Set-Oriented Functional Style of Programming

Conrad Steven Martin Mueller
Department of Computer Science
The University of the Witwatersrand, Johannesburg

Abstract

A novel way of programming, which starts with specific details of a particular example and abstracts these details into generalised sets, is proposed. The whole program is constructed around the definition of these sets. Once the structure and type of the sets have been specified, the relationship between the sets is added to these set defintions. Thus, there is a move away from seeing the description of the data and relationships between the data as two separate parts, i.e., the data definition and the body of the block. This paper concentrates on the method used to develop programs, using a tutorial approach to illustrate the ease of programming, debugging and modifiablity. An interesting aspect of the method is how a program can be developed by studying the details in the problem domain using a procedural approach and abstracting them into a final declarative definition.

Introduction

The author is currently involved in research considering the value of removing the need to specify order in a program. As part of this research, a language was developed which requires a new approach to programming. This paper focuses on this method of programming rather than the language. While the language is used to illustrate the method, it is not dealt with formally, but is explained in the context of examples.

A program can be seen as a relationship between two sets: the domain (i.e. the input) and the range (i.e. the output) of the relationship. The definition of a program is usually broken up into two parts: the data definition describing the sets and the body describing the relationship.

Considerable attention has been given to ways in which to express these relationships and, in different styles of programming, relationships are expressed in different ways. Programs in the von Neumann style of language are expressed as procedures, which, when applied to an element in the domain, produces an element in the range. In functional style languages, a program is a function expressed as a composition of primitive functions [Glasser 1984]. Predicates, as in logic programming, can also be used to express relationships [Clocksin 1984].

The definition of the sets, between which these relationships are defined, has received less attention. The only way of specifying these sets is by using types, which only became widely accepted with the general use of Pascal [Jensen 1974]. With object-oriented [Goldberg 1983] and abstract data types [Ghezzi 1982], there has been a movement towards grouping the data definition with the description of the relations on that data. The approach to programming proposed in this paper takes this further: the definition of relationships between the sets forms part of the definition of the data, and the programming task revolves around the specification of sets rather than of relationships.

The method proposed here uses a functional style to express relationships, but instead of expressing a relationship as a composition of functions, a relationship is expressed as related sets. Programming becomes a task of specifying sets, and included as a part of this definition is how a set is related to other sets. The emphasis is moved from programming relationships to defining sets and the most important part of programming becomes that of identifying what sets are required.

This method of programming is explained using a tutorial style. In this way, it is possible to illustrate the new methodology. A particular aim is to show that a procedural view of programs can be used for development, but the final specification of the program is declarative. Other features of the method are also highlighted, such as the ease of modification and debugging.

Without extensive experimentation, it is difficult to conclusively show that the method proposed is a better method than any other. However the method does appear to have some advantages over other styles of programming. The programmer is dealing with sets, which are much more concrete and tangible than descriptions of relationships. The method is based on a functional style of programming and thus has the desirable mathematical properties of this style. Thus the method has the potential to be easy to use, while at the same time being mathematically sound.

Summation

The first example is used to introduce the method. This example shows how simple iteration works. The problem is to add up a list of numbers which are terminated with a zero.

The method's first step is to study the problem, and relate elements identified in the problem area with sets in the language (i.e. the computational domain). This is done by identifying elements in the problem area as belonging to sets with certain computational properties. In this example, there are two easily identifiable sets: the set of all lists of numbers, and the set consisting of all sums of lists of numbers. These sets need now to be associated with a type in the computation structure of the program which can be done as follows:

```
list : SEQUENCE
    number : INTEGER

summation : INTEGER
```

The next step is to define a relationship between these two sets. If one assumes that an operation 'sum' is defined, which maps a sequence of integers onto an integer, then the relationship between the two sets can be specified as follows:

```
summation : INTEGER <- ,sum, list;
```

The above line reads as: 'summation' is a set of integers which is defined as 'sum' of 'list'. The set 'list' can in turn be defined in terms of the input as follows:

```
list : SEQUENCE <- input:
    number : INTEGER <- input;
```

and the complete program is as follows:

```
summation : PROGRAM <- list:

    list : SEQUENCE <- input:
        number : INTEGER <- input;

    summation : INTEGER <- ,sum, list;

summation : END.
```

If no operation 'sum' exists then the relationship needs to be defined in terms of some intermediate set or sets. The next step is to establish what these sets are. A possible method for establishing what these sets are is to calculate the relationships manually for a given element in the domain. The element, {5, 6, 7, 8, 0} in the set 'list', has been chosen for the manual calculation giving the following results:

list		running		summation
		total	result	
number (1) 5	step (1)	5		
number (2) 6	step (2)	11		
number (3) 7	step (3)	18		
number (4) 8	step (4)	26		
number (5) 0	step (5)		26	26

The intermediate step in the calculation can be expressed by a set 'running' consisting of a sequence of 'step's each of which is made up of the cartesian product of 'total' and 'result'. The particular element of 'running' in the manually calculated example is: {[5, undefined], [11, undefined], [18,undefined], [26,undefined], [undefined,26]}. The set 'running' can be associated with a type as follows:

```
running     : SEQUENCE
   step     : CARTESIAN
      total  : INTEGER
      result : INTEGER
   step     : END
```

Having determined what sets are required, the next stage is to establish the relationships between the three sets: 'list', 'running' and 'summation'. The relationship between 'running' and 'summation' is easy to specify as the only defined set, 'result' in the sequence 'running', needs to be mapped onto 'summation'. Since only one of the sets 'result' in 'running' is defined, they can all be mapped onto 'summation' as follows:

```
summation : INTEGER <- running.(step).result;
```

The brackets around 'step' specifies that all the sets 'step' in the sequence 'running' must be mapped onto 'summation'.

The next set to consider is 'running' which is define in terms of a more complex relationship. The domain of this relationship needs first to be established. Writing down the calculation in more detail gives a clearer picture as to what the domain of this relationship is.

list +	last ->	running	
number (1) 5	0	total (1) 5	
number (2) 6	total (1) 5	total (2) 11	
number (3) 7	total (2) 11	total (3) 18	
number (4) 8	total (3) 18	total (4) 26	
number (5) 0	total (4) 26		result (5) 26

From the above calculation, there appears to be two sets which map onto 'running'. The one is the sequence 'list' and the other is the sequence 'running' prefixed with the constant set zero. Thus to be able to define the domain of 'running' (such that, the corresponding sets in the domain map into the range), it is necessary to define a new sequence, 'last', which is made up of the constant set, zero, followed by all the sets in the sequence 'running'. This set can be defined as:

```
last:= {step:= [total:= 0], running.,}
```

The curly brackets specify a sequence whereas the square brackets define a cartesian product. The dot comma after 'running' indicates that all the sets of the sequence, 'running', form part of the sequence, 'last'.

The definition of 'running' can now be extended to incorporate its domain as follows:

```
running     : SEQUENCE <- list, last:= {step:= [total:= 0], running.,}:
   step     : CARTESIAN
      total  : INTEGER
      result : INTEGER
   step     : END
```

How the domain maps onto the range, is specified in terms of how each of the sets in the sequences forming the domain, map into the corresponding sets in 'running'. This is done by

defining the sub-component 'step' in 'running'. Since 'step' is also a complex component, it in turn needs to be defined in terms of its sub-components. However, the domain of 'step' must be specified first. The domain consists of the sets: 'number' in 'list', and 'step' in 'last'. These domain sets can be added to the definition as follows:

```
running     : SEQUENCE <- list,+ last:= {step:= [total:= 0], running.,}:
   step     : CARTESIAN <- number, step :
      total  : INTEGER
      result : INTEGER
   step     : END
```

The definition of 'step' defines how the i^{th} set 'step' in 'running' is defined in terms of the i^{th} set 'number' and 'step' in the domain. Everything to the left of the arrow (<-) refers to the range, thus 'step' to the left of the arrow refers to 'step' which is part of 'running'. Everything to the right of the arrow refers to the domain, thus 'step', here, is part of 'last', and 'number' is part of 'list'.

The final stage is to define the sub-components 'total' and 'result'. 'Total' is equal to the last 'total' plus the next 'number', if 'number' is not zero and 'result' is equal to the last 'total', if 'number' is zero. This relationship can be expressed as:

```
running     : SEQUENCE   <- list,+ last:= {step:= [total:= 0], running.,}:
   step     : CARTESIAN  <- number, step :
      total  : INTEGER    <- |number,<>,0| total,+,number;
      result : INTEGER    <- |number,= ,0| total;
   step     : END;
```

In the expressions above (which are italicized), bars bracket the guards which specifies when the expressions, to the right of the guards, hold and can be read as:

```
total (an integer set) = total + number        if number <> 0
result(an integer set) = total                 if number = 0
```

The above definition of sets can now be incorporated into a program as follows:

```
summation : PROGRAM <- list:

   list          : SEQUENCE <- input:
      number     : INTEGER <- input;

   running       : SEQUENCE <- list, last:= {step:= [total:= 0], running.,}:
      step       : CARTESIAN <- number, step :
         total   : INTEGER <- |number,<>,0| total,+,number;
         result  : INTEGER <- |number,= ,0| total;
      step       : END

   summation     : INTEGER <- running.(step).result;

summation : END.
```

Books teaching a language such as Pascal spend a good proportion of the book covering aspects such as: the structure of a program, iteration, selection and data defintions [Atkinson 1981, Keller 1982, Wilson 1978]. In this example, these concepts are introduced in a relatively short space.

Greatest Common Divisor

This example shows the important features of selection and iteration in one program. The algorithm used in this example is Euclid's classical algorithm for calculating the greatest common divisor (gcd) as discussed in Dijkstra's book on structured programming [Dijkstra 1976]. This example is also used to illustrate how the program can be modified.

The problem domain consists of the set of the cartesian product of two sets of numbers and the set of all possible gcds. The initial design assumes that the numbers are positive, and the program is then modified to show how to extend the program to cater for all possible integers. The two sets in the problem area can be associated with the computational structure as follows:

```
numbers : CARTESIAN                gcd : INTEGER
   x    : INTEGER
   y    : INTEGER
numbers : END
```

The next step is to define a relationship between these two sets. As there is no existing operations which can directly express this relationship, the programmer is required to define the relationship via some intermediate set or sets. The task is to establish what this set is, or these sets are. As before, the suggested approach is that the programmer should manually calculate the relationship between the two sets for a particular element in the domain. The resulting values calculated can then be related to a set or sets. Below is the calculation of the gcd using Euclid's algorithm for the numbers, where x is 30 and y is 18.

	numbers	calculation gcd				
		step1	step2	step3	step4	
x	30	12	12	6		
y	18	18	6	6		
result					6	6

The intermediate step can be expressed as the set 'calculation' which is made up of a sequence of sets called 'step' where each 'step' is the cartesian product of the sets 'x', 'y' and 'result'. The set 'calculation' can be associated with the computational domain as follows:

```
calculation : SEQUENCE
    step    : CARTESIAN
       x        : INTEGER
       y        : INTEGER
       result   : INTEGER
    step : END
```

At this point, the programmer is ready to specify the relationship between the sets. The programmer can deal with each set independently of the other and in any order. Here, the mapping onto the set 'calculation' is chosen to start with. Each set, 'step', in the set 'calculation' is defined in terms of the previous 'step'; except for the first set which is defined in terms of the set 'numbers'. Hence the domain of the relationship which maps onto the set 'calculation' is the sequence whose first set is the set 'numbers' followed by all the sets in the sequence 'calculation'. The domain, of the relationship mapping onto 'calculation', can thus be expressed as {step:=numbers, calculation.,}. The first line of the specification of 'calculation', which specifies that 'calculation' is a sequence and what the domain of the relationship is, can now be completed.

```
calculation : SEQUENCE <- last:= {step:=numbers, calculation.,}:
```

How the domain relates to the range is specified in terms of each sub-component, 'step', in the range. The next line of the specification follows easily on from the first because each 'step' in the domain maps onto each 'step' in the range. The definition of 'calculation' can thus be extended as follows:

```
calculation : SEQUENCE <- last {step:=numbers, calculation.,}:
   step : CARTESIAN <- step:
```

The definition of each 'step' is defined in terms of the sub-components 'x', 'y' and 'result' as follows:

```
calculation : SEQUENCE <- last:= {step:=numbers, calculation.,} :
   step : CARTESIAN <- step:
      x : INTEGER <- |x,>,y| x,-,y |x,<,y| x;
      y : INTEGER <- |y,>,x| y,-,x |y,<,x| y;
      result : INTEGER <- |x,=,y| x;
   step : END;
```

The definition of set 'gcd' is simple, in that, all the 'result' components of the set 'calculation' are mapped onto it. The complete program looks as follows:

```
gcd : PROGRAM <- numbers:

   numbers : CARTESIAN <- input:
      x : INTEGER <- input;
      y : INTEGER <- input;
   numbers : END;

   calculation : SEQUENCE <- last:= {step:=numbers, calculation.,} :
      step : CARTESIAN <- step:
         x : INTEGER <- |x,>,y| x,-,y |x,<,y| x;
         y : INTEGER <- |y,>,x| y,-,x |y,<,x| y;
         result : INTEGER <- |x,=,y| x;
      step : END;

   gcd : INTEGER <- calculation.(step).result;

gcd : END.
```

Going back to the manual calculation, each of the numbers written down can be associated with an element in a set defined within the program as follows:

number
x = 30
y = 18

calculation

step(1)	step(2)	step(3)	step(4)
x = 12	x = 12	x = 6	
y = 18	y = 6	y = 6	
			result = 6

gcd
6

The assumption is that the numbers are always positive. The program can now be modified to deal with all integers. The set 'numbers' can first be mapped onto the set 'positive' which in turn is mapped onto the set 'calculation'. The programmer may first wish to develop the relationship, 'positive', first before incorporating it into his program. This can be done with the following program.

```
positive : PROGRAM <- numbers:

    numbers : CARTESIAN <- input:
        x : INTEGER <- input;
        y : INTEGER <- input;
    numbers : END ;

    positive : CARTESIAN <- numbers:
        x : INTEGER <- |x,>=,0| x  |x,<,0| ,-,x;
        y : INTEGER <- |y,>=,0| y  |y,<,0| ,-,y;
    positive :END;

positive : END.
```

Once the programmer has tested this relationship, it can then be incorporated into the original program as follows:

```
gcd : PROGRAM <- numbers:

    numbers : CARTESIAN <- input:
        x : INTEGER <- input;
        y : INTEGER <- input;
    numbers : END;

    positive : CARTESIAN <- numbers:
        x : INTEGER <- |x,>=,0| x  |x,<,0| ,-,x;
        y : INTEGER <- |y,>=,0| y  |y,<,0| ,-,y;
    positive :END;

    calculation : SEQUENCE <- last:= {step:= positive, calculation.,} :
        step : CARTESIAN <- step:
            x : INTEGER <- |x,>,y| x,-,y |x,<,y| x;
            y : INTEGER <- |y,>,x| y,-,x |y,<,x| y;
            result : INTEGER <- |x,=,y| x;
        step : END;

    gcd : INTEGER <- calculation.step.result.;

gcd : END.
```

Sort

A sort example is a slightly more complex example than a typical first year exercise, and yet one that can be expressed concisely. Sorting is an essential operation found in almost every aspect of computing and hence is an important example to look at. The example given here is based on the insert sort algorithm [Knuth 1973]. This example is also used to show how a programmer goes about debugging his program. The program is developed with some aspects not correctly specified and some idea is given as to how these errors can be detected and corrected.

The problem consists of elements in two sequences: the set of unsorted sequences and the set of sorted sequences. In this example, both the sorted and unsorted sequences are sequences of integers. These two sequences can be then associated with the computational structure as follows:

```
unsorted : SEQUENCE
    number : INTEGER

sorted : SEQUENCE
    number : INTEGER
```

The exercise is to define the relationship between these two sets. As in the previous examples, the programmer needs to establish what intermediate sets are required to map the set of unsorted sequences onto the set of sorted sequences. An element in the domain set, 'unsorted', can be selected, say {3, 8, 4, 2, 5}, and the relationship with an element in the range set, 'sorted', can be established manually. Below this relationship is calculated manually using the insert sort algorithm.

unsorted	sort				sorted
	pass(1)	pass(2)	pass(3)	pass(4)	pass(5)	
3	3	3	3	2	2	2
8	8	8	4	3	3	3
4			8	4	4	4
2				8	5	5
5					8	8

The intermediate values can be abstracted into a set called 'sort' which is made up of a sequence of 'passes'. Each 'pass' set consists of a sequence of 'numbers'. The set 'sort' can thus be associated with the computational domain as follows:

```
sort : SEQUENCE
    pass : SEQUENCE
        number : INTEGER
```

The programmer now needs to define the relationships between the three sets: 'unsorted', 'sort' and 'sorted'. The set 'sort' is defined in terms of the set 'unsorted' and itself, as each 'pass' of 'sort' inserts the next unsorted number into the previous pass of 'sort'. The domain set specifying the previous pass needs to be defined. The previous pass is the set 'sort' itself with some initial condition which needs to be chosen. In the manual calculation the initial set is the empty sequence giving the following definition:

```
previous:= {pass:= {}, sort.,}
```

The definition of the set 'sort' can now be extended as follows:

```
sort : SEQUENCE <- unsorted, previous:= {pass:={}, sort.,}:
    pass : SEQUENCE
        number : INTEGER
```

Each pass within the sequence 'sort' has as domain the previous 'pass' and the next set in the sequence of 'unsorted'. To decide what the domain for each 'pass' is, it is useful to look in more detail at the manual calculation as to how each set in 'pass' is calculated. An element in the fifth set 'pass' in 'calculation' is considered below:

first pass(4)	second pass(4)	next unsorted ->	sort pass(5)	
	number (1) 2	number (5) 5	number (1) 2	*second*
number (1) 2	number (2) 3	number (5) 5	number (2) 3	*second*
number (2) 3	number (3) 4	number (5) 5	number (3) 4	*second*
number (3) 4	number (4) 8	number (5) 5	number (4) 5	*next*
number (4) 8		number (5) 5	number (5) 8	*first*

The unsorted set needs to be slotted into the previous 'pass' between the two consecutive sets where one is greater and the other is less than or equal to the next 'unsorted' set. In the calculation above, there are three sequences used in defining the next 'pass' of 'sort' which are 'first', 'second' and 'next'. The sequence 'first' consists of the previous 'pass' prefixed with some initial condition. The programmer is now forced to think about this boundary case. For simplicity the numbers to be sorted will be assumed to be greater than zero in which case this set can be the set zero. The sequence 'second' consists of just the 'previous' 'pass' and the sequence 'next' consists of the set 'number' in the sequence 'unsorted' which is the same throughout the sequence 'next'. The domain of each pass can now be added to the definition of the set 'sort' giving:

```
sort : SEQUENCE <- unsorted, previous:= {pass:={}, sort.,}:
    pass : SEQUENCE <- next:= {number..}, second:= pass, first:={number:=0,pass.,}:
        number : INTEGER
```

In the definition of 'next', the double dot specifies that the set 'number' is the same throughout the sequence 'next'. Each set 'number' in 'pass' within 'sort' is defined in terms of the 'next' unsorted number and two consecutive sets 'first' and 'second' as follows:

```
number : INTEGER <-   |second.number,<=,next.number| second.number
                      |first.number,>,next.number| first.number
                      |first.number,<=,next.number,&,
                      next.number,<,second.number| next.number
```

In the above line it is necessary to qualify the sets 'number' in the domain as 'number' is a sub-component of 'first', 'second' and 'next'. These sets are qualified by prefixing their identifiers with the identifiers of the sets they are sub-components of. The complete definition of 'sort' can now be extended to:

```
sort : SEQUENCE <- unsorted, previous:= {pass:= {}, sort.,}:
    pass : SEQUENCE <- next:={number..}, second:= pass, first:={number:=0, pass.,}:
        number : INTEGER <-   |second.number,<=,next.number| second.number
                              |first.number,>,next.number| first .number
                              |first.number,<=,next.number,&,
                              next.number,<,second.number| next.number;
```

At this point, it is useful to re-examine the manual example to check if the relationships have been correctly specified.

unsorted 3, 8, 4, 2, 5

sort
domain= (unsorted= { 3,8,4,2,5}, previous= {{},{3,}..

pass (1)
domain= (next= {3..}, first= {0}, second= {})
number (1) nothing defined **error** should have been 3

pass (2)
domain= (next= {8,8..}, first= {0,3}, second= {3})
number (1) = 3 because second(1) <= next(1)
number (2) nothing defined **error** should have been 8

pass (3)
domain= (next= {4,4,4..}, first= {0,3,8}, second = {3,8})
number (1) = 3 because second(1) <= next(1)
number (2) = 4 because first(2) <= next(2) < second(2)
number (3) nothing defined **error** should have been 8
•••
•••

There is clearly a problem with the boundary conditions in the example above as no relationship holds for the last set in the sequence. This error was made in determining the initial set for the sequence 'previous' for which the empty sequence was chosen. It now becomes clear that this set is the boundary condition which specifies the end of the pass and a simple solution is to make this set equal to the maximum number allowed. Also this sentinel should be propagated to all subsequent passes. The definition of the sort can be altered to cater for this as follows:

```
sort : SEQUENCE <- unsorted, previous:= {pass:= {number:=Max}, sort.,}:
   pass : SEQUENCE <- next:= number, second:= pass, first:= {number:=0, pass.,}:
      number : INTEGER <-    |second.number,<=,next.number| second.number
                             |first.number,>,next.number| first.number
                             |first.number,<=,next.number,&,
                             next.number,<,second.number| next.number ;
```

Going back to the example, the change results in the following relationships:

unsorted 3, 8, 4, 2, 5, 6

sort

pass (1) *next={3,..}, first={0,Max}, second={Max}*

number (1) = 3 ; second(1) <= next(1)
number (2) = Max ; first(2) > next(1)

pass (2) *next={8,8,..}, first={0,4,Max}, second={3,Max}*

number (1) = 3 ; second(1) <= next(1)
number (2) = 8 ; first(2) <= next(2) < second(2)
number (3) = Max; next(3) < first(3)

pass (3) *next={4,4,4,..}, first={0,3,8;Max}, second={3,8,Max}*

number (1) = 3 ; second(1) <= next(1)
number (2) = 4 ; first(2) <= next(2) < second(2)
number (3) = 8 ; next(3) < first(3)
number (4) = Max; next(4) < first(4)

pass (4) *next={2,2,2,2,..}, first={0,3,4,8,Max}, second={3,4,8,Max}*

number (1) = 2 ; first(1) <= next(1) < second(1)
number (2) = 3 ; next(2) < first(2)
number (3) = 4 ; next(3) < first(3)
number (4) = 8 ; next(4) < first(4)
number (5) = Max ; next(5) < first(5)

pass (5) *next={5,5,5,5,5,..}, first={0,2,3,4,8,Max}, second={2,3,4,8,Max}*

number (1) = 2; second(1) <= next(1)
number (2) = 3; second(2) <= next(2)
number (3) = 4; second(3) <= next(3)
number (4) = 5; first(4) <= next(4) < second(4)
number (5) = 8; next(5) < first(5)
number (6) = Max; next(6) < first(6)

sorted 2, 3, 4, 5, 8

Defining the 'sort' sequence is the most complex relationship and the remaining relationships are fairly straightforward, resulting in the following program:

```
sorted : PROGRAM <- unsorted:

   unsorted : SEQUENCE <- input:
      number : INTEGER <- input;

   sort : SEQUENCE <- unsorted, previous:= {pass:= {number:=Max}, sort.,}:
      pass : SEQUENCE <- next:={number..}, second:= pass, first:= {number:=0, pass.,}:
         number : INTEGER <- |second.number,<=,next.number| second.number
                             |first.number,>,next.number| first .number
                             |first.number,<=,next.number,&,
                             next.number,<,second.number| next.number

   sorted : SEQUENCE <- sort, unsorted :
      number : INTEGER <- |unsorted.number,=,end|sort.(pass).number;

sorted : END.
```

There are two important boundary conditions in the insert sort which determine either end of the previous pass into which the next unsorted set is being inserted. These two boundary conditions are required as initial sets in the definition of two domain sequences thus making it difficult to accidentally overlook these boundary conditions.

Another interesting aspect of the above program is that although the algorithm was manually worked out in a procedural way, the program itself is declarative. The program only expresses relationships between sets.

Conclusions

Programming using this method is made easier by allowing the programmer to start by working with the actual details of a problem. The programmer is able to take an actual case and work through it. The programmer works through an example any way he wishes, which is likely to be procedural. Having established a relationship for a particular element or elements in the domain, he can abstract this relationship, so that it holds for all elements in the domain set. The first part of the abstraction is to describe the sets as belonging to types or composition of types. The structure of these sets is suggested by the elements used in the manual calculation. The programmer is still able to associate with the problem domain and, in particular, the example he worked out, because he is relating elements in the example with sets in the program. The second part of the abstraction is to express the relationships between sets. The relationships are expressed as simple expressions and these relationships hold for all the elements in the sets. The programmer is in a sense working bottom-up: from the details to an abstract definition. At the bottom, he is dealing with elements in a procedural way, which are easy to work with. These details are abstracted into a declarative specification of sets and relationships between sets, which is mathematically sound and easy to understand.

Debugging a program developed using this method becomes a process of establishing that the relationships between sets are correct. An error in one relationship does not invalidate the correctness of any other relationship. Thus, the relationships between sets can be checked independently of the rest of the program. These relationships can be verified by comparing that the domain elements map onto the appropriate range elements. As the examples show, this can be done by associating the elements in the sets with the names of the sets and desk checking can easily be done in this way. Debugging can also be done using the computer; by recording every element (with its set identifier) created during the computation of the relationship. These elements can then be sorted according to their identifiers and printed out. The relationships can then be manually checked to verify that an element in the domain of a relationship maps onto the correct range element. Errors appear to be easily identifiable as illustrated by the sort example. Also, the source of the error is obvious. Again, the concepts of sets is useful in that the programmer is dealing with identifiable sets to which elements belong and the programmer can check that the relationships between these elements are correct. These sets have identifiers which should correspond with the problem area, and so knowledge of the particular problem should assist in debugging.

The gcd example shows how easy it is to modify a program. The relationship of one set in terms of its domain is independent of other relationships within the program. Also, a program is a definition of sets and the definition of a set includes how it is related to other sets. The two above facts make a program easy to modify as a modification is localised to only those sets which need to be changed. The programmer is not concerned with the effect a change will have on the overall program. A modification is to a set and its correctness is determined only by whether the relationship with its domain is correctly specified. A program, which is described using sets, does not have an overall structure — only sets are structured. Thus a change to the structure of a set does not affect the overall structure of a program.

Even though this research has not developed far, it does show promise of finding a better tool for programming. The method relies heavily on a new way of expressing programs which requires a different style of language. While a language has been designed and implemented to write and test the examples given in this paper, there are questions, as to whether this can be done efficiently.

REFERENCES

- Atkinson L., [1981], *Pascal Programming*, John Wiley and Sons, Chichester.
- Clocksin W.F., and Mellish C.S., [1984], *Programming in Prolog*, 2nd ed., Springer Verlag, New York.
- Dijkstra E.W., [1976], *A Discipline of Programming*, Prentice-Hall.
- Ghezzi C. and Jazayeri M. [1982], *Programming Language Concepts*, John Wiley, New York.
- Glaser H., Hankin C. and Till D., [1984], *Functional Programming*, Prentice-Hall.
- Goldberg A. and Robson D., [1983], *Smalltalk-80: The Language and its Implementation.*, Addison Wesley, Reading, Massachusetts.
- Jensen K. and Wirth N., [1974], *Pascal User Manual and Report.*, Lecture Notes in Computer Science 18, Springer-Verlag, New York.
- Knuth D.E., [1973], *The Art of Computer Programming: Volume 3 / Sorting and Searching*, Addison-Wesley, Reading, Masachusetts.
- Keller A.M., [1982], *Computer Programming using Pascal*, Computer Science Press, MacGraw-Hill, New York.
- Wilson I.R. and Addyman A.M., [1978], *A Practical Introduction to Pascal*, MacMillan, London.