Technische Hogeschool Twente

MEMORANDUM NR. INF-84-8

Exception handling constructs considered unnecessary.

Maarten M. Fokkinga

April 1984.

Twente University of Technology
Department of Informatics
P.O. Box 217
7500 AE Enschede.

Onderafdeling der Informatica

Contents Page

Abstract In his thesis "Exception handling: the case against" (Univ. of Oxford, January 1982) Andrew P. Black shows convincingly that normal control structures, together with the data type discriminated union, suffice to replace exception handling facilities in a satisfactory way. We want to propagate his ideas with an example. Moreover we show that (i) even with the discriminated union approach incremental program construction is still possible, and that (ii) the programs using discriminated unions resemble the programs using exception handling facilities more than Black suggests.

1. Introduction

In his thesis Black gives a thorough analysis of what usually is meant by "exception" and "exception handling", and he also shows how the examples which are used to motivate exception handling facilities can be programmed by normal constructs. I will not repeat his arguments, but for the treatment of one example which clearly shows the most important technique. In this way I hope to propagate his ideas.

The second aim of this paper is the following. One of the advantages of exception handling facilities is claimed to be the possibility of incremental program construction. That is, first a program is constructed which is correct for the normal case, i.e. assuming that the input entities satisfy the precondition, and then the program is adapted so that the exceptional inputs are dealt with satisfactoryly too; see (Bron & Fokkinga 1977). This aspect of exception handling, facilitating a separation of concerns, is not considered by Black in his thesis. We show that incremental program construction is also possible with the normal constructs advocated by Black to replace the exception handling constructs.

Our third aim is the following. We propose a suitable syntax for the operations of a discriminated union so that the elimination of exception handling constructs brings about only minor modifications of the program texts; in our example the texts with and without exception handling constructs resemble each other more than in a similar example treated by Black. To be honest, however, we note that Black already hinted at such syntactic sugar.

*****

Black gives various techniques to handle exceptions using normal constructs, the most important of which is the discriminated union. Discriminated unions look like the union construct of Algol 68 and the variant record of Pascal; they are explained in detail in Section 3. Another construct which is sometimes needed, is the procedure as parameter; in our example we can do without it. Finally, often "local termination" is needed, that is, a construct to terminate a textually enclosing program fragment, like statements, blocks, repetitions and procedure bodies. Most programming languages contain some such constructs: exists, returns, leaves, Zahn's events, and so on. It is like hitting a musquito with a sledge-hammer if one introduces exception handling constructs solely for local terminaton. Indeed, it is generally the purpose of "raising an exception" to terminate dynamically invoking (rather than textually enclosing) blocks.

It turns out that the main technique to eliminate exception handling constructs boils down to the following. First, the dynamically nested block incarnations are now terminated locally; each one separately from the others. Second, the case analysis (exception analysis and handling) is now written precisely at the place where the exception is detected "by the user" (after suitable notifications by invoked procedures), rather than at the end of the block which is to be terminated. Thus the program becomes now more "structured" in the sense that more of its behaviour (more of its correctness proof) can be deduced by local inspection of its text.

The remainder of the paper is organized as follows. First, in Section 2 we show the example program using exception handling constructs; we take this program for granted. In Section 3 the exception handling constructs are eliminated; discriminated union is introduced and explained. Some further elaborations are given in Section 4 and we conclude with Section 5.

2 An example with exception handling

We consider the following problem statement. "Construct a procedure sum which yields as its result the sum of several numbers, the denotations of which are placed on the input separated by one or more spaces. For the conversion of a string (a denotation) to its integer value the procedure s2i (short for: string to int) is to be used".

To be more specific we assume the following context for the definition of procedure sum. There are several exceptions, some of which are used by s2i and +.

```
exc overflow, badformat(char), fatal-error, too-large;
proc s2i: (string --> int) possibly raising badformat, too-large;
operation +: (int, int --> int) possibly raising overflow;
```

Procedure s2i raises the exception badformat(c) if its argument is not a proper denotation; c equals the first invalid character. If on the other hand the number represented by the denotation exceeds maxint, the exception too-large is raised. Similarly, + raises overflow if the result would exceed maxint. Exception fatal-error is a standard one; we assume that a programmer cannot handle it, i.e. if it is raised the complete program is terminated (and handled by the operating system).

Here is the definition of sum; some explanatory notes follow it.

```
proc suml: int;
   exc none;
   var s: int;
   proc readint: int possibly raising none;
        var denot: string;
        begin skip spaces:
                  while not eof(input) cand input ↑= space
                  do get(input)
                  od;
              if eof(input) then raise none;
              accumulate denotation:
                  denot := input ↑; get(input);
                  while not eof(input) cand input ↑/= space
                  do denot := denot concat input ↑;
                     get(input)
                  od;
              return s2i (denot)
        end readint;
   begin s := 0;
         while true do s := s + readint od[none => skip];
         return s
   end
```

figure 1

Notes.

1. We do not claim that the program is elegant; we only show it to illustrate some use of exception handling.
2. On the second line a new exception is introduced: none. It is raised in the middle of readint where it turns out that there is no more denotation on the input. In the third line from below the while-statement is possibly terminated due to the raising of none from within readint; the handling of that exception consists of doing <u>skip</u> followed by normal continuation of the execution after the while-statement.

3. In the heading of readint it is not specified that it possibly raises the exceptions too-large and bad-format. Therefore these exceptions are by default reraised at the end of readint as fatal-error. In other words, the <u>end</u> of readint actually reads:

   <u>end</u>[badformat(c) => <u>raise</u> fatal-error, too-large => <u>raise</u> fatal-error]

4. Similarly, overflow exceptions from within the body of sum are reraised by default as fatal-error.

One should note that during the construction of the program only the normal situation is taken into account. An exceptional input is not at all considered, and the programmer need not even be aware of the fact that + and s2i possibly raise exceptions. Nevertheless the program is robust and reliable: erroneous input does not lead to unreliable results.

Now we want to deal with exceptional inputs too. First the requirements for sum are extended. If too large a denotation is encountered, or the sum of the members exceeds maxint, exception overflow is to be raised. And if a denotation in bad format is encountered, a new exception badf(d) is to be raised, where d is the invalid denotation, i.e. a string. In a previous paper (Bron & Fokkinga 1977) we have argued that exception handling will do well for this purpose. Indeed, we may adapt the program by inserting additions only; nothing of the original text needs to be rewritten or changed. The new text is shown in figure 2.

Even if sum is to deliver maxint as result rather than raising overflow, it is easy to adapt the program. This, and similar variations, are left to the reader.

<u>3. Exception handling eliminated</u>

We now show how the handling of exceptions can be expressed using normal programming language constructs. For our example we need constructs to terminate a textually enclosing block and repetition, and the discriminated union. We first explain our syntax and the semantics of the discriminated union, and then present the new formulations of sum1 and sum2. It should be obvious from the strong resemblance with the old versions that exactly the same reasoning and methodology can be applied to derive sum1′ and sum2′, or to adapt sum1′ to sum2′, as was done before with sum1 and sum2.

```
exc badf (string);
proc sum2: int possibly raising overflow, badf ;
    exc none;
    var s: int;
    proc readint: int possibly raising none , too-large, badf ;
        var denot: string;
        begin skip spaces:
                    ------------
                if eof(input) then raise none;
                accumulate denotation:
                    ------------
                return s2i (denot) [bad format (c) =>
                                     raise badf (denot)]
        end readint;
    begin s := 0;
         while true do s := s + readint od [none => skip]
         return s
    end [too-large => raise overflow]
```

figure 2

The discriminated union is a data structure with the following operations: injection (postfix $\uparrow i$), projection (postfix $\downarrow i$), inspection (postfix ?i) and case selection (postfix ?[f1,..., fn]). Their semantics is explained by the following axioms; i and j are constants 1, 2, 3, ...

$$\text{expr} \uparrow i \downarrow j = \begin{cases} \text{expr} & \text{if } i=j \\ \underline{\text{fatal}}\ \underline{\text{error}} & \text{if } i/=j \end{cases}$$

$$\text{expr} \uparrow i \ ?j = \begin{cases} \text{true} & \text{if } i=j \\ \text{false} & \text{if } i/=j \end{cases}$$

$$\text{expr} \uparrow i \ ?[f1,\ldots,fn] = fi(\text{expr})$$

The type of a discriminated union is written as t1+...+tn where t1,...,tn are the types of its summands. So e.g. an injection expr $\uparrow i$ is well-typed only if expr has type ti and the whole has type t1+..+ti+..+tn, for some types t1,...,tn.

For the sake of readability we often give "summand identifiers" in the type (like field identifiers in records) and use them instead of 1,2,3,... in the operations; and if possible we also label the cases of a case selection with the summand identifiers. Moreover we define two abbreviations (coercions, syntactic sugar) in order that the use of discriminated unions can compete notationally with exception handling constructs: the operations ↑1 and ↓1 need not be written, but are automatically inserted (by the compiler) if the context so requires. Note that expr↓1↑1 may be written expr, and yields fatal error if expr?1=false.

We are now ready to present the transliteration, eliminating the exception handling constructs and using the discriminated union and exits and returns instead. The program text is given in figure 3; it is to be interpreted in the following context.

```
proc s2i: (string --> int + bf: char + too-large: void)
oper + : (int, int --> int + ovf: void)
```

Void is a type with only one element, denoted by: empty.

```
proc sum1': int;
    var s: int;
    proc readint: int + none: void;
        var denot: string;
        begin skip spaces:
                    -----------
              if eof(input) then return empty↑none;
              accumulate denotation:
                    -----------
              return s2i(denot)
        end readint;
    begin s:=0;
        while true
        do s := s + readint?[i: i, none e: exit]    (*)
        od;
        return s
    end
```

(*) Notation. The texts "i: i" and "e: exit" are notations for functions with formal parameter i resp. e and body i resp. exit. Note that the second case is labelled with summand identifier 'none', see the type of readint.

figure 3

Note that s2i(denot) actually means s2i(denot) ↓1↑1, so that an error results when denot has a bad format or represents too large an integer. A similar remark applies to (s + readint?[...]). Note also that the programmer needs only know, and take account of, the first summand of the result types of s2i and +.

Again we now may extend the specification of the procedure, and extend the program text accordingly, so as to take exceptional inputs into account. Procedure sum2′ is given in figure 4.

```
proc sum2′: int [+ ovf: void + badf: string];
    var s: int;
    proc readint: int + none: void [+ too-large: void + badf: string];
        var denot: string;
        begin skip spaces:
                  ---------
              if eof(input) then return empty ↑ none;
              accumulate denotation:
                  ---------
              return s2i (denot) [?[i: i
                                  , bf c: denot ↑ badf
                                  , too-large e: e ↑ too-large
                                  ]]
        end readint;
    begin s := 0;
        while true
        do s := (s + readint?[i: i
                             , none e: exit
                             [, too-large e: return e ↑ ovf
                              , badf s: return s ↑ badf]
                             ]
                ) [?[i: i, ovf e: return e ↑ ovf]]
        od;
        return s
    end
```

figure 4

Especially from figure 4 and 2 it is clear that the elimination of exception handling constructs gives rise to longer program texts. One reason is that the flow of control is expressed explicitly; termination of the chain of dynamically invoking blocks is programmed by several local terminations of only textually enclosing blocks. Another reason is to be found in the syntax for case selection; our syntax requires that each case is treated separately and explictly, in contrast to the exception handlers. Some abbreviation is quite well concievable; after all, seven of the nine cases are merely an identity or an identity followed by a return.

One may also dislike the exists and returns out of subexpressions. Rightly so. It is caused by our wish to take procedure sum1 and sum2 as a starting point. In those programs the exits and returns exist as well, but they are not written explicitly! Our aim was to simulate sum1 and sum2 as precise as possible, and in this respect we are quite succesfull.

We found it furthermore quite surprising to observe that auxiliary variables like s and denot kept their original types. At first we had expected that some of them should get a discriminated union type. By now it is clear that no such thing will happen due to the elimination of the exception handling constructs. (However, such types may appear of course during programming if the programmer so wishes. Discriminated unions are a normal data structure, like arrays and records).

4. Some further elaborations

We first give an exercise for the reader and then discuss alternative ways for writing s := s + readint. Nothing new is explained in this Section.

In our example we have used both get and eof to operate upon the input. These two procedures might be replaced by just one: readchar, resulting in an exception end-of-file if no more character is present. We leave the adaptations of sum1 and sum2, and the transition to sum1′ and sum2′, as an exercise to the reader. (Notice that now there is a greater similarity between readchar and readint. However one could as well replace readint by a pair of procedures, int-eof and int-get say, which cooperate via a private look-ahead variable. This is another technique advocated by Black to eliminate exception handling constructs.)

Now we consider the statement s := s + readint. Assume for simplicity that readint does indeed yield an integer as result --without exceptions--. Suppose we had written s +:= readint, and we wanted to eliminate the exception handling of overflow caused by the + operation. The problem is where to put the case-selection ?[i: i, ovf e: return e↑ovf] without changing anything of the given program text. The solution is simple. Together with the abbreviation s +:= expr for s := s + expr, one should also devise an abbreviation for s := (s + expr)?[...], for example s +:=?[...] expr. Thus incremental programming is still possible.

The reader may now wonder how to deal with plus-and-becomes (s, readint) instead of s := s + readint. There is no problem here too. Indeed, let plus-and-becomes be defined as follows.

```
proc plus-and-becomes: (var x: int, e: int --> void)
                             possibly raising overflow;
begin x := x + e;
      return empty {a void result!}
end
```

According to our elimination scheme this is translated as follows.

```
proc plus-and-becomes: (var x: int, e: int -->
                             {normal:} void + ovf: void);
begin x := (x + e)?[i: i, ovf e: return e ↑ovf];
      return empty {with the coercion: ↑1}
end
```

writing in sum1′ respectively in sum2′:

- plus-and-becomes(s, readint) {with the coercion: ↑1}
- plus-and becomes(s, readint) ?[e: e, ovf e: return e ↑ovf]

Again we see that the exceptional termination of the body of plus-and-becomes is programmed explicitly, rather than implicitly via the raise of an exception from within the dynamically enclosed (i.e. invoked) + operation.

5. Conclusion

We have shown by means of one nontrivial example how constructs for local termination and discriminated union may replace exception handling constructs. The scheme seems quite uniformly applicable. Its main characteristic feature is that the dynamically determined jumps of control are replaced by accumulating textually determined control jumps. This slightly increases the text, but may have its benefits with respect to readability and efficiency.

Black argues that the notion of "exception" is ill-defined and that the programmer should take account of exceptional situations right from the beginning, thus reducing their status to normal situations. We do not want to discuss this position, but only remark that incremental programming (i.e. taking care of exceptional situations only after a correct program for the normal cases has been constructed) is as feasable with exception handling constructs as with their replacements.

Finally we stress once more that we wanted to propagate the technique of Black without repeating all of his 238 page Thesis.

References

Black, A.P.: Exception handling, the case against.
Ph.D. Thesis, University of Oxford, 1982, 238 pages

Bron, C. & Fokkinga, M.M.: Exchanging robustness of a program for a relaxation of its specification. TW-Memorandum nr 178, September 1977, Twente University of Technology, Netherlands.