the debugger, in that they are generated in "impossible" situations. Although his estimate is unsubstantiated, it is clearly common to use Mesa signals for this purpose. In such a situation the programmer is able to examine the local variables of the routine which generated the signal and determine the cause of the problem. It may even be possible to use the debugger to alter those variables and resume the signal.

It should be observed, however, that access to the local variables of a routine by another part of the *program* is prohibited by the principle of modularity and the scope rules of the language (except in the degenerate case where a signal is handled in the routine which generates it). This distinction between program and programmer is well-made by Liskov [62]. While observing that the CLU mechanism is designed to provide *programs* with the information required to recover from undesired events, she points out that nothing in the design precludes an implementation which reports exceptions to the programmer, should he be available, before terminating the activations which generated them. In this way as much state as is required may be inspected. Moreover, there is no implied restriction that only *unhandled* exceptions should be reported to the programmer. It would be possible for the exception mechanism to be restricted to report a specified subset of exceptions to the programmer, who could then decide whether to

handle them himself or to let them be handled by the program.

None of the above is intended as a claim that the ability to resume an exception does not add power to the language. That it does so is indisputable: one may wish to argue only that the extra power is not needed, or is not worth its cost in terms of semantic complexities, or should be provided by some other mechanism. My point is that in putting these arguments the question of debugging is quite irrelevant.

The Mesa language has been in use at Xerox PARC for seven or eight years, and considerable experience in the use and misuse of signals has been built up. Such experience is not common, and I think it worthwhile quoting the following communication from Jim Morris at length (excerpted from [51]).

> Like any new and powerful language feature, Mesa's signal mechanism, especially the *unwind* option, should be approached with caution. Because it is in the language, one cannot always be certain that a procedure call returns, even if he is not using signals himself. Every call on an external procedure must be regarded as an exit from your module, and you must clean things up before calling the procedure, or include a catch phrase to clean things up in the event that a signal occurs. It is hard to take this stricture seriously because it is really a hassle, especially considering the fact that the use of signals is fairly rare, and their actual exercise even rarer. Because signals are rare there is hardly any reinforcement for following the strict signal policy; i.e. you will hardly ever hear anyone say "I'm really glad I put that catch phrase in there; otherwise my program would never work". The point is that the program *will* work quite well for a long time without these precautions. The bug will not be found until long after the system is running in Peoria.
> ... It should be noted that Mesa is far superior to most languages in this area. In principle, by using enough catch phrases, one can keep control from getting away. The non-local transfers allowed by most Algol-like languages preclude such control. It has been suggested that systems programming is like mountaineering: one should not always react to surprises by jumping; it could make things worse.

Cedar Mesa is a version of the Mesa language being designed and implemented as part of the Cedar Programming Environment. Associated with

the Cedar design effort is a restricted form of the signal mechanism. The restrictions are not enforced by the compiler but are adopted by convention: enforcement may follow if it is found that the restrictions are not too irksome. The aim is to simplify the mechanism and to make it mesh better with the principle of encapsulation. The restrictions are:

(i) that the ANY catchall be eliminated;

(ii) if a signal crosses more than one abstraction boundary it is changed to a globally known exception denoting a programming error. This captures the fact that the signal has left the only abstraction in which it could reasonably be caught.

(iii) There should be a "calling error" signal indicating that a module has been provided with illegal arguments.

(iv) Neither calling errors nor programming errors should be resumed or have parameters.

At present I have no information about the effect of these restrictions. It will be some time before it is known whether they help to alleviate the problems discussed by Morris.

3.4 Levin's Proposals for Exception Handling

In his Ph.D. thesis [59] Roy Levin made proposals for an extensive exception handling mechanism. Realising the dangers of adding to any language, he sought a construct that would be *verifiable*, *uniform* over a large class of exceptions, *adequate* for real problems in both parallel and sequential programming, and *implementable* with reasonable efficiency. The opening sections of his thesis elaborate these goals and ought to be compulsory reading for anyone contemplating the invention of a new programming construct.

However, by laying such a firm foundation Levin invites a more stringent evaluation of his mechanism than was appropriate for those designed to less exacting standards.

The introduction to his thesis admits that there is no adequate definition of "exception"; whether or not a particular event should be classified as an exception depends largely on its context. It is claimed that the exception mechanism should be used whenever the details of the handling of a particular case need to be suppressed, so that other (more common) cases can be emphasised. His mechanism is capable of dealing with anything from general interprocess communication to interrupt handling.

The chief difference between Levin's mechanism and the others I have described is in the selection of the modules which may be given an opportunity to handle an exception. Levin argues that all of the *users* of a module should be considered. The classic example supporting this argument is a storage allocation module used by several other modules which maintain private caches of store. If the allocator cannot satisfy a request from one of its users it generates an exception 'Insufficient Store'. In which module or modules should this exception be raised? Clearly not just in the module whose request cannot be satisfied. It should at least be *potentially* possible for the exception to be raised in all the modules which have used the allocation system, because any of them may be willing to return some of their cache. Of course, it may be that the first module to receive the exception is able to release sufficient store, and that there is no need to raise it in other modules. Levin allows such selective exception raising, but insists that the initial set from which the selection is made must include all the users of the abstraction which generates the exception.

Another distinguishing feature of Levin's mechanism is that an exception may be generated on both data abstractions and control abstractions. The users of a control abstraction are its callers; in this case propagation of the exception along the call chain is appropriate. The users of a data abstraction form a hierarchy independent of the call chain. A particular invocation may raise either kind of exception. For example, 'ReadfromFile(f)' may generate 'ProtectedFile' if the invoker does not have read permission, and 'FileInconsistent' if there is an inconsistency in the disk structure used to represent 'f'. In the first case the exception should be raised in the invoker of 'ReadfromFile', in the second it should perhaps be raised with all users of 'f'.

Although I have spoken of a hierarchy, Levin is emphatic that exceptions should propagate through exactly one level. An exception must be raised with the *immediate* user of the appropriate abstraction; indirect users cannot even know that the offending module has been called, still less how to handle its exceptions.

It may be that a single handler for an exception will be appropriate for the entire lifetime of a datum. If that datum exists only during the activation of a single routine (i.e. is a local data structure), then this situation can be dealt with as in CLU by attaching a suitable handler to the body of that routine. However, data structures are frequently created by one module and then passed to others, so their lifetime transcends that

of their creator.* Levin therefore introduces a notation for associating default handlers with data structures, and a convention that, in the absence of an explicit default handler, a null default handler be assumed to exist. Unfortunately he uses the same notation for defining these default handlers as for dealing with the exceptions generated on the control abstractions of the module itself.

Before proceeding further some concrete syntax and examples may be helpful. Levin uses the notation and terminology of Alphard [103]; I have translated them into a more Algol-like notation. Handlers may be attached to a statement by appending to it a list of exception conditions and handlers enclosed in brackets. For example, the 'ReadfromFile(f)' example might be written

```
ReadfromFile(f) [ f.FileInconsistent : handler1
                | ReadfromFile.ProtectedFile : handler 2 ]
```

The part of the condition name before the point is the name of the abstraction on which the exception is generated; the part after the point is the name of the exception itself.

If the statement to which a handler is appended is a block, and if the abstraction on which the exception may be generated is a routine, Levin's interpretation is that a separate copy of the handler be attached to each invocation of the routine within that block. This is also the interpretation when the abstraction is a data object, provided that the block is not the module which defines that data type. Thus the following are equivalent.

---

* Such structures are often said to reside on a *heap*, but this term has connotations of a particular implementation.

```
begin
    var x : integer
    x := ReadfromFile(f)
    WritetoFile(g,x)
    WritetoFile(Increments, x+1)
end [ f.FileInconsistent : h1
    | WritetoFile.Protected File : h2 ]

begin
    var x:integer
    x := ReadfromFile(f) [f.FileInconsistent : h1]
    WritetoFile(g,x) [Write to File.ProtectedFile : h2]
    WritetoFile(Increments, x + 1)
                    [WritetoFile.ProtectedFile : h2]
end
```

In both of the above fragments, the handler 'h1' is attached to the 'FileInconsistent' exception of 'f' only for the duration of the invocation 'x := ReadfromFile(f)', which represents only a small part of the lifetime of 'f'. If 'f' is local to some routine 'WriteIncrements' which contains one of the above blocks, then a handler can be attached to 'f.FileInconsistent' for the remainder of the lifetime of 'f' as follows:

```
procedure WriteIncrements =
begin
    var f : File of integer
    begin
        var x : integer
        x := ReadfromFile(f)
            ...
    end [f.FileInconsistent : h1 | ...]
end WriteIncrements [f.FileInconsistent : h3]
```

When 'ReadfromFile(f)' is invoked both 'h1' and 'h3' are *enabled*. However, 'h1' *masks* 'h3', so only 'h1' is *eligible* to handle 'f.FileInconsistent'. Precise definition of these terms - enabled, masked and eligible - are beyond the scope of this thesis. The interested reader is referred to

[59, Section 4.6]. These definitions are central to an understanding of Levin's mechanism, and I confess to finding them difficult to understand. This may be because the definitions are phrased in terms of implementation rather than linguistic concepts, but at least part of the difficulty lies in the inherent complexity of the mechanism.

Levin's mechanism becomes interesting when we allow *sharing* of data abstractions. At present, sharing is thought to be an important concept in systems programming. Allowing sharing is equivalent to introducing the concept of *reference** at the semantic level. A simple language such as 3R [11] needs only *names* and *values* and a mapping between them to describe the actions of parameter passing and assignment. Algol 60 can be described in this way because of the copy rule for procedure activation. Languages such as Fortran, Algol W, Algol 68 and Pascal introduce references more or less explicitly. Their semantics can only be described by introducing a set of *locations* and two mappings, the *environment* which takes names into locations, and the *store* which takes locations into values. In these languages declarations associate a name with a location, and assignments associate a location with a value. (Algol 68 either simplifies or confuses the issue, depending on one's point of view, by unifying locations and values.) CLU does not introduce explicit references but allows sharing by using them implicitly. In CLU, *assignment* changes the *environment*, while the *store* is changed only by built-in procedures that update arrays and records.

Levin's storage allocator can now be examined in more detail. Suppose that the modules 'CachedCommunicateWithDisk' and 'Columnate' both use the

---

* Also known as pointer or access value.

services of 'FreeStore', which allocates storage from a *shared* storage pool 'FSPool'; suppose further that a *reference* to 'FSPool' has been passed to both of the user modules as parameter. These modules are shown below.

```
module CachedCommunicateWithDisk
is  var CachedPages : IndexedList of DiskPage
    var StorageZone : ref pool
        ...
    procedure ReadPage(PageNr : integer)
    is if IsinList(PageNr, Cached Pages)
       then ...
       else const p : ref Page =
                          Allocate(StorageZone, DiskPage)
                 ...
            p := BasicDiskRead(PageNr)
            EnterinList(CachedPages, p)
               [StorageZone.InsufficientStorage: skip]
                 ...
       fi
    end of ReadPage [StorageZone.InsufficientStorage:
                  ReturnSomebutkeep(CachedPages, PageNr)]
              ...
end of CachedCommunicateWithDisk
            [StorageZone.InsufficientStorage:
                       ReturnSome(CachedPages)]


module Columnate
is  var ThisPage : List of Line
    var CentralPool : ref pool
      ...
    procedure Out(s : ref CharStream, c : char)
    is if IsFull( LastLine(ThisPage) )
       then const l : ref Line =
                                Allocate(CentralPool, Line)
            ThisPage := Append(ThisPage, l)
               [CentralPool.InsufficientStorage: skip]
       fi
              ...
    end of Out
end of Columnate [CentralPool.InsufficientStorage:
                                        Compress(ThisPage)]
```

'Allocate(z,t)' attempts to find enough storage in the zone referred to by 'z' to create a location of type 't'. If this is impossible it generates

the 'InsufficientStorage' exception on 'z'. Levin's mechanism assumes that all handlers will return control to the generator of the signal, so after the appropriate handlers have been invoked 'Allocate' can again attempt to satisfy its caller. (Of course, if this second attempt fails, some other action should be taken.)

Suppose the 'Allocate' request on line 9 raises the 'InsufficientStorage' exception on 'StorageZone'. The handlers on lines 16-17 and 20-21 are enabled, but that on lines 16-17 masks that on lines 20-21. If 'CentralPool' and 'StorageZone' refer to the same 'pool', then the handler on lines 37-38 is also enabled; the *eligible* handlers are thus those on lines 16-17 and 37-38.

If parallelism is allowed the situation is more complicated, because it is possible for 'Out' in 'Columnate' to be active at the time the exception is raised in 'ReadPage'. Suppose execution has reached line 33; the handler attached to the invocation of 'Append' would then be enabled, and since it is not masked, eligible. The purpose of this handler is to mask the handler on lines 37-38 while the structure 'ThisPage' is being modified.

Levin distinguishes two levels of selection in the above process. The method used to chose which handler in each module is eligible at a given instant is of secondary importance. He uses a static, single level mechanism because it is in accord with conventional scope rules, but states that a different policy could be used if required. What is central to his thesis is the composition of the set of *contexts* within which a handler is selected. He emphasises that every user of a data structure should be able to handle exceptions raised on that structure, whether or not that particular user caused the exceptional condition.

Having decided that several handlers in different contexts may be

eligible to deal with a condition, which should actually be invoked, and in what order? Levin answers this question by defining several *selection policies*. 'Broadcast-and-wait' indicates that all eligible handlers are invoked (potentially) in parallel, and that execution of the generator resumes when they have all terminated. 'Broadcast' relaxes this restriction: the generator initiates all the handlers but does not wait for any of them to complete. The generator thus executes in parallel with the handlers. The last policy Levin defines is 'Sequential-conditional', which associates a predicate with the raise statement. If the predicate is true or there are no eligible handlers, the raise statement has completed. If the predicate is false a handler is selected and initiated, and the handler removed from the set of eligible handlers. When the handler terminates the raise statement is re-entered with the reduced set of handlers.

Much of the above assumes that handlers will return to the generator of the exception. This is indeed a requirement of Levin's mechanism. The raise statement works exactly like a routine call; it even has a post-condition which the handlers must satisfy. This is in contrast to the CLU mechanism, which forbids resumption, and that of Mesa, which allows both termination and resumption. Levin justifies this in the following way:

> Handlers are invoked (in most cases) to process an unusual condition that cannot be handled entirely within the signalling module. They have an obligation, expressed by the [postcondition]..., and the signaller will assume, upon completion of the raise, that the obligation has been satisfied. If handlers were able to terminate ... the execution of the signaller [then] the abstractions of the signalling module might not be maintained. The signalling and handling modules should be viewed as mutually suspicious subsystems [83]; neither should be able to influence ... the execution of the other.

Although handlers cannot directly alter the flow of control within the signaller, they may do so indirectly by their effects or by the results they return. In the storage allocation example, after 'Allocate' has invoked the handlers for 'InsufficientStorage', it is natural for 'Allocate' to look again at the storage pool before deciding how to proceed.

In contrast, Levin does allow a handler to influence the flow of control within the block in which it is declared. That block may have been interrupted by the invocation of the handler (if it was executing in parallel with the generator): the handler is *not* executed in parallel with other routines in its scope. He devises a mechanism which enables a handler to "post" the location at which execution of its enabling block is to resume. Interested readers should see [59, §4.9].

The above description of Levin's mechanism is incomplete; it attempts to illustrate the distinguishing features of the mechanism rather than to define anything. The definitive description is much longer and more involved, and therein lies a major problem. Levin's mechanism is undoubtedly very complicated. The defence ought to be that at least Levin has given axioms of conditional correctness for his mechanism, so his definition must be explicit and rigorous even if it is complex. Unfortunately this argument is unsound, for only the most straightforward part of his mechanism is actually axiomatised.

Levin's proposals may be roughly sub-divided as follows:

(i) The rules for associating handlers with particular data and control abstractions;

(ii) The action of the raise statement itself;

(iii) The action of the various selection policies;

(iv) The synchronization requirements of the handlers.

The most complicated part of his mechanism is undoubtedly (i). This is also his most significant contribution to the subject because of the prominence given to the association of handlers with data abstractions as well as control abstractions. Unfortunately none of this is formalized. The semantic significance of attaching a handler to a block is explained only in terms of rewriting rules which copy the handler bodies into the appropriate statements. The only guide to this process is the informal definition in terms of *activations, instances, loci of control* and other implementation concepts. The single most important idea, that of the *user* of an abstraction, is never defined formally. Moreover, this use of rewriting rules ignores a fundamental principle of the axiomatic method: one should be able to write the assertions in the program without rearranging the statements.

What *is* defined axiomatically is the semantics of the various different forms of the raise statement, i.e. raise under the various selection policies, and the semantics of handler invocation. The assumption is made that the handlers are *interference-free*, i.e. that their potentially parallel composition in the case of the broadcast policies will have the same semantics as sequential composition. The synchronization requirements between the handlers themselves, and between the handlers and their contexts, are not formalized.

Levin is, of course, aware of these shortcomings, and points out that some of them arise from the weakness of the proof system. He claims that "the problem of verifying parallel programs is a difficult one and distinct from the exception handling issues of this thesis". The first part of that claim is undoubtedly true, and the problem of verifying programs using sharing is also difficult. Nevertheless, both of these problems had been

treated by various authors at the time that Levin was writing (see [6] for a survey). Although such studies were then and remain incomplete, Levin should perhaps have attempted to use them to shed light on his mechanism.

The alternative is to decide that these ill-understood areas, particularly parallelism, should be avoided when designing an exception mechanism. It seems reasonable to try to gain a full understanding of both exceptions and parallelism in isolation and only then to examine their interactions. It is much easier to enlarge a language than to make it smaller; perhaps the temptation to include mechanisms concerning two such volatile topics as exception handling and parallelism should have been resisted.

3.5 Exception Handling in Ada

Ada [96] is intended as a language for real-time applications requiring a high degree of reliability and maintainability. To promote this usage, and to enhance reliability in particular, the designers tried as far as possible to incorporate only well-understood constructs which had been proved in practical applications. However, this desire sometimes conflicted with the contractual requirements set out in the "Steelman" [95] document; in the areas of interprocess communication and exception handling, Ada goes significantly further than any existing commercial language.

Exception handling in sequential Ada is similar in conception to the proposals discussed above. There are five predefined exceptions generated implicitly in certain situations: 'constraint_error', 'numeric_error', 'select_error', 'storage_error' and 'tasking_error'. A 'constraint_error' occurs in many situations including index out of bounds in an array access and trying to access a non-existent variant of a record. A 'numeric_error' *may* be generated when the result of a pre-defined numeric operation does not

lie within the implemented range; an implementation is not compelled to generate this exception. A 'storage_error' occurs when dynamic storage space is exhausted; 'select_error' and 'tasking_error' relate to interprocess communication, and will be considered later. There is also, for each task* 't', an exception 't'failure' which (although pre-declared) can only be raised explicitly. In addition to these exceptions the user may declare his own, which can be generated by the raise statement. Such user-declared exception names are subject to the usual scoping rules.

An Ada block** has the form

```
preamble
    declarations
begin
    statements
exception
    exception handler
end
```

where the syntactic form of 'preamble' varies between the different categories of block. The 'exception handler' is like a Mesa catch phrase: sequences of statements are associated with exception names or with the symbol others. The execution of a raise statement causes the appropriate exception to be

---

* Ada uses the term 'task' to mean an independent, dynamic entity that may operate in parallel with other tasks, i.e. what is usually called a *process*. However, despite the influence of a major U.S. computer manufacturer the English word *task* always means the piece of work rather than the agent which performs it. I will use process in preference to task whenever possible.

** *Block* will be used in the sequel to include blocks, subprogram bodies, package bodies and task bodies.

raised *within* the current block; this is like Mesa rather than CLU. If there is an exception handler within the block, and if it names the exception or has an others clause, it is executed. The handler acts as a substitute for the remainder of the block; if the block is a routine body the handler may execute a return statement, which will terminate the routine.

If there is no exception clause in the block, or if it does not name the appropriate exception, the exception is raised again at the point of invocation of the block; in Ada terminology it is *propagated*. Exceptions can be passed through on arbitrary number of levels of the call hierarchy in this way. The headings of procedures and functions are not required (or even permitted) to list the exceptions which they may propagate.

Because a function may raise an exception the designers of Ada had to decide how to deal with exceptions raised during the elaboration of the declarations of a block. Arguably the handler in the block body should not apply because some of the variables it manipulates may not be declared when the exception occurs. The current Ada definition [96] accepts this argument and limits the scope of the exception handler to the *statements* of the block. An exception generated within the declarations is raised with the invoker of the block; the handler in the body of the block is ignored. (Interestingly, [55, §12.5.2] states that this arrangement was "rejected for implementability reasons".)

Unfortunately the rules outlined above do not apply in all situations. There are undesirable interactions between exceptions, packages and processes. For example, an exception raised in the declaration part of a task body is propagated to whoever caused the task activation, whereas an exception raised in the statement sequence of the same task body (and not handled locally) is *not* propagated at all; the task is simply terminated. The

very fact that there are eleven separate clauses concerning unhandled exceptions [96, §11.4.1-2] indicates that it is not always clear just when an exception will be propagated.

The propagation rules also clash with the scope rules. Since exception names, like identifiers, are subject to the scope rules of package and block structure, an exception can be propagated beyond the scope of its name. It can then be caught by an others handler; it is even possible to re-raise it.

As was mentioned in Section 1.5, the rules about optimization in the presence of functions which may generate exceptions [96, §11.8] also substantially complicate the language. They introduce semantic traps of which the programmer must beware.

The following is a slight modification of the stack package from Section 12.4 of the reference manual. A function 'Top' has been added and, in the interests of simplicity, the generic clause has been omitted. The specification* is now:

```
package StacksforElems
is  type ElemStack is private
;   procedure Push(s: in out elemstack; e: in elem)
;   procedure Pop(s: in out elemstack; e: out elem)
;   function Top(s: in elemstack) return elem
;   Overflow, Underflow: exception
end StacksforElems
```

Although this specification does not show it, 'underflow' can be raised

---

* i.e. syntactic specification, for the benefit of a compiler. For this reason Ada insists that the *representation* of ElemStack be given as part of the "specification". I omit it.

by invocations of 'Pop' and 'Top'. If one attempts to supply the missing predicate 'IsEmpty' it is easy to fall foul of [96, §11.8], for example:

```
function IsEmpty (s: in elemstack) return boolean
is  e:elem
begin   e := Top(s)
;       return false
exception
    when underflow => return true
end
```

Since 'e' is never used within the function body, the value of the 'Top' operation is clearly not needed. Applying the rule that "the operation need not be invoked at all if its value is not needed, even if the invocation would raise an exception", it is clear that an implementation is entitled to elide the call to 'Top', and to represent 'IsEmpty' as the constant function 'false'. Ada exceptions may be useful when warning of a departure from the specification, as with an implementation insufficiency. However, it seems that their semantics are irregular and difficult to formalize, and that their use to communicate a result *required* by the specification is ill advised.

Luckham and Polak [66][67], writing before the promulgation of the above rules, describe some of the other changes which should be made to Ada to permit verification of programs using exceptions. I have already mentioned the lack of a propagation declaration in subprograms; Luckham and Polak propose the addition of a form of declaration which includes an assertion describing the state whenever the subprogram terminates by raising that exception. They also add an assertion to the handler which gives the precondition for its invocation. With these additions the precondition for each raise of a named exception can be determined, and it is possible to

check that the handlers achieve the postcondition of the subprogram in which they appear.

However, various aspects of Ada exception handling are not so easily axiomatized. Exceptions raised by function invocations cause problems because the axiomatic method relies on importing functional expressions into the assertion language. This can only be done if functions in the programming language emulate their mathematical analogue; a function which raises an exception clearly does not do so.*

The others clause in a handler also gives rise to difficulties, because a raise statement in such a handler can propagate an unnamed exception. The method of Luckham and Polak assumes that exceptions are only propagated within their scope. They recommend therefore that the raiseing of an unnamed exception should be avoided, and that a globally known exception 'error' be raised instead.

There are three pre-defined exceptions that have not yet been considered. 'Select_error' and 'tasking_error' relate directly to the interprocess communication primitives of Ada. They both represent programming errors, such as trying to communicate with an inactive process. A 'select_error' is generated when a select statement has all its guards false; a simpler alternative would have been to require the inclusion of an else part.

The exception 't'failure' can only be generated by an explicit raise statement, in task 't' or in any other task from which the name 't' is

---

* Provisional Ada [54] made a valiant attempt to ensure that functions (as distinct from procedures with results) were really functions. However, this distinction has been dropped from the Revised language [96].

visible. The report advises that a task's failure exception be generated only as a last resort, when attempts at ordinary communication have failed. Because the failure exception may interrupt a process at an arbitrary point the only sensible action that it can take is some clearing up followed by termination; however, this is solely at the discretion of the interrupted process. At the time of writing, the American National Standards Institute has proposed that the task failure exception be removed from Ada before it is accepted as a U.S. standard. Forcible termination of another process could still be performed with the abort statement. The normal method of achieving termination is of course by co-operation: one process passes a message to another informing it that its services are no longer required.

3.6 Summary

There is a great diversity in the intended scope of the exception handling mechanisms described in this chapter. That of CLU is probably the most constrained; it can be considered as a way of providing multiple returns. Levin's mechanism and Mesa signals are the most elaborate, but in different ways. The Ada designers clearly conceived of a limited mechanism, but the interaction between it and other features of the language gives rise to considerable complexity.

Chapter 4

EXCEPTION HANDLING IN ACTION

It is encumbent on anyone who proposes a language feature to show that it is useful. If a programming language is to remain simple one must be very cautious about including a new feature. It is well known that any computable problem can be solved with only an alternative construct and either an unbounded repetitive construct or the ability to name and call pieces of program. Anything in excess of this minimum must justify the complexity it adds to the language by removing even more complexity from typical programs.

Levin [59] is one of the few authors to take this responsibility at all seriously. He admits that an exception handling mechanism must be capable of solving a variety of "real world" problems naturally. But the requirement is actually stronger: there must be no natural solution to those problems without the mechanism, for if there is then we are guilty of enlarging a language gratuitously.

Any use of an exception handling mechanism to achieve termination of a routine can be simulated by making the result of the routine a union of types. Alternatively, it may be appropriate to replace the routine by several routines, each applicable to only a part of the domain of the original. An exception mechanism like that of Mesa, when used to achieve resumption, is semantically equivalent to the provision of one or more procedure parameters to the routine which generates the exception. Levin's mechanism can be similarly replaced, with some complication if the exception is raised on a data abstraction rather than a control abstraction. In this

case procedure parameters to the data structure may be needed. As I will show, in the presence of multiple processes it may be appropriate to use inter-process communication in place of Levin's mechanism.

Sections 4.1 and 4.2 of this chapter examine programming problems provided by Levin and other authors as illustrations of the utility of their exception handling mechanisms. Section 4.3 presents some uses of the Mesa mechanism extracted from real software. Consideration is given in each case to ways in which the problem might be solved without an exception handling mechanism. An attempt is made to compare the various formulations and to decide which is simpler, more natural, easier to modify and less prone to error. Such judgements must of their nature be subjective, and it is of course possible that the reader may disagree with my assessment of these examples. In part, such disagreements account for the multiplicity of programming languages. But the existence of these differences of opinion is in itself an argument in favour of simpler languages: it is much easier to have two language designers agree on the structure and features of a minimal language than on a large, eclectic one with many interactions between its constructs. This principle of language design may be summarised as "if in doubt, leave it out".

Given two means of expressing a particular concept, and agreement that one is superior to the other on methodological grounds, it still may be prudent to choose the inferior construct if it permits a vastly more efficient implementation. Although a small increase in execution time may be an acceptable price to pay for methodological advantage, I must be sure that my proposals for avoiding exception handling do not introduce major inefficiencies. Section 4.4 examines some actual and hypothetical implementations.

It is not possible for me to include an examination of every example

which purports to justify exception handling. I have chosen the examples which seem to put the strongest case. In the case of Mesa I have included some examples which have not been previously published. One objection which may be raised against the material presented here is that all the examples are small. This is partly because of the inherent limitations of the medium of presentation. A full examination of a large piece of software would make excessive demands of the reader, as well as lengthen a chapter which may already be too long. More importantly, most of the reader's efforts would not be directed at the use made of the exception mechanism, but rather at understanding the overall structure of the system. Once one's attention is directed at a single exception-generating routine and its caller, the situation in a large example is fundamentally the same as in a small one.

One benefit that may accrue from the use of an exception mechanism in a large system is better design. It is common to use syntactic procedure interfaces to sketch out the module structure of a system, but it is unfortunately not yet common to define the semantics of these procedures during the design stage. If the procedure interfaces are written in a language like CLU, which requires that all the exceptions a procedure may raise must be mentioned in the procedure heading, then the system designer may be encouraged to think of all the cases with which the procedure must deal.

It is only fair to point out that the same benefits can be obtained by a disciplined use of some other means of communicating exceptional results, such as union results or procedure arguments. It should also be observed that languages like Mesa and Ada which do *not* require exceptions to be listed in the procedure heading encourage the opposite mode of thinking. The less common cases are likely to be excluded from the design under the belief that they can be added later using exceptions. The result of doing this is likely to be a system structure that is, from its very inception, inappropriate for

many of the events that will occur within it.

I have also excluded some examples because the case they make for exception handling is very weak. For example, Wasserman [99] discusses a routine called 'search' which determines whether a name is in a table. He presents it in three forms:

(i) as a function with a boolean result;

(ii) as a procedure which raises two exceptions, 'Found' and 'Notfound';

(iii) as a procedure which always raises 'found', but which returns a (character!) result indicating whether the name was really found.

Form (i) results in very transparent code which is not improved by the introduction of exception handling. Wasserman does at least give comparative examples; unfortunately, the comparison is spoiled by failing to use the search routines to solve a stated problem. (Incidentally, form (iii) was an attempt to simulate Zahn's situation-case statement [104]. I am unsure which construction benefits by the comparison.)

In attempting to provide comparative examples I am faced with a number of difficulties. The most obvious is choosing a specification and a programming style which are neutral with regard to exceptions.

Those who advocate exception handling mechanisms sometimes specify problems by asking for a routine which provides a "normal" service 'A' and "exceptional" services 'B' and 'C'. They then find that such problems are elegantly solved by their exception mechanism! By posing their problems in such terms, they disqualify any solution which is symmetric with respect to 'A', 'B' and 'C', even though such a solution may be just as appropriate in the surrounding context.

When discussions about the merits of the goto statement were popular, advocates would ask: "How do I get from here to there without a goto?" As soon as one attempts to answer such a question the argument is lost, for one is arguing about flow of control rather than how to solve a programming problem. Such arguments have lead to the introduction of exit, break and leave statements which do indeed get you from here to there without a goto - but roses smell as sweet by any name. The only way to deal with such questions is to abstract the requirements away from the implementation and then to construct a program (without any jumps) that satisfies them. I have to do a similar thing with problems illustrating exception handling.

Chosing a programming style is again a subjective problem. The options are best illustrated by examining a real example.

4.1 The CLU Sum_Stream Example

This example is taken from the CLU Reference Manual [61, §12.3]. The requirement is to produce a

> procedure 'sum_stream' which reads a sequence of signed decimal integers from a character stream and returns the sum of those integers. The stream is viewed as containing a sequence of fields separated by spaces; each field must consist of a non-empty sequence of digits, optionally preceded by a single minus sign.

The CLU manual then specifies that the heading of the procedure is

```
sum_stream = proc(s:stream) returns(int)
                signals( overflow, badformat(string)
                       , unrepresentable_integer(string)
                       )
```

and states the conditions under which the various exceptions are raised.

Another way of looking at this problem is to consider 'sum_stream' as a procedure which returns one of four different kinds of result under different circumstances. Specifically, let the result be of type 'SumResult', where

```
type SumResult = oneof( overflow : singleton
                      ; int
                      ; unrepresentable_integers : string
                      ; bad_format : string
                      )
```

The declaration '$\underline{\text{type}}$ O = $\underline{\text{oneof}}(T_1; T_2;\ldots; T_n)$' defines 'O' to be the discriminated union of the types denoted by 'T_1', 'T_2',... and 'T_n'. It is *not* required that all these types are different; if 'T_i' and 'T_j' happen to denote the same type, the oneof does *not* collapse in the way a set-theoretic union would do. That is why I have chosen the symbol oneof rather than union. It *is* required, however, that the 'T_i' are syntactically distinct names; to facilitate this some of the 'T_i' may be renamed by writing 'n_i:T_i'. This requirement arises because there are '3n' functions contained in type 'O', which must have distinct names.

```
From_T_i : (T_i) → O
To_T_i   : (O)   → T_i
Is_T_i   : (O)   → boolean
```

The '$\text{From\_}T_i$' functions are the injection operators, i.e. those used to construct values of the oneof type 'O'. Thus a 'SumResult' can be constructed by writing 'From_bad_format("-12s4")' or 'From_int(256)'. The projection functions '$\text{To\_}T_i$' are obviously partial; '$\text{To\_}T_i(x)$' is only

defined if 'x' was constructed from type 'T_i', which is determined by the predicate 'Is_T_i(x)'. (These functions are formally defined in Chapter 5.)

The name 'singleton' denotes a type which contains only one value. This value has no explicit denotation; instead operations on a 'singleton' are applied to an empty parameter list. Thus a 'SumResult' value can be constructed by the application 'From_overflow()'.

Having explained the notation that will be used for types, we can return to the specification of 'sum_stream'. In my formulation 'sum_stream' has the heading

```
sum_stream = proc(s:stream) returns SumResult
```

Should the sum of the numbers in the stream (or an intermediate sum) exceed the implemented range of integers 'sum_stream' will indicate that an overflow has occurred. If one of the numbers in the input stream exceeds the implemented range of integers then 'sum_stream' will indicate that an 'unrepresentable_integer' was found. If the stream contains a string which does not denote an integer 'sum_stream' will indicate that its input was in a 'bad_format'. These indications will be provided either by exception signals or by the oneof result as appropriate.

The CLU version uses the services of two library routines, 'getc' and 's2i'. The first has type

```
proctype(stream) returns(char)
            signals( end_of_file, not_possible(string) )
```

and returns the next character from the stream unless the stream is empty, in which case 'end_of_file' is raised. The signal 'not_possible' is generated if the operation cannot be performed on the stream, as would be

the case if it were an output stream or did not consist of characters. The CLU manual assumes that 'getc' is always possible on the given stream.

The procedure 's2i' converts character strings to integers; its type is

```
proctype (string) returns(int)
                  signals( invalid_character(char)
                         , unrepresentable_integer
                         , bad_format
                         )
```

The signal 'unrepresentable_integer' is generated if the string represents an integer outside the implemented range; 'invalid_character' is signalled if the string contains a character which is neither a digit nor a minus sign, and 'bad_format' is signalled if a minus sign follows a digit, if there is more than one minus sign, or if there are no digits.

My version will assume the existence of some similar routines. 's2i' will have type

```
proc (string) returns
              oneof( int
                   ; invalid_character : char
                   ; bad_format : singleton
                   ; unrepresentable_integer : singleton
                   )
```

For the sake of variety, I will not use a procedure 'getc' which returns a oneof result, but will instead use a predicate 'Endof' and a procedure 'Nextc'. Their types are

```
Endof : proc (stream) returns boolean
Nextc : proc (stream) returns char
```

'Nextc' is only partially defined: if not 'Endof(S)' then 'Nextc(S)' returns the next character from 'S'. Like the CLU authors, I will assume that 'Nextc' is applicable to the given stream: a predicate 'IsNextcPossible'

would permit this assumption to be checked.

```
sum_stream = proc(s:stream) returns(int)
             signals( overflow
                    , unrepresentable_integer(string)
                    , bad_format(string));
    sum:int := 0;
    num:string;
    while true do
        % skip over spaces between values;
        % sum is valid, num is meaningless
        c:char := stream$getc(s);
        while c = ' ' do
            c := stream$getc(s);
        end;
        % read a value; num accumulates new number,
        % sum becomes previous sum
        num := '';
        while c ≠ ' ' do
            num := string$append(num,c);
            c := stream$getc(s);
        end;
        except when end_of_file: end;
        % restore sum to validity
        sum := sum + s2i(num);
    end;
    except
        when end_of_file: return(sum);
        when unrepresentable_integer:
                    signal unrepresentable_integer(num);
        when bad_format, invalid_character(*):
                    signal bad_format(num);
        when overflow: signal overflow;
    end;
end sum_stream;
```

Figure 4.01: The sum_stream procedure.

The CLU implementation of 'sum_stream' is shown in Figure 4.01. The outer loop contains two inner loops, the first to skip spaces and the second to accumulate digits. If the end of the stream 's' is encountered in the second inner loop the raising of the 'end_of_file' exception causes a jump out of the loop to the null handler at line 21. Control then passes

to line 23: the outer loop is only terminated when 'getc' is invoked again at line 12. This raises the 'end_of_file' exception a second time and causes the execution of the return statement in the handler on line 26.

```
    proc sum_stream = (s:stream) returns SumResult
        var sum : int = 0
        repeat
4           {skip over spaces between values; sum is valid}
            var c : char
            repeat
                if Endof(s) return From_int(sum) fi
8               c := Nextc(s)
                when c ≠ ' ' exit
            again
            { read a value; num accumulates new number,
12            sum becomes previous sum }
            var num : string = ' '
            repeat
                num := string.append(num,c)
16              when Endof(s) exit
                c := Nextc(s)
                when c = ' ' exit
            again
20          {restore sum to validity}
            if Is_int( s2i(num) )
            then
                const newsum : oneof(int; overflow:singleton)
24                                  = sum + To_int(s2i(num))
                if Is_int(newsum)
                then sum := To_int(newsum)
                else return From_overflow()
28              fi
            elif Is_unrepresentable_integer( s2i(num) )
            then return From_unrepresentable_integer(num)
            else return From_bad_format(num)
32          fi
        again
    end of sum_stream
```

Figure 4.02: sum_stream without exceptions.

The discussion in the CLU manual states that most of the handlers have been placed at the end of the outer loop to avoid cluttering the code.

One of the consequences of this is that 'num', which is otherwise used only between lines 16 and 23, must have a much larger scope so that it is available for the handlers. This is the reason for the comment at line 9.

My formulation of 'sum_stream' without exceptions appears as Figure 4.02. It follows the CLU example line by line as far as is possible. I have used the return statement in four different places to exit from what would otherwise be an infinite loop; these four cases correspond to the four different exception handlers in the CLU version. I also wished to avoid the use of a while ... do construction for loops in which the test does not naturally come at the beginning: I have used a loop delimited by repeat ... again within which when 'b' exit clauses may appear.

All the exceptions generated by the CLU version of 'sum_stream' originate as exceptions raised by lower-level routines. 'Sum_stream' passes them on to its caller: although some of the names may be the same, the meaning of the exceptions is different at the different levels. For example, the 'bad_format' exception generated by 'sum_stream' has a broader meaning than the 'bad_format' exception signalled by 's2i' (it also has a result value). The 'overflow' exception is generated by the '+' operator; for consistency I have assumed in my version that '+' returns a result of the type 'oneof(int; overflow:singleton)'.

The type 'SumResult = oneof(overflow : ...)' and the type of 'newsum', that is 'oneof(overflow : singleton; int)' both have operators 'From_overflow()'. Allowing distinct types to use the same operator name is known as *overloading* and occurs with many built in types. For example, '+' is usually overloaded with definitions between reals and between integers, and may also have other definitions such as between vectors or matrices. The potential ambiguity thus introduced is resolved either

implicitly by examining the types of the arguments and result, or explicitly by qualification, as in 'SumResult.From_overflow()'. (CLU uses $ instead of the point.) How much overloading is allowed, and in which situations qualification is required, varies from one language to another; no difficulty should be found in resolving the examples.

Comparing the two versions of sum_stream, the main difference is that in mine an explicit test must be made for each exception. I consider this to be an advantage because it forces the programmer to think about all the possible cases.

To illustrate this, suppose that the specifier or programmer had forgotten to consider the case where the last number in the stream is not followed by a space. In the CLU version this would lead to the omission of the 'end_of_file' handler at line 21. If this program were then applied to an input stream without trailing spaces, the effect would be to exclude the last number in the stream from the total. This would happen because the 'end_of_file' exception raised by 'getc' on line 19 would be caught by the handler at line 26; this handler performs an immediate return without adding in the digits collected in 'num'. If the same oversight were made when constructing my version, the effect would be to omit line 16. If this program were applied to the same input stream, 'Nextc' at line 17 would be called when no characters remained in the stream. This would cause an immediate abort. In this sense my methodology increases robustness*.

The robustness arises because of the explicit association of "handler" code with the routine which discovers the problem. Indeed, the existence of a call of 'Nextc' not guarded by a test of 'Endof' ought to alert the programmer to check the problem specification. However, the fact that partial functions were used rather than routines which return oneof results

---

* This sense of the word is that adopted by Dijkstra [23], p. 56 , Hehner [42], p. 278 and Bron and Fokkinga [12a].

is irrelevant. The end result is the same if, instead of 'Nextc' and 'Endof', the library had supplied a routine

```
getc : proc(stream) returns oneof( char
                                 ; end_of_file:singleton
                                 ; not_possible:singleton
                                 )
```

The loop which accumulates characters would then have appeared as

```
repeat
    num := string.append(num, To_char(c))
    c := getc(s)
    when Is_end_of_file(c) exit
    when To_char(c) = ' ' exit
again
```

and omission of the 'Is_end_of_file' test would have lead to an illegal invocation of the 'To_char' function, which would also cause an abort.

This argument does not apply in the case of CLU signals. CLU expressly permits a call that raises an exception not to be followed by a handler; in such a situation, the handler from the surrounding lexical scope is used.

It may be that the reader is worried by the introduction of a loop construction with exits. This is the inevitable consequence of trying to imitate a program which uses exceptions to achieve premature termination of loops. If the program is written in a top down manner without reference to the CLU version no exists or returns are necessary because no loops are used. This alternative formulation is shown in Figure 4.03. It uses Dijkstra's guarded command notation [22] and call and refinement [42]; both of these techniques produce programs which occupy a greater number of lines, but one should not be mislead into thinking that this implies greater complexity. In my view Figure 4.03 is easier to understand than Figure 4.02.

However, the object of this thesis is to argue the merits of exception handling, not of loops. The remaining examples in this chapter will be written (without exception handling) in a way that follows as closely as

```
proc sum_stream = (s:stream) returns r:sumresult
    var sum:int = 0
    var c:char
    var Num:string
    Add Numbers in stream to sum

where Add Numbers in stream to sum is
    if Endof(s) => r := From_int(sum)
    [] ¬Endof(s) =>
            c := Nextc(s)
            if c = ' ' => Add Numbers in stream to sum
            [] c ≠ ' ' => Read number and add it to sum
            fi
    fi

where Read number and add it to sum is
    Num := ''
    Append Non_space chars to Num
    if Is_int(s2i(Num)) =>
            Add Num and Numbers in stream to sum
    [] Is_unrepresentable_integer(s2i(Num)) =>
            r := From_unrepresentable_integer(Num)
    [] Is_bad_format(s2i(Num))
       or Is_invalid_character(s2i(Num)) =>
           r := From_bad_format(Num)
    fi

where Append Non_space chars to Num is
    num := string.append(Num,c)
    if Endof(s) => skip
    [] ¬Endof(s) =>
            c := Nextc(s)
            if c = ' ' => skip
            [] c ≠ ' ' => Append non_space chars to Num
            fi
    fi

where Add Num and numbers in stream to sum is
    const NewSum:oneof(int; overflow:singleton) = sum + s2i(Num)
    if Is_int(NewSum) =>
            sum := To_int(Newsum)
            Add Numbers in Stream to Sum
    [] Is_overflow(NewSum) =>
            r := From_overflow()
    fi

end of sum_stream
```

Figure 4.03: Sum_stream without exits or exceptions.

possible the original solution (with exception handling); return and exit statements will be used where necessary to contrive the similarity. I feel that the proponents of exception handling will be inclined to favour such a style; in any case it facilitates the comparison of the two versions. It should be clear from the above that this method of presentation neither means that exits are necessary if exception handling is removed, nor that I support their use.

4.2 Levin's Examples

In Section 7 of his thesis [59] Levin presents five examples to demonstrate the "practical applicability" of his mechanism. He claims both that his solutions are natural and that the problems cannot be successfully handled by other mechanisms. In this section I discuss each of his examples in turn. In the case of arithmetic exceptions I examine the CLU formulation as well as Levin's.

4.2.1 The Symbol Table Problem

For his first example Levin presents the symbol table data type. Looking up a name in a symbol table is representative of a common class of operations which may return different kinds of results.

Levin defines a 'symbol_table' to be a set of pairs '<name, value>' where the first elements of the pairs are distinct. The cardinality of the set is at most 'n', i.e. the symbol table is of bounded size. If the pair '<s,v>' has previously been inserted into 'st', the result of 'lookup(st, s)' is 'v', otherwise it is a notification that 's' is absent. If an attempt is made to insert a new name into a table which is already full then provision for some "exceptional" action must be available.

Levin's specification is in a style close to that of Alphard [103]; it is reproduced as Figure 4.04. Considering first the 'lookup' function, it will be seen that it may generate two exceptions, 'absent' and 'present', and that the latter has a parameter of type 'V'. The preconditions of these exceptions tell us that each invocation of 'lookup' will generate exactly one of them. No pre- or postconditions are given for 'lookup' itself. Levin states that an omitted precondition is identically 'true', and that the omission of a postcondition means that "all parameters remain unchanged". Thus 'lookup' is a total function in the mathematical sense.

```
form symbol_table(n:integer, T:form<=, :=>, V:form<=, :=>) =
beginform
specifications
    requires n ≥ 1;
    let symbol-table = assoc: {<s:T,v:V>};
    invariant
        cardinality(assoc) ≤ n
            and a,b ε assoc => (a.s = b.s => a.v = b.v);
    initially symbol-table = {};
    functions
        lookup(st:symbol-table, str:T)
        raises
            absent on lookup policy broadcast
                pre = a∉st => a.s≠str;
            present(v:V) on lookup policy broadcast
                pre = ∃a∈st st a.s=str and a.v=v
        endraises

        insert(st:symbol-table, str:T, val:V)
        raises
            full on insert policy broadcast
                pre = cardinality(st)=n and (a∉st => a.s≠str)
        endraises
        post normal = if ∃a∈st st a.s=str
                      then st = st' - {a} U {<str,val>}
                      else st = st' U {<str,val>}
        post full = st=st';
```

Figure 4.04: Levin's symbol table specification.

I have tried to interpret this specification according to the verification methodology set out in [59].

The reader will recall that after an exception is raised and handled, control returns to the procedure which generated the exception. Section 6.2 of [59] states that "a module that defines a particular exceptional condition on its abstraction also defines two predicates that hold before and after the condition has been handled". I would therefore expect the specification of 'lookup' to provide postconditions for 'absent' and 'present' as well as preconditions. In the absence of explicit postconditions I can only assume them to state that "all parameters remain unchanged". But which set of parameters, those of the exception or those of 'lookup'? If the former, then the postcondition of 'absent' is vacuously 'true' (because it has no parameters) and so the postcondition of 'lookup' must also be 'true'. This contradicts the previous assumption. So I must assume the latter interpretation, that the implicit postcondition for an exception states that all of the parameters of the operation remain unchanged. This means, for example, that the handler for 'absent' is debarred from changing the symbol table: in particular it cannot insert the name which was found to be absent. I do not believe that Levin intended to define such a restrictive mechanism but see no other way of interpreting his specification and verification methodology.

Similar problems arise with 'insert'. Levin tells us that the last four lines represent the postcondition for 'insert'. The notation is intended to separate the part of the postcondition which applies in the "normal" case from the part which applies when 'full' has been raised. The actual postcondition is stated to be

(¬raised(full) and post normal) or (raised(full) and post full)

where 'raised(c)' if and only if 'c' was generated by 'insert'. There seems to be no advantage in separating the terms in this way because 'raised(full)' is not available in the program. Nevertheless, it seems clear that the handler for 'full' is prohibited from altering 'st'.

```
representation
    unique
        names:vector(T,1,n),
        values:vector(V,1,n),
        last:integer
            init last := 0;
    rep(last,names,values) =
            {<names[i],values[i]> | i ε [1,last]};
    invariant
        last ε [0,n] and ((i,j ε [1,last] and i ≠ j)
                                    => names[i] ≠ names[j])
implementation
    body lookup =
        first j:upto(1,last) suchthat names[j]=str
        then raise present(values[j])
        else raise absent;
    body insert
        out normal = ∃iε[1,last] st
                (names[i]=str and values[i]=val and
                ∀jε[1,last'] ∃kε[1,last] st
                (names[j]'=names[k] and i ≠ j =>
                values[j]'=values[k]))
        out full = last'=n and ∀iε[1,last'] names[i] ≠ str
        begin
            first j:upto(a,last) suchthat names[j]=str
            then values[j] := val
            else if last<n
                 then last := last+1; names[last]:=str;
                        values[last] := val
                 else raise full
                 fi
        end
endform
```

Figure 4.05: Levin's symbol table implementation.

The implementation of Levin's symbol table is reproduced as Figure 4.05. The out assertions are the concrete versions of the appropriate postconditions. All the statements which generate exceptions are the last actions of the procedures in which they occur; Levin's resumption mechanism is used to simulate a termination mechanism.

```
shared t:symbol-table(47, string, integer)
unique r,s:string, v,w:integer
    ...
1:begin
        < set s >
        lookup(t,s) [present(x): v:=x | absent: v:=0]
        < use v >
            ...
        < set r and w >
        insert(t,r,w) [full: → leave 1]
            ...
end
```

Figure 4.06: Sample use of Levin's symbol table.

Figure 4.06 shows a sample use of the symbol table. Its action is described in [59]:

> The invocation of 'lookup' sets 'v' to the value associated with 's', if any, or zero if 's' does not appear in 't'. The invocation of 'insert' expects to enter the pair '<r,w>' into 't', but will leave the block labelled '1' if the table is full.

In seeking to provide a similar example which does not use an exception mechanism one is faced with a number of choices. 'Lookup' is naturally represented by a function with a result of type 'oneof(V; absent:singleton)'. Using a union for the result of 'lookup' enables the symbol table abstraction to treat both results on an equal footing and leaves it to the user of the

abstraction to decide which result, if either, is "exceptional". It was to gain the same advantage that Levin introduced two exceptions rather than just one.

Such symmetry is not required when inserting a new name into a possibly full table. One technically sound (but inefficient) way of dealing with this eventuality is to make 'insert' a partial operation and to provide a predicate 'Is_insertable(st,n)' which is true if and only if 'insert(st,n,v)' is permitted.

Another method of dealing with a full table is to invoke a procedure parameter. Such a parameter could be provided either when the table is first created or on each call of 'insert'. If the first course is adopted it might be wise to provide an operation on symbol tables which changed the handler procedure. This is very similar to what was done in the AED Free Store package (see Section 2.1). My formulation of 'insert' uses a procedure parameter to deal with full tables. This has an implementation advantage over the use of partial operations: it is easier to check that a total operation is provided with the correct number of parameters than it is to check that a partial operation is invoked only from valid states. It also has the didactic advantage of introducing a new technique - that of *notification procedures* - into our list of alternatives to exception handling mechanisms. This technique requires only a minimal transformation of Levin's code.

```
type symbol-table(n:integer; T:type with =, :=
                           ; V:type with =, := )
is
specifications
    requires n ≥ 1
    let SymbolTable = assoc {<s:T, v:V>};
    invariant cardinality(assoc) ≤ n
          and a, b ε assoc => (a.s=b.s => a.v=b.v);
    initially symbol-table = {};
    operations
        function lookup(st:symbol-table; str:T)
                 returns r:oneof(V; absent:singleton)
            post = if ∃a ε st st a.s=str
                   then r = From_V(a.v)
                   else r = From_absent()
        procedure insert(var st:symbol-table; str:T; val:v; h:proc())
            let tablefull = (cardinality(st)=n) and (a ε st => a.s≠str)
            pre = tablefull => pre h
            post = if tablefull
                   then post h
                   else if ∃ a ε st st a.s = str
                        then st = st' - {a} ∪ {<str, val>}
                        else st = st' ∪ {<str,val>}
```

Figure 4.07: Revised symbol table specification.

The revised specification is shown in Figure 4.07. The 'lookup' function has a single postcondition and the default precondition of 'true'. Procedure 'insert' has an extra parameter 'h', a parameterless procedure. The precondition of 'insert' is that the precondition of 'h' should be satisfied whenever the table is full. The let declaration simply defines 'tablefull' as an abbreviation. The postcondition of 'insert' depends on that of 'h'; I feel that this is a more realistic formulation than Levin's which debarred the handler for 'full' from altering 'st'.

```
representation
    unique
        names : vector(T,1,n),
        values : vector(V,1,n),
        last : integer
            init last := 0;
    rep(last,names,values) = {<names[i],values[i]> | i∈[1,last]};
    invariant
        last ∈ [0,n] and ((i,j ∈ [1,last] and i ≠ j)
                                    => names[i] ≠ names[j])
implementation
    body lookup =
        first j:upto(1,last) suchthat names[j]=str
        then r := To_V(values[j])
        else r := To_absent();
    body insert
        out = (∃i∈[1,last] st
                    (names[i]=str and values[i]=val and
                    ∀ j∈[1,last'] ∃ k∈[1,last] st
                        (names[j]'=names[k] and i≠j =>
                            values[j]'=values[k]))
              ) or post h
        begin
            first j:upto(1,last) suchthat names[j]=str
            then values[j]:= val
            else if last<n
                 then last := last+1; names[last]:=str;
                      values[last] := val
                 else h()
                 fi
        end
endform
```

Figure 4.08: Symbol table without exceptions

My symbol table implementation is presented in Figure 4.08. Two of Levin's *raise* statements have become assignments to a result variable; the third has become a procedure call. The *out* assertion has been modified in a similar way to the abstract specification.

```
shared t:symbol-table(47, string, integer)
unique r,s:string
     , v,w:integer
     , u:oneof(integer; absent:singleton)
     ...
l:begin
     <set s >
     u := lookup(t,s)
     if Is_integer(u) then v := To_integer(u)
                      else v := 0
     fi
     <use v>
         ...
     <set r and w>
     insert(t, r, w, proc:leave l)
         ...
end
```

Figure 4.09: Using the revised symbol table.

Comparing Figures 4.06 and 4.09, it does not seem to me that the use of an exception mechanism produces a 'lookup' function whose use is easier to understand. Moreover, Levin's example violates his own restriction that handlers must be procedure calls; if the two assignments to 'V' are turned into procedure calls his example is considerably more difficult to understand. This is perhaps unfair, because the handlers are abnormally simple; if they were more complicated I would have wanted to use procedure calls in the limbs of my if statement.

Nevertheless, these considerations lead us to ask *why* Levin's handlers are so simple. Assigning 'v' to zero and continuing with the computation is sensible in two kinds of situation. The first occurs when zero is not valid as a value to be inserted in the table. This implies that the table abstraction should have been instantiated with some other 'V', say 'Nautral'. Thus 'v' is really of a union type: it is either a 'V' or it is zero. The program fragment represented by '<use v>' will presumably test 'v' to see if it is zero before deciding

how to proceed.

There is nothing new or surprising about this use of zero as an "impossible" value: it has a history going back to the days of assembly language and Fortran. However, the advantages of languages like Pascal, CLU and Alphard is that one can construct types containing just the values one needs and thus obtain clearer code and more checking at compile time. Using union types preserves these benefits at the cost of having to make explicit checks and type conversions. Adding "impossible values" to existing types allows those conversions to be avoided, but negates many of the benefits of strong typing. I return to this question in Chapter 6.

The second kind of situation in which it is sensible for 'v' to be assigned to zero occurs when zero is some sort of default value, and the computation which follows can be performed with equal validity on both zero and non-zero values. This is an unlikely possibility when a compiler's symbol table is involved, but imagine for a moment that Levin's exception mechanism has been used to implement a sparse array. Failure of a lookup function to find the requested element raises an exception; at the next level of abstraction the handler for this exception substitutes the appropriate default value (in this case zero).

In this second situation, the result of the operation which accesses an element of the sparse array is clearly of type integer, and not of any union type. I cannot therefore object to Levin's 'v := 0' handler on methodological grounds. However, an implementation of 'lookup' which returned a union result

would be in this case too just as readable as one which raised an exception. The relative efficiency of the two mechanisms is considered in Section 4.4.

4.2.2 The Inconsistent String Problem

Levin's second example is based upon his first but is intended to demonstrate a different kind of use for his exception mechanism. He shows how exception handling can be used to repair a data structure which has become inconsistent because of a failure of "memory" (i.e. a malfunction of the computer store).

He assumes that the strings stored in the symbol table are values of a user-defined data type and that their representation contains some redundant information. If this redundancy reveals an inconsistency the string module generates an exception. This exception is raised in the module using the string, which in this case is the symbol table module. In order that the symbol table may perform some recovery action Levin alters it to maintain a duplicate list of strings in the vector 'dnames'.

Figure 4.10 is part of Levin's string module; the function which tests equality is emphasised. It generates the exception 'bad-string' if one of its argument strings has a negative length. If the handler for 'bad-string'

does not correct the condition, it sets the erstwhile inconsistent string to the null string and generates a second exception, 'reset-string'.

```
module string(n:integer)
begin
    private length:integer, chars:vector(character,1,n)
    init length := 0
    condition reset-string policy broadcast
    condition bad-string(var s:string)
            policy broadcast-and-wait
        ...
    function =(s1,s2:string) returns b:boolean
    raises reset-string on s1,s2
    raises bad-string on s1,s2
    begin
        if s1.length<0
        then
            raise s1.bad-string(s1)
            if s1.length<0
            then s1.length := 0; raise s1.reset-string;
                 return
            fi
        fi
        if s2.length<0
        then
            raise s2.bad-string(s2)
            if s2.length<0
            then s2.length := 0; raise s2.reset-string;
                 return
            fi
        fi

        if s1.length = s2.length
        then
            first i:upto(1,s1.length)
            suchthat s1.chars[i] ≠ s2.chars[i]
            then b := false
            else b := true
        else b := false
        fi
    end
end
```

Figure 4.10: Levin's string module with exception detection.

```
condition lost-entry policy broadcast-and-wait

function lookup(st,s)
raises lost-entry on st
raises absent on lookup
raises present(v) on lookup
begin
    for i:upto(1,last)
    do
        if =(names[i],s) [names[i].bad-string(str):str := dnames[i]]
        then raise present(values[i]); return
        fi
    od [ names[i].reset-string: fixtable(st,i) → return ]
    raise absent
end

routine fixtable(st:symbol-table, i:integer)
begin
    names[i]:= names[last]
    dnames[i]:= dnames[last]
    values[i]:= values[last]
    last := last-1
    raise st.lost-entry
end
```

Figure 4.11: Re-implementation of lookup.

This string module is used by the revised 'lookup' operation shown in Figure 4.11. If 'bad-string' is raised on a name in the symbol table the value of the appropriate string in the duplicate vector 'dnames' is copied into it. If the copy is also bad then 'string' will raise 'reset-string'. This will be handled by the routine 'fixtable', which uses local information to remove the corrupted entry from both 'names' and 'dnames'. It then generates the exception 'lost-entry' so that the users of the symbol table know that it is not reliable.

Despite its heading 'lookup' is not a function: it is liable to alter its symbol-table parameter. Moreover, if an irrepairably damaged entry should be

found in the table neither 'absent' nor 'present' will be generated even though the name 's' supplied as a parameter to 'lookup' is present in 'names'. Similarly, not only is the string equal operation not a function (because it may alter its parameters), it is not even symmetric. Levin's exposition does not bring out these difficulties because it does not provide a specification.

I cannot believe that this example illustrates an exception handling problem that might arise in a real-world system. It is unrealistic for at least three reasons.

First, remember the assumption that the negative string length arises from "memory failure". On a conventional machine, any unreliability in the store will be at least as likely to corrupt the duplicate strings and the code of the program (and thus the exception handlers) as it is to cause a negative string length in the name table. Levin's example begins to be realistic only with a programming language and architecture that allow one to specify that the code, duplicate strings and name table reside in different physical stores.

Second, why does the representation of strings reserve a bit for the sign of the length, when it must always be positive? Is it solely for error detection? If so, it would be better to use it as a parity bit.

Third, the separation of the detection and correction of string errors into two different levels of abstraction is a design error. And it is this very error which gives rise to the need for exception propagation between levels. If the string module is going to operate with an unreliable store, and if a duplicate list of names is going to be maintained, it should clearly be the string module which maintains it. This increases the likelihood of error detection as *any* corruption in either of the strings can be detected. Levin's formulation detects a change in the sign bit only. On the other hand, since errors in the duplicate strings are presumably just as likely as errors in the original strings, the available error correction is minimal. With one hundred per cent redundancy it is possible to do a

lot better than this.

Despite the artificial nature of the problem it is instructive to see how it might be solved without exception handling. Since the 'bad-string' and 'reset-string' exceptions relate to an individual string, it is natural to replace them by notification procedures private to each string. Such procedures might be provided as parameters to the 'NewString' routine, or they might more conveniently be "frozen" into a 'NewStringwithStandardRecovery' routine. In either case routines to retrieve and change the notification procedures must be provided. Part of a 'string' module is shown in Figure 4.12. The routines 'BadStringRt' and 'SetBadStringRt' are provided to access the badstring notification procedure; within the 'string' module it can be accessed as a component of the representation, but this is assumed to be hidden from users of 'string'. If the string equality routine finds a negative length it first assigns the corrupted string variable to the result of applying the 'Badstr' procedure. If the length is still negative it is zeroed and the 'Reset' procedure of the string is applied. This parallels exactly the action of Levin's equality routine.

When the routine is used, instead of enabling exception handlers one must set the appropriate recovery procedures. This is illustrated in Figure 4.13. Before the strings are compared for equality, 'lookup' changes the bad string routine of the appropriate element of 'names'; after the comparison it is reset. This resetting could be accomplished more conveniently if each string contained a stack of recovery procedures. If this change is made then my equality and lookup routines are essentially identical to the *implementation* Levin gives for his mechanism. This would seem to boost Levin's claim that his exception handling mechanism is a high level feature which reduces the amount of code the programmer must write.

Figure 4.12: String module with notification procedures.

```
module string
    defines =, BadStringRt, SetBadStringRt, NewString.
          , ResetRt, SetResetRt, NewString, ...
    representation is
        record( chars:array 1:n of character
              ; Badstr:proc() returns string
              ; Reset:proc()
              ; length:integer
              )

    function BadStringRt(s:string) returns p:proc() returns string

    is p := s.Badstr
    end of BadStringRt

    proc SetBadStringRt( var s:string; p:proc() returns string )

    is s.Badstr := p
    end of SetBadStringRt

    function ResetRt(s:string) returns p:proc()
    is  p := s.Reset
    end of ResetRt

    proc SetResetRt(s:string; p:proc() )
    is s.Reset := p
    end of SetResetRt

    proc =(var s1:string; var s2:string) returns b:bool is
        if s1.length < 0
        then s1 := (s1.Badstr)()
             if s1.length < 0
             then s1.length := 0
                  (s1.Reset)()
                  return
             fi
        fi
        if s2.length < 0
        then s2 := (s2.Badstr)()
             if s2.length < 0
             then s2.length := 0
                  (s2.Reset)()
                  return
             fi
        fi

        if s1.length = s2.length
        then first i:upto(1,s1.length)
             suchthat s1.chars[i] ≠ s2.chars[i]
             then b := false
             else b := true
        else b := false
        fi
    end of =
end of string
```

```
proc Lookup(t:symboltable; s:string) returns
                        r:oneof(Value; Notfound:singleton)
is
    for  i:upto(1,last)
    do
         proc FixTable() is
             names[i] := names[t.last]
             dnames[i] := dnames[t.last]
             values[i] := values[t.last]
             last := last-1
             t.LostEntry()
         end of FixTable

         proc UseDuplicate() returns s:string is
             s := t.dnames[i]
             SetResetRt(s, FixTable)
         end of UseDuplicate

         const OldBadStringRt : proc() returns string
                                = BadStringRt( names[i] )

         SetBadStringRt(t.names[i], UseDuplicate)
         if =(t.names[i], s)
         then SetBadStringRt(t.names[i], OldBadStringRt)
              r := From_Value(t.values[i])
              return
         fi
         SetBadStringRt(t.names[i], OldBadStringRt)
    od
    r := From_Notfound()
end of Lookup
```

Figure 4.13: Lookup using inconsistent string module with notification procedures.

There are two defences against this argument. The first is that there is very little extra code present in my version. Two fields and their access routines have been added to the string module, and three routine calls to 'lookup'. Given that the example was constructed specifically to demonstrate the advantages of Levin's mechanism I was surprised that the same effect can be achieved with very little extra effort. Do similar examples occur often enough to justify the complexity of Levin's mechanism?

The second defence is that the use of an exception handling mechanism,

by hiding what is really happening, actually *hinders* our understanding. This is a bold claim which I will justify through my experiences in constructing Figures 4.12 and 4.13.

Having written the code fragments shown, I naturally looked for ways of simplifying them. The repeated setting and re-setting of the notification procedures is displeasing and could be avoided. The same bad string routine is applicable anywhere in the symbol table module after a name has been inserted: the routine could be set just once and never changed. However, if I did that I would lose the exact parallel with Levin's routine which does indeed set up the exception handler just for the duration of the equality test.

It is also clear from the figures that the original 'Reset' routine of 'names[i]' is never used. Before 's1.Reset' is called, 'dnames[i]' has been assigned to 's1'. Similarly, the 'Badstr' routine of 'dnames[i]' is never applied: 'dnames' is only accessed as a consequence of applying the bad string routine of 'names'. Further, the preconditions established by 'Lookup' before calling these routines are identical: in both cases the string has a negative length. It is thus possible to simplify the representation of string *and* to provide more levels of recovery by using just one notification procedure.

Part of a string module which takes advantage of this simplification is shown in Figure 4.14. The local routine 'IsOK(s)' returns 'true' if the length of its parameter is positive or zero. If it is negative, the recovery routine of 's' is applied. The result of this routine is either the singleton 'Unrecoverable' or a new string value. If the former, 'IsOK' returns 'false'; if the latter, the new string is assigned to 's'. Because

```
type string ...
    representation is
        record( chars:array 1:n of char
              , Recovery:proc() returns
                    oneof(string, Unrecoverable:singleton)
              , length:integer
              )

    proc =(var s1:string, var s2:string) returns b:bool is
        proc IsOK(var s:string) returns b:bool is
            var r : oneof(string, Unrecoverable:singleton)
            if s.length < 0
            then r := s.Recovery()
                 if Is_string(r)
                 then s := To_string(r)
                      b := IsOK(s)
                 else b := false
                 fi
            else b := true
            fi
        end of IsOK

        if IsOK(s1) and IsOK(s2)
        then if s1.length = s2.length
             then first i:upto(1, s1.length)
                  suchthat s1.chars[i] ≠ s2.chars[i]
                  then b := false
                  else b := true
             fi
        else b := false
        fi
    end of
```

Figure 4.14: String module with single notification procedure.

the new string itself contains a recovery procedure, the recovery procedure of 's' is overwritten by this assignment. When 'IsOK' invokes itself recursively to test if 's' is now consistent, a further inconsistency will cause the *new* recovery procedure to be invoked.

In this way users of the revised string equality routine can provide as many levels of recovery as they wish, from zero upwards, rather than being constrained to supply exactly two. As far as I can see this solution cannot be programmed with Levin's mechanism; only one handler could be provided for a 'recovery' exception generated by '='. (Of course, it would be possible for that handler to apply the value of a routine variable, but Levin's exception mechanism was presumably designed to make such variables unnecessary.)

This gain in simplicity was not due to a flash of insight. It was due to the discipline of having to write out exactly what was happening in Levin's example. I obtained a simpler version *because* I did not use an exception handling mechanism rather than in spite of not using one. And this is true even though the example was inverted specifically to demonstrate the utility of Levin's mechanism.

4.2.3 Arithmetic Exceptions

Arithmetic overflow and underflow must bear much of the responsibility for creating the subject of exception handling. It is interesting to see how these exceptions are handled both with Levin's mechanism and in CLU.

Levin presents part of the specification of a form for floating point arithmetic. The exception handling is indirect because the handlers return to the operation with a result. For example, the 'overflow' exception

generated by 'add' has a var parameter: in the call

```
z := add(x,y) [overflow(v): v := maxfp]
```

the assignment of 'v' in the handler causes 'z' to become 'maxfp'.

The specification given for real in the CLU Reference Manual [61] defines a type which is easier to use in simple cases. The types of two of the operations are

```
add: proctype(real, real) returns(real)
            signals(overflow, underflow)
div: proctype(real, real) returns(real)
            signals(zero_divide, overflow, underflow)
```

Axioms are given to define the results of these operations when the exceptions do not occur. The exceptions are generated in the situations one might expect; an exact definition is unnecessary for our purposes.

Typical calls of these routines might be

```
z := add(x,y) except when overflow: z := Real_Max
              end
z := div(x,y) except when underflow:
                        if sign(x) = sign(y)
                        then z := Real_Min
                        else z := -Real_Min
                        end
              end
```

How can such a data-type be represented without the use of exceptions? Note that 'add' will return *either* an approximation to the sum of its arguments *or* an indication that the sum is outside the range of real. Similar observations hold for 'div' and the other operators. A natural

representation for this seems to be a oneof result: the headings of the 'add' and 'div' operations would then become

```
proc add(a:real; b:real) returns c:
    oneof(real; underflow:singleton; overflow:singleton)

proc div(a:real; b:real) returns c:
    oneof( real; zero_divide:singleton
         ; underflow:singleton; overflow:singleton )
```

I will omit the full specification because the characterization of approximate arithmetic is not my goal. Instead I will illustrate how these operators might be used:

```
var o:oneof(real; underflow:singleton; overflow:singleton)
o := add(x,y)

if Is_overflow(o) then z := Real_Max else z := To_real(o)
fi

if Is_underflow(div(x,y))
then if sign(x) = sign(y) then z := Real_Min
                          else z := -Real_Min
     fi
else z := to_real(div(x,y))
fi
```

These examples are very close parallels to those written in CLU. In the first, if 'add(x,y)' results in an overflow, 'z' is assigned to the value of 'Real_Max'. If it results in an underflow the CLU code will generate the exception 'failure("unhandled exception: underflow")' (assuming that there is no handler for 'underflow' on an enclosing block). In my version the function 'To_Real' will fail and one can expect the message "o is of type underflow, not real". I assume that this is what the

programmer intended: he may know that two very large positive numbers are being added and that underflow cannot occur, and that 'Real_Max' (rather than, say, '-Real_Max') is a suitable result in the case of overflow. Similar context considerations could make the second example realistic. My version assumes that the implementation will optimize the two identical function calls 'div(x,y)'; this avoids the introduction of a temporary variable and is a trivial optimization because 'div' is a pure function.

The most common use of these operations would be where no exceptions are expected and no sensible continuation is possible should one occur. The occurrence of an exception would indicate a programming error in some other module. Suitable diagnostics are obtained in CLU by simply writing

```
z := add(x,y)
```

which would generate the failure exception should underflow or overflow occur. This will provide the appropriate alarm provided that no enclosing block has a handler for underflow and that no higher level module handles the 'failure' exception.

With my functions one would have to write

```
z := To_real(add(x,y))
```

which would apply the 'To_real' function outside of its domain if underflow or overflow occurs. This will provide an implementation dependent alarm which cannot be intercepted at a higher level.

Of course, I do not expect every addition to be written with such a cumbersome syntax. There are several ways of abbreviating it. One is to introduce implicit coercions, i.e. to allow 'add(x,y)' to be written in

contexts where a 'real' is expected and for the 'To_real' projection to be provided implicitly. Another is to provide two sets of arithmetic operators: partial functions such as

P_add: (real; real) returns real

and total functions such as 'add' with the oneof result. The total functions would be used when the programmer was prepared to deal with the exceptions, and the partial ones when he was not.

I favour the second solution because I consider implicit coercions to be dangerous, but the choice is one for the language designer to make after considering how it will affect other aspects of the language. A third possibility is available in CLU because infix operators are considered to be syntactic sugar for invocations. For example, if 'expr1' is of type 'T', the syntactic form 'expr1 + expr2' is defined to be shorthand for 'T$add(expr1, expr2)'. There are no semantic constraints: the shorthand form is legal exactly when the corresponding invocation is legal. The expansion could equally well have been chosen to be 'To_T(T$add(x,y))' and the desired syntactic brevity could then be obtained without altering the semantics of the underlying language.

Another way in which an exception handling mechanism helps the programmer achieve brevity is by permitting a handler to be attached to a whole sequence of statements rather than just one. Levin's mechanism is easier to use in this way because it is parameterized and always returns control to the routine raising the exception. For example, suppose that zero is an acceptable result whenever 'div' detects underflow within a numerical routine. With Levin's mechanism one can write

```
begin
    ... body of routine using div ...
end [underflow on div(v): v := 0] .
```

In CLU this is not possible both because there is no parameter to set to zero and because the division routine is terminated before the handler is invoked.

The solution to this problem in CLU (and using my oneof notation) is to declare a local division function with the required specification. In CLU one would write

```
begin
    Localdiv = proc(a:real, b:real) returns real
        return (div(a,b))
        except when underflow: return 0.0 end
    end Localdiv

    ... body of routine using Localdiv ...
end .
```

Using my notation the routine would be similar except for the definition of 'Localdiv', which would be

```
proc Localdiv(a:real, b:real) returns r:real is
    if Is_underflow(div(a,b))
    then r := 0.0
    else r := To_real(div(a,b))
    fi
end of Localdiv .
```

Both of these code fragments contain a few more lines than those which use Levin's notation. What this modest extra expenditure has bought us is

greater security. We can be sure that *only* those divisions which are performed by 'Localdiv' will return zero when an underflow is detected; there is no possibility of inadvertently handling other exceptions. In my opinion this is a worthwhile exchange if one is attempting to write reliable software. If brevity is the most important consideration one might be better using APL.

4.2.4 The Storage Allocation Problem

Storage allocation was mentioned in Chapter 3 as an example where exception propagation should not be along the call chain. Levin presents a module 'pool' which provides 'allocate' and 'release' functions. I will abstract a little from his example (because it makes unrealistic assumptions about the management of the pool) but preserve its essential features.

I will first assume that only one process is active. The 'pool' module will make use of a lower level module 'FreeStore' which deals with untyped blocks of store. The function 'MaxBlockSize(f)' returns the size of the largest contiguous block of store available in the FreeStore 'f'. The procedure 'NewBlock(f,n)' returns a reference to a new storage area of size 'n ≤ MaxBlockSize(f)', and the procedure 'ReturnBlock(f, p, n)' makes the block referenced by 'p' available for reuse. Some type conversion functions will also be introduced. With these modifications Levin's 'pool' module appears in Figure 4.15.

```
module pool
    condition pool-low policy sequential-conditional
    condition pool-empty policy broadcast
        ...

    proc Allocate(p:pool; t:type) returns d:ref t
        raises pool-low on p
        raises pool-empty on Allocate
    is
        const f:FreeStore = p.FS
        function adequate returns b:bool is
            b := Size(t) ≤ MaxBlockSize(f)
        end of adequate

        if not adequate()
        then   raise pool-low until adequate()
               if not adequate()
               then   raise pool-empty
                      return
               fi
        fi
        d  := Forcetotype(t, NewBlock(Size(t)))
    end of Allocate

    proc Release(p:pool; d:ref any) is
        ReturnBlock(d, Size(Type(d)))
    end of Release
end of pool
```

Figure 4.15: Pool module for a single process, after Levin.

I will ignore details of how the pool is created originally. 'Allocate' accepts a 'pool' and a type as parameters, obtains sufficient storage from the 'FreeStore' associated with that pool to represent an object of the required type, and returns a reference to it. If adequate storage is not available to do this the exception 'pool-low' is generated. It is raised in turn with each of the users of the pool until either adequate resources become available or there are no more handlers. If the handlers do not succeed in releasing enough store the exception 'pool-empty' is generated. The 'Release' operation is straightforward.

Note that the 'pool-empty' exception is raised only with the caller of 'Allocate', and not with all the users of the pool. This is because the pre-condition of the exception, 'Size(t) > MaxBlockSize(f)', is meaningful only to that caller.

A use of the 'pool' module is illustrated in Figure 4.16. The normal handler for 'pool-low' invokes 'squeeze' to try and compact 'm'. However, this handler is masked while 'AddtoStructure' and 'AnotateStructure' are applied to 'm'.

```
module Usepool
    var p:ref pool
    var m:structure
    condition impossible policy broadcast

    proc Update(info:data) raises impossible is
        var d1:ref obj1
        var d2:ref obj2
        proc local-clearup is
            release(p, d1)
            raise impossible
        end local-cleanup

        d1 := Allocate(p, obj 1) [pool-empty: raise impossible → return]
        d2 := Allocate(p,obj2) [pool-empty: local-cleanup → return]
        ... set fields of d1 and d2 using info ...
        begin
            AddtoStructure(m, d1)
            AnotateStructure(m, d2)
        end [pool-low: ]
    end Update

    proc Squeeze is
        || performs compaction of m, calling release(p, d)
        || to return any d removed from m
    end of Squeeze
end of Usepool [pool-low: Squeeze()]
```

Figure 4.16: Levin's use of the pool.

Equivalent program fragments which do not use exceptions are easy to formulate. The pool-empty exception is replaced by a oneof result to 'Allocate', and the 'pool-low' exception by a notification procedure. Whenever 'Usepool' assigns 'p' to a reference to the pool it also adds 'Squeeze' to the pool's list of notification procedures. The compaction could be inhibited while 'm' is being updated either by temporarily removing 'Squeeze' from the list or by setting a variable which 'Squeeze' tests. For the sake of variety I have chosen the second alternative. Code composed according to these considerations is presented in Figures 4.17 and 4.18.

```
module pool
    representation is
        record( FS:FileSystem; poollowrts:List proc() )

    proc Allocate(p:pool; t:type) returns
            d:oneof(Descriptor:ref t; Empty:singleton)
    is const f:FreeStore = p.FS
       function adequate returns b:bool is
           b := size(t) > MaxBlockSize(f)
       end of adequate

       if not adequate()
       then for r in p.poollowrts until adequate do r()
            if not adequate()
            then d:= From_Empty()
                 return
            fi
       fi
       d := From_Descriptor( Forcetotype(t, NewBlock(Size(t))) )
    end of Allocate

    proc AddPoolLowRt(p:pool; r:proc()) is
        Append(p.poollowrts, r)
    end of AddPoolLowRt

    proc RemovePoolLowRt(p:pool; r:proc()) is
        RemovefromList(p.poollowrts, r)
    end of RemovePoolLowRt
end of pool
```

Figure 4.17: The pool module without exceptions.

```
module Usepool
    var p:ref pool
    var m:structure
    proc Update(infor:data) returns
        r:oneof(Possible:singleton; Impossible:singleton)
    is  var d1:ref obj1
        var d2:ref obj2
        const qd1:oneof( Empty:singleton
                       ; Descriptor:ref obj1 ) = Allocate(p, obj1)
        if Is_Empty(qd1)
        then r := From_Impossible();  return
        else d1 := To_Descriptor(qd1)
        fi
        const qd2:oneof( Empty:singleton
                       ; Descriptor:ref obj2 ) = Allocate(p, obj2)
        if Is_Empty(qd2)
        then r := From_Impossible(); release(p,d1); return
        else d2 := To_Descriptor(qd2)
        fi

         ... Set fields of d1 and d2 using info ...

        begin
            m.Updating := true
            AddtoStructure(m,d1)
            AnotateStructure (m,d2)
            m.Updating := false
        end
    end update

    proc squeeze is
        if m. Updating
        then skip
        else || performs compaction of m, calling
             || release(p, d) to return any d removed
        fi
    end of squeeze
end of Usepool
```

Figure 4.18: Using the pool module without exceptions.

So far this example has been similar to the inconsistent strings problem. However, it is often necessary to perform resource allocation in what Levin calls "a fully parallel environment". He presents the problem in such a context. I have not done so initially because I first wanted to develop comparative examples in the sequential case.

In a parallel environment 'pool' is a data structure shared by a number of different processes. One might therefore expect it to be protected by a monitor. Unfortunately this would lead to deadlock. If a call of 'allocate' generates the 'pool-low' exception and a handler attempts to call 'release', the handler will be suspended awaiting completion of the 'allocate' request. However, 'allocate' cannot complete until each of the handlers has done so. Levin cannot therefore use monitors and is forced to do his synchronization "by hand". He introduces two semaphores, 'inner' and 'outer'. 'Allocate' signals 'inner' before the 'pool-low' exception is generated. It waits again on 'inner' after the handlers have completed, and signals both semaphores before returning. The 'release' procedure waits on 'inner' before commencing, and signals it on completion: it does not use 'outer'. The skeleton of 'pool' with these semaphore operations in place is shown in Figure 4.19.

```
module pool
    inner:semaphore
    outer:semaphore
        ...
    proc allocate(...) ...  is
        P(outer); P(inner)
        if not adequate()
        then
            V(inner)
            raise pool-low ...
            P(inner)
            if not adequate()
            then V(inner); V(outer) ...  return
            fi
        fi
          .....
        V(inner); V(outer)
    end of allocate

    proc release(...)   is
        P(inner)
           ...
        V(inner)
    end of release
end of pool
```

Figure 4.19: Pool with semaphores.

The clutter that the 'P' and 'V' operations introduce is considerable. Although it is fairly easy to see whether 'pool' will work correctly as it is written, it is not at all clear how to generalize this method to a module with several access procedures each capable of generating several exceptions. It seems as though a complexity explosion cannot be avoided.

Levin hopes to solve these problems by an appeal for "more sophisticated constructs". I wish to appeal for simplicity, not sophistication. I also believe that parallelism is a more fundamental concept than exceptions. I will therefore consider the parallelism first and afterwards see how to deal with the case of inadequate resources.

Let us first assume that monitors are indeed the chosen method of interprocess communication. *Condition variables* and the associated 'wait' and 'signal' operations were introduced to deal with resource scarcity [46]. Although these operations are not as simple as they first appeared [1], they are considerably easier to reason about than semaphores.

Suppose, then, that 'pool' has a condition 'MoreStoreAvailable', and that 'Allocate' executes a 'MoreStoreAvailable.Wait' command when it cannot satisfy a request. This will release the monitor locks and enable other processes to invoke 'Release'. When some store has been returned to the pool 'MoreStoreAvailable.Signal' is executed by 'Release', and 'Allocate' can again attempt to satisfy the request.

Of course this arrangement does not inform the users of the storage allocator that there is a shortage of resources. It is a trivial matter to provide a monitor entry 'StorageScarce' which delivers true if a process is waiting on 'MoreStoreAvailable', or even if the available storage drops below a certain threshold. User processes could then test 'StorageScarce' at convenient intervals and compact their local data structures if necessary.

Nevertheless, such voluntary co-operation seems to be different from the exception handling in Levin's example where the handlers are invoked by the *allocator*, even though they execute in the environment of the users. In some sense Levin achieves tighter coupling.

However, the price of this tight coupling is that monitors cannot be used to control the synchronization. The monitor concept requires that the *only* calls which affect the data of another process are monitor calls. But Levin wishes *handler* calls to do just that: a handler invoked under the control of one process manipulates the data structures of another process. Levin is prepared to sacrifice monitors in order to use his exception mechanism for interprocess communication. In order to simulate the communication he thus obtains, I must also abandon monitors. However, rather than resorting to semaphores I will adopt disciplined communication primitives.

For the purposes of this example I will use Communicating Sequential Processes (CSP) [47] [58]. Although not ideal for this example because the number of users to the allocator must be bounded, the emphasis placed by CSP on communication as a primitive is appropriate. CSP assumes that each process has its own local store, but there seems to be no objection to a storage allocator controlling access to a central shared store.

The allocator in Figure 4.20 differs slightly from those discussed above. It is more persistent: if an allocate request cannot be satisfied immediately it continues to signal 'pool-low()' to all the user processes until enough store is available. During this time no further 'Allocate' requests will be accepted, but 'Release' requests will be handled correctly.

Figure 4.20: The storage allocator in CSP notation.

```
Pool is [ FreeStore :: AandR || Chooser :: CHOOSE ]

AandR is
    *[  (i:1..n) p:pool; t:type; User(i)?Allocate(p,t) → A
    []  (i:1..n) p:pool; d:ref any
    ;   User(i)?Release(p,d) → R
    ]

A is
    done:bool; done := false;
    *[  not done →
        [   adequate() →
            User(i)!Forcetotype(t,NewBlock(Size(t)));
            done := true
        []  not adequate → MORE
        ]
     ]

R is    ReturnBlock(d, Size(Type(d)))

MORE is
    [   *[(j:1..n) d:ref any; User(j)?Release(p,d) → R]
    ||  i:integer; Chooser?i; User(i)!pool-low()
    ]

CHOOSE is
    *[  s:set of integer; s := {1..n}
    ;   *[  s ≠ {} →  i:integer; i := oneof(s);
                      s := s-i; FreeStore!i
        ]
    ]
```

Figure 4.21: A pool user in CSP notation.

```
Usepool is
    *[  info:data; client?Update(info)  → Updater
    []  pool?pool-low() → squeeze
    []  ...
    ]

Updater is
    [   [   [   d1:ref obj1; d2:ref obj2;
            ;   pool!Allocate(p, obj1); pool?d1
            ;   pool!Allocate(p, obj2); pool?d2
            ;   ... set fields of d1 and d2 using info ...
            ]
        ||  [   pool?pool-low() → squeeze   ]
        ]
    ;   AddtoStructure(m, d1)
    ;   AnnotateStructure(m, d2)
    ]
```

A user is shown in Figure 4.21. Normally it is prepared to accept either 'Update' requests from its client or 'pool-low' requests from 'pool' (and requests for its other services, represented by the elipsis). Whilst obtaining and filling-in 'd1' and 'd2', 'pool-low' requests are still accepted and acted upon, but whilst 'm' is being updated they are not. Requests arriving at that time will be dealt with after the updating has been completed; in Levin's example they would be ignored. (It would be possible to accept and ignore 'pool-low' signals while 'm' is updated, but unless this takes a very long time there seems to be no reason to do so.)

The purpose of this excursion into CSP is to illustrate that Levin's example is a problem in parallelism rather than exception handling. I could have avoided this issue by adding semaphores to the solution with notification procedures, thus producing a program with the same effect as Levin's which would bear line-by-line comparison. I have refrained from doing so because that introduces more problems than it solves. The notion of co-operating processes has permitted great advances in the way we think about systems. It is applicable to a wide range of problems and seems to be more fundamental than that of exceptions. The exception handling in the example becomes just one more communication; synchronization cannot be subsumed by exception handling with the same ease.

4.2.5 The Real-Time Update Problem

The final example in Levin's thesis is undoubtedly the strangest. Whereas all of the preceding exceptions can be characterized by Parnas's phrase *undesired event* [76], the events considered in this example are *essential* for normal progress. In other words, they are not in the least exceptional.

The events concerned are *interrupts* from another process indicating

that input of data from an external sensor has been completed. Levin shows that his mechanism can be used to implement such communication. He deduces from this that "it is sufficiently robust to handle situations in the gray area between exceptions and communication".

The specific problem which Levin poses is that of a collection of processes which maintain a display of data computed from external sensors and changing in real time. A concrete example is an aircraft control system which displays altitude, latitutde and longitude, cabin pressure and temperature, and so on. The need for exceptions is introduced by assuming that

(i) there are fewer processes than items to be displayed, so each process must compute more than one display item;

(ii) some of the calculations are more important than others. If new sensor data arrives while the less important parameters are being calculated, then those calculations should be abandoned and the more important ones recommenced.

Levin's solution is shown in Figure 4.22. Sensor data is collected in 'buf' which broadcasts the "exception" 'new-data' to all its users. All the 'grind' routines share 'buff'; they also share an output structure into which the results are placed, called 'display'.

Two kinds of computation are shown in 'grind1'. Those which can be abandoned at any point associate the leave '1' handler with 'new-data': lines 12 and 16 are examples. Computations which must be completed use a handler which sets a variable as on line 14 and a subsequent test as on line 15. Typically this will be necessary while related results are transmitted to 'display', so that, for example, the latitude and longitude relate to the same instant in time.

```
module crunchers
begin
    shared buf:sensor;buffer
    shared display:screen

    function grind1 =
        while true
        do
          1:begin
                private restart:boolean
                restart := false
                < perform computation 1 >
                < perform computation 2 >
                                [new-data: restart := true]
                if restart then leave 1 fi
                < perform computation 3 >
                ...
            end [new-data:   leave 1]
        od

    function grind2 =
        < similar to the above >
        ...
end
```

Figure 4.22: Levin's real-time computation module.

Levin admits that such a problem could be (and probably has been) solved without an exception handling mechanism. The incentive for me to produce such a solution is thereby removed. Instead I will make some observations on Levin's solution.

(i) Some synchronization is necessary between the 'grind' routines and 'buf' to prevent the data changing while related values are being read. Levin glosses over this problem, presumably because it is not amenable to solution by exceptions. If 'StartRead' and 'EndRead' primitives are introduced and used by 'grind' to bracket all related accesses to 'buf' then the situation reduces to the well known readers and writers problem [15][56].

(ii) There is only one 'new-data' exception but several different sensors will be providing 'buf' with data. If 'new-data' is generated whenever *any* of the data change then most of the calculations will be aborted for no good reason. One solution to this problem would be to introduce a different exception for each datum, and for the grind routines to set up handlers for just those exceptions whose corresponding datum was of a high enough priority. The resulting code would become quite difficult to follow.

(iii) A system structured as Levin suggests is very fragile: it depends crucially on the relative speeds of the processes. If more efficient sensors are fitted to the aircraft so that data is supplied to 'buf' more frequently, *less* information would appear on the display. In the limit the 'new-data' exception could be generated so frequently that the pilot is supplied with no information at all! This scenario is particularly dangerous because it is counter-intuitive: one would expect that a more efficient sensor would provide more information, not less.

All of these problems can be avoided if interruption is abandoned and the 'grind' processes *ask* 'buf' if there is any new data. There is no need for 'buf' to remember when a particular 'grind' process last queried a particular datum: the determination of whether the datum is new is most easily carried out by the interested 'grind' process asking 'buf' for the current value and comparing it with that it last obtained. A general communication mechanism such as CSP is quite adequate both for the solution of the readers and writers problem associated with 'buf' and the determination of when to re-start a computation with new data.

I presume that Levin presented this example because he considered it desirable that his mechanism be used for interprocess communication. I consider it undersirable that such a use is even possible. Some of the dangers and difficulties of using his mechanism in this way were indicated above. However, there is another, more general, objection.

It is obvious, and freely admitted by Levin, that his mechanism is not adequate for general interprocess communication. This implies that some other mechanism is required. Programmers are immediately faced with the problem of reasoning about programs which use two disparate mechanisms to achieve the same end. Dealing with the semantics of parallelism is difficult enough without having to consider the interactions between two mechanisms which impose different constraints on the communications they allow.

A reason for objecting to Levin's mechanism in particular is that interrupts seem to be a very intractable communication device. Despite ten or fifteen years' experience with interrupts, it was only with the discovery of more constrained facilities for communication and synchronization that techniques for reasoning about parallelism have been developed. The invention of mathematical techniques which permit one to prove theorems about parallel programs must be considered as a major advance [6] [48] [49]. Although it is possible that techniques could be found to help us reason about interrupts, Levin does not offer any guidance. The synchronization aspects of his mechanism are defined only informally.

In 1975 Jack Dennis said that "interrupts are an example of a very bad programming construct which one simply should not be using at all" [98, p. 33]. I wish to take the weaker position that interrupts should only be

used when they provide a clear, simple and correct solution to an otherwise intractable problem. Levin's example does not illustrate such a problem.

4.3 Experience with Mesa Signals

In [51] Horning describes the Mesa signal mechanism (see Section 3.3) and states that in his experience there are situations where the use of Mesa signals greatly simplifies what would otherwise be a thorny programming problem. I wrote to Dr Horning asking if he could show me one such example. My query was circulated to many people at Xerox Palo Alto Research Center and resulted in some very interesting correspondence.

Horning's own response was to admit that it is very difficult to give a convincing specific example that demonstrates the utility of signals. He attributed this to the fact that signals are a mechanism for the control of complexity. In simple examples there is not enough complexity, and more complex systems are too big to be good examples. This seems to be a weak argument. Procedures and abstract data types both exist to control complexity: the literature is nevertheless full of simple examples illustrating their usefulness. My own feeling is that a facility will be of little use in a real system if it is not even of use in a specially constructed example.

The other responses covered the whole spectrum of opinion, from those who found signals very useful at one extreme to those who deprecated them at the other. One of the less enthusiastic responses was from Crowther:

> I do not use signals. That is a long standing policy of mine. I cite two exceptions: I use signals freely to get to the Debugger, and usually provide provisions for resuming them. I used a signal in the BTree package ... I now consider this to be an error in design; the resulting code was quite obscure, and there were other straightforward ways to work around the problem...

If one accepts the need for run-time debugging then using signals to access the debugger seems to be unobjectionable. As was pointed out in Section 3.3, Mesa signals are implemented in such a way that all the relevant information is still on the stack when an otherwise uncaught signal is intercepted by the debugger. Of course, the same would be true if the debugger were simply *called*; I do not know if this is prohibited in Mesa.

Some of the more positive comments restate the arguments in favour of exceptions that have already been discussed elsewhere. I will consider here three of the more novel applications.

4.3.1 Dealing with Oversights in Existing Software

Consider the file enumerator

```
ListFiles:PROCEDURE[ d:directory
                   ; TakeFile:PROCEDURE[ FileHandle ]
                   ]
```

'ListFiles' applies 'TakeFile' to every 'FileHandle' in 'd'. Such procedures exist in standard packages at Xerox, and can be used but not changed.

The problem is how to convey information from 'TakeFile' to the invoker of 'ListFiles', such as details of archived files or file titles. 'TakeFile' returns no results. For our purposes we would prefer the declaration of 'ListFiles' to use the Mesa type ANY, a union of all types, and have a heading

```
ListFiles:PROCEDURE[ d:directory
                   ; TakeFile:PROCEDURE[ FileHandle ] RETURNS ANY
                   ] RETURNS SEQUENCE ... OF ANY
```

where the result of 'ListFiles' is formed by concatenating all of the results of the individual calls of 'TakeFile'. 'ListFiles' cannot be redefined because it is in a standard package. However, because Mesa signals

are not part of a procedure's type, the same effect can be obtained by a 'TakeFile' procedure which signals its results to the caller of 'ListFiles'. Signals can pass right through the intermediate level.

Of course, there are other ways of solving this problem, the most obvious being to adopt the amended design suggested above. However, the number of possible 'ListFile' procedures is large. Suppose it is required to terminate the enumeration before the directory is exhausted. A procedure

```
ListFiles:PROCEDURE[ d:directory
                   ; TakeFile:PROCEDURE[ FileHandle ]
                                        RETURNS[ halt:BOOLEAN ]
                   ]
```

might then be appropriate. Perhaps it is impossible to ensure that there are enough procedures in the library to satisfy every user.

The problem arises because procedure parameters blur the distinction between levels of abstraction. The caller of 'ListFiles' is in some sense the *direct* caller of 'TakeFile': it is the only procedure to know the identity of the actual procedure which is provided for 'TakeFile'. One way of exposing this relationship is to reformulate the 'ListFiles' procedure either as a *process* which *outputs* 'FileHandles', or as a *stream*. The latter alternative was adopted for file enumerators at the Programming Research Group nearly ten years ago and have proved very flexible [91].

Briefly, one applies the procedure 'EntriesfromFile' to a directory: the result is a *stream*, say 'S'. Streams are objects to which the procedure 'Next' and the predicate 'Endof' are applicable. Successive calls of 'Next' return successive file handles: the user may do whatever he wishes with them. A call of 'Reset[S]' will restart the enumeration from the beginning; when termination is required the stream can be disposed of by calling 'Close[S]'.

Whether a language should directly support the use of streams is a question open to debate. CLU [61] [64] includes them in a restricted form as iterators. Whatever one's opinion on this matter, it is unconvincing to advocate Mesa signals because they provide a means of patching up design errors. Furthermore, this usage is only possible because Mesa does not require signals to be specified in routine headings. Most other advocates of exception handling insist on full specification.

4.3.2 Using Handlers to Clean Up Test Code

J. Morrison reports that in test code it is common to have *long* lists of test operations. Using handlers instead of procedures can help to clean up the code.

Consider a procedure 'Try[TrialProc, ...]' which calls 'TrialProc' from an environment within which its behaviour can be debugged. (This is necessary because of the large number of exceptions which may be generated: there is no way in Mesa for the compiler to check that only the expected ones will be propagated.) A test program might therefore contain code like:

```
test1:PROC = {stuff1};
test2:PROC = {stuff2};
test3:PROC = {stuff3};
...
Try[test1, ...];
Try[test2, ...];
Try[test3, ...];
```

Morrison finds this code very messy. His solution is to construct and use another version of 'Try' that doesn't take a procedure argument:

```
Try2[... ! Test => Stuff1];
Try2[... ! Test => Stuff2];
Try2[... ! Test => Stuff3] .
```

Instead, 'Try2' generates the exception 'Test'. This invokes the trial operation 'Stuffi'.

Again, the problem here has nothing to do with exception handling; Morrison is really complaining about an asymmetry in Mesa. The original code was "messy" because Mesa insists that procedures must be named. The problem would not arise if Mesa allowed lambda expressions to appear in contexts where procedures are required. In contrast, handler bodies are arbitrary statements and are written where required rather than being named and then used later. This particular inconsistency in Mesa is deprecated in [26].

4.3.3 Implementing Dynamic Binding

The binding of signal values to handler bodies in Mesa resembles the binding of atoms to values in Lisp. Geschke and Satterthwaite point out [26] that signals can be used to implement dynamic binding for variables.

Suppose one wishes to dynamically bind an identifier 'Id'. This may be achieved by defining a signal

```
FetchId: SIGNAL RETURNS[value:T] .
```

The body of each procedure which declares a local variable 'Id' must enable the handler

```
FetchId => RESUME[Id];
```

the signal call 'FetchId[ ]' will then return the dynamically bound value of 'Id'.

This scheme would be very inefficient because of the high overheads of the exception handling mechanism. Efficiency can be improved by caching 'Id' or by a method closer to a "shallow binding" implementation of Lisp.

Interested readers should see [26] which contains an elaborate example of an application.

Most modern applicative languages (e.g. Lispkit Lisp [43]) use lexical binding precisely because dynamic binding often produces unpleasant surprises. Nevertheless, dynamic binding appears to be useful on occasions, and a language designer might consider that this usefulness counteracts the dangers sufficiently often to warrant the inclusion of dynamic binding as a language option (e.g. Lisp/370 [53]). However, going on from there to include exception handling because it provides a means of simulating dynamic binding does not seem to be sound reasoning.

4.3.4 Assessing the Use of Mesa Signals

Since the publication of Dijkstra's famous letter warning of the dangers of the goto statement [20], practising programmers have gradually realised that unconstrained power does not necessarily make a good programming language. Features such as untyped variables, non-local gotos and dynamic binding, although sometimes convenient, were found to be "too much of an invitation to make a mess of one's program". They have gradually disappeared from programming languages.

Over the same period our ability to reason about programs both formally and informally has improved. Both of these changes have sprung from the growing realization that programs are objects which *should* be reasoned about. Their purpose is not just to command a machine to perform some operation in a minimal number of cycles: it is also to communicate with other human beings.

It seems to be fatuous to design a language without non-local gotos, with strong typing and with lexical binding, and then to provide a facility which circumvents these restrictions. Mesa signals are such a facility.

The above examples seem to me to illustrate why unconstrained exception handling mechanisms are dangerous.

4.4 Implementation Efficiency

An issue often raised in connexion with exception handling is the efficiency of the code produced by a reasonable compiler. One question of interest is whether an exception handling mechanism enables exceptional results to be dealt with more efficiently than would be possible without such a mechanism. The usual criteria for judging an implementation of an exception handling mechanism is that its effect on the "normal" case should be minimal and that exceptions should be handled with reasonable (but not maximal) efficiency.

Arithmetic exceptions are sufficiently simple for various different implementations to be compared in some detail. Let us first consider the code which might be generated for the CLU phrase

```
z := real$add(x,y) except when overflow: z := Real_Max end
```

The examples will be worked on a hypothetical stack machine; we will first assume that it sets condition codes when overflow or underflow occur. A compiler for CLU would recognise 'real$add' as a built-in operation, and might generate the code that follows.

```
        push x              ;push the value of x onto the stack
        push y              ;push the value of y onto the stack
        add                 ;pop the top two stack values, and
                            ;push their sum
        jio OHandler        ;jump if overflow occurred
        jiu Failure         ;jump if underflow occurred
        pop z               ;store top of stack in z and pop it
next:   ...
```

```
OHandler:
        topl Real_Max       ;set the top of the stack to the
                            ;literal value Real_Max
        pop z               ;store top of stack in z and pop it
        goto next           ;unconditionally
```

Six instructions are required to perform a non-exceptional addition. In the case where overflow occurs, seven instructions are executed. Note that the compiler can always determine whether an appropriate handler exists, and that if one does its address is known. The above code assumes that the overflow handler is positioned away from the code which calls it: before the beginning or after the end of the routine would be suitable places. If it is positioned with the code, an extra instruction would be necessary to branch around it. It is fairly easy for a compiler to arrange such remote positioning of the handler, but it does require buffering space and may necessitate the use of long format jumps under some conditions.

Now consider the equivalent phrase using type unions:

```
o := real.add(x,y)
if Is_Overflow(o) then z := Real_Max
                     else z := To_Real(o) endif
```

Remember that the add operation now has the heading

```
proc add(a:real; b:real) returns c:
     oneof(real; underflow:singleton; overflow:singleton)
```

Following the treatment of CLU, we will assume that 'real.add' is known to the compiler and is expanded in line. The following code might be generated.

```
        push x              ;push the value of x onto the stack
        push y              ;push the value of y onto the stack
        add                 ;pop the top two stack values, and
                            ;push their sum
        jno else            ;jump if overflow did not occur
then:   topl Real_Max       ;set the top of the stack to the
                            ;literal value Real_Max
        pop z               ;store top of stack in z and pop it
        goto next           ;unconditionally
else:   jiu Failure         ;jump if underflow occurred
        pop z               ;store top of stack in z and pop it
next:   ...
```

This code is based on the assumption that 'o' is a temporary variable not used outside the code fragment shown. If some other part of the program needs to access 'o' (rather than 'z') then code would have to be added to set and store an appropriate union value. (Similar code would be necessary in the CLU case.) The above example also assumes that the underflow condition code is persistent over an executed jump. This seems to be a reasonable assumption: it can be avoided by re-ordering the code so that the else branch comes first:

```
        push x              ;push the value of x onto the stack
        push y              ;push the value of y onto the stack
        add                 ;pop the top two stack values, and
                            ;push their sum
        jio then            ;jump if overflow occurred
        jiu Failure         ;jump if underflow occurred
else:   pop z               ;store top of stack in z and pop it
        goto next           ;unconditionally
then:   topl Real_Max       ;set the top of the stack to the
                            ;literal value Real_Max
        pop z               ;store top of stack in z and pop it
next:   ...
```

Both of these code fragments execute seven instructions to perform non-exceptional addition. When overflow occurs, six instructions are executed. The difference from the CLU code is that the "handler" for overflow is now in the middle of the code which uses it, and an extra instruction is needed to branch around it. Conversely, no return jump is needed after the handler completes, so the exceptional case is one instruction shorter. Of course, the above code could easily be transformed into the CLU code by moving the "then" part elsewhere. However, it is not as reasonable to expect the compiler to do this with half of an "if" statement as with an except statement, which may be interpreted as an assertion that one of the execution paths will be much more common than the other.

It is difficult to compare the implementation of Levin's mechanism with those treated above. The obvious implementation of

```
z := add(x,y) [overflow(v): v := maxfp]
```

involves calling the handler as if it were a procedure, having previously set up 'v' as a reference or value/result parameter. The code for the normal case will be similar to that generated for add with a oneof result. Because it is not possible in general to locate the correct handler for a

particular exception until run-time, the code for the exceptional case will be rather more complicated than the examples presented above. The code between 'then' and 'next' would be replaced by code which searches for the correct handler, creates its parameter list, calls it, and finally copies the value of the output parameter into 'z'. Clearly, the generality of Levin's mechanism has its price in the exceptional case; unfortunately, it also has a cost in the normal case. The method Levin proposes for locating handlers involves linking handler descriptors to a known part of the stack. Although the descriptor itself can be allocated at compile time, the linking must be done dynamically. The cost of link and unlink code must therefore be added to the normal as well as to the exceptional case. Instead of six or seven executable instructions, there are ten or eleven. In the particular example we are considering, the choice of handler can be made statically, but it is an open question whether a compiler which implemented Levin's mechanism in its full generality could reasonably be expected to optimise such cases.

Thus, when handlers need to be attached to individual arithmetic operations, there is little to choose between the efficiency of CLU exceptions and type unions, while Levin's mechanism may be considerably more expensive. When a handler should be attached to a whole sequence of operations rather than one, as in the examples on page 122 Levin's mechanism fares better. Using both CLU exceptions and type unions, convenience dictates the declaration and use of the auxiliary function 'Localdiv'. If this function is not expanded in line, many extra procedure calls will take place; if it is expanded, many copies of the handler will exist. Levin's mechanism requires only hne handler, and involves calls only if an exception actually occurs.

It is also interesting to compare the execution efficiencies of exception raising routines written by the user. CLU signals are implemented by embedding a table in the code of each procedure. Each handler in the procedure has an entry in the table; each entry lists the exceptions with which the handler will deal, its code address within the procedure, its scope (in terms of the addresses of the statements to which it is attached) and other information connected with the arguments and results of the exception. When an exception is generated by a signal statement, the routine which called the generator is first determined*. The handler table for that routine is then examined, and the handler which applies to the call which raised the exception is found. Naturally, the time taken to find the right handler depends on the number of handlers in the table, and thus in the calling routine. Finally, the stack is adjusted to allow for the results of the signal, and a "return" is made from the generator, not to the point from which it was called, but to the appropriate handler.

Now consider a 'lookup' operation on a symbol table like that discussed in Section 4.2.1. So that both cases can be treated symmetrically, Levin made both "absent" and "present" exceptions. A CLU programmer, I am told, would not do this: instead he would give 'Lookup' the type

proctype(symbolTable, T) returns(V) signals absent

---

* Since CLU procedures may not be nested, this determination can be made from a list of the start address of each procedure. Such a list does, of course, occupy space in main store, but it can be argued that it is so useful for debugging purposes that it ought to be in main store anyway.

so that the search for a handler (for 'absent') would take place only when the item is not in the table. Whether this occurs frequently or infrequently is, of course, dependent on the use to which 'Lookup' is put.

If a similar routine is written using a oneof result, the search for a handler is traded for the extra instructions required to return a tag with the result and to test it. It is difficult to avoid this tag through optimisation, because part of the optimisation must take place in the calling routine and part in the called routine. The CLU signal mechanism can be regarded as a way of making the optimisations easier by identifying where they may occur. Whether the code that results from the CLU implementation is actually faster depends both on the number of handlers in the caller and the relative frequency with which items are not found. If the CLU mechanism is used in the way that I have suggested, the verdict must be "not proven". In particular, it is important to observe that the programmer who writes a general purpose library routine such as 'Lookup' knows nothing about the environment in which it will be used. He can only guess whether 'present' or 'absent' will occur more frequently. Even if he knows that the lookup procedure will be used in an Algol compiler, he does not know whether the Algol programs the compiler is called upon to process will have been written by an ex-Fortran programmer who habitually forgets to declare his variables or by a block-structure enthusiast who habitually uses the same names in each block. In some cases, of course, the programmer does have a good idea of the relative frequencies of the possible outcomes of his routine. An example is a disk read routine, where failing to read the data on the disk is a very rare occurrence. In such a case the use of a CLU signal can provide a small (but significant) gain in efficiency.

If the CLU signal mechanism is used to emulate Levin's example, i.e. if a 'Lookup' routine is written which has type

proctype(symbolTable, T) signals absent, present(V) ,

then it seems that using CLU signals would be less efficient than using type unions. However, as I mentioned above, this can be regarded as an unfair comparison because CLU programmers are aware of the efficiency tradeoffs involved in the current implementation of the CLU exception mechanism. They would never use it in the symmetric way suggested by Levin, however symmetric the application.

In fact, it is fair to point out that *any* implementation that makes exceptions more expensive than normal returns will discourage programmers from using exceptions in the symmetric way suggested by Levin. Even Levin's own implementation suggestions had this property. With such an implementation, a programmer is pressured into making one of the possible results "normal" and the other "exceptional", regardless of whether that classification is natural. For example, if exceptions were to be used to implement sparse arrays, pragmatics would suggest that the case of an element being non-zero be made the "exception", since zero elements would occur more frequently. In cases such as this, the use of an exception mechanism is likely to impair readability.

4.5 Conclusion

This chapter could not include all of the examples I have found which purport to justify exception handling. However, I believe that I have illustrated a fair selection; I certainly have not knowingly excluded any example because I could not deal with it.

It must remain a matter of opinion as to whether the inclusion of an exception handling mechanism in a language is worthwhile. I hope that this chapter has provided evidence which may be used as a basis for forming such an opinion.

Chapter 5

EXCEPTION HANDLING AND ABSTRACT DATA TYPES

A data type is a set of values and a set of operations on those values. An abstract data type is a type whose implementation is hidden; the user has no knowledge of the representation of the values, or of the algorithms corresponding to the operations. For such a type to be useful, the user must be provided with a semantics for the type: he must be told what operations are available and how they behave. A specification is therefore an essential part of an abstract data type.

The problem of specifying a data type precisely is difficult because the values and operations are highly dependent. New values are generated by the action of the operations, but the operations themselves must be applied to values. Which should be defined first? One solution to this chicken and egg problem is to define them both simultaneously by means of axioms.*

The technique of axiomatic specification of types was proposed by Guttag [34] [35]. It is closely related to the algebraic approach of the ADJ group [28]; the differences are beyond the scope of this thesis.

5.1 Errors and Exceptions in Axiomatic Type Definitions

Several papers have dealt with the supposed difficulty of including errors and exceptions in axiomatic type definitions (e.g. [27], [38], [28], [69]). Some of the proposed solutions appear to be very complicated. This chapter

---

* There are various other methods of specifying data types which will not be considered here. A survey will be found in [65].

argues that there are no errors in abstract data types, and that "exceptions" do not need special facilities for their description.

This view is in direct opposition to that of Goguen, who claims [27]

> that the specifier of an abstraction must also specify exactly what error messages are to be produced and under exactly what circumstances; the error messages should ... not be left to the whim of the implementer ...
>
> The situation is actually worse than this, in that error states are an *essential* and intrinsic feature of certain data types (such as 'stack', with 'UNDERFLOW'), so that these types actually *cannot* be handled correctly at all without some systematic provision for error states. Previous attempts to specify types like 'stack' abstractly have either been wrong in this sense, or else unnecessarily complex. Abstract data types *must* include abstract errors.

Errors and exceptions have been introduced into abstract data types as a way of completing the definition of partial operations. The next section explains why this is difficult, and proposes an alternative solution.

5.2 *Defining Partial Operations*

In mathematics it is a common practice to avoid partial functions by making them total. This seems to be easy so long as the domain of the function is computable; the range is extended so that all previously undefined applications of the function map to a distinguished value. (If the domain is uncomputable then the function cannot be made total: to do so would lead to an immediate contradiction.)

The same technique was adopted by the pioneers of axiomatic type definition. As an illustration, consider the stack of integers, an example which has perhaps been overworked, but which demonstrates the problem concisely. A stack provides five operators whose names and signatures are

```
Empty: () → StackofInt
Push: (StackofInt, Int) → StackofInt
Pop: (StackofInt) → StackofInt
Top: (StackofInt) → Int
IsEmpty: (StackofInt) → Bool .
```

I have used a notation closer to programming languages than mathematics: 'Push: (StackofInt, Int) →' indicates that 'Push' takes as argument a Cartesian produce of 'StackofInt' and 'Int'. The *meanings* of these operations can be defined by axioms:

```
Pop(Empty) = Empty
Pop( Push(s,i) ) = s
Top( Push(s,i) ) = i
IsEmpty(Empty) = True
IsEmpty( Push(s,i) ) = False .
```

However, this definition is incomplete: it does not say anything about applying 'Top' to 'Empty'. Intuitively, this is an impossible operation, just as is taking the head of a null list. 'Top' is a partial function, and is not defined on empty stacks.

In his thesis [34] Guttag attempted to capture this idea by writing

```
Top(Empty) = error
```

where "'error' is a distinguished value with the property that the value of any operation applied to an argument list containing 'error' is 'error'". In other words, Guttag used the standard mathematical technique for making a partial function total. Unfortunately, this technique gives rise to various complications, which will be explored in the next section.

An alternative would have been simply to admit that 'Top' is partial, and to include in the specification the clause

```
pre Top(s) ≡ not IsEmpty(s)
```

One can argue that the same effect could be achieved simply by omitting the axiom for 'Top(Empty)'. However, this would not only make errors of omission difficult to detect, but would specify the domain of 'Top' only implicitly.

An alternative way of indicating the restricted domain of 'Top' would be to change its signature to

$$\text{Top}(\{s \in \text{StackofInt} \mid \underline{\text{not}}\ \text{IsEmpty}(s)\}) \to \text{Int}\ .$$

One could even argue that by so doing one renders 'Top' total. Two reasons for not using this notation are that it confuses datatypes with sets of values, and that it is not always sufficiently powerful. For example, subtraction over the natural numbers can be characterized by

$$-\colon (\text{Nat}, \text{Nat}) \nrightarrow \text{Nat} \qquad \underline{\text{pre}}\ a-b \equiv a \geq b,$$

whereas there is no convenient signature which defines the restricted domain. For these reasons I prefer to specify partial functions by means of preconditions.

Partial operations occur naturally both in computer science and in the world outside the computer; we are used to household articles which are partially applicable. A gas heater is provided with a specification which states that its operation is safe provided that ventilation is adequate. An electric space heater should not be submerged in the bath. Both articles are useful despite their restricted domains of application.

For an example from computer science, consider Euclid's algorithms for greatest common divisor [23].

```
x := X, y := Y;
do x > y => x := x-y
[] x < y => y := y-x
od
```

This code fragment achieves 'x = y = GCD(X,Y)' provided both 'X' and 'Y' are positive. Subtraction defined only over the natural numbers is fully adequate for this program. 'x' and 'y' are always positive, and the guards in the repetitive statement ensure that the precondition of '-' is always true. Nothing whatsoever is to be gained by using a subtraction operator which is total: if '-' is invoked outside its domain something is seriously wrong with the implementation of the algorithm, and defining subtraction for all integers will not set it right.

For these reasons I believe that partial operations are now and will always be useful; the only caveat is that their preconditions must be rigorously specified. One practical difficulty is that any implementation of a partial operation must actually do something if the precondition is not satisfied. If programmers come to rely on that action then the portability of programs is severely compromised. This problem can be mitigated by

(i) detecting as many illegal calls as possible before the program is run, and

(ii) making it difficult for the programmer to take advantage of the response to an illegal call.

The second principle is the one which prompted me to state in Section 2.1 that termination of the program is an appropriate response to Hill's problem.

Of course, these preconditions in no way reduce the responsibility of the data type designer to require only those preconditions which can be checked economically and conveniently. A compiler which will only operate

on syntactically correct programs is about as useful as a room heater which requires the ambient temperature to be above 30 degrees C: neither is salable. As with the symbol table example, such considerations may lead the designer to provide a total operation which returns different sorts of result under different circumstances. However, before presenting a formal definition of my mechanism for returning such results, it is instructive to see why Guttag's original approach to the definition of partial functions is unsatisfactory.

5.3 The Problem with Error

Guttag's original definition of stack [34, p. 29] not only defined 'Top(Empty)' as 'error' but described the functionality of 'Top' by 'Top: (StackofInt) → Int'. This implies that 'error' is an 'Int', and is the cause of several difficulties.

First, it is counter-intuitive. The result of applying 'Top' to the empty stack is *not* an integer: that is why 'error' was introduced. Second, if one wishes to distinguish between different errors, every time a new data type such as 'QueueofInt' or 'StringofInt' is created new exceptional elements must be added to 'Int'. In general, adding any new type would require altering the definitions of all the types it uses; apart from being cumbersome and intellectually unappealing, this would mean that a library of type definitions could not exist. Third, in attempting to extend the existing types, consistency problems arise in the axiomatization. A full description of the difficulties is available in [28, §3.5]; what follows is a brief summary.

If 'error' is an 'Int' then axioms must be given which explain the action of "Int" operations on it. The usual philosophy is to ensure that

once an error occurs it is propagated, that is, if any argument of an operation is 'error' then so is its result.* If one attempts to implement this by adding new axioms one rapidly runs into trouble. One would need to add rules like

```
Sum(n, error) = error                                   (i)
Product(error, n) = error                               (ii)
```

to the axiomatization of the integers. Of course, it still contains other rules describing the more conventional properties, such as

```
Sum(n, 0) = n                                           (iii)
Product(n, 0) = 0                                       (iv)
```

Using the above four rules some interesting results can be obtained:

```
0 = Product(error, 0)                                   (by iv)
  = error                                               (by ii)
```

and

```
n = Sum(n, 0)                                           (by iii)
  = Sum(n, error)                                       (by above)
  = error .                                             (by i)
```

These derivations show that all integers are equal to 'error'. A similar problem arises within the stack data type; one must add a new constant 'StackError' of type stack and a new axiom

```
Push(s, error) = StackError
```

---

* Another way of saying the same thing is to assert that all operations are strict in 'error'.

to ensure that errors propagate. In combination with the more usual axioms it is easy to show that all stacks are equal to 'StackError'. This instance of the problem is dealt with very fully in [69].

Of course, such problems can be resolved; various techniques for doing so are described in [27], [28] and [69]. Some mechanism is introduced to ensure that the "normal" axioms are only employed when no "error" axiom is applicable. I do not intend to criticize the mathematical intricacies of these approaches; instead I wish to avoid the whole issue, and for two reasons. First, it is very complicated. Second, partitioning the world into "error" and "OK" states is inappropriate when one is trying to formalize 'LookUp' and similar operations. As I have emphasised throughout this thesis, whether the result of a particular application of "LookUp" is an error can only be determined by the caller: it is not a property of the 'SymbolTable' data type.

"Error algebras" are necessary only if one insists that 'Top(Empty)' is an integer. If one permits 'Top(Empty)' to be of some other type one can avoid error algebras, consistency problems and the need to augment existing types when a new one is created.

It seems likely that this was what Guttag originally intended. In [34, p. 47] he states that 'error' will not normally be included in any of the types involved in a definition, contradicting the inference that it was an integer. In the later papers [36] and [37] the signature of 'Top' is given as

Top: (StackofInt) → Int ∪ {UNDEFINED}

where the range is stated to include the "singleton set '{UNDEFINED}'".

Unfortunately no more details of the meaning are given. The difficulty is that set operations such as union cannot be applied to types; the next section defines a *data type* union operation which can be used to formalize this definition.

5.4 A Rigorous Definition of Data Type Union

The union operation I will define is exactly the oneof type generator introduced in Chapter 4 and used there in the examples. First consider the somewhat more restricted type given in Figure 5.1.

```
type U2
operators
    Is_First: (U2) → Bool
    Is_Second: (U2) → Bool
    From_First: (First) → U2
    From_Second: (Second) → U2
    To_First: (U2) ↛ First
        pre To_First(u) ≡ Is_First(u)
    To_Second: (U2) ↛ Second
        pre To_Second(u) ≡ Is_Second(u)

axioms
    Is_First( From_First(f) ) = true
    Is_First( From_Second(s) ) = false
    Is_Second( From_First(f) ) = false
    Is_Second( From_Second(s) ) = true
    To_First( From_First(f) ) = f
    To_Second( From_Second(s) ) = s
end of U2
```

Figure 5.1: Specification of a union of two types.

'U2' is the discriminated union of two data types, 'First' and 'Second'. It has six operators with the functionalities given. The use of '↛' in the signatures of 'To_First' and 'From_First' indicatesthat they are partial operations. This is emphasised by the presence of explicit preconditions. 'f', 's' and 'u' are free variables with types 'First',

'Second' and 'U2' respectively. Some styles of specification would declare 'f', 's' and 'u' explicitly; I have not done so because their types can be inferred from the signatures of the operators.

The only way values of type 'U2' can be generated is by application of 'From_First' and 'From_Second', the injection operations. 'U2' values can be examined only by application of the two projection functions 'To_...' and the two inspection functions 'Is_...'. Thus 'U2' can be completely defined by specifying the result of applying the four inspection and projection operations to applications of the two injection operations. This reasoning indicates that at most eight axioms are required; in fact, because the projection functions are partial there is no need to specify the meaning of 'To_First(From_Second(s))' and 'To_Second(From_First(f))'. Indeed, the preconditions emphasize that these expressions have no meaning. The six axioms of Figure 5.1 are thus sufficient to completely specify the type.

'U2' is restrictive in two ways: 'First' and 'Second' are not parameters, and there are only two of them. Both of these restrictions can be lifted, although there are difficulties in doing so in a completely formal framework.

The simplest way of promoting 'First' and 'Second' to the status of parameters is to view 'U2' not as a type but as a *type schema*: the meaning of 'U2(a, b)' is then given simply by substituting 'a' and 'b' uniformly for 'First' and 'Second', as in a macro expansion. The generic parameters of Ada are given roughly this interpretation [96]. Note that in doing this the names of the operations remain unchanged: 'U2(a,b)' has the two injection operators

From_First: (a) → U2(a, b)
From_Second: (b) → U2(a, b) .

'U2(a, a)' is thus perfectly well behaved, and 'From_First(a)' is not the same as 'From_Second(a)'. 'U2' is a disjoint or non-absorbing union; set-theoretic union is absorbing in the sense that '{a} ∪ {a}' is '{a}'. It is to emphasize this distinction that I have adopted the name oneof for my type generator.

The shortcomings of the macro substitution definition of parameterization are illustrated by the recursive type definition

type R = U2(integer, R) .

There are other problems too: each instantiation must be type-checked independently, and the macro expansion approach cannot deal with recursive procedures with type parameters [33]. These difficulties can be avoided by restricting the language, or they can be faced by trying to define a semantics for the concept of type. An apparently successful attempt to do the latter is the Russell project of Cornell University [24] [18] [12] [17]. In the Russell language 'R' is perfectly legal and well-behaved.

The full power of the Russell type system is not necessary to relax the restrictions of 'U2' if one is prepared to permit a little license in the description. By assuming oneof to be a primitive of the programming notation I will avoid the question of whether the power thus admitted is available for the definition of the user's own types.

First, oneof will be permitted to take an arbitrary number of argument types; the only restrictions are that there shall be at least one argument and that each shall be represented by a syntactically distinct identifier.

Second, the *names* of the operators will be parameterized as well as their functionalities; this is a great help to the reader. The resulting type schema is shown in Figure 5.2.

```
type O = oneof(Id1; ...;Idn)
operators
    Is_Id1: (O) → Bool
       .
       .
       .
    Is_Idn: (O) → Bool
    From_Id1: (Id1) → O
       .
       .
       .
    From_Idn: (Idn) → O
    To_Id1: (O) ↛ Id1      pre To_Id1(o) ≡ Is_Id1(o)
       .
       .
       .
    To_Idn: (O) ↛ Idn      pre To_Idn(o) ≡ Is_Idn(o)
axioms
    Is_Idi( From_Idi(x) ) = true    ∀i ε [1..n]
    Is_Idi( From_Idj(x) ) = false   ∀i, j ε [1..n] st i ≠ j
    To_Idi( From_Idi(x) ) = x       ∀i ε [1..n]
end of oneof
```

Figure 5.2: Axiomatic Specification of oneof

The other extension used in Chapter 4 is purely syntactic: the declaration

```
type O = oneof(Id1:T1; Id2:T2)
```

is shorthand for

```
type Id1 = T1
type Id2 = T2
type O = oneof(Id1; Id2)
```

and provides a convenient way of declaring types with distinct names.

This proved especially useful for renaming 'singleton'.

The type 'singleton' itself is trivial to define:

```
type singleton
operators
    : () → singleton
end of singleton
```

Singleton contains one generator which I denote by the empty operator symbol. No axioms are necessary because there are no enquiry operators whose results must be defined.

One further point which needs clarification is the meaning of the type declaration

```
type Id = T
```

This introduces a new type known as 'Id', isomorphic to 'T' but distinct from it. Thus, after the definitions

```
type underflow = singleton
type overflow = singleton
```

the type checking mechanism will prevent confusion of 'underflow' with 'overflow'.

5.5 Formal Specification of Two Data Types

Having set up the required machinery by defining oneof and 'singleton', it is possible to present a formal definition of a stack. The added complexities of defining a generic 'Stack' will be avoided here by remaining with 'StackofInt'.

```
type StackofInt
Operators
    Empty:() → StackofInt
    Push:(StackofInt, Int) → StackofInt
    Pop:(StackofInt) → StackofInt
    Top:(StackofInt) → oneof(Int; Underflow:singleton)
    IsEmpty:(StackofInt) → Bool

Axioms
    Pop(Empty) = Empty
    Pop( Push(s,i) ) = s
    Top(Empty) = From_Underflow()
    Top( Push(s, i) ) = From_Int(i)
    IsEmpty(Empty) = True
    IsEmpty( Push(s,i) ) = False
end of StackofInt
```

Figure 5.3: Specification of StackofInt

The definition of Figure 5.3 avoids the contradictions which troubled Majster [69]. In her analysis, the functionality of 'Top' is 'StackofInt → Int'. If 'Top(Empty)' is substituted for 'i' in the second axiom, one obtains

```
Pop( Push(s, Top(Empty)) ) = s
```

However, the axiom 'Top(Empty) = error' and the rule of propagation of errors give

```
Pop( Push(s, Top(Empty)) ) = error .
```

According to my definition, the troublesome expression is simply a type error. 'Top(Empty)' is of type oneof(Int; Underflow) whereas the second parameter of 'Push' must be of type 'Int'. Moreover, it is impossible to convert 'Top(Empty)' to an 'Int'.

The 'IsEmpty' operator is actually redundant with the definition of stack given in the figure: 'IsEmpty(s)' can always be simulated by the

expression 'Is_Underflow(Top(s))'. Nevertheless, a designer might include it for reasons of efficiency or convenience. In contrast, if 'Top' is a partial operation then 'IsEmpty' is essential because it defines the domain of 'Top'.

```
type SymbolTable(Ident, Attr)
operations
    EmptyST: () → SymbolTable
    InsertST: (SymbolTable, Ident, Attr) → SymbolTable
    LookUpST: (SymbolTable, Ident) → oneof(Attr; absent:singleton)

axioms
    LookUpST(EmptyST) = From_absent()
    LookUpST(InsertST(st,i,a),j) =
            if Ident.Equal(i,j) then From_Attr(a)
                                else LookUpST(st,j)
end of SymbolTable
```

Figure 5.4: An unbounded symbol table.

The symbol table of Section 4.2.1 provides an example where partial functions are inappropriate. A complete axiomatic specification of an unbounded symbol table is presented in Figure 5.4. The type there specified differs from that of Figure 4.07. Most significantly, the latter uses procedures rather than pure functions; it is also bounded. Figure 4.07 can be cast into a form where it uses Figure 5.4: the semantics of an 'insert' procedure which alters its parameters might be given as

{st=st'} insert(st, id, val) {st=InsertST(st', id, val)}.

However, the details of such a specification of a symbol table are not relevant to the subject of this thesis, and will not be described further here.

5.6 Conclusion

The study of data type specifications has given rise to another instance of the "abstract errors" myth. Attempting to solve the problems thus created may be an interesting mathematical exercise, but not one that seems to be particularly relevant to the production of quality programs.

Specifications like that of the symbol table need oneof results. Error algebras are not an adequate tool because they require that one of the results of 'Lookup' be designated an error, and that once an error has been noted the program remains in an exceptional state [27].

The specification of oneof itself could be done with error algebras. It could also be done denotationally. I have preferred to use an axiomatic definition with explicit preconditions for two reasons, the first of which is that it avoids the introduction of a lot of extra mathematical machinery.

The second reason is that axioms have the unique property of allowing one to say nothing when there is nothing one wishes to say; in other words, they enable one to leave things intentionally unspecified. In contrast, a denotational definition insists that all results are defined, even if they are $\perp$. Allowing the implementor sufficient freedom to optimise seems to me to be an essential property of a specification. One of the beauties of the abort statement [23] is that *any* implementation is correct. The same benefit accrues to type specifications when explicit preconditions are allowed.

Chapter 6

PROGRAMMING WITHOUT EXCEPTION HANDLING

The examples of Chapter 4 demonstrate that it is indeed possible to program without exception handling mechanisms, and to do so clearly in a modular, strongly typed language. However, the notation I have used is not that of any existing language, and it is necessary to examine the features I have introduced and ensure that my cure is not worse than the disease.

The most obvious extension I have used is the oneof constructor for discriminated unions. In certain of the examples I have also assumed that procedures and functions are values which can be manipulated within the language. I will now consider these extensions in more detail.

6.1 Programming with Discriminated Unions

Unions of different types actually pre-date the inclusion of user definable data types in programming languages. In Fortran and Algol 60 a lookup function would typically operate on an array, and return either the index of the array element containing the required value or some integer such as zero which is not a valid array index. The result of such a function is indeed a union of distinct types, although Fortran and Algol 60 do not provide a means of expressing this.

The programming language Pascal [101] [50] was the first to include types which are subranges of other types. Subranges make it possible to specify the range of a lookup function exactly: instead of saying that the result is an integer, the programmer can express the constraint that it is in the range '1..n'. This is a desirable thing to do because it both makes the intent of the program clearer and makes it easy for the compiler to

deduce that no range checks are necessary when the array is accessed. However, the inclusion of even one value which is not a valid index in the result type of the lookup function means that it is different from the subscript domain of the array. The need to use different type names reduces readability and makes some range checks necessary. Used in this way the type '0..n' is indeed a union of '1..n' and zero.

A similar situation arises in languages such as BCPL [80] where structures are accessed by address. A lookup operation typically returns the address of the appropriate structure. If no such structure exists an invalid address is returned as result.

This practice was recognized by the designers of Algol 68 and Pascal. References in Algol 68 and pointers in Pascal may either refer to a value of the appropriate type or be null, i.e. not refer to anything.

When a function like a lookup is used the programmer must always bear in mind the possibility that the key was not found. Usually the result must be checked to see if it is a valid array index, address, pointer or reference. However, the context may sometimes be such that the programmer is sure that the key will be found: in some compilers, for example, certain standard identifiers will always be in the symbol table because they are inserted by the set-up phase. Unfortunately, even when programmers are sure of something, they may be wrong: in the compiler example, it is possible that the set-up phase is not working correctly because of a bug in the software (or a failure of the hardware).

The above methods of indicating failure of the search share a pleasant property. Even if the programmer decides that no explicit test of the validity of the result is required, an attempt to misuse an invalid result

will lead to an immediate alarm. At least, the other features of the language are such that a reasonable implementation will provide checks which will catch this mistake. The application of an array with bounds '1..n' to a subscript of zero ought to generate an alarm, and by using zero to represent the fact that the key was not found, the programmer is taking advantage of a built-in check which was already part of the implementation. Similarly, in a BCPL implementation there is usually some negative or very large positive value which the hardware will detect as an illegal address. Indeed, this hardware characteristic is probably responsible for the inclusion of nil in Algol 68; it could very well have been omitted, and is one of the few facilities of that language which are available by other means [70].

Both Algol 68 and Pascal provide other ways of forming unions in addition to those mentioned. Indeed, programmers familiar with those languages and faced with a need to implement my oneof constructor may have immediately looked to Algol 68's union types and Pascal's variant records without considering null references and subranges at all, for these facilities are not often recognized as unions. Yet they should be so recognized, both because a language should provide one way of representing an abstract concept rather than three, and because the implementations used are often particularly efficient, and should be adopted as widely as possible. Since the most appropriate implementation for a particular union is governed by the hardware, its selection ought to be left to the compiler. If there are three different source language constructs which compile into different code but which express the same idea, the programmer must know all about the hardware and the implementation to make a rational choice between them. This is clearly undesirable.

The oneof type constructor defined in Chapter 5 is designed to be convenient in use both when the type of a result is known and when it is not. Consider the 'LookUpST' function defined in Figure 5.4 and assume first that the programmer has proved, either formally or informally, that the result of 'LookUpST' will be an 'attr'. He may then use the appropriate projection function, as in

```
Attribute := To_attr(LookUpST(st, Id)) .
```

This expression is valid exactly when the assertion {Is_attr(LookUpST(st, Id))} is true. If the argument of 'To_attr' is not a oneof value injected from an 'attr' then the expression is an error. With the example of a standard identifier in a compiler's symbol table this error would indicate that the part of the program or computer which initialised the symbol table was not working correctly. A reasonable implementation of 'To_attr' will try to halt the program and generate an error message (but the semantics of oneof does not require that all implementations are reasonable). The second case occurs when the programmer is unsure whether an 'Id' will be in the table. It is then incumbent on him to write code which deals with both possible results, as in the following example.

```
var r : oneof(attr; absent)

r := LookUpST(st, Id)
if Is_absent(r) => SyntaxError('Undeclared identifier', Id)
[] Is_attr(r) => Attribute := To_attr(r)
fi
```

Essentially, the alternative construct reduces the second case to the first: within the guarded commands it is known whether 'Id' is in 'st'.

In Algol 68 the type 'attr' would be represented by some structured "mode" attr. One way of representing the result of 'LookUpST' is as a reference to an attr. This would permit the use of the null reference to indicate the absence of an item. Because the dereferencing operator is implicit, the case where the programmer is sure of the result could be coded as

```
Attribute := LookUpST(st, Id) .
```

If 'Id' was not, after all, found in 'st' this expression would lead to an attempt to dereference the null pointer, which is an undefined operation.

On the other hand, if the programmer does not know whether or not 'Id' is in 'st', he must test the result, as in the following program fragment

```
ref attr r = LookUpST(st, Id)
if r is ref attr (nil)
then SyntaxError('Undeclared identifier', Id)
else Attribute := r
fi .
```

These expressions are fairly clear, but have some drawbacks. Defining the result of 'LookUpST' to be ref attr does not make it obvious that a null reference may be produced, or say anything about what this means: all reference types contain nil whether or not the programmer wants to use it. In contrast, 'oneof(attr; absent)' is clearly used for the express purpose of allowing 'absent' as a result, and the name 'absent' is itself an aid to understanding. Another problem with using nil to indicate an "exceptional" result is that only one such result can be represented: there is only one null reference of each type.

Using a reference result rather than the structure itself may also be expensive. Returning a reference requires that the storage referred to outlasts the invocation of the function. In the particular case of a table look-up this requirement is fairly easily met because the table entries will usually be global. But in general it may be necessary to copy a local structure onto the heap simply so that a reference may be returned.

The use of a null reference may be expensive in another way too. While the machines of the late 1960's and early 1970's usually trapped attempts to access an invalid address, this is no longer always the case. Many mini- and micro-computers have a physical store as large as their address space, so there is not bit pattern available to represent nil. Others may simply interpret any address modulo the size of the store.

Any of these difficulties may cause the programmer to look for better methods of representing different types of result than the use of null references. Most strongly typed languages provide some sort of union constructor and some way of declaring new types such as 'absent'.

Algol 68 [97] provides a built-in type constructor union which has some of the properties of oneof. A type (called a "mode") can be named, as in

```
mode AttrorAbsent = union(attr, absent) ;
```

such a type can be used in exactly the same way as a primitive type. However, this declaration does *not* create a *new* type; it names an existing one. There is no way to create new types in Algol 68; in particular there is no way to create the singleton enumeration type absent. The declaration

```
mode absent = void
```

simply provides a new name for void. If one also declared

```
mode AttrorVoid = union(attr, void)
```

the two tags AttrorVoid and AttrorAbsent would identify the same type.

Another difficulty with the union constructor of Algol 68 is that it is absorbing. In the context of the declarations

```
mode BadFormat = string;
mode UnrepresentableInteger = string;
mode SumResult = union( int, Badformat, UnrepresentableInteger );
```

the mode SumResult is indistinguishable from both union(int, string, string) and from union(int, string).

These difficulties are to some extent mitigated by the rule that two record types ("structures") are only identical if the field names are the same. Thus

```
mode BadFormat = struct(string BadFormat);
mode UnrepresentableInteger = struct(string UnrepresentableInteger)
```

identify two distinct types. However, because of the (quite unorthogonal) restriction that a field of a structure cannot have mode void, absent must be declared as something like struct(bool absent).

In Algol 68, injection (which I have indicated by use of operators 'From_...') is available implicitly as a coercion. In the right context, such as the result clause of a procedure delivering a union, a value of one of the component types will be automatically "united" to the required type. This provides the briefest possible syntax with minimal loss of information or readability, because the type required is always obvious from the context. In places where the type is not obvious it can be made so by use of a "cast",

such as 'AttrorAbsent(expr)'.

While this makes operations like 'LookUpST' easy to write, they are, unfortunately, difficult to use. This is because the inspection and projection operations do not have simple realizations in Algol 68. Instead the language provides a construction called the "conformity clause": this is the *only* means of taking a union apart. It is quite suitable for dealing with the second example, where the programmer does not know the type of the result.

```
case LookUpST(st, Id)
in (absent): SyntaxError('Undeclared identifier', Id)
,  (attr r): Attribute := r
esac
```

Unfortunately it is hopelessly cumbersome for the case where the programmer is sure of the type of the result. The simple assignment to 'Attribute' must be coded as follows:

```
Attribute := case LookUpST(St, Id)
             in (attr a): a
             ,  (absent): (    print('Impossible happened')
                          ;    stop
                          )
             esac .
```

If one is sufficiently confident of the reliability of the program to omit the alarm, this can be shortened to

```
Attribute := case LookUpST(st, Id)
             in (attr a): a
             esac
```

but this is still very inconvenient if union is used as frequently as I have advocated.

It is possible to define Algol 68 procedures which simulate the 'To_attr' and 'Is_absent' operators, but such procedures must be written for every component of every union, and there is no "procedure schema" mechanism in Algol 68 to help produce the definitions.

In Pascal [101] [50] the situation is also unsatisfactory. The problem is compounded by two factors:

(i) the orthogonal mathematical concepts of union and Cartesian product have been combined into a single linguistic construction, the "variant record";

(ii) functions may not return records as their result; a pointer to a record must be used instead.

The Pascal variant record is a structure containing an optional *fixed part* and a *variant part*. The latter consists of a tag field of some enumerated type and several alternative variants, one for each tag value. The intention is that only one of the variants should be current and that its identity be given by the tag. A variant record used to represent the result of 'LookUpST' would have no fixed part:

```
type    WhetherAttrorAbsent = (ATTR, ABSENT)
;       AttrorAbsentRecord =
            record case WhichType: WhetherAttrorAbsent
                   of ATTR: (attribute: Attr)
                   ; ABSENT: ()
            end
;       AttrorAbsent = ↑AttrorAbsentRecord
```

As it stands this construction is not type-safe. The tag field is optional and can in any case be assigned to independently of the rest of the record, indeed there is no way of constructing a record without doing so. Variant records were actually intended to create a loophole in the type system so

that machine dependent code (such as device drivers) could be written [102]. For this reason many implementations will *not* generate any run-time error message when a non-existent variant is accessed.

When Pascal variant records are disassembled there is a significant improvement over Algol 68: a case statement need not be introduced if case analysis is not required. When one knows the type of the result of 'LookUpST' one may write

```
Attribute := LookUpST(st, Id)↑.attribute .
```

Where enquiry is necessary an if statement can be used, as in

```
var r : AttrorAbsent;

r := LookUpST(St, Id)↑;
if r.WhichType = ATTR
then Attribute := r.attribute
else SyntaxError("Undeclared identifier", Id) .
```

Writing 'LookUpST' is more difficult in Pascal than in Algol 68 because the result must be explicitly created with the procedure 'new'. To avoid this inconvenience, and to ensure that the tag field is always correctly set, injection functions like those of Figure 6.1 could be used. As in Algol 68, such functions must be written by the programmer for each union which requires them.

```
function FromAbsent : AttrorAbsent;
var result : AttrorAbsent;
begin new(result)
;     result↑.WhichType := ABSENT
;     FromAbsent := result
end;

function FromAttr(a:Attribute) : AttrorAbsent;
var result : AttrorAbsent;
begin new(result)
;     result↑.WhichType :=  ATTR
;     result↑.attribute := a
;     FromAttr := result
end;
```

Figure 6.1: Pascal functions to create union values.

The programming language Ada perpetuates the confusion of product and union introduced by Pascal. Type safety is improved by making the tag-field a constant: it can be changed only by updating the whole record.

Ada does not have a facility powerful enough to permit the declaration of the oneof type schema as I have defined it. The possibility of "generic" types is allowed, but only the functionalities of the operators may be parameterized, not their names. In other words, the generic type 'U2' (see Section 5.4) can be defined, but not generalized to oneof. So the result type of 'LookUpST' will probably be represented as

```
type WhetherAttrorAbsent is (IsAttr, IsAbsent)
type AttrorAbsent(WhichType: WhetherAttrorAbsent) is
    record case Which Type
           is  when IsAttr => attribute: Attr
           ;   when IsAbsent => null
           end case
    end record
```

Values of this type are constructed explicitly using record aggregates such as

```
(WhichType => IsAttr, Attr => Atable[i])  .
```

One might choose to define conversion functions 'FromAttr' and 'FromAbsent' as in Figure 6.2. Ada allows overloading of function names so that two different oneof types can both have 'FromAttr' and 'ToAttr' operators; this is not permitted in Pascal.

```
function FromAttr(a:Attr) return AttrorAbsent is
begin return(WhichType => IsAttr, attribute => a)
end;

function FromAbsent return AttrorAbsent is
begin return(WhichType => IsAbsent)
end;
```

Figure 6.2: Ada routines for construcing oneof values.

Variants may be disassembled in Ada just as in Pascal by accessing the components with the dot notation.* If a component which does not exist is referenced the exception 'constraint_error' is generated. Unfortunately it is possible to suppress this action, thus breaching what would otherwise be a strong type system.

The CLU language [61] [64] provides a type constructor called oneof which is similar to the oneof used in Chapter 4. The result of 'LookUpST' could be declared in CLU as

```
AttrorAbsent = oneof[Attribute:Attr, Absent:null]
```

'Attribute' and 'Absent' are labels; 'AttrorAbsent' is different from another oneof with fields of the same types but with different labels. The CLU oneof is non-absorbing:

```
oneof[ Integer:int, Bad_format:string
     , unrepresentable_integer:string ]
```

is useful in CLU because it is possible to ask whether a string value indicates

---

* The first Ada definition [54] did not allow the result of a function call to be subject to component selecton. This omission has now been rectified.

a 'Bad_format' or an 'Unrepresentable_integer'. For each label 'l' in the oneof there is a predicate 'is_l', a function 'make_l', and a procedure 'value_l' which correspond fairly closely to the 'Is_...', 'From_...' and 'To_...' operations defined in Figure 5.2. A minor difference is that in CLU the operators depend on the labels, not the names of the types. The most important difference is that the CLU 'value_l' procedures generate exceptions if their argument is not of the correct type, whereas my 'To_ ...' operators require that their argument be of the correct type.

CLU also provides a control structure called the tagcase statement, which may be used for dissecting unions when case analysis is required. The two examples may thus be written in CLU as

```
Attribute := Value_Attribute(LookUpST(st, Id))
```

and

```
tagcase LookUpST(st, Id)
tag Attribute(a:attr): Attribute := a
tag Absent: SyntaxError('Undeclared identifier', Id)
end
```

The CLU oneof provides all the capability necessary for handling results of differing types. If the definition of the CLU oneof did not rely on the signal mechanism, it would be possible to view CLU exception handling as an alternative syntax for oneof results.

Few would argue against some kind of union mechanism in a strongly typed language. Per Brinch Hansen is one who has done so, and the Edison language does not include unions in any form [40].

Edison is derived from (but simpler than) Pascal. Variant records were eliminated because they were both complicated and insecure. Nevertheless,

when unions are really needed, as they are in the Edison compiler for describing symbol table entries [39], some way of simulating them must be found. Hansen used the retyping mechanism of Edison and a knowledge of the size of the representations of different types to simulate union. Such implementation dependencies seem to be a high price to pay for the elimination of a type generator.

I recommend that any strongly typed language should have a built-in type generator like oneof. Its type mechanism should allow new types to be created easily. Concise notations should also be available for constructing values of oneof types and for projecting them into their component types. In particular, the language should not insist on case analysis unless there are cases to analyse. A statement like the Algol 68 conformity clause or the CLU tagcase can sometimes be convenient, but it is never essential, and should not be allowed to obscure the simplicity of oneof.

The motivation for the Algol 68 conformity clause seems to be the removal of partial operations. Whereas applications of the projection operators of oneof can be undefined, the conformity clause is always total. Nevertheless, Algol 68 does not shrink from using partial operations in other places: division, dereferencing and application of arrays to subscripts are all partial.

6.2 Manipulating Procedures in Programming Languages

In certain of the examples considered in Chapter 4, procedure parameters were used in lieu of exception handlers. In the inconsistent string problem (see Section 4.2.2) it was also assumed that record components could take on procedure values. Are these facilities one would expect to find in modern, structured languages, and are they more justified than exception handling?

Procedures as parameters are allowed in PL/I, BCPL, Algol 68, RTL/2, Pascal, Ada and Edison, as well as in Fortran and Algol 60. However, they are not available in Modula, Euclid, 3R and some Pascal dialects, such as UCSD Pascal [94]. Clearly the designers of these languages had their reasons for omitting procedure parameters; it is instructive to speculate about what they might be.

The first reason is probably that procedure parameters can be awkward to implement. In languages with Algol-like scope rules it is possible to declare both variables and procedures in an inner block. Within the body of such a procedure such variables can be used. However, because they are neither resident in the procedure's own stack frame nor may be allocated statically, there is some difficulty in compiling code to access them. So long as all calls of the procedure are explicit, providing a pointer to the appropriate stack frame is fairly easy: it becomes more difficult if the procedure may be passed as a parameter.

These problems do not arise in Fortran implementations which allocate all storage statically. They have been faced and surmounted by Algol 60 implementors many years ago, and the solutions are well established. BCPL [80] avoids the problem altogether by allowing a procedure to access only its local variables (including, of course, its parameters) and those free variables of the surrounding context which have been declared to be statically allocated. In practice these restrictions cause few problems and permit procedure calls to be implemented very simply and efficiently.

Another reason, which probably applies in the case of Euclid, is the fear that procedure parameters complicate the semantics. Certainly the formulae of a semantic description become larger, but I do not think that

they become much more profound. Consider a procedure which performs numerical integration of a function: the function and the limits of the integration must be parameters of the integration procedure because it is impossible to define integration without referring to them. This is true whether one interprets the word "define" with mathematical rigour or as a vague mental image: it is the *concept* of integration which is parameterized, not any particular realization of it. It should come as no surprise, therefore, to discover that the semantics of the procedure is parameterized by the semantics of the function on which it operates.

When predicates are used as a means of formalizing semantics, the pre- and postcondition of the calling routine will depend on the pre- and postcondition of the parametric routine. This is exactly what happened in my formulation of Levin's symbol table example (see Section 4.2.1). The insert operation took a procedure parameter which was invoked only when the table was full; its semantics had the form

```
{ not tablefull or pre proc }
            Insert(st, id, attr, proc)
                    { (tablefull and post proc)
                      or (not tablefull and<id,attr> ε st) }  .
```

The technicalities of specifying the semantics of procedure parameters are rather far from the subject of exception handling. Readers interested in particular formalisms are referred to [84] or [86]. It should be noted, however, that exception handling mechanisms which permit handlers to resume the routine which generated the exception also provide a way of parameterizing the semantics of routines. (The mechanisms of Levin and Mesa have this property.) If one objects to procedure parameters on semantic grounds, one must also object to resumption exception handling. In the examples I have

used the former only to provide the latter.

The third objection to procedure parameters is that as provided in Algol 60, PL/I and Pascal; they make type checking difficult. The forms of parameter specification included in those languages do not permit the programmer to specify the types of the parameters of a parametric procedure. Type checking a call of such a parametric procedure thus involves examining every call of its parent procedure. The simple way of resolving this objection is to require full specification, as do Algol 68 and Edison.

The final reason for not wishing to permit procedure parameters is that they are unnecessary. I have argued strongly that unnecessary features should not be included in programming languages. Let us see, then, what happens if procedure parameters are removed from a language.

Suppose procedure 'A' has a procedure parameter 'P', and that 'A' is called from 'n' different places in the program, with actual procedure parameters 'P1', 'P2', ..., 'Pn' in place of 'P'. This program can be transformed by placing the declarations of 'P1', ..., 'Pn' *inside* 'A', and replacing the procedure parameter 'P' with a parameter of an enumeration type (or an integer) which takes a different value at each call. Inside 'A' the appropriate 'Pi' is selected by testing the enumeration value. It should be clear that the semantics of the program are unchanged by this transformation.

However, what has been drastically changed is the modular structure of the program: it has been totally destroyed. This should be obvious when 'A' is called 'Integrate' and is available in a numerical analysis library. My transformation implies that every function it could possibly be called upon to integrate must be declared inside it, so that the user can select the function he requires by supplying the appropriate enumeration

value as a parameter! Even in the case where the calling and parameteric routine are written by the same programmer and compiled at the same time, the modularity and modifiability of the program are greatly enhanced by using a procedure parameter rather than an enumeration value. Since these are amongst the most desirable properties of a program, it seems clear that procedure parameters are indispensable.

A note about Ada is in order here. Ada procedures may have procedure parameters, but they must be declared as "generic". Syntactically, the effect of this is to require that the actual procedure parameter is supplied in a "generic instantiation", which creates and names a partially parameterised procedure. This "instance" is then called in the normal way, when ordinary constant and variable parameters are supplied. Semantically, Ada generic procedure parameters appear to be usable in the same ways as procedure parameters in Algol 60: the syntactic alum seems not to affect the expressive power of the language.

Procedure variables are more difficult to defend than procedure parameters. All the above objections to procedure parameters apply also to procedure variables, but with increased force.

The implementation problems of procedure parameters become real semantic problems when translated into the domain of procedure variables. A procedure declaration which accesses non-global free variables can be interpreted in two different ways. The free variables can be evaluated at declaration time, and "frozen" into the procedure once and for all. The more usual interpretation is that the free variables are evaluated every time the procedure is called. In this case the scope of the procedure (i.e. the area of the program in which it is meaningful) is the same as that of the free variables.

The problem with procedures whose scope is restricted in this way is that if assignment of procedure values is permitted, a procedure with a given scope may be assigned to a variable with a larger scope. (The same problem arises if a language permits references to name scoped variables.) Algol 68 avoids these dangling reference problems by prohibiting the assignation of any value to a variable whose scope is larger; unfortunately this restriction is very difficult to enforce at compile time. My first solution to Levin's inconsistent string problem (see Section 4.2.2) breaks this rule, and would thus not be legal in Algol 68. The restriction can be circumvented in the same way as in the example of Section 2.5. First, a local string variable is declared and initialised to the value of the appropriate element of 'dnames'. Then the bad string routine of the local string is set to 'UseDuplicate' and the local string is compared with the parameter of 'LookUp'. The code that results (Figure 6.3) is in one respect simpler than the original (Figure 4.13) because there is no need to save and restore the old bad string procedure.

Another major difficulty with procedure variables is that it can be very difficult to determine what will happen when they are invoked. Like references, procedure values are unprintable: unlike references there is not even a printable value at the end of the chain. This problem is especially acute when the variable (or the structure of which it is part) is accessed and assigned in many different places. Of course, this objection applies equally to Levin's exception handling mechanism: the handler for a given exception may have been set and reset in various different places. One way of avoiding this problem is to allow procedure constants but not procedure variables. When a structured value with procedure valued components

is first created those components are initialized: they may not subsequently be changed.

```
proc Lookup(t : symboltable; s : string) returns
                    r : oneof(Value; Notfound : singleton)
is for  i:upto(1, last)
   do
        proc FixTable() is
            names[i]:= names[t.last]
            dnames[i] := dnames[t.last]
            values[i] := values[t.last]
            last := last - 1
            t.LostEntry()
        end of FixTable

        proc UseDuplicate() returns s:string is
            s := t.dnames[i]
            SetResetRt(s, FixTable)
        end of UseDuplicate

        var LocalString : string := t.names[i]
        SetBadStringRt(LocalString, UseDuplicate)

        if =(LocalString, s)
        then r := From_Value (t.values[i])
             return
        fi
    od
    r := From_Notfound()
end of Lookup
```

Figure 6.3: Lookup using inconsistent string module with notification procedures and Algol 68 scope rules.

The *envelope* data structure of PascalPlus [13] permits the declaration of constant fields. It is based on the Simula class concept, as are many similar structures in other languages. A 'string' might be defined as in Figure 6.4.

```
envelope string(procedure BadStr; procedure Reset)
;   type *strchars = array [1..n] of char
    ;   var chars:strchars
        ;   length:integer
    ;
    procedure *SetStr(var s:string; c:strchars; l:integer)
    ;   var j:integer
    ;
    begin   s.length := l
    ;       for j := 1 to l do s.chars[j] := c[j]
    end
;
    function *EqualStr( var s1:string
                      ; var s2:string ):boolean
    ;   function IsOK(var s:string):boolean
        ;   var c:strchars
            ;   i:integers
        ;
        begin   if s.length<0 then
                begin   s.BadStr(c,i)
                ;       SetStr(s, c, i)
                ;       if s.length<0 then
                        begin   s.Reset
                        ;       SetStr(s, c, 0)
                        ;       IsOK := false
                        end
                        else IsOK := true
                end
                else IsOK := true
        end

    ;   var i:integer
        ;   AreEqual:boolean
    ;
    begin
        if IsOK(s1) and IsOK(s2) then
            if s1.length = s2.length then
            begin
                i := 0
            ;   AreEqual := true
            ;   while i < s1.length and AreEqual do
                begin
                    AreEqual := s1.chars[i] = s2.chars[i]
                ;   i := i+1
                end
                Equal := AreEqual
            end
            else Equal := false
        else Equal := false
    end
;
begin length := 0; *** end
```

Figure 6.4: A string envelope in PascalPlus

This string envelope is adequate for the solution of the inconsistent strings problem. The 'BadStr' and 'Reset' procedures can be initialized when the strings are first placed in the symbol table, and do not subsequently need to be changed. Of course, the transformation which produced a 'string' with only *one* procedure field (see Figure 4.14) relies on being able to assign procedure values, and could not be applied in PascalPlus. Nevertheless, one should recall that the simplest solution to the inconsistent string problem is for the string module to contain the list of duplicate names; no procedure parameters, variables or exception handling mechanisms are then needed.

I am clearly in danger of becoming too involved with the pros and cons of procedure variables. My experience in BCPL and Algol 68 makes me realize that they can be difficult to use and to debug, but also makes me appreciate that they can provide a simple, efficient and flexible solution to certain programming problems. I do not wish to take sides on the question of whether procedure variables should be included in programming languages. I have shown that exception handling mechanisms can be replaced by procedure parameters and procedure constants alone, but that procedure variables can be very useful. The decision on whether to include them must be taken by the language designer in the light of the aims and objects of the language. Procedure valued fields in structures seem to be an integral part of languages such as CLU and Mesa which aim to support "object oriented programming". By this is meant a style of programming in which one's primary concern is with blocks of storage representing *objects*, which have the property that their identities remain fixed while their fields may be changed. Such languages rely on references at the semantic level but can be

implemented very efficiently on certain architectures. On the other hand, procedure variables are foreign to languages which compute with *values*. Indeed, this idea is taken to its logical conclusion in so-called applicative languages which discard the notions of variable and assignment altogether.

6.3 Conclusion

In order to program without exception handling I have used procedures which return results of different types. I have also used procedure parameters when a procedure needs some assistance. Both of these techniques are more than twenty years old. oneof results are almost certainly as old as programming itself: unfortunately many recent strongly typed languages have made them difficult to use. The idea of procedure parameters interfering with the action of a procedure was described by Rutishauser in 1961 [82]; it has become more important over the years as programmers have realized the importance of modularity.

In a programming language designed for manipulating "objects", particularly "shared objects", Levin's exception handling mechanism provides facilities which the others do not. These same facilities can always be obtained by the use of procedure variables, and often by procedure constants. Whether "object oriented programming" is a desirable development is possibly the subject for another thesis. Nevertheless, procedure variables are usually found in languages which support this style of programming. Indeed, the implementation of Levin's mechanism relies on being able to manipulate lists and sets of exception handlers, and thus the implementation language at least must support manipulation of procedures.

Chapter 7

CATASTROPHE HANDLING

One of the central themes of this thesis has been that "exception" is not a well-defined concept, and that it is therefore not appropriate to incorporate exception handling into a programming language. In contrast, the term "catastrophe" can be rigorously defined, and the increasing emphasis on reliability has made it important to do so.

This chapter investigates the difference between catastrophes and exceptions, and examines some examples of "catastrophe handling". Thus prepared, it becomes clear that the sort of exception handling mechanism discussed in Chapters 2 and 3 does not address the problem of surviving catastrophes. Instead it will be seen that the manner in which catastrophes are detected and handled and the degree of recovery that is possible are important design attributes of a system. In a database, for example, the provision made for recovery after failure can influence the structure of the entire system [31].

Having provided a mechanism for surviving catastrophes, it is of course possible to (mis)use it for other purposes. The chapter concludes with a discussion of the advisability of this practice.

7.1 Catastrophes, Exceptions, Errors and Faults

A *catastrophe* occurs when the behaviour of a component deviates from its specification. This use of the term catastrophe coincides with the meaning of *failure* in the literature on reliability [57] [71] [78]. However, writers on exception handling often use *failure* in a more general sense, and so it seems worthwhile to introduce a new term.

The connexion with reliability should be obvious: the reliability of a system is a measure of its conformance to its specification. An absolutely reliable system is one which never fails, i.e. in all circumstances behaves as its specification requires. Randell [78] emphasises that without an authoritative specification it is meaningless to talk about reliability.

It should be clear that catastrophes are, of their very nature, unexpected. If a program invokes a routine which is specified as sorting an array, but finds that on completion the contents of the array is not a permutation of its original contents, then a catastrophe has occurred. That which was specified as impossible has taken place: this must be an unexpected event.

In contrast, exceptions must represent anticipated and well-defined events. Exceptions are only generated in the expectation that they will eventually be handled. For this to be possible, a routine which can generate an exception must declare this fact as part of its specification. Full details of all of the consequences of the exception must be provided if it is to be handled effectively.

The terms *error* and *fault* are used subjectively in the reliability literature. In [78] a state is said to be *erroneous*

> when there exist circumstances (within the specification of the use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which we do not attribute to a subsequent fault. The subjective judgement ... derives from the use of the phrases "normal algorithms" and "which we do not attribute" in this definition.

The term *error* thus designates a datum: that part of the state which is

incorrect. A *fault* is an event: the mechanical or algorithmic cause of an error.

As an example of the use of these terms, consider a filing system. A file as an abstraction is a sequence of characters, but its representation may be a linked list of disk pages. Suppose that because of a mechanical fault in the disk heads or because of an algorithmic fault in the file stream, one of the link fields is written incorrectly. An error is thus introduced. This error will be latent until an attempt is made to read the affected file. If the representation of files is redundant, e.g. the pages are doubly linked, then the error may be detected by the system. If it is not, reading the file will produce gibberish. In either case, the system has failed to meet a specification which requires that reading a file will generate the same sequence of characters as was originally written.

The *fault* in the filing system can be repaired by replacing the disk head or the stream program. Repairing the *error* involves changing the representation of the file from its current erroneous state to one which will permit the correct operation of the system.

7.2 Fault-tolerant Computing

A computer system can be viewed as a series of compartments nested one within the next. Each compartment can be considered as a "block box" which processes information according to some specification. In a fault-tolerant system, each box does not assume that inner boxes will conform to that specification. Instead the performance of subcomponents is monitored in an attempt to detect errors, and the compartments are made "water tight" so that the effects of a catastrophe are contained.

The need to detect errors implies that there must be some redundancy in

data representations and that some redundant computation is performed. The error detection capability can be improved by increasing the redundancy, but one must be aware that this also increases the cost. In certain applications where reliability is paramount, the whole computer system has been duplicated; such systems offer an extreme example of the use of redundancy. Once an error is discovered, recovery may be attempted in either of two ways. Backward error recovery involves "backing up" the state of the system to a previous state (which one hopes is error-free), and then attempting to continue further processing. Forward error recovery attempts to make further use of the erroneous state. Both techniques rely on "watertight bulkheads" or "firewalls" to contain the error and its consequences.

7.2.1 Backward Error Recovery

In some sense, of course, all recovery must be forward, because time runs continuously in one direction. Nevertheless, because we reason about complex systems at one level of abstraction at a time, backward recovery can exist at a given level of abstraction. Forward progress at an underlying level can simulate the regress of a higher level, provided that the state of the higher level has previously been recorded and provision has been made for its reinstatement. Various techniques have been developed to do this: an extensive discussion and further references will be found in [78].

An early example of backward recovery is provided by the Algol 60 system of the Elliot 803 computer, in which the compiler remained in core while the user's program was executing. Ideally the Algol system would have sealed the user program into a watertight compartment so that the system sould be protected from even the most wayward programs. However, the hardware characteristics of the machine were such that this was not possible at a reasonable cost. Instead, the containment was simulated; when the program terminated the compiler sumchecked its own code in an attempt to detect interference. If an error was found the compiler would try to re-load

itself from magnetic tape, thus effecting recovery by restoring the system to the state which existed before the offending user program was submitted.

This example illustrates two essential points. On detecting a catastrophe the system falls back to an environment which expects much less of its subcomponents: the illegal behaviour is now considered a mere fault. For this to be possible, all the effects of the failing component which are significant to its caller must be undoable.

Most operating systems make some provision for surviving an errant user program. In a simple system designed to run student programs on a micro-computer, the physical confines of the processor may provide sufficient containment boundaries and the 're-boot' switch an adequate recovery mechanism. The experimental operating system developed at the Programming Research Group in Oxford provides a somewhat more comprehensive mechanism, but one which can be implemented economically without special hardware and is available recursively to user programs themselves.

The recovery block scheme invented as part of the reliability project at the University of Newcastle upon Tyne is a more elaborate mechanism. It provides extensive facilities for recovery after failure but at a substantially greater cost. These two examples of catastrophe handling will be presented in more detail.

7.2.2 Running Programs under an Operating System

One of the functions of an operating system is to provide the means of loading a program and initiating its execution. It is also usually expected that the system be able to restore itself to some well-defined state after the program has completed. Because the specification of the program is not known to the system, the recovery mechanism must work independently of whether or not a catastrophe occurred.

Recovery can be made to mean restoration of the system's state to exactly that which existed before the program was ever submitted. But this definition is inconveniently strong, because it means that no program can every change the filing system or the state of the main store. Some weaker definition is usually adopted, with the result that some programs occasionally produce unpleasant effects, such as deleting an important file. Other *ad hoc* recovery processes, usually requiring human intervention, may then be available, such as restoring the file from a back-up tape.
As Stoy and Strachey observed in 1972 [91], the whole concept of "program" is only necessary because of the possibility of failure. They considered a program as that part of the nest of routine activations which should be abandoned if a catastrophe is detected. This definition was developed within the framework of the Programming Research Group's experimental operating system OSPRG.* Nevertheless, it seems to capture the essential meaning of "program"; such matters as whether all of the program was loaded at one time are irrelevant. Because these concepts are clearly separated in OSPRG it provides a concrete framework within which the pragmatic business of surviving catastrophes can be discussed.

Despite its experimental nature, OSPRG is a "real" operating system. Its various incarnations have been in use in Oxford for twelve years on various hardware: the system has also been ported to other sites and other machines [89]. The complete text of a 1972 version is available as [92].

---

* The system has evolved from OS1 in 1969 to OSExp16B at the time of writing: OSPRG is a convenient name for a hypothetical system which adopts the underlying philosophy.

OSPRG takes the form of a nest of load and go loops. Somewhere near the tope of the hierarchy is a virtual machine monitor, but because this monitor itself provides facilities for the loading and obeying of programs, it cannot be said to be *at* the top. In fact, in OSPRG *any* program may load or obey any other.

The system provides a primitive routine 'Run' which takes a parameterless routine as parameter: 'Run[p]' is similar in effect to the simple call 'p[]' except that certain parts of the state are recorded before 'p' is applied, and restored after it terminates. The particular values saved and restored are the state of the free store systems for code and data, the standard input and output streams, and various values connected with the production of diagnostics when a catastrophe is detected. This choice prevents 'p' from returning as a result a structure in main store, because the restoration of the old free store state will erase it. Stoy and Strachey observe [91]:

> This is something of a disadvantage. It is possible to regard the processor and core store as an evaluating mechanism which always leaves its results as files in the backing store. But in practice one wishes to leave results in core, and the desire to do this is counterbalanced by the desire that the system should not allow any permanent changes to the core in case they turn out to be mistakes.

In practice this difficulty can usually be overcome because 'Run' does not undo all the changes to the store. Global variables which are caused to refer to routines or constants by the loader are reset when the code is unloaded, as will happen at the end of the 'Run'. However, globals which are set by explicit assignment are *not* reset at the end of the 'Run': this is partly for reasons of economy, and partly because it is useful to be

able to return results from programs.

It should be emphasised that 'Run' is not restricted to "supervisor mode", indeed there is no such concept in OSPRG. 'Run' is also fully recursive: any procedure may 'Run' any procedure, to any depth, subject only to the finite size of the machine. A user program implementing a command interpreter may 'Run' the programs indicated by its input: this in no way affects the degree of protection provided to the underlying system which in turn 'Run's the command interpreter.

The execution of a program may terminate for one of three reasons: it may itself determine that it has completed its task, it may decide that it has failed catastrophically, or it may be forcibly interrupted. The action of 'Run' is the same in all cases: the mere fact that a program completes normally is no guarantee that it has complied with its specification.

Normal termination is usually achieved simply by returning from the routine which was 'Run'. This implies that the writer of routine need not know whether it will eventually be 'Run' or simply called. It is a principle of OSPRG that, as far as the writing is concerned, a program and a routine are identical [91].

If the program detects that it has catastrophically failed, the correct thing to do is to call 'GiveUp'. The semantics of 'GiveUp' are those of abort, so that the program may assume nothing after calling it. The implementation of 'GiveUp' is rather more helpful, however.

'GiveUp' is a variable routine and may be freely altered by the programmer. The value of 'GiveUp' is preserved and reset by 'Run'. It is initialized by the system to a default value, which typically prints some diagnostic information and offers the user the option of a complete dump of the state of his machine, which may be analysed at leisure. The

final action of 'GiveUp' is to abandon all the procedure invocations of the current 'Run' by simulating a return from the top level routine, i.e. the routine provided as a parameter to 'Run'.

Many system routines call 'GiveUp' if they discover that their preconditions have been violated. For example, the routine which creates an output stream for overwriting a file requires as its precondition permission to write to that file. For safety, it checks that writing is actually permitted, and if not outputs a message and calls 'GiveUp'.

Some system primitives are implemented directly in the "hardware" of the OSPRG virtual machine. These include not only basic operations like division but some quite elaborate buffering primitives intended to speed up the operation of streams. If any virtual machine instruction discovers that its precondition is false, a virtual machine error is signalled to the interrupt routine, which in turn calls 'GiveUp'. Thus, from the user's point of view, an attempt to divide by zero is treated in exactly the same way as an attempt to write to a protected file.

If the user forcibly interrupts a program by means of the appropriate key on his terminal, a similar effect is obtained. The terminal stream causes an interrupt, which can call 'GiveUp' after some confirmatory dialogue on the terminal.

The final act of 'GiveUp', i.e. abandoning all the invocations in the current 'Run', is accomplished by calling the routine 'Finish[]'. This is the only non-local* jump available in OSPRG (apart from routine calls).

---

* Local jumps are available only through "structured" primitives like for and while: goto is never used.

It is useful because it enables users to write their own 'GiveUp' routines, but it is actually rather dangerous.

It sometimes happens that a program realises its task is completed when it is deep in routine calls. Most frequently this happens because of interaction with the user. It is tempting to call 'Finish[]' in such a situation, but this has two unfortunate effects. The first of these is that the placement of 'Run's now affects the semantics of the program. Replacing an invocation 'Prog[]' by 'Run[Prog]' ought to be an invisible action as far as the semantics of a working program are concerned. The effect will be visible only if 'Prog' fails, i.e. either aborts by calling 'GiveUp' or does something more subtle, such as omitting to close a stream which it creates. However, because this substitution alters the place to which 'Finish[]' jumps, there will be surprise effects if a procedure uses 'Finish[]' to achieve normal termination. Such a procedure is using information about its environment which it ought not to have: in this case it is the knowledge that the "firewall" created by 'Run' for the purposes of recovery is also a suitable place to which to return when the user requests clean termination.

The second unfortunate effect arises because of the clearup problem (see Section 2.2). OSPRG maintains a stack of routines and environments called the 'ClearUpChain'. Whenever an activity is initiated which requires some clearing up should a failure occur, the appropriate concluding action is entered on the 'ClearUpChain'. If the activity completes normally it removes the entry from the chain. Any entries remaining on the 'ClearUpChain' when the program terminates are removed and invoked by 'Run'.

As an example of this action of the 'ClearUpChain', consider the creation

of a file stream. Invoking the routine 'BytestoFile[f]' constructs a stream for overwriting 'f'. It places on the 'ClearUpChain' a routine 'FailClose' which deletes the partially constructed new body of 'f', leaving the old body of 'f' intact. 'FailClose' is removed from the chain if the stream is closed normally; it will only be invoked if, because of some catastrophe, the current 'Run' is terminated without the stream being closed.

Using 'Finish[]' to achieve normal termination means that the *normal* closing down actions of the program must be placed on the 'ClearUpChain'. At worst, this can lead to confusion between those actions appropriate to catastrophe and those appropriate to normal termination. At best it reduces the readability of the program, because it is more difficult to check that the appropriate final action is taken in every circumstance.

The dangers of 'Finish[]' were not fully appreciated when OSPRG was first designed, and it has become such an accepted part of the system that its withdrawal would be strongly opposed. Indeed, the philosophy of the system is to permit a user to access any routine he wishes. Nevertheless, the decision to use a system routine rather than the BCPL command finish is quite deliberate: "the meaning of the concept of finishing a program depends entirely on the particular operating system, and its *ad hoc* nature is quite out of place in the semantics of a language with an hierarchical structure" [91].

Some authors have proposed 'Exit' or 'Leave' as an alternative to 'Finish' [93]. The caller of 'Exit(procedure)' must be a dynamic descendant of 'procedure'; the effect is equivalent to a normal return from 'procedure'. Such a facility is available in the UCSD Pascal System [94]. It makes a nonsense of hierarchical structuring because a routine should not be

permitted to know that it is a dynamic descendant of another, let alone that that procedure is an appropriate one to terminate. In contrast, the *raison d'être* of 'Run' is to declare to its dynamic descendants that it is the place to which return should be made in the event of a catastrophe.

7.2.3 Recovery Blocks

Recovery blocks [52][77] are a structuring scheme which provides a means of surviving a catastrophe by confining its effects. Whereas in a general purpose operating system the specification of a program is unknown, recovery blocks require that it is made explicit.

An essential part of a recovery block is an *acceptance test*, a Boolean expression which the action of the block ought to render true. In general it will be too expensive for the acceptance test to check all of the block's postcondition, and some weaker test implied by the postcondition will be chosen. For example, the acceptance test for a sort routine might check that the array is ordered and that the sum of its elements is the same as before sorting. Such a test is quite likely to detect faults in the sorting process whose exact location and significance are unknown. Having detected such an unanticipated failure, recovery blocks provide a means of dealing with it.

The recovery block scheme is similar to the provision of "stand-by spares" in hardware engineering. Having detected that a component has failed, a spare component is switched in to take its place. Unlike the hardware analogue, this component is not a mere copy of the one which failed, but is of independent design. It is hoped that this will reduce the likelihood that the same data will cause the "spare" component to fail also. Another difference from the hardware case is that the spare software

component is invoked to deal with only the particular set of circumstances which caused the main component to fail. This is because such failures are assumed to be due to *residual* design faults, and should therefore occur only rarely.

A recovery block, therefore, consists of an acceptance test which approximates a postcondition, a normal algorithm intended to satisfy that postcondition, and zero or more spare algorithms or *alternates* which are independently capable of achieving the postcondition, although possibly in less efficient or desirable ways. The term *block* is used to refer to the structuring unit of the program design, be it module, routine, paragraph or whatever. The text of any particular recovery block may include calls on subordinate blocks in the usual hierarchical way.

When a recovery block is invoked its first action is to execute the normal algorithm. If a failure is detected during its execution, due perhaps to attempting an illegal operation or to the failure of a subordinate recovery block, then the next algorithm in the block is attempted. This is also done if, after completion of the normal algorithm, the acceptance test does not evaluate to true. However, before the alternate algorithm is invoked the state of the program is reset to that which existed when the recovery block was entered. Thus, everything that was done by the failing algorithm is discarded. If the first spare algorithm does not satisfy the acceptance test, the state is again reset and the next alternate is invoked; this process is repeated until either the test is passed or the set of alternates has been exhausted. In the latter case the recovery block as a whole fails, and further recovery is only possible at a higher level.

Central to the recovery block scheme is the resetting of the state

before invoking the alternates. Because the mechanism is designed to deal with unanticipated catastrophes,the only thing known when a failure is detected is that the state of the program is erroneous. Any assumptions made about that state are therefore likely to be wrong, and there is precious little prospect of satisfactory continuation. However, by going back to a previous state and trying again with a different algorithm there is a reasonable chance that the specification will be satisfied. Pilot studies [5] show that a high degree of recovery can be achieved. In part this is because the assumption that all the alternates start from the same state enables their designs to be independent. The designer of an alternate need not concern himself with the designs of the other alternates, or even know if they exist. He certainly should not have any responsibility for repairing the damage they may have caused.

The automatic resetting of the state is partly accomplished by a mechanism called a recursive cache. A description of the recursive cache is beyond the scope of this thesis; essentially it detects assignment to non-local variables and retains their prior values as well as their new ones. The interested reader is referred to [52] and [5]. Consideration of the additional problems introduced by the failing process communicating with other processes (be they mechanical or human) will be found in [77] and [78].

Clearly, replacing simple routine calls by recovery blocks will greatly increase the size of programs. However, it is argued that this does not imply any increase in complexity, because of the essential independence of the alternates [78]. The form of failure handling employed is actually very simple; this would seem to be an essential requirement of any scheme

designed to improve software reliability. No attempt is made by the program to determine what went wrong or why: detected failures are simply logged for later inspection by the maintenance staff.

7.2.4 Backward Recovery and Programming Languages

Neither OSPRG Runs nor Recovery Blocks are programming language features. They are means of marshalling programs into a system, and represent a level of structure above that of the programming language.

The Run mechanism of OSPRG was designed to take up where the routine mechanism of BCPL left off. In a more conventional operating system 'Run' would only be expressible in the job control language. The fact that Run is actually a BCPL routine derives from the decision that BCPL should be the job control language too.

Similarly, although recovery blocks are often illustrated in a programming language-like syntax, they are really a tool for collecting together programs to form reliable systems. This vital distinction cannot be stressed too strongly. They are intended to detect and recover from residual design errors, not as a control structure. The programmer who 'Runs' a procedure rather than calling it directly, or who replaces a simple call by a recovery block, does not do so to change the semantics of the program. His interest is to guard against the "impossible" and thus improve reliability.

7.2.5 Forward Recovery

Recall that, by definition, recovery is necessary because the state of the system is detectably inconsistent. Forward recovery is an attempt to regain a consistent state by making use of the inconsistent data. This is obviously a tricky procedure. Randall, Lee and Treleaven comment as follows [78]:

> The relative simplicity of backward error recovery is due to two facts: first, that questions of damage assessment and repair are treated quite separately from those of how to continue to provide

> the specified service; and second, that the actual damage assessment takes virtually no account of the nature of the fault involved. In forward error recovery these questions are inextricably intermingled, and the technique is to a much greater extent dependent on having identified the fault, or at least all its consequences.

A forward recovery scheme must be designed as an integral part of a system, rather than forming an essentially separate mechanism as with backward recovery. There is thus a considerable danger that the increase in the complexity in the system this generates will reduce rather than increase reliability. However, because certain environments cannot be backed-up, forward recovery is sometimes necessary. Once a missile has been launched no amount of backing-up of the control computer will cause it to return to base. Some positive instruction to the guidance system is necessary in order to disarm or destroy the missile.

7.2.6 Exception Handling as a Forward Recovery Mechanism

Having recognized that forward recovery is a difficult task, we must ask whether the presence of exception handling mechanisms in the programming language will make that task easier.

Faced with the problem of recovering from an inconsistent state, the programmer must ignore some aspects of that state and pay extra attention to others. How is he to choose which data to use and which to ignore? There are two possibilities. The first is to apply his knowledge of the failure modes of the components. For example, it may be that if the two outputs of a particular routine contradict each other, then the first should be ignored in favour of the second. However, all of the information the programmer has about his components ought to come from its specification: he must not be permitted to write a program which depends on the "secret"

behaviour of a component. Why? Because such a program is unmaintainable: replacing one component with another of identical specification but with a different "secret" will cause unpredictable changes in the behaviour of the overall system. In this particular case, the recovery algorithm will make matters worse instead of better.

If our intent is to construct a fault-tolerant system, we are thus lead to the conclusion that details of the failure modes of each component must be included in its specification. Observe, however, that once this has been done, the mode of operation under consideration is no longer a failure. On the contrary, it is part of the defined behaviour of the component. However undesirable that behaviour may be, it is now openly recognized. A programmer using the component now has both the responsibility of dealing with the undesirable behaviour and the knowledge necessary to do so.

At the start of this section I mentioned that there were two ways in which a programmer might be helped to proceed when faced with an inconsistent state. The first, just discussed, amounts to changing the specification so that the state is no longer inconsistent, i.e. so that the data the programmer possesses unambiguously tell him all that is necessary. This approach has been criticised as "defining the problem away". In my view it does just the opposite. It insists that all the details of the problem are openly admitted. This seems to be an essential first step in the process of finding a solution.

The second way in which a programmer can be given more confidence in some aspects of an inconsistent state than in others, is if in some manner *independent of his program* he is assured that some data come from a more reliable source. The recursive cache can be viewed as an example of this:

an error in an alternate may well give rise to a state which does not satisfy the acceptance test, but it is most unlikely to corrupt the data in the cache. This is because the cache is not administered by the programmer, but is part of the underlying abstract machine on which the program runs.

In backward recovery, the data in the cache is used to rollback the state *en masse*. Even if this is not possible, say because some irreversible action has been performed, it is easy to envision the cached data being used to determine which aspects of the current (inconsistent) state are correct and which are incorrect. Notice, however, that before it is reasonable to use data in this way, one must have a very high degree of faith in their correctness.

At the point where a failure is discovered, one has little or no faith in the correctness of data maintained by the program's code: it has, after all, just failed. However, data maintained by an underlying level of abstraction is unaffected by programming errors at the higher level. Its correctness is therefore not suspect.

This is the essence of the argument that some form of protection mechanism is necessary to perform recovery, and that such a mechanism should *not* be part of the programming language. An exception handling mechanism embedded in the programming language is irrelevant to the business of recovering from failure. Exception handling mechanism can assist in the construction of components which behave in a more complicated way instead of failing; as was shown in Chapter 4, other language mechanisms can do this too.

Another way of exhibiting the difference between recovery and exception handling is through the observation that recovery is appropriate when an unanticipated error occurs. This may be a residual design error that was not discovered when the program was specified and constructed, or it may be due to a hardware failure, the precise characteristics of which must be unknown until it occurs. In contrast, exception handling involves predicting faults and their consequences *before they happen*, i.e. when the program is written. Thus, while it may be useful for dealing with anticipated unsatisfactory behaviour such as the provision of invalid input, it cannot be used to provide design fault-tolerance. This rather traditional view is well expressed in [77]:

> The variety of undetected errors which could have been made in the design of a non-trivial software component is essentially infinite. Due to the complexity of the component, the relationship between any such error and its effect at run time may be very obscure. For these reasons we believe that diagnosis of the original cause of software errors should be left to humans to do, and should be done in comparative leisure. Therefore our scheme for software fault tolerance in no way depends on automated diagnosis of the cause of the error - this would surely result only in greatly increasing the complexity and therefore the error proneness of the system.

In fact, exception handling and recovery, whether forward or backward, are distinct techniques designed for dealing with different problems. The following example is designed to bring out this difference; it is based on the appendix to [71]. The basic form of the program is the recovery block:

```
ensure consistent_inventory
by process_updates
elseby refuse_updates
else fail .
```

The program is assumed to manipulate an inventory file whose consistency must be maintained. The procedure 'process_updates' invokes another procedure 'checknum' to read and check the updates. 'Checknum' assumes

```
proc process_updates is
    var num:integer
    ...
    while updates_remain
    do update_number := update_number + 1
        ...
        num := checknum()
        ...
    end do
    ...
end of process_updates

proc checknum returns j:integer is
    var count:integer = 0
    proc message is
        count := count + 1
        if count < 3
        then write('please try again')
             retry
        else write ('three input errors: update abandoned')
             fail
        fi
    end of message
    ...
    read(j) [ overflow, underflow, conversion: message()
              ioerror:  fail ]
end of checknum

proc refuse_updates is
    write('sorry - last update accepted was number')
    write(update_number)
end of refuse_updates
```

Figure 7.01: Complementary use of exception handling and recovery blocks.

the existence of an exception handling mechanism and a procedure 'read' which is used to obtain input and may generate exceptions.

In Figure 7.01 explicit exception handling is used to allow the user three attempts to produce the correct input. The statement retry specifies that the statement which raised the exception should be executed again. fail specifies that the current alternate of the recovery block has failed; the recovery mechanism will then reset the state and attempt the next alternate. In this example the fail in 'message' will cause the abandonment of 'process_updates' and the invocation of 'refuse_updates'. The same thing will happen if 'process_updates' fails to pass the acceptance test *for any reason*, including a residual design or programming error in its code, or in the implementation of the exception handling mechanism.

'Checknum' is actually a slightly simplified version of a routine of Wasserman's [99]. The simplification is possible because the recovery block structure will take care of clearing-up. Wasserman introduces 'checknum' not just as an exercise for his exception handling mechanism but specifically to justify the inclusion of the retry primitive. It is therefore instructive to program 'checknum' using neither facility. Figure 8.02 illustrates the result of this exercise.

The version of 'Checknum' in Figure 7.02 uses (tail) recursion instead of retry. It assumes that instead of generating exceptions, 'read(j, r)' uses two output parameters; 'r' is an enumeration value indicating the fate of the read, and 'j' is the result as before.

```
proc checknum returns j:integer is
    var count:integer = 0
    ...
    read_j

where read_j is
    var r:oneof( readOK:singleton
               ; overflow:singleton
               ; underflow:singleton
               ; conversion:singleton
               ; ioerror:singleton
               )
    read(j, r)
    if  Is_readOK(r) => skip
    []  Is_ioerror(r) =>fail
    []  Is_overflow(r) or  Is_underflow(r) or
              Is_conversion(r) => message
    fi

where message is
    count := count + 1
    if  count < 3 =>
            write('Please try again')
            read_j
    []  count ≥ 3 =>
            write ('three input errors: update abandoned')
            fail
    fi
end of checknum
```

Figure 7.02: checknum without exception handling

This example is trivial compared to many of those in Chapter 4; it does however serve to illustrate the difference between exception handling and recovery. Exception handling is just another language feature, and one whose function is subsumed by procedures, arguments and results. It is thus essentially irrelevant to the problem of providing fault-tolerance in software, the problem which recovery blocks were designed to solve. Any reasonable response to residual design errors must avoid attempting automatic diagnosis of the fault, "a task whose complexity is such as to

be productive of design faults, rather than conducive to the design of reliable systems" [78].

7.3 Using a Recovery Mechanism for Exception Handling

I have argued above that exception handling is inappropriate as a recovery mechanism. But what of the converse: can a recovery mechanism be used for exception handling? In other words, can a recovery mechanism be used to deal with events which are not failures (even though they may be unusual in some subjective sense)?

The answer to this question is "yes", and the availability of 'Finish[]' in OSPRG (see Section 7.2.2) demonstrates one possibility. It is also conceivable that recovery blocks could be used to implement backtrack programming [9] [90]. A more interesting question is whether one *should* use a recovery mechanism in this way.

The argument in favour of such a use is simply stated. The recovery mechanism exists, and happens to help the programmer achieve the desired effect. Clearly it should be used to do so, rather than constructing a second device to do the same thing.

The argument against can be generalized from the above discussion of 'Finish'. Readability is impaired because one can no longer be sure that invocation of the recovery mechanism implies the occurrence of a catastrophe. More importantly, the use of the recovery mechanism for its original purpose is jeopordized.

My own experience in this area is admittedly limited. Although I must admit to using 'Finish', I may at least claim that I always felt guilty about doing so - without really knowing why. I could not ascribe

this feeling simply to a dislike for non-local jumps, because I had no reservations over the use of 'GiveUp'. One of the benefits of writing this chapter has been the realization that 'Finish' and 'GiveUp' are not symmetrical primitives, and that the placement of 'Run's to provide optimal recovery after a 'GiveUp' may not correspond to the placement required by 'Finish'. It is then tempting, and highly dangerous, to adjust the 'Run' structure to deal correctly with calls of 'Finish' and to let 'GiveUp' take care of itself.

The danger is illustrated by an analogy due to C.A.R. Hoare. The installation of a fire alarm system involves the fixing of loud bells so that all the occupants of the building can hear them. It is obviously convenient to use such bells to indicate the presence of callers at the front door. A short ring could represent a visitor, and a continuous ring a fire.

Economical as this scheme may be, there are at least two reasons why it is prohibited. The first is that the continuous ringing of the fire alarm might be attributed to a small boy putting chewing-gum into the bell push. The second is that the professor, tired of being continuously interrupted by tradespeople, will contrive to disconnect his alarm bell.

The programming notation 3R [11] [2] provides an example of the consequences of misusing a catastrophe mechanism. 3R is designed for the production of publishable programs [87]. It is a minimal language, but originally included the notion that any statement could fail. A basic operation such as assignment or multiplication might fail because of an implementation malfunction or insufficiency. A guarded command [22] would fail if no guard was true; to emphasise this the closing bracket was represented as otherwise fail rather than fi. An invocation would fail if

any command in it failed.

In order to report on and perhaps clear-up after such catastrophes, the concept of a tested invocation was introduced. Whereas the simple invocation 'routine' would cause the invoking block to fail if 'routine' itself failed, the tested invocation test 'routine' would not. Instead it was possible to examine the state of the computation to see whether 'routine' had failed or not, and to act on the result.

Because no state restoration was performed by test, trying to continue after a failure was very risky, or indeed impossible in the case of a machine failure. Attempts to axiomatize fail and test demonstrated this very clearly. However, it had meanwhile been observed in practice that machines fail rarely, and that deliberate use of the test and fail commands to simulate a boolean result could sometimes prove to be very convenient. By the time I formulated a rigorous definition of 3R [11], fail had become just another control construct, completely stripped of any connotation of catastrophe. If a programmer was using fail to communicate essential results, it would have been disastrous if a faulty guard might cause the failure flag to be set unexpectedly. The semantics of the guarded command were therefore changed so that the failure flag was not set. To reflect this the closing bracket of the alternative construct was rendered as otherwise chaos; I hoped this would better convey the consequences of failing to ensure that at least one of the guards was true. Chaos is the state in which false is true, in other words the catastrophe state defined at the beginning of this chapter.

Since this time 3R has been used for further projects by Euro Computer Systems. They have concluded that, while sometimes very convenient, fail

has not earned itself a place in the language. Indeed, it has not been used in any of the modules written recently. In summary, one sees that what was originally intended as a catastrophe mechanism was modified until it became a primitive exception handling mechanism. As such, it was found to be insufficiently useful and has been abandoned. In practice, it seems that attempts to misuse catastrophe mechanisms for other purposes are likely to fail, or even lead to chaos.

Chapter 8

CONCLUSION

In the preceding chapters of this thesis I have argued that exception handling mechanisms are both unnecessary and undesirable. I have also argued that catastrophes, i.e. the totally unexpected, cannot be "handled" but must be survived, and that a catastrophe is a very different concept from an exception. It is appropriate here to summarize these arguments.

8.1 Exception Handling is Unnecessary

Recently the belief has grown up that some form of exception mechanism is an essential part of both specification and programming notations. Some programmers have been led to believe that the awesome responsibility of dealing with all possible cases could be lifted from their shoulders by adding a construct to their programming language - if only the right construct could be found.

I do not share this belief, and one of the principal motivations for this thesis was my desire to point out both that abstract specifications can be written without abstract errors (see Chapter 5), and that a programming language exception mechanism neither prevents the construction of erroneous programs nor provides a way of dealing with errors that absolves the programmer from thinking about them.

Nevertheless, because the popular press and the authors of at least one programming language (see Section 1.5) seem to be confused as to the capabilities of exception handling mechanisms, let us examine the various kinds of error that may occur during the construction of a program, and see whether exception handling mechanisms are relevant to their correction.

I believe, as Dijkstra has so aptly put it [23], that "programming is a goal directed activity". I am convinced that one should decide what one is trying to do before setting out to do it, or in more technical terms, that specification should precede implementation.

Once given a specification, the programmer is responsible for implementing it. Of course, there are specifications which are impossible to implement.

One hopes that designers will not generate such impossible specifications frequently, and that programmers will tell specifiers if they have erred. Indeed, it is one of the functions of an engineer, whether of hardware or software, to recognize the constraints of Physics and Mathematics. If a civil engineer is asked to build a railway bridge across the First of Forth out of bamboo, he should feel free to say that it is impossible. A software engineer should say the same about a request for a square root program applicable to all real numbers. Recognizing an impossible specification has nothing to do with generating an exception - it happens before or during the construction of the program, not while it is running.

When the program is completed and delivered, it should comply with its specification: I do not believe that this should be exceptional. If it does not, it is because the programmer has made an error, such as overlooking the consequences of an action or making an unjustified assumption. Such errors are also not exception occurrences: they should be corrected before the program is delivered, not handled while it is being run.

Finally, if the program is correct but is invoked from a state which does not satisfy its specified precondition, what should happen? Anything at all - for that too is an (implicit) part of the specification.

The ability to leave the action of a component unspecified when it is invoked from an illegal state is one of the most useful aspects of a specification: it allows the implementor the maximum freedom to optimise and to use existing components. If a program is used without first achieving its precondition, we may deduce that the very reasoning on which the invoking program was based is faulty. By the same argument as above, that is not an exception; it is a programming error. However, because such errors do occur, it is often appropriate for a program to be suspicious of its inputs and to provide an alarm if it detects that they are not as specified. Programs which are written in this way are said to be robust.

It seems necessary to require the specification of a program to define the conditions (if any) under which exceptions will be generated. If exceptions are not so specified, an invoking program could never be prepared to handle them and their existence would be pointless. So an alarm from a robust program reporting illegal inputs is also not an exception, for such an alarm is by definition not part of its specified behaviour. In any case, "raising an exception" is hardly a suitable alarm: there is no ground for believing that the reasoning behind the invoker's "exception handling" is any less faulty than that on which the original invocation was based.

From this reasoning it follows that exceptions have nothing to do with "errors" or "failures"; they are relevant only when a program complies with its specification.

The fact remains that exception handling mechanisms have been proposed and implemented, and we may therefore ask what facility they add to a programming language. The answer is that they are a new control structure, in some languages carefully restricted in application, and in others so general as to replace the goto. Exception handling mechanisms are unnecessary because

they can always be replaced by union results, parametric procedures and partial operations (see Chapters 4, 5 and 6). The question remains whether an exception handling mechanism is a desirable addition to a modern programming language.

8.2 Exception Handling is Undesirable

Programming languages are not just arbitrary notations. If they were, then it would indeed be advisable to choose that with the largest and richest sponsor or the fastest compiler [45]. But in fact the choice of notation has an important influence on the way we reason. The notation of mathematics is an important aid to logical reasoning: equations can be manipulated symbolically with an ease and rapidity that would be quite impossible if their meaning had to be considered at every step.

Similarly, a programming language should be constructed with the aim of making programs written in it easy to understand and reason about. Simplicity is one desirable property of such a language. Thus great care should be taken before any nonessential feature is included.

Exception handling is just such a nonessential feature. It is nothing more nor less than an additional control structure for programming languages. Before adding any such structure it should be encumbent on language designers to ensure that it encourages clear, structured and modular programming.

The reason that exception handling mechanisms are undesirable is that, whereas they are irrelevant to the correction of erroneous programs, the difficulties they introduce are very real. Rather than making it easier to write reliable programs, they make it more difficult. It is instructive to look at some of the problems some exception handling mechanisms have introduced.

(i) They permit non-local transfers of control, re-introducing the dangers of the goto and the problem of clearing-up.

(ii) They can cause parallel processes to interact other than through the normal communication channels. This greatly complicates reasoning about the co-operation of those processes.

(iii) If "functions" are allowed to generate exceptions, a rift is introduced between the concepts of "function" in the programming language and in mathematics. This complicates the semantics of the language and makes reasoning about programs more difficult.

(iv) Recursive exception handling is extremely complex. If an exception occurs while another exception is being handled, an arbitrary decision must be made as to which has precedence. The problem arises because the ultimate action of a handler is often to perform a non-local transfer of control while transmitting some result. Two conflicting transfers or result values produced by recursive application of the exception handling mechanism cannot both be honoured.

(v) An exception handling mechanism may be used to deliver information to a level of abstraction where it ought not to be available, violating

the principle of information hiding.

(vi) If a list of possible exceptions is incorporated into the syntactic interface of a procedure, that interface is thereby complicated, and interfaces are the very part of a program which one wishes to keep as small and simple as possible. On the other hand, if exceptions are not so specified then the dangers of using them are increased, and many of the advantages of strong typing are lost.

To be harmless, an exception handling mechanism must be constrained to avoid these problems. In fact, rather than doing so, many proposals for exception handling suffer from excessive generality: they are powerful enough to subsume procedures, parameters, results, coroutines, gotos, dynamic binding and abort. The CLU exception handling mechanism is one of the most constrained. It completely avoids problems (i), (ii) and (v) by confining itself to one level of procedure call, resolves (iv) by jumping and delivering results *before* invoking the handler, leaves resolution of (iii) to the programmer, and on point (vi) decides that a complicated specification is better than none at all. This should not be taken as a criticism of CLU; on the contrary, CLU's exception mechanism is markedly superior to most others. For example, the Mesa unwind primitive makes problem (iv) doubly serious, and the absence of any mention of exceptions at procedure interfaces is a throwback to the days before type checking. The Ada mechanism is objectionable on all of the above counts and on several others.

It seems clear, then, that existing exception handling mechanisms are not a desirable addition to a programming language because of their interaction with other, more necessary, language features. But how can we be sure that this implies that exception handling mechanisms are bad *per se*, rather than merely that existing mechanisms are insufficiently constrained?

Without a definition of exception it is, of course, quite impossible to show that a mechanism for handling exceptions can never be useful. I myself have argued that *catastrophe* handling mechanisms can be very useful, so extending the meaning of "exception" to include "catastrophe" would immediately provide an example of useful exception handling. Nevertheless, if one restricts exception handling to mean something similar to the action of the mechanisms described in Chapter 3, then a mechanism which avoids the problems listed above seems to be necessarily somewhat like that of CLU. Such an exception mechanism must provide a way of dealing with "unusual" results. If it is constrained to deal with a single procedure invocation then there is no gain in brevity or clarity over the use of results values which may be oneof a number of different types. On the other hand, if the mechanism may apply a single handler to several invocations, one achieves brevity but is simultaneously assailed with the dangers of defaults. It is far too easy to omit a necessary handler and to have an exception caught by an inappropriate handler from a surrounding scope. This was clearly illustrated by the example of Section 4.1.

Now that the dangers of large languages with excessively general features are becoming appreciated, one is entitled to ask why exception handling mechanisms were ever proposed. The motivation seems to have been twofold. One aspect of the case was a desire to make available the "full

power of the machine" to the high level language programmer. When PL/I was designed it was observed that the hardware provided "traps" for arithmetic underflow, and ON conditions were included as a way of allowing the programmer to use them. Extending ON conditions to deal also with conditions detected by the run-time system was an attempt to create a homogeneous interface between the language and the implementation by modelling the former on the latter.

Since the 1960's our attitude towards computers has matured. Instead of a programming language being visualised as a tool for controlling a computer, the computer is considered as a tool for implementing a language. Language features are now assessed as much on the help they provide to the programmer as on the ease and efficiency with which they may be implemented. This is not to say that efficiency of implementation is unimportant, for however fast and cheap computers become it will always be faster and cheaper to use them efficiently [45]. But efficiency of the object code is not the only consideration: if the costs of programming are increased or the reliability of programs reduced, then the saving of machine cycles has been counterproductive.

An encouraging example of the trend towards simple, programmer-oriented languages is provided by Edison [40].

The variant of the Edison language for PDP-11 computers, Edison-11, is implemented by ignoring interrupts completely, even at the machine level. The only synchronizing primitive is a form of the conditional critical region. Input and output are controlled by explicitly testing the state of the appropriate registers in the device. If a process is suspended

because a device is not ready it must wait until the other processes (if any) terminate or are suspended, and then test the device registers again. This simple scheduling scheme reduces the overhead of process switching to one-fifth of that of an interrupt drive concurrent Pascal implementation [39].

Now consider the main loop of an Edison program for backing-up a disk to magnetic tape described in [41].

```
while more_disk do
    i := not i
    cobegin 1 do write_tape(x[not i])
    also 2 do read_disk(x[i])
    end
end
```

The cobegin statement denotes concurrent execution. The disk is read and the tape is written simultaneously, but no communication is necessary because the two processes operate on disjoint buffers. Since 'i' and 'not i' must be different, and 'i' is only changed when execution is purely sequential, the integrity of the program is guaranteed. The simplicity is achieved by re-creating the reading and writing processes for each cylinder. This is feasible because the simple form of concurrency used in Edison makes process creation and termination very cheap. When backing up a disk on the PDP 11/55, re-creating these processes imposes an overhead of less than 0.06 per cent.

The lesson to be drawn from this illustration is that trying to make use of the "full power of the machine" (in this case interrupts) not only makes programming more difficult by complicating the language but is often quite unnecessary: efficiency can also arise from simplicity.

The second consideration which seems to have motivated exception handling is a desire to write the code for the normal case first and to avoid cluttering it with tests for less desirable cases. Considering these other cases later permits concerns to be separated in a commendable way. However, the same separation can be achieved by writing the normal case code as a procedure which is called only when tests have established that the case really is normal. This method has the additional benefit of making the conditions for normality explicit.

A related motivation is that checking the results of an operation to see if a particular event has occurred can be a chore for the programmer. This is quite true. However, if the action to be taken on discovering the "unusual" event is the same for several invocations, then procedures provide a suitable means of abbreviation. On the other hand, if different actions are required at each invocation then setting up a different exception handler at each place is even more of a chore than writing the equivalent *if* statements.

The associated complaint that testing is a serious source of inefficiency [88] seems to be vacuous. Since the result of the operation will be on the stack, comparing it with a constant and branching on the result will take very few operations - far fewer, in fact, than the overhead imposed on procedure calls by the implementation of most exception handling mechanisms. One implementation which does not place an overhead on calls is that chosen for the Mesa mechanism; if exceptions are never generated the only extra costs at run time are increased code size and a single branch instruction per handler definition. As a consequence, generating an exception in Mesa is fairly expensive: indicating an unusual event with a signal costs about four hundred times as much as using a result [26].

It is nevertheless true that certain conditions may be known to occur so rarely that there may be an overall saving of execution time in spite of this ratio. This is even more true with other implementations. As was shown in Chapter 4, the restricted nature of the CLU mechanism makes it implementable with little overhead, and a few instructions may be saved even when frequencies differ by a much smaller ratio. It is my view, however, that while this sort of argument may have been an appropriate motivation for the FREQUENCY statement of Fortran II, it should not be allowed to influence the design of a modern programming language. Methodological considerations are far more important.

8.3 Exceptions and Catastrophes

In contrast to exceptions, catastrophe is a well-defined abstract concept. One of the reasons that exception handling is not just undesirable but positively dangerous is that it encourages the belief that catastrophes and exceptions differ only in degree, whereas in fact they differ in kind.

A catastrophe occurs when some component fails to meet its specification. Even if all software components are proved correct, the possibility that the hardware may fail is ever present: catastrophes will occur.

Catastrophes differ from exceptions in that they are totally unexpected. The accepted view of an exception is as an indication that some event has occurred. Whether or not it is a desirable event is unimportant: what is significant is that the programmer was aware that it might happen, was able *a priori* to fully understand the implications of its occurrence, and has coded a handler which makes some well-specified response. In contrast, a catastrophe indicates that the impossible has happened. The

programmer had no way of even knowing that this particular impossibility might occur, and certainly did not understand all its implications. Because the symptoms of a catastrophe are likely to be totally unrelated to its cause, it is more appropriate to speak of *surviving* a catastrophe than *recovering* from it. Because the effects of a catastrophe may be widespread and unquantifiable, the only hope of survival lies in containment. Catastrophe handling involves dividing a system into secure compartments protected from each other by "firewalls" which one hopes will be proof against catastrophes. Firewalls are likely to be strongest when they are directly supported by the underlying hardware. For example, in a multiprocessor configuration in which each processor has its own physical memory, a catastrophe in one processor is unlikely to corrupt the state of the others. However, if they share a store segmented by software or even firmware, the possibility of cross-contamination is increased.

It is also important to remember that there is a cost associated with establishing a "firewall". In the case of the 'Run' structure of OSPRG the cost is small, but the degree of protection provided is correspondingly limited. Recovery Blocks provide much better containment but at a greatly increased cost. Because of this cost it is quite inappropriate to set up firewalls at each abstraction boundary. Whereas exceptions are supposed to be trapped as close to their source as possible so that there is sufficient context for their correct analysis, the handlers for catastrophes may be quite widely spaced. No automatic analysis of the cause of the catastrophe is even attempted: from the other side of the firewall only survival is important. This may take the form of an attempt to re-create the failed component or a continuation without it.

8.4 On Subjectivity and Proof

A potential objection to this thesis is that it is nothing but an expression of my own opinion. That is an oversimplification. While I have not hesitated to express my opinion where it seems relevant, I have also presented the evidence on which it is based. Part of that evidence is the experience of other programmers with exception handling, and part is the series of comparative examples.

It is pertinent to ask how my argument could be made more objective and rigorous. The thesis that the occurrence of unusual or undesirable events can be dealt with in the same way as any other programming problem is not susceptible to formal proof. Any decision about what should and should not be included in a programming language must be subjective because the language will be written and read by people. Rejecting all subjective assessment means that any language is as good as any other, only providing they are all universal (in the sense of a universal Turing machine). Those who believe this will not be convinced by my arguments; indeed, they will probably not have read this thesis. On the other hand, my argument can be disproved by a single counter-example; this thesis could be interpreted as a challenge to the proponents of exception handling to produce an example of a well-handled exception.

The qualities of a convincing proof vary from one reader to another. Even in a mathematical proof it is necessary to make assumptions about the size of the step that a reader will be able to understand: except in the most trivial theorems, resorting to proof theory at each stage would be impossibly tedious for the theorem proover and would render the proof incomprehensible to the reader. The same problem faced me when writing this thesis. Some people may have been convinced of my hypothesis

before they read this thesis, while others may find the leaps in intuition required by my argument are too large. I hope I have made a reasonable compromise, and that even those who are not convinced by my arguments may at least be prompted to ask for substantiation of the often unsupported counter-claim that exception handling mechanisms must be available in the programming language.

8.5 Suggestions for Further Work

It is conventional for a doctoral thesis which has proposed some new idea to outline areas where further investigation is required and directions in which extension might be possible. This thesis, by *disposing* of a language feature, disposes also with all need for smoothing its rough edges. What further work is appropriate to substantiate this thesis?

The most obvious suggestion is that the experiment of writing programs without exception handling mechanisms should be conducted as widely as possible. Of course, this "experiment" has been in progress for many years, and the results to date have been encouraging. Many readable programs have been written, and some even published, which cope with diverse behaviours without using exception handling mechanisms. I hope that language designers encourage the continuation of this experiment by excluding exception handling mechanisms from their products, and devoting their efforts to other issues.

Two language design issues which *are* raised by this research are the availability of data type unions and procedure-valued variables. It certainly seems to be true that unions have not been carefully designed in many existing languages. As was shown in Chapter 5, there is no difficulty in determining the semantics of a disjoint union. However, there are engineering problems in integrating such a facility with the rest of the

language. The conventional wisdom is that in order to achieve type security at an acceptable cost an additional syntactic structure like the CLU tagcase statement is required. Whether the implementation techniques suggested in Chapter 6 make this unnecessary is an open question. Whether or not injecting to (or projecting from) a union should ever be implicit operations is a decision which can only be made in the context of a particular language. These questions can only be resolved with experience of the use of oneof types for representing different kinds of result.

Treating procedures as manipulable values seems to be harmless enough in isolation. However, when combined with nested procedure declarations, block structure and free variables, problems are posed for the compiler writer. Some languages have eliminated non-global free variables in order to avoid these problems; in others procedure parameters and procedure assignment have been omitted. With the development of module structures which provide for explict exposure of names, it may be that nested procedure declarations and block structure are the least useful facility. Or perhaps all these features are sufficiently important to warrant the small increase in implementation complexity required to accommodate them. It is even possible that as implementations of applicative languages become more efficient assignment itself may become an error-prone anachronism.

The trend towards simpler languages presents their designers with both a challenge and an opportunity. In a baroque design each irregularity is camouflaged by many others. In a small, compact language every facility must be manifestly worth its place; any special cases are plainly visible. The design of a programming notation which is so simple and natural that its semantics are obvious on inspection provides opportunities enough for the exercise of ingenuity and skill. There is no need to search for ever more sophisticated constructs which will inevitably perplex rather than enlighten.

REFERENCES

[1] Adams, J.M. and Black, A.P. On Proof Rules for Monitors. Operating Systems Review Vol. 16 Nr. 2 (April 1982) pp. 18-27.

[2] Alcock, D.G. Readability of Design Programs. Proc. Colloq. Interface between Comput. and Design in Structural Engineering, Bergamo (Sept. 1978) pp. III.1-III.10.

[3] American National Standards Institute. Standard FORTRAN X3.9-1966.

[4] American National Standards Institute. Programming Language PL/I X3.53-1976.

[5] Anderson, T. and Kerr, R. Recovery Blocks in Action: a system supporting high reliability. Proc. Int. Conf. Softw. Eng., San Francisco, October 1976. IEEE and ACM.

[6] Andrews, G.R. Parallel Programs: Proofs, Principles and Practice. Commun. ACM Vol.24 Nr.3 (March 81), pp. 140-146.

[7] Atkinson, R. and Liskov, B.H. Aspects of Implementing CLU. Proc. ACM Annual Conf. 1978, pp. 123-129.

[8] Best, E. and Cristian, F. Systematic Detection of Exception Occurrences. Technical Report, University of Newcastle upon Tyne (Feb. 1981).

[9] Bitner, J.R. and Reingold, E.M. Backtrack Programming Techniques. Commun. ACM Vol.18 Nr.11 (Nov. 1975) pp. 651-656.

[10] Black, A.P. Exception Handling and Data Abstraction. Technical Report RC 8059, IBM T.J. Watson Research Center, Yorktown Heights, New York. (Jan. 1980).

[11] Black, A.P. Report on the Programming Notation 3R. Technical Monograph PRG-17, Oxford University Computing Laboratory. (Aug. 1980)

[12] Boehm, H., Demers, A. and Donahue, J. An Informal description of Russell. TR 80-430, Cornell University (October 1980).

[12a] Bron, C. and Fokkinga, M.M. Exchanging Robustness of a program for a Relaxation of its Specification. Department of Applied Math, Twente University of Technology, Memo Nr. 178, September 1977.

[13] Bustard, D.W. A user manual for PascalPlus. Department of Computer Science, Queen's University Belfast (April 1978).

[14] Corbató, F.J. and Vyssotsky, V.A. Introduction and Overview of Multics. Proc. AFIPS Fall Joint Comp. Conf. 1965, pp. 185-196.

[15] Courtois, P.J., Heymans, F. and Parnas, D.L. Concurrent Control with "Readers" and "Writers". Commun. ACM Vol. 14 Nr. 10 (Oct. 1971) pp. 667-668.

[16] Cristian, F. Exception Handling and Software Fault-tolerance. Proc. 10th Int. Symp. on Fault Tolerant Computing, Kyoto, Japan. IEEE 1980.

[17] Demers, A. and Donahue, J. The Russell Semantics: an Exercise in Abstract Data Types. TR80-431, Cornell University (September 1980).

[18] Demers, A., Donahue, J. and Skinner, G. Data Types as Values: Polymorphism, Type-checking, Encapsulation. Conf. Rec. Fifth ACM Symp. Principles Prog. Lang. (January 1978), pp. 23-30.

[19] DeMorgan, R.M., Hill, I.D. and Wichmann, B.A. Modified Report on Algol 60. Comp. J. Vol. 19 Nr. 4 (Nov. 1976), pp. 364-379.

[20] Dijkstra, E.W. Goto Statement Considered Harmful. Commun. ACM Vol. 11 Nr. 3 (March 1968).

[21] Dijkstra, E.W. Notes on Structured Programming. Structured Programming. Academic Press, 1972.

[22] Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM Vol. 18 Nr. 8 (August 1975), pp. 453-457.

[23] Dijkstra, E.W. A Discipline of Programming. Prentice Hall, 1977.

[24] Donahue, J. On the semantics of "Data Type". SIAM J. Comput. Vol. 8 Nr. 4 (November 1979), pp. 546-560.

[25] Geschke, C.M., Morris, J.H. and Satterthwaite, E.H. Early Experiences with Mesa. Commun. ACM, Vol. 20 Nr. 8 (Aug. 1977), pp. 540-553.

[26] Geschke, C.M. and Satterthwaite, E.H. Exception Handling in Mesa. Technical Report, Xerox Palo Alto Research Center. (Draft, Sept. 1977)

[27] Goguen, J.A. Abstract errors for abstract data types. Formal Description of Programming Concepts. (Neuhold, E.J., Ed.) North Holland (1978), pp. 491-525.

[28] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness and implementation of abstract data types. Current Trends in Programming Methodology Volume IV: Data Structuring. (Yeh, R.T., Ed.) Prentice Hall (1978), pp. 80-149.

[29] Goodenough, J.B. Structured Exception Handling. Conference Record of the 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, California. (January 1975), pp. 204-224.

[30] Goodenough, J.B. Exception Handling: Issues and a proposed notation. Commun. ACM Vol. 18 Nr. 12 (Dec. 1975), pp. 683-696.

[31] Gray, J. et al. The Recovery Manager of the System R Database Manager. Comput. Surv. Vol. 13 Nr. 2 (June 1981), pp. 223-242.

[32] Gries, D. Some Comments on Programming Language Design. Invited lecture, Fachtagung über Programmiersprachen, Erlangen, Germany (Marcy 1976).

[33] Gries, D. and Gehani, N. Some Ideas on Data Types in High-level Languages. Commun. ACM Vol. 20 Nr. 6 (June 1977), pp. 414-420.

[34] Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. Thesis - Computer Systems Research Group Report CSRG-59. Department of Computer Science, University of Toronto. (September 1975), 154 pp.

[35] Guttag, J.V. Abstract Data Types and the Development of Data Structures. Commun. ACM Vol. 20 Nr. 6 (June 1977), pp. 396-404.

[36] Guttag, J.V., Horowitz, E., Musser, D.R. Abstract data types and software validation. ISI/RR-76-48 University of Southern California Information Sciences Institute. (August 1976), 45 pp.

[37] Guttag, J.V., Horowitz, E., Musser, D.R. The design of data type specifications. Current Trends in Programming Methodology Volume IV: Data Structuring. (Yeh, R.T., Ed.) Prentice Hall (1978), pp. 60-79.

[38] Guttag, J.V., Horowitz, E., Musser, D.R. Some extensions to algebraic specifications. Proc. ACM Conf. Language Design for Reliable Softw. (Wortman, D.B., Ed.) North Carolina (March 1977), pp. 63-67.

[39] Hansen, P.B. The Design of Edison. Softw. Pract. Exper. Vol. 11 Nr. 4 (April 1981), pp. 363-396.

[40] Hansen, P.B. Edison - a Multiprocessor language. Softw. Pract. Exper. Vol. 11 Nr. 4 (April 1981), pp. 325-361.

[41] Hansen, P.B. Edison Programs. Softw. Pract. Exper. Vol. 11 Nr. 4 (April 1981), pp. 397-414.

[42] Hehner, E.R.C. do considered od: a Contribution to the Programming Calculus. Acta Informatica Vol. 11 Fasc 4 (April 1979), pp. 287-304.

[43] Henderson, P. Functional Programming: Application and Implementation. Prentice Hall Int. (1980), 355 pp.

[44] Hill, I.D. Faults in Functions, in Algol and Fortran. Comp. J. Vol. 14 Nr. 3 (August 1971), pp. 315-316.

[45] Hoare, C.A.R. Hints on Programming Language Design. Invited address at SIGACT/SIGPLAN Symp. Principles Prog. Lang. (Oct. 1973).

[46] Hoare, C.A.R. Monitors: an Operating System Structuring Concept. Commun. ACM Vol. 17 Nr. 10 (Oct. 1974), pp. 549-557.

[47] Hoare, C.A.R. Communicating Sequential Processes. Commun. ACM Vol. 21 Nr. 8 (Aug. 1978), pp. 666-677.

[48] Hoare, C.A.R. A Model for Communicating Sequential Processes. Technical Monograph PRG-23, Oxford University Computing Laboratory. (1981).

[49] Hoare, C.A.R., Brooks, S.D. and Roscoe, A.W. A Theory of Communicating Sequential Processes. Technical Monograph PRG-16, Oxford University Computing Laboratory. (May 1981).

[50] Hoare, C.A.R. and Wirth, N. An Axiomatic definition of the Programming Language Pascal. Acta Inf. Vol. 2 Nr. 4 (December 1973), pp. 335-355.

[51] Horning, J.J. Language features for Fault Tolerance. Program Construction (Eds. Bauer and Broy). Lecture Notes in Computer Science, Vol. 69. Springer-Verlag 1979, pp. 508-516.

[52] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M. and Randell, B. A Program Structure for Error Detection and Recovery. Operating Systems, Lecture Notes in Computer Science Vol. 16, pp. 171-187. Springer-Verlag 1974.

[53] IBM Corporation. LISP/370 Program Description/Operations Manual. Form SH20-2076-0 (March 1978).

[54] Ichbiah, J.D. et al. Preliminary Ada Reference Manual. SIGPLAN Notices Vol. 14 Nr. 6 Part A (June 1979).

[55] Ichbiah, J.D. et al. Rationale for the design of the Ada programming language. SIGPLAN Notices Vol. 14 Nr. 6 Part B (June 1979).

[56] Kessels, J.L.W. An alternative to event queues for synchronization in Monitors. Commun. ACM Vol. 20 Nr. 7 (July 1977), pp. 500-503.

[57] Kohler, W.H. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. Comput. Surv. Vol. 13 Nr. 2 (June 1981), pp. 149-183.

[58] Kuo, S.S., Linck, M.H. and Saadat, S. A Guide to Communicating Sequential Processes. Technical Monograph PRG-14, Oxford University Computing Laboratory. (Aug. 1978).

[59] Levin, R. Program Structures for Exceptional Condition Handling. Ph.D. Thesis, Carnegie-Mellon University (June 1977).

[60] Liskov, B.H. Exception Handling. CLU Design Note 60. Massachusetts Institute of Technology, Laboratory for Computer Science (Aug. 1976).

[61] Liskov, B.H. et al. CLU Reference Manual. Computation Structures Group Memo 161. Massachusetts Institute of Technology, Laboratory for Computer Science. (July 1978), 138pp.

[62] Liskov, B.H. and Snyder, A. Structured Exception Handling. Computation Structures Group Memo 255, Massachusetts Institute of Technology, Laboratory for Computer Science. (Dec. 1977).

[63] Liskov, B.H. and Snyder, A. Exception Handling in CLU. IEEE Trans. Softw. Eng. Vol. 5 Nr. 6 (Nov. 79), pp. 546-558.

[64] Liskov, B.H., Snyder, A., Atkinson, R. and Schaffert, C. Abstraction Mechanisms in CLU. Commun. ACM Vol. 20 Nr. 8 (August 1977), pp. 564-576.

[65] Liskov, B.H. and Zilles, S. Specification Techniques for Data Abstraction. Proc. Intl. Conf. Reliable Softw. (April 1975), pp. 72-87.

[66] Luckham, D.C. and Polak, W. Ada Exceptions: Specifications and Proof Techniques. Report Nr. STAN-CS-80-789 Computer Science Department, Stanford University. (1980).

[67] Luckham, D.C. and Polak, W. Ada Exception Handling: An Axiomatic Approach. ACM Trans. Prog. Lang. and Syst. Vol. 2 Nr. 2 (April 1980), pp. 225-233.

[68] MacLaren, M.D. Exception Handling in PL/I. Technical Report, Digital Equipment Corporation, Maynard, Mass. Also appears (abbreviated) in SIGPLAN Notices Vol. 12 Nr. 3 (March 1977), pp. 101-104.

[69] Majster, M.E. Treatment of partial operations in the algebraic specification technique. Proceedings Specifications of Reliable Software IEEE (1979), pp. 190-197.

[70] Meertens, L. Mode and Meaning. New directions in Algorithmic languages 1975. (Schuman, S.A., Ed.) IRIA (1975), pp. 125-138.

[71] Melliar-Smith, P.M. and Randell, B. Software Reliability: the Role of Programmed Exception Handling. Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Notices Vol. 12 Nr. 3 (March 1977), pp. 95-100.

[72] Mitchell, J.G., Maybury, W. and Sweet, R. Mesa Language Manual Version 5.0 (April 1979) CSL 79-3. Xerox Palo Alto Research Center, Systems Development Department.

[73] Naur, P. et al. Revised Report on the Algorithmic Language Algol 60. Commun. ACM Vol. 6 Nr. 1 (Jan. 1963), pp. 1-17, and Comp. J. Vol. 5 Nr. 4 (Jan. 1963), pp. 349-367.

[74] Noble, J.M. The Control of Exceptional Conditions in PL/I Object Projgrams. IFIP Congress 68, pp. C78-C83.

[75] Organick, E.I. The Multics System: An Examination of its Structure. MIT Press, Cambridge, Massachusetts and London (1972). xviii + 392pp.

[76] Parnas, D.L. and Würges, H. Response to Undesired Events in Software Systems. Proc. Second Intl. Conf. on Softw. Eng. (October 1976).

[77] Randell, B. System Structures for Software Fault Tolerance. Proc. Int. Conf. on Reliable Softw. (April 1975), pp. 437-457. SIGPLAN Notices Vol. 10 Nr. 6. Also in IEEE Trans. Softw. Eng. Vol. 1 Nr. 2 (June 1975), pp. 220-232.

[78] Randell, B., Lee, P.A. and Treleaven, P.C. Reliability Issues in Computing System Design. Comput. Surv. Vol. 10, Nr. 2 (June 1978), pp. 123-165.

[79] Richard, F. and Ledgard, H.F. A Reminder for Language Designers. SIGPLAN Notices Vol. 12 Nr. 12 (Dec. 1977), pp. 73-82.

[80] Richards, M. and Whitby-Stevens, C. BCPL — The language and its Compiler. Cambridge University Press (1978).

[81] Ross, D.T. The AED Free Storage Package. Commun. ACM, Vol. 10 Nr. 8 (Aug. 1967), pp. 481-492.

[82] Rutishauser, H. Interference with an Algol procedure. Annual Review Automatic Programming Vol. 2.(Goodman, R., Ed.) Pergamon Press (1961).

[83] Schroeder, M. Cooperation of Mutually Suspicious Subsystems. Ph.D. Thesis, Massachusetts Institute of Technology. MIT MAC-TR-104 (1972).

[84] Schwartz, R.L. An Axiomatic Semantic Definition of Algol 68. UCLA-34P214-75, Computer Science Department, University of California, Los Angeles. (July 1978), xiii + 218pp.

[85] Schwartz, R.L. An Axiomatic Treatment of Aliasing. Department of Applied Mathematics, Weizmann Institute of Science. (Sept. 1979).

[86] Schwartz, R.L. An Axiomatic Treatment of Algol 68 Routines. Sixth Colloq. Automata, Languages and Programming. Lecture Notes Computer Science Vol. 71. Springer-Verlag (July 1979), pp. 530-545.

[87] Shearing, B.H. The Forpa Programmer's Manual. Design Office Consortium, Guildhall Place, Cambridge (1977).

[88] Wilkes, A.J., Gibbons, J.J. and Singer, D.W. Exception Handling in BCPL. Rainbow Group Note, University of Cambridge Computer Laboratory (September 1980).

[89] Snow, C.R. An Exercise in the Transportation of an Operating System. Softw. Pract. Exper. Vol. 8 Nr. 1 (Jan.-Feb. 1978), pp. 41-50.

[90] Stallings, W. An Application of Coroutines and Backtracking in Interactive Systems. Int. J. Comput. Inf. Sci. Vol. 5 Nr. 4 (Dec. 1976), pp. 303-313.

[91] Stoy, J.E. and Strachey, C. OS6: An Operating System for a Small Computer. Technical Monograph PRG-8, Oxford University Computing Laboratory. Also in Comp. J. Vol. 15: Part 1, pp. 117-124; Part 2, pp. 195-203. (1972).

[92] Strachey, C. and Stoy, J.E. The Text of OSPub (Text and Commentary). Technical Monograph PRG-9, Oxford University Computing Laboratory (July 1972).

[93] Thimbleby, H. Leave and Recall: Primitives for Procedural Programming. Softw. Pract. Exper. Vol. 10 Nr. 2 (Feb. 1980), pp. 127-134.

[94] UCSD Pascal Users' Manual. SofTech Microsystems, Inc., San Diego, (Feb. 1980).

[95] United States Department of Defense. Requirements for High Order Computer Programming Languages: Steelman (June 1978). Also in [100], pp. 298-315.

[96] United States Department of Defense. Reference Manual for the Ada Programming Language - Proposed Standard Document. (July 1980).

[97] van Wijngaarden, A. et al. Revised Report on the Algorithmic Language Algol 68. Springer-Verlag (1976), 236pp.

[98] Wasserman, A.I. (Ed.) Special Issue on Programming Language Design. SIGPLAN Notices, Vol. 10 Nr. 7 (July 1975).

[99] Wasserman, A.I. Procedure-Oriented Exception Handling. Technical Report Nr. 27, University of California, Laboratory of Medical Information Science (February 1977).

[100] Wasserman, A.I. Tutorial: Programming Language Design. IEEE Catalog Nr. EHO 164-4.

[101] Wirth, N. Revised report on the programming langauge Pascal. Pascal User Manual and Report. (Jensen, K. and Wirth, N.), Eleventh printing. Springer-Verlag (1979), pp. 133-167.

[102] Woodger, M. (Ed.) Supplement to the Algol 60 Report. Commun. ACM Vol. 6 Nr. 1 (Jan. 1963), pp. 18-20.

[103] Wulf, W.A., London, R. and Shaw, M. An introduction to the construction and verification of Alphard programs. IEEE Trans. Softw. Eng. Vol. 2 Nr. 4 (Dec. 76), pp. 253-265.

[104] Zahn, C.T. A Control Statement for Natural Top-down Structured Programming. Programming Symposium (Ed. Robinet, B.). Lecture Notes in Computer Science, Vol. 19. Springer-Verlag (1974), pp. 170-180.