RC 8059 (#34988) 1/14/80
Computer Science 14 pages

# Research Report

## Exception Handling and Data Abstraction

Andrew P. Black
Programming Research Group, University of Oxford
and Thomas J. Watson Research Center, IBM Corporation



IBM Research Division
San Jose · Yorktown · Zurich

RC 8059 (#34988) 1/14/80
Computer Science 14 pages

# Exception Handling and Data Abstraction

Andrew P. Black†
Programming Research Group, University of Oxford
and Thomas J. Watson Research Center, IBM Corporation

**Abstract:** A simple treatment of exception conditions in algebraic specifications is presented. The method uses type unions and is also applicable to programs. The extensive use of unions is not practical in many programming languages because of the verbose and baroque constructions required to deal with them. Rather than introduce specialized facilities for exception handling it is proposed that a simpler syntax be adopted for unions.

**Key Words and Phrases:** Abstract Data Types, Algebraic Specifications, Data Abstraction, Data Types, Errors, Exceptions, Partial Operators, Type Unions, Unions, Variant Records.

**C.R. Categories:** 4.2, 4.22, 4.34.

† Author's present address: Oxford University Computing Laboratory, 45, Banbury Road, Oxford, OX2 6PE, England

## 0 Introduction

A great deal has been written on the supposed difficulties of dealing with errors in abstract data types, e.g., [5], [9], [6] and [17]. Some of the proposed solutions appear to be very complicated. In this paper we retain simplicity by taking the view that there *are no errors* in an abstract data type.

Two situations which are often cited as errors are failure of an implementation and production of an exceptional result. The first kind of error does not exist at all at the abstract level; it resides solely in the implementation and so the question of how to deal with it in the type definition never arises. Examples are attempting to represent the integers by fixed length bitstrings or unbounded stacks by bounded arrays. We do not consider such errors in this paper.

Of course, there is no reason why a designer should not axiomatize a restricted range of integers or a stack with a maximum capacity if that is what he really wants. In doing so he may need to define functions which produce exceptional results in certain circumstances, but such results are certainly not 'errors'; on the contrary, they are exactly what is required.

To illustrate this point, consider the well known (unbounded) stack of integers. Application of the interrogation function *Top* to such a Stack will usually yield an integer as the result. However, application of *Top* to the empty stack is *exceptional* in the sense that the result is *not* an integer. Whether or not applying *Top* to an empty stack constitutes an *error* depends entirely on what the routine *using* the stack abstraction does with the result of *Top*. If it attempts to add *7* to the result then an error has occurred; on the other hand, if it tests the result to see if the stack was empty then no error has occurred. Corresponding to the observation that only the way a result is used determines whether it is an error, we believe that it is desirable to use the *same* data type definition module in both of these situations. This implies that the same specification be valid in both situations too.

This paper is divided into 6 sections. Section 1 describes the problem of specifying types with exceptions. Section 2 uses an example to illustrate our proposed specification technique and Section 3 describes the impact of this technique on programs, programming languages and language implementations. Section 4 explores a mathematical background consistent with these implementations. Section 5 is a brief survey of the history of exception handling. Section 6 is the conclusion.

## 1 The Problem of Specifying Exceptions

We make no apology for drawing on the stack of integers as an example; simple as this type is, a lot of the published specifications are either incomplete, overly complex or wrong.

Most definitions of stack contain an axiom along the lines of

$$Top(Empty) = EXCEPTION$$

where for *EXCEPTION* read variously *UNDERFLOW*, *UNDEFINED*, *ERROR* and so on. What exactly does such an axiom mean? This is usually not at all clear.

In Guttag's Thesis [7] the functionality of *Top* was described by

$$Top: Stack \rightarrow Int.$$

This implies that *EXCEPTION* is an *Int*, which has some serious consequences. First, it is counterintuitive, because as we noted above the result of *Top(Empty)* is *not* an *Int* – that is why

it is an exception. Second, every time we create a new data type such as *QueueofInt* or *StringofInt* we are forced to add new exceptional elements to *Int*: in general, adding any new type requires altering the definitions of all the types it uses, which is both impractical and intellectually unappealing. Third, in doing this consistency problems arise in our axiomatization; while there are several ways to resolve them much additional machinery is needed.

Our last point, the problem of consistency, deserves some elaboration; we present a brief summary of the excellent exposition found in [6] Section 3.5. If *EXCEPTION* is an *Int* then axioms must be given which explain the action of *Int* operations on it. The usual philosophy is to assure that once an exception occurs it is propagated, that is, if any argument of an operation is *EXCEPTION* then so is its result.† If we attempt to implement this by simply adding new axioms we rapidly run into trouble. We must add rules like

$$Sum(n, EXCEPTION) = EXCEPTION \quad \text{(i)}$$
$$Product(EXCEPTION, n) = EXCEPTION. \quad \text{(ii)}$$

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

$$Sum(n, 0) = n \quad \text{(iii)}$$
$$Product(n, 0) = 0. \quad \text{(iv)}$$

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

$$\begin{aligned} 0 &= Product(EXCEPTION, 0) && \text{(by iv)} \\ &= EXCEPTION && \text{(by ii)} \end{aligned}$$

and

$$\begin{aligned} n &= Sum(n, 0) && \text{(by iii)} \\ &= Sum(n, EXCEPTION) && \text{(by above)} \\ &= EXCEPTION && \text{(by i)} \end{aligned}$$

which show that all integers are equal to *EXCEPTION*. A similar problem arises within the stack datatype; we must add a new constant *STACK EXCEPTION* of type stack and a new axiom

$$Push(s, EXCEPTION) = STACK\ EXCEPTION$$

to ensure that errors propagate. In combination with the more usual axioms it is simple to show that all stacks are equal to *STACK EXCEPTION*. This instance of the problem is dealt with very fully in [17].

Of course, such problems can be resolved - various techniques are described in [5], [6] and [17]. Our aim is not to criticize these mechanisms (indeed the mathematical intricacies are such that we would find this difficult) but simply to observe that forcing the result of *Top(stack)* to be an integer gives rise to serious problems which we would rather avoid.

More recently some authors ([8] and [10]) have changed the problem by specifying *Top* as

$$Top: (Stack) \rightarrow Int \cup \{EXCEPTION\}$$

Here again it is necessary to ask exactly what this means. We assume that the minor syntactic change of placing the domain in parenthesis is meaningless; this notation, together with the use of commas to denote a cartesian product, makes the specification more like most programming

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

languages. One possible interpretation for the right hand side, suggested when the word *UNDEFINED* is used in place of *EXCEPTION*, is that *Top* is a partial function, i.e. it is defined on some stacks only. In this case one might expect not to find any axiom giving the meaning of *Top(Empty)*, but then one could never be sure whether this was due to intention or oversight. So an axiom *Top(Empty) = UNDEFINED* might reasonably be added to indicate that the designer of the type had deliberately left the result of this application unspecified.

We will not pursue the use of partial functions any further, not because we doubt the ability of mathematicians to produce a consistent theory but because we are interested in using data types in programs. Being told that a program has an undefined result is not very useful; what tends to happen is that different implementations define the result in different ways and programmers become reliant on these local definitions. We should not forget that one of the functions of type specifications is to help avoid such implementation dependence.

The interpretation that remains is the one that will be studied in this paper. {*EXCEPTION*} denotes the datatype *Exception* (which has *EXCEPTION* as its only value) and ∪ denotes the union operation on datatypes. This interpretation also has its problems, but we feel that they have simpler solutions than those discussed above. We go on to outline our approach in more detail.

## 2 Axiomatic Specification of Exceptions

Figure 1 is our definition of *StackofInt*. We assume *Bool* to be a pre-defined type with values *TRUE* and *FALSE*, and *Union* to be a pre-defined *type schema* which denotes datatype union. We use the term *type schema* (after the usage of mathematical logic, e.g., [14]) to indicate that *Union* is a parameterized type definition, like *Array* or *Set*. (Indeed most frequently *Stack* would be a type schema too, but we do not want to become involved with that here.) The type *Union*[*ta, tb*] has operators *Is*[*ta*]: *(Union*[*ta, tb*]*)* → *Bool*, *From*[*ta*]: *(ta)* → *Union*[*ta, tb*] and *To*[*ta*]: *(Union*[*ta, tb*]*)* → *ta* (and similarly for *tb*) representing inspection, injection and projection respectively. The details of *Union* will be deferred to Section 4. This is because we first wish to motivate our definition by showing how we expect *Union* to be used. In particular we make no statement now about the result of *To*[*ta*] when applied to a union value that was not formed from type *ta*.

***Type*** *StackofInt*

***Uses*** *Underflow, Bool, Int*

***Operators***
*Empty: ( )* → *StackofInt*
*Push: (StackofInt, Int)* → *StackofInt*
*Pop: (StackofInt)* → *StackofInt*
*Top: (StackofInt)* → *Union*[*Int, Underflow*]
*IsEmpty: (StackofInt)* → *Bool*

*Pop(Empty)* = *Empty*
*Pop( Push(s, i) )* = *s*
*Top(Empty)* = *From*[*Underflow*](*UNDERFLOW*)
*Top( Push(s, i) )* = *From*[*Int*](*i*)
*IsEmpty(Empty)* = *TRUE*
*IsEmpty( Push(s, i) )* = *FALSE*

Figure 1: Specification of *StackofInt*

In using the names *Is*[*ta*], *From*[*ta*] and *To*[*ta*] we are inviting ambiguity between the operators of the type *Union*[*ta, tb*] and those of the type *Union*[*ta, tc*]. We assume that this can be resolved if necessary by qualifying the operator name with the type name, as in *Union*[*ta, tb*].*From*[*ta*]. We will not need to use this device in this paper because we will deal with a limited number of types, but it introduces no new semantic issues.

The definition of stack shown in Figure 1 uses only total functions and yet avoids the contradictions that troubled Majster [17]. *Pop( Push(S, Top(Empty)) )* is quite simply a domain error. The result of *Top(Empty)* is a *Union*[*Int, Underflow*], which is clearly not of the correct type to be used as the second parameter of *Push*, whose domain is *Int*. This is the kind of error we consider in this paper: note that it is not an error in the datatype but in the *use* of that type. Obtaining the result *UNDERFLOW* from *Top* is an error if and only if the program receiving that result was prepared to accept only integers. If it was prepared to find the stack empty then no error has occurred, although the situation is still an exception in the sense that the program will not perform the same operations on *UNDERFLOW* as it would have performed on an integer.

## 3 Programming with Union Types

The ease with which the abstraction of Figure 1 may be implemented and the degree of protection provided for the representation depend on the programming language used. Newer languages generally recognize the need to construct new types from old by the use of union and Cartesian product, to declare completely novel types by enumeration, to permit functions whose results are of arbitrary type, to encapsulate data representations by prohibiting access by functions other than the type operators, and so on. We intend to see how easy it is to define operators returning results of union types in various languages. As far as possible we will ignore the other facilities which aid or hinder the implementation of the rest of the type.

For the purposes of example we will concentrate on the function *Top*. Since it will be written only once, the ease of definition of *Top* is less important than its ease of use. There are two categories of use: one where the calling program is sure that the stack is not empty (because some other reliable module has just told it so) and one where being empty is acceptable, indeed sometimes expected. Throughout this section *stack* and *i* are *StackofInt* and *Int* variables respectivly.

In the first case, if the stack does indeed turn out to be empty then a real error has occurred. Our reliable module is not as reliable as we thought! But this eventuality is not something for which we should need to explicitly test; the burden of having to do that is intolerable. We suggest that

*i* : = *To*[*Int*](*Top(stack)*)

be the syntax used in this case, and that the *To*[*Int*] operator have the following meaning. If the type of its parameter is a *Union* with an *Int* component, and if the current value of the *Union* has been injected from a value of type *Int*, then the result is that value. Otherwise, an error has occurred. Whether that error is detected at compile-time or run-time is largely irrelevent and depends on how much work the compiler does. Some errors one would obviously expect the compiler to detect, a trivial example being the application of *To*[*Int*] to a *Bool* argument. On the other hand, where the actual type of a union depends on input data the error cannot be detected until run-time. We expect that in any reasonable implementation the run-time system will halt the program and produce a suitable error message, but we prefer to leave its choice to the implementor. The statement *i* : = *To*[*Int*](*Top(stack)*) should thus be interpreted as a message from the program writer to the program reader which says 'I know that *stack* is not empty so I am making *i* the top integer'.

The second case is illustrated by the following example.

```
PrintStack is
        variable e is bool
        e : = IsEmpty(stack)
        if ¬e do
                PrintInt(To[Int](Top(stack)));
                stack : = Pop(stack);
                PrintStack
        if e do
                PrintString("---bottom---")
        endif
```

Here an explicit test of the state of the stack is made. Note that with the specification of Figure 1 the function *IsEmpty* is redundant: the assignment *e* : = *IsEmpty(stack)* could be replaced by *e* : = *Is*[*Underflow*](*Top(stack)*) without changing the semantics. Of course, this is not true of a stack in which *Top* is a partial function: in that case *IsEmpty* is essential. Note further that both *Top* and *To*[*Int*] can have exactly the same semantics as in the previous case; we can be confident that no error will occur because of the way they have been used.

In what follows we will see how these two cases must be handled in various programming languages. The same examples will be treated and the reader is invited to make comparisons with the notation presented above.

Algol 68 provides a built-in type schema ***union*** which has some of the properties of our *Union*. We may easily name a type (called a 'mode'), as in

```
mode IntOrUnderflow = union(int, Underflow);
```

such a type can be used in exactly the same way as a primitive type. Note, however, that this does *not* create a *new* type; it names an already existing one. There is no way to create new types in Algol 68; in particular there is no way to create the enumeration type ***Underflow***. In practice we can be fairly confident that ***Underflow*** is distinct from other types by making it name a record ('structure') with only one field which is selected by the name *Underflow*. Such a structure is distinct from other structures with a different number of fields or a different selector. The mode of the field cannot be ***void*** (which would be nice, as no storage would then be allocated for it) so let us assume it is ***bool*** (which should waste the minimal amount).

```
mode Underflow = struct(bool Underflow)
```

In this particular example we could avoid the use of a distinct type ***Underflow*** altogether by declaring

```
mode Underflow = void
```

but this would not be useful if more than one exception could occur. This is a consequence of the rules for mode equivalencing. In the context of the declarations

```
mode Overflow = void;
mode Underflow = void;
mode StackResult = union(int, Underflow, Overflow)
```

the mode ***StackResult*** is indistinguishable from ***union(int, void, void)*** and from ***union(int, void)***. Thus such declarations would not enable one to distinguish between underflow and overflow.

Very often there is useful additional information which an operation would like to return when an exception occurs. In these cases declaring the exception types to be structures does not *waste* storage but *uses* it. For example, a procedure which reads a string of digits from an input stream might normally return an integer, but if a character other than a digit is found the result might be a string. The mode ***union(int, BadFormat)*** would be suitable for the result of such an operator, where ***mode BadFormat = struct(string*** *BadFormat).*

In Algol 68, injection (i.e. the formation of a union value from a component value), which we have indicated by using the *From*[ ] operators, is available as an implicit 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 context is not obvious it can be made so by enclosing the expression in a 'cast', such as ***IntOrUnderflow(****Top(stack)).* Thus it is easy to write the *Top* operation in Algol 68. Unfortunately it is difficult to use it, because our *Is*[*ta*] and *To*[*ta*] operations do not have simple counterparts. Instead there is a construction called the 'conformity clause' which must be used even if the programmer knows the current type of the union. Thus to perform the simple assignment appropriate to our first case we would have to write

```
i := case Top(stack)
     in (int result) :    result
     , (Underflow) : (    print ("Underflow found where Int expected");
                          stop
                     )
     esac.
```

This certainly makes clear what error action will be taken in the event that the type is not ***int***, but we deem the need to specify this a disadvantage because we are looking for a construct to use when the type is known to be right. The above is so verbose that one cannot really expect it to be used every time the top integer of a stack is examined. The conformity clause *is* suitable for use in the second case, when either an ***int*** or ***Underflow*** is expected. The *Printstack* example can be coded as

```
proc Printstack = void :
     case Top(stack)
     in (int result) :    (    print(result);
                               stack := Pop(stack);
                               Printstack
                          )
     , (Underflow) : print("---bottom---")
     esac.
```

It is possible to write operators equivalent to *To*[*Int*] and *Is*[*Int*] which simplify the first example.

```
proc ToInt = (IntOrUnderflow iou) int :
     case iou
     in (int i) :         i
     , (Underflow) : (    print ("Underflow found where Int expected");
                          stop
                     )
esac;
```

```
proc IsInt = (IntOrUnderflow iou) bool :
        case iou
        in (int) :          true
        , (Underflow) :     false
        esac.
```

The problem is that similar procedures must be written for every component of every union, and there is no 'procedure schema' mechanism in Algol 68 to help us produce them.

A similar, somewhat unsatisfactory, situation exists in Pascal. The problem is compounded by the need for *Top* to return a pointer to its result (functions may not return records) and because the basically orthogonal mathematical concepts of union and Cartesian product have been combined in a single linguistic construction, that is, the variant record.

The Pascal variant record is a structure containing an optional *fixed part* and a *variant part*, which 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. An example is

```
type symbol =    record    xloc, yloc : real;
                           area : real;
                           case  s : shape of
                           triangle :    (side : real;
                                          inclination, angle1, angle2 : real );
                           rectangle :   (side1, side2, inclination : real);
                           circle :      (diameter : real)
                 end
```

As it stands this construction is not type-safe, for several reasons. 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. Thus, as Berry and Schwartz observe in [1], the attitude of Pascal in this area is strictly *caveat programmor* (programmer beware). It is possible to construct some functions which, if used as the only method of accessing variant records, will prevent representation dependent results in the absence of parallelism. Figure 2 contains some samples. A significant improvement over Algol 68 occurs in the disassembly of the variant record, where there is no need to introduce a case statement unless case analysis is really required. Note that the *ToInt* function does not assign to *IsInt* if *iou*↑.*IsInt* = *false*. This is illegal, and one would expect the run-time system to produce an error. But it does not really matter what happens: *ToInt* is only applied when we have already made certain that its argument is an integer. We will not deal more fully with Pascal because so many semantic details are left unspecified by the Report [21] that it is impossible to do so. Instead we will take a look at Ada [12], [13], which specifies its intentions more completely.

The Pascal-like confusion of product and union is still present in Ada. Type safety is improved by making the tag-field (called a *discriminant*) a constant: it can be changed only by updating the whole record. Components of the variant part are accessed by the dot notation; if a component which does not exist is referenced the exception *DISCRIMINANT_ERROR* is raised. Unfortunately it is possible to suppress this exception, thus breaching what would otherwise be a strong typing system.

Ada does not have a facility powerful enough to permit the declaration of the *Union* type schema as we have envisioned it. The possibility of 'generic' types is allowed, but only the functionalities of the operators may be parameterized: we would like to parameterize the *names* of the operators as well. In Ada we could indeed write an instantiation

```
package IntOrUnderflow is new Union(Integer, Underflow)
```

```
type    Underflow = (UNDERFLOW);
        IntOrUnderflow = ↑IntOrUnderflowRecord;
        IntOrUnderflowRecord =  record case IsInt : Boolean of
                                        true :  (Int : integer);
                                        false :()
                                end;

function FromUnderflow (u : Underflow) : IntOrUnderflow;
var result : IntOrUnderflow;
begin   new(result);
        result↑.IsInt := false;
        FromUnderflow := result
end;

function FromInt (i : integer) : IntOrUnderflow;
var result : IntOrUnderflow;
begin   new(result);
        result↑.IsInt := true;
        result↑.Int := i;
        FromInt := result
end;

function IsInt (iou : IntOrUnderflow) : Boolean;
begin IsInt := iou↑.IsInt
end;

function ToInt (iou : IntOrUnderflow) : integer;
begin   if iou↑.IsInt
        then ToInt := iou↑.Int
        else writeln( ' Underflow found where Int expected ' )
end;
```

Figure 2: Pascal routines to encourage use of unions

but the operators would have to be called things like *IsType1* and *ToType2*, which is hardly satisfactory.

The Ada version of the type definition is shown below.

```
type IntOrUnderflow is
        record
                IsInt : constant Boolean;
                case IsInt of
                        when true => Int : integer;
                        when false => null;
                end case;
        end record;
```

Values of this type are constructed explicitly using record aggregates such as *(IsInt => true, Int => 17)* and *(IsInt => false)* (using the named selector notation). Since the type of such aggregates is not obvious we may chose to define conversion operators *FromInt* and *FromUnderflow*; samples are given in Figure 3.

Assignment of a union to an integer variable is accomplished by component selection. However, the syntax does not permit the direct application of selection to the result of a function: it is necessary to use an intermediate variable of type *IntOrUnderflow*. Assuming *iou* to

```
function FromInt (i : integer) return IntOrUnderflow is
begin return (IsInt => true, Int => i)
end;

function FromUnderflow (u : Underflow) return IntOrUnderflow is
begin return (IsInt => false)
end;
```

Figure 3: Ada routines for constructing union values

be such a variable one may write

```
iou := Top(stack);
i := iou.Int     -- May raise DISCRIMINANT_ERROR
```

There is no obvious reason for the proscription of the more direct *Top(stack).Int*. It is certainly a breach in orthogonality and probably arises through oversight. Nevertheless, the inconvenience of having to introduce a temporary name (which must be declared at the top of a 'unit', not where it is needed) is such that one will probably want to define a *ToInt* operator.

An interesting question now arises as to whether such an operator can be considered an Ada ***function***. It is not a mathematical function because it does not always return a value: if its argument is not formed from an integer it raises an exception instead. Additionally, such operators do not preserve commutativity as required by [13] Section 7.4. However, the raising of an exception is *not* one of the things disallowed in a function body. Indeed, the language definition [12] states that if an invocation of a function does not cause a ***return*** statement to be executed then the exception *NO_VALUE_ERROR* is raised. It also gives an example of a function containing an ***assert*** statement, which raises an exception if the assertion is false. The basic problem is that handling exceptions by jumping is an operational concept and does not have a place in functional programming. For the purposes of our example we will follow the letter rather than the spirit of the Ada definition and write

```
function ToInt (iou : IntOrUnderflow) return integer is
begin return iou.Int     -- May raise DISCRIMINANT_ERROR
end.
```

Of the languages discussed so far, Ada is the only one which provides facilities for encapsulating the definition of a data type. By this we mean that it provides a linguistic construction behind which the representation of a type may be concealed whilst other operations on it are exposed. Thus Ada supports data abstraction more specifically than does Algol 68 or Pascal.

There are a whole host of data abstraction languages which pre-date Ada. We will look at only one, the programming language CLU [16], [15]. We will not study the data abstraction facilities because that is not the main subject of this paper. We will examine the way unions are treated.

CLU has a basic type schema called the ***oneof*** which creates an object which is one of a set of alternatives. It was not designed as an exception handling mechanism; there are extensive (separate) features for exception handling. A ***oneof*** type has the form

***oneof*** $[l_1 : T_1, l_2 : T_2, \ldots, l_n : T_n]$

where the $l_i$ are labels ('tags') and the $T_i$ are types. All the $l_i$ must be distinct, but this is *not* required of the $T_i$. This overcomes the problem of creating unique types present in Algol 68; ***oneof*** [*Integer* : ***int***, *Underflow* : ***null***, *Overflow* : ***null***] is useful because it is possible to tell

whether a value ***nil*** (of type ***null***) indicates *Underflow* or *Overflow*. This type is also distinct from another ***oneof*** with fields of the same types but with different labels. A ***oneof*** is not a true union because ***oneof*** [ *i*: ***int***, *b*: ***oneof*** [ *i*: ***int***, *b*: ***bool*** ] ] cannot be simplified; it is not, for example, the same as ***oneof*** [ *i*: ***int***, *b*: ***bool*** ].

***Oneof*** objects must be created explicitly: for each tag $l_i$ there is a constructor *make_*$l_i$*(  )* which takes an argument of type $T_i$. Taking a ***oneof*** apart is more difficult. It is necessary to use a ***tagcase*** statement; this is similar to the Algol 68 conformity clause except that the choice is made according to the tag rather than the type. Thus, although CLU makes it easy to create the types which are needed to handle exceptions through unions, it does not offer appropriate constructions and syntaxes to use them for this purpose.

We conclude this section on programming by summarizing the properties a language should have in order to facilitate the use of unions to handle exceptions. It should be easy to create new types which cannot be confused with existing types. It should be easy to construct types which are unions of other types; the operations of creating values of the union and projecting such values back into the appropriate type must be possible without syntactic verbosity. The language should not insist on the use of case analysis unless there are cases to analyse. If there is no built in union type schema with these properties then there should be a type parameterization mechanism sufficiently powerful to create one. None of these requirements need compromise efficiency or type safety.

## 4 A Mathematical Basis for Union

In order to have made any advance on previous work we must state exactly what is meant by our type definitions using *Union*, which means giving a semantics for *Union*. Figure 4 is an attempt to provide an axiomatic specification of *Union*.

***type*** *Union*[*ta : Type, tb : Type*]

***uses*** *Bool*

***Operators***
*Is*[*ta*] *: (Union*[*ta, tb*]*) → Bool*
*Is*[*tb*] *: (Union*[*ta, tb*]*) → Bool*
*From*[*ta*] *: (ta) → Union*[*ta, tb*]
*From*[*ta*] *: (tb) → Union*[*ta, tb*]
*To*[*ta*] *: (Union*[*ta, tb*]*) → ta ...*
*To*[*tb*] *: (Union*[*ta, tb*]*) → tb ...*

*Is*[*ta*]*( From*[*ta*]*(a) ) = true*
*Is*[*ta*]*( From*[*tb*]*(b) ) = false*
*Is*[*tb*]*( From*[*ta*]*(a) ) = false*
*Is*[*tb*]*( From*[*tb*]*(b) ) = true*
*To*[*ta*]*( From*[*ta*]*(a) ) = a*
*To*[*ta*]*( From*[*tb*]*(b) ) = ...*
*To*[*tb*]*( From*[*ta*]*(a) ) = ...*
*To*[*tb*]*( From*[*tb*]*(b) ) = b*

Figure 4: Partial Specification of *Union*

As the ellipses indicate, this specification is incomplete. An application *To*[*tb*]*( From*[*ta*]*(a) )* is clearly a type error and we would expect a program containing it to halt and say so. The usual way of accommodating this in a functional semantics is to define the result of such a misapplication to be ⊥ (bottom), the totally undefined value, and to ensure that

all operations in the language are strict in $\perp$, i.e. $f(\perp) = \perp$ for all operations $f$. Now it is indeed correct for a program to halt when it encounters *To*[*tb*]( *From*[*ta*](*a*) ). Since this expression evaluates to $\perp$, the whole computation must evaluate to $\perp$. It is unnecessary to perform any further applications: one can simply stop and say that the answer is $\perp$.

The requirement that every operation be strict in $\perp$ means that $\perp$ must be in the domain and range of every operation. This can be ensured by implicitly adding '$\cup\ \perp$' to both sides of all the functionalities in our specifications. To the axioms we implicitly add $f(\perp) = \perp$ for all operators $f$. Note that we cannot use *Union* to extend the functionalities because making $\perp$ into a type, say *Bottom*, would mean that *Is*[*Bottom*]( *From*[*Bottom*](⊥) ) = *true* where strictness requires that it should be $\perp$. In accordance with this convention we can complete the above specification by deleting the ellipses after *To*[*ta*] and *To*[*tb*] in the list of functionalities and replacing them by $\perp$ in the axioms.

It may be argued with some justification that using $\perp$ to represent the result of an erroneous application is not very satisfactory. One would be even more justified in complaining about its use to denote a non-terminating computation. To rectify these faults requires a slightly more complicated set of domains in our semantic theory. But in a real language *To*[ ] will not be the only operator requiring such embellishments for its complete formal definition. It seems reasonable to define *To*[ ] with the same degree of rigour as the rest of the language. We do not intend to dictate here what this should be. What is important is that the ordinary programmer should know exactly what the operators of *Union* will do.

It may seem that in leaving open exactly how to specify the exceptional cases of the *To*[ ] operators we have left the original problem unsolved. This is not so. We started with the problem of finding a simple way of dealing with exceptions in user defined data types. This problem has been reduced to that of dealing with exceptions in *one* type, and one which is built into the programming or specification language. And that problem is not new; it has been with us ever since we began to attempt the rigorous definition of languages.

The conventional interpretation of a data type is that it defines a set of values, i.e. each value belongs to exactly one type. It is clear that we must be very careful to avoid the paradoxes of set theory if we wish to treat *types* as values, which is the natural way of dealing with type parameters [19]. That is why we have preferred to speak of *Union* as a type schema rather than as a function *type* × *type* → *type*; the usual interpretation of a schema is by syntactic substitution.

Recently there has been some interesting work on types as sets of operations [3], [2]. The underlying value space is considered to be untyped and there is no fundamental problem in dealing with types as values: the paradoxes mentioned above are avoided. Another advantage of this approach is that it is closely related to the way in which values are dealt with in real implementations. In a computer a value is an untyped string of bits whose *interpretation* depends on the type of the operators used to examine it. Although this formalization is still a research topic it offers hope of providing a rigorous semantics for languages with type definition facilities.

## 5 A Historical Perspective

The problem of handling exceptions is not a new one to programming, nor is it confined to operators implementing data abstractions. An example as old as Fortran is a routine which searches an array $A$ for a number $x$ and if it is found returns an index $i$ such that $A[i] = x$. The exceptional case of $x$ not being found was traditionally indicated by the return of $0$ or some other value not valid as an index into $A$.

This is indeed a reasonable solution in a language like Algol 60 or Fortran in which it is not possible to indicate the range of the search function beyond saying that that the result will be an integer. In Pascal it is possible to specify that the range is exactly the same as the subscript domain of *A*, and it is advantageous to do so because it makes the intention of the program clearer to both human readers and the compiler. The latter may then use a more compact representation for the result and avoid inserting the run-time checks that would otherwise be necessary when the result is used to index an array. These advantages are lost if the range of the function is extended to include even one value which is *not a valid index*, and that is exactly what we are doing if we add *0*. What is needed is some mechanism for indicating explicitly that *0* is a an exceptional value not intended as an index and that only values in *1..n* will be used in this way. What the proposal of Section 3 amounts to is just such a mechanism: instead of writing the result type as *0..n* we advocate writing *Union*[*NotFound*, *1..n*]. It is reasonable to expect the compiler to notice that only the *1..n* component is used as an index and to elide tests accordingly. Thus the proposal of this paper represents nothing new: it is merely the translation of a well known technique into a notation compatible with specific typing.

Another historically significant method of dealing with exceptions was common in the days of assembly code. When a routine detected an exceptional condition it did not return normally at all; it either stopped with an error message or made an abnormal return. For example, the standard return point for a subroutine call might be to a place two instructions after the call; a return to the instruction immediately following the call would indicate an exception. A different way of achieving a similar effect is to pass as a parameter an address to which the exceptional return should be made. This can be programmed in Algol 60 where non-local labels may be passed as parameters. Use of these techniques has declined as the dangers of non-local and dynamic jumps and routines with multiple exits became widely appreciated and languages which restricted such facilities were developed. There has been a recent reversal of this trend with the appearance of languages like CLU, Mesa [4] [18] and Ada which attempt to promote such jumps to the status of a high level feature. The exception handling in CLU is by far the simplest of the three; it nevertheless gives rise to what is by far the most involved part of the semantics [20]. Experience with Mesa, which has a much more complicated exception mechanism, has shown that it confounds readability and reliability [11]. It is still too early to evaluate exception handling in Ada as a formal definition has yet to be published and there is no experience with its use. However, examination of the manual [12] gives one cause for concern. The global nature of exceptions violates the principle of locality that has been maintained in the rest of the language. The way an exception will be handled in the multi-tasking environment does not even seem to be deterministic. One can only hope that things will turn out to be better than they appear.

## 6 Conclusion

The method of dealing with exceptions proposed in this paper avoids all the difficulties mentioned above. It does this by achieving a separation of concerns. All exceptions are confined to the *Union* type schema, leaving the programmer free to create libraries of data types and operators whose semantics can be stated by simple axioms.

The ease of use of this technique depends to some extent on how willing one is to allow implicit conversions. In the example *i* : = *To*[*Int*](*Top(stack)* it might be reasonable for the *To*[*Int*] to be syntactically omissible. The context here is such that a compiler could quite easily insert it. Although such abbreviations can be attractive they can also give rise to misunderstanding. Whether they should be allowed, and if so in which contexts, could be the subject of another paper. The considerations are human rather than mathematical; it may turn out that the place to insert the coercions is not in the compiler but in the editor used to construct the program text. This paper aims to provide a semantic base on which such syntactic shorthands can be built.

Our method involves certain changes in style which some programmers may be reluctant to make. At first our way of handling exceptions may seem strange. We have used it to program the example given in [15] Section 12.3, in which many exceptions occur at different levels. With suitable defaults we find the use of unions produces more readable code than exception handlers. The reader is invited to try some examples himself. As a bonus there is no exception handling scheme to understand, and data types defined using unions can be specified with the simplest axiomatic technique.

## Acknowledgements

This paper was written while on leave at the IBM T. J. Watson Research Center, Yorktown Heights, New York; the support of IBM Corporation and IBM World Trade Corporation are gratefully acknowledged.

The author wishes to thank J. W. Thatcher and C. A. R. Hoare for constructive comments on an earlier draft of this paper. R.-D. Fiebrich has helped improve the presentation in many ways.

## References

[1] Berry, D.M. and Schwartz, R.L. United and discriminated record types in strongly typed languages. *Inf. Processing Letters 9,* 1 (July 1979) pp 13–18

[2] Demers, A. and Donahue. J. Revised report on Russell *TR 79-389* Department of Computer Science, Cornell University. (September 1979) pp 42

[3] Donahue, J. On the semantics of 'Data Type' *SIAM J. Comput. 8,* 4 (November 1979). pp 546–560

[4] Geschke, C.M., Morris J.H. Jnr. and Satterthwaite, E.H. Early experiences with Mesa. *Comm. ACM 20,* 8 (August 1977). pp 540–553

[5] Goguen, J.A. Abstract errors for abstract data types. *Formal Description of Programming Concepts. (Neuhold, E.J., Ed.)* North Holland (1978). pp 491–525

[6] 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

[7] 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). pp 154

[8] 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). pp 45

[9] Guttag, J.V., Horowitz, E., Musser D.R. Some extensions to algebraic specifications. *Procs of an ACM Conference on Language Design for Reliable Software (Wortman, D.B. Ed.)* North Carolina (March 1977) pp 63–67

[10] 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

[11] Horning, J.J. Language features for fault tolerance. *Lecture Notes on Computer Science 69: Program Construction International Summer School. (Bauer, F.L. and Broy, M., Eds.)* Springer-Verlag (1977). pp 508–512

[12] Ichbiah, J.D. et al. Preliminary Ada reference manual. *SIGPLAN Notices 14,* 6 Part A (June 1979).

[13] Ichbiah, J.D. et al. Rationale for the design of the Ada programming language. *SIGPLAN Notices 14,* 6 Part B (June 1979).

[14] Kleene, S.C. *Mathematical Logic.* Wiley, New York (1967).

[15] Liskov, B.H., Moss, E., Schaffert, C., Scheifler, R.W. and Snyder, A. CLU Reference Manual. *Computational Structures Group Memo 161.* Massachusetts Institute of Technology, Laboratory for Computer Science. Draft (July 1978). pp 138

[16] Liskov, B.H., Snyder, A., Atkinson, R. and Schaffert, C. Abstraction mechanisms in CLU. *Comm. ACM 20,* 8 (August 1977). pp 564–576

[17] Majster, M.E. Treatment of partial operations in the algebraic specification technique. *Proceedings Specifications of Reliable Software* IEEE (1979). pp 190–197

[18] Mitchell, J.G., Maybury, W. and Sweet R. Mesa Language Manual. *Technical Report CSL-78-1.* Xerox Palo Alto Research Centre (1978).

[19] Reynolds, J.C. Towards a theory of type structure. *Lecture Notes in Computer Science 19: Programming Symposium. (Robinet, B, Ed.)* Springer-Verlag (1974). pp 408-425

[20] Scheifler, R.W. A Denotational Semantics of CLU. *M.S. Thesis - MIT/LCS/TR-201.* Massachusetts Institute of Technology, Laboratory for Computer Science. (May 1978). pp 175

[21] Wirth, N. Revised report on the programming language Pascal. *Pascal User Manual and Report. (Jensen, K. and Wirth, N)* Eleventh printing. Springer-Verlag (1979). pp 133–167