The South African Institute for Computer Scientists and Information Technologists

ANNUAL RESEARCH AND DEVELOPMENT SYMPOSIUM

23-24 NOVEMBER 1998
CAPE TOWN
Van Riebeeck hotel in Gordons Bay

Hosted by the University of Cape Town in association with the CSSA, Potchefstroom University for CHE and The University of Natal

PROCEEDINGS

EDITED BY
D. PETKOV AND L. VENTER

SPONSORED BY

The South African Institute for Computer Scientists and Information Technologists

ANNUAL RESEARCH AND DEVELOPMENT SYMPOSIUM

23-24 NOVEMBER 1998
CAPE TOWN
Van Riebeeck hotel in Gordons Bay

Hosted by the University of Cape Town in association with the CSSA, Potchefstroom University for CHE and The University of Natal

GENERAL CHAIR : PROF G. HATTINGH, PU CHE

PROGRAMME CO-CHAIRS:
PROF. L VENTER, PU CHE (Vaal Triangle), PROF. D. PETKOV, UN-PMB

LOCAL ORGANISING CHAIR: PROF. P. LICKER, UCT - IS

PROCEEDINGS

EDITED BY
D. PETKOV AND L. VENTER

SYMPOSIUM THEME:

Development of a quality academic CS/IS infrastraucture in South Africa

SPONSORED BY

Proceedings of the 1998 Annual Research Conference of the South African Institute for Computer Scientists and Information Technologists.
Edited by Prof. D. Petkov and Prof. L. Venter
Van Reebeck Hotel, Gordons Bay, 23-24 November 1998

ISBN: 1-86840-303-3

Keywords: Computer Science, Information Systems, Software Engineering.

Office of SAICSIT: Prof. J.M.Hatting, Department of Computer Science and information Systems, Potchefstroom University for CHE, Private Bag X6001, Potchefstroom, 2520, RSA.

Produced by the Library Copy Centre, University of Natal, Pietermaritzburg.

FOREWORD

The South African Institute for Computer Scientists and Information Technologists (SAICSIT) promotes the cooperation of academics and industry in the area of research and development in Computer Science, Information Systems and Technology and Software Engineering. The culmination of its activities throughout the year is the annual research symposium. This book is a collection of papers presented at the 1998 such event taking place on the 23rd and 24th of November in Gordons Bay, Cape Town. The Conference is hosted by the Department of Information Systems, University of Cape Town in cooperation with the Department of Computer Science, Potchefstroom University for CHE and and Department of Computer Science and Information Systems of the University of Natal, Pietermaritzburg.

There are a total of 46 papers. The speakers represent practitioners and academics from all the major Universities and Technikons in the country. The number of industry based authors has increased compared to previous years.

We would like to express our gratitude to the referees and the paper contributors for their hard work on the papers included in this volume. The Organising and Programme Committees would like to thank the keynote speaker, Prof M.C.Jackson, Dean, University of Lincolshire and Humberside, United Kingdom, President of the International Federation for Systems Research as well as the Computer Society of South Africa and The University of Cape Town for the cooperation as well as the management and staff of the Potchefstroom University for CHE and the University of Natal for their support and for making this event a success.

Giel Hattingh, Paul Licker, Lucas Venter and Don Petkov

Table of Contents Page

A STATIC PROGRAMMING PARADIGM

Conrad Mueller
Department of Computer Science
University of the Witwatersrand, Johannesburg
email: conrad@cs.wits.ac.za

1 Introduction

Formal methods have not as yet succeeded in becoming an accepted way to develop software. This paper explores whether a possible reason for this is the complexity of the tools we use, in particular, the programming paradigms. Existing programming paradigms view a program as something that is executed or evaluated, requiring reasoning about the dynamic behaviour of a program. Reasoning about something dynamic is considerably more complex than a static situation. The question considered here is whether it is possible to reason about a program in a static way, and in this way reduce the complexity of the tools we work with.

An alternative programming paradigm is proposed in which it is possible to reason about a program as a static entity. The conceptual jump is to reason about programs as if they have completed rather than describing their dynamic behaviour. A general definition of programming is developed that forms the principles upon which the alternative paradigm is based. The definition describes a program in terms of elements defining all the possible values that can be computed. An element is defined in terms of a unique reference and a value.

The paradigm exploits this view of a program as a description of elements. The value of an element is defined as static relationships to the values of other elements. These relationships can be defined as simple algebraic expressions. Inherent in the paradigm is the ability to define infinitely many elements. Two mechanisms are required to achieve this: the first is to be able to reference infinitely many elements uniquely, and the second is to be able to define these elements. This is achieved by the reference consisting of an identifier and an index or indices, as well as being able to express generalizations for classes of elements.

This paradigm is used to evaluate whether moving to a more static way of reasoning reduces the complexity of programming. Some simple examples are used to explore how it is possible to reason in a formal way about a program in this paradigm. The advantages illustrated by these examples are used to justify the need to take this research further and to consider if we need better tools if we are to succeed in encouraging formal methods in becoming the norm.

2 Need to explore alternatives

Up to now we have been influenced by early models of computation. Operational semantics has played an important role in these models, that rely on understanding how the computation takes place. Even the models that have tried to move away from the instruction based view still land up requiring the programmer to have some notion of how the computation takes place. For example in Prolog[1], the order of the rules are important.

The consequence of this are that programs are seen as a mechanism to compute values. Programming then means having to design and understand a mechanism. A mechanism by its nature is dynamic and the order in which events take place is important. This adds an extra layer of complexity in the task of programming. Consider the following simple example of a program:

```
sum := 0;
count := 0;
While number[count] <> 0 do
    Begin
        sum:= sum + number[count];
        count := count +1
    End
average := sum / count;
```

The following aspects relate directly to it being a mechanism:

- the need for control structures like "while" statement,
- the order in which the statement are executed are important thus we cannot put the statement "sum:=0" at the end of the program,
- the meaning of an identifier can have multiple meanings within a program and an identifier like "sum" will reference different values throughout the execution of the program,
- since we are manipulating data, the mechanism needs to provide mechanisms for storing the data that is illustrated by the array "number" and the manipulation of the index "count",
- the semantics are dependent on the execution of the program in particular the control constructs.

What is even more disturbing about this is that we start to accept that this is an inherent part of programming and do not see the complexity resulting from viewing a program in this way.

Few would question that the properties a paradigm should have are:

- simple semantics — no need to understand the underlying mechanism,
- referential transparency — each reference has a single meaning [2],
- static semantics — no need to take into account the order of evaluation,
- few concepts both in terms of the syntax and the semantics,
- straightforward notation based on commonly accepted principles, and
- semantics that are easy to reason about.

In our example we see that all the properties are negatively influenced by the way in which we view programs as dynamic objects. Our argument is that one needs to explore the possibility of developing a paradigm that views a program as a static object.

3 New approach: describing data

We propose that the focus be shifted to looking at the possible values computed in a program and that these are viewed independently of how they are computed. The paradigm becomes one of describing the relationship between these values. Let us develop this idea by considering an example of finding the average of a list of numbers 5, 2, 8 and 6.

One way is to list all the values in the computation and describe them by giving them a meaningful names as follows:

$$
\begin{array}{ccccc}
 & number_0 = 5 & number_1 = 2 & number_2 = 8 & number_3 = 6 \\
sum_0 = 0 & sum_1 = 5 & sum_2 = 7 & sum_3 = 15 & sum_4 = 21 \\
 & & average = 5.25 & &
\end{array}
$$

Based on this example we can describe the relationship that holds between the values as shown below:

$$\begin{aligned} average &\stackrel{def}{=} sum_4/4 \\ sum_0 &\stackrel{def}{=} 0 \\ sum_{i+1} &\stackrel{def}{=} sum_i + number_i \end{aligned}$$

We can extrapolate this relationship to one that is valid for finding the average for any list of four numbers. This relationship can be extended to hold for lists of any length with the following modification where *LoL* is the length of the list:

$$\begin{aligned} average &\stackrel{def}{=} sum_{LoL}/LoL \\ sum_0 &\stackrel{def}{=} 0 \\ sum_{i+1} &\stackrel{def}{=} sum_i + number_i \end{aligned}$$

The important point we wish to illustrate with the above example is that we have viewed a program as a static object. We have not considered at all how the computation will take place. In the next section we formalise the concept and develop the idea into a paradigm.

4 A definition of a program

Before we can make progress in formalising our concept we need to have a starting point. We need to have a clear understanding of what a program is. In defining what a program is we need one which will be all embracing and not specific to a particular programming paradigm. Very important in this respect is that the definition does not include any concept like ordering that implies needing to reason about the dynamic behaviour of a program. As with formal languages we feel it is important to separate out the definition from the representation or expression of a program.

A particular execution or instance of a program results in values being computed. For each of these values we can associate a tuple $e = (r, v)$ where r is a unique reference and v is the value computed. We shall refer to such a tuple as an *element*. An outcome of a program can thus be represented by a set of elements. We refer to such a set $\Im$ as an *instance*. An *image* ξ_p is the set of all possible instances for a program p. A program is defined in terms of its image.

Our formal definition of a program is:

> An *image* ξ_p is the set of all possible *instances* of a program p
> An instance $\Im \in \xi_p$ is the set of *elements*
> An element $e \in \Im$ consists of a tuple (r, v) where r is a *reference* and v is a *value*
> ξ_p is valid if $\nexists \Im \in \xi_p \ni \exists (r_1, v_1), (r_2, v_2) \in \Im$ and $r_1 = r_2$

The above definition is useful in that it allows us to view a program as a static object. There is no mechanism implied as to how the values will be computed or the order in which they will be computed. It establishes a basis on which to build the formalism in that we have elements that are given meaning in terms of a reference and a value. A program representation becomes the task of defining these elements.

5 Program representation

In this section we look at how we can develop a program representation that exploits the static view of a program based on the above definition of a program. The definition relies on the concept of an element and our proposed representation is one of defining these elements as simple algebraic expressions.

Our representation consists of a set of definitions that define each element or classes of elements. Each of these definitions are independent of the definitions of other elements and the relationships an element has with other elements.

In this introductory paper, we have left out all the formal definitions of a language with its semantics. Instead we look at the examples of the three important forms of element definitions.

Let us consider the simple case of defining an individual element:

$$area \stackrel{def}{=} breadth * width$$

with the semantics

$$\frac{(breadth, b), (width, w) \in \Im \text{ and } (b, w, a) \in *}{(area, a) \in \Im}$$

A class of elements can be defined using universal quantification as follows:

$$sum_{i+1} \stackrel{def}{=} sum_i + number_i$$

with the semantics

$$\text{for any } i \quad \frac{(sum_i, s), (number_i, n) \in \Im \text{ and } (s, n, s') \in +}{(sum_{i+1}, s') \in \Im}$$

Note the universal quantification is implied where there are unbounded indices.

A element can be expressed using partial functions in the following way:

$$\begin{aligned} area &\stackrel{def}{=} length * breadth \textbf{ if } object = rectangle \\ &\stackrel{def}{=} radius^2 * pi \textbf{ if } object = circle \end{aligned}$$

with the semantics

$$\frac{(length, l), (breadth, b), (object, o) \in \Im \text{ and } (o, rectangle, true) \in =, (l, b, a) \in *}{(area, a) \in \Im}$$

$$\frac{(radius, r), (object, o) \in \Im \text{ and } (r, r') \in \textbf{square}, (o, circle, true) \in =, (r', pi, a) \in *}{(area, a) \in \Im}$$

A complete program representation consists of a set of these descriptions. There is no significance to the order in which they are expressed.

Our previous example of finding an average turns out be a representation of a program:

$$average \stackrel{def}{=} sum_{LoL}/LoL$$
$$sum_0 \stackrel{def}{=} 0$$
$$sum_{i+1} \stackrel{def}{=} sum_i + number_i$$

with the following formal semantics:

$$\frac{}{(\text{sum}_0, 0) \in \Im} \tag{1}$$

$$\forall i \quad \frac{(\text{sum}_i, s_i), (\text{number}_i, n_i) \in \Im \text{ and } (s_i, n_i, v_i) \in +}{(\text{sum}_{i+1}, v_i) \in \Im} \tag{2}$$

$$\frac{(\text{sum}_e, s), (\text{LoL}, l) \in \Im \text{ and } (s, l, a) \in /}{(\text{answer}, a) \in \Im} \tag{3}$$

that enables us to reason as follows about the program:

$(\text{sum}_0, 0) \in \Im$ by (1)

$(\text{sum}_1, n_0) \in \Im$ where $(\text{number}_0, n_0) \in \Im$ by (2)

...

$(\text{sum}_i, \sum_{k=0}^{i-1} n_k) \in \Im$ where $(\text{number}_k, n_k) \in \Im \ \forall \ 0 < k < i$ by (2) repeatedly

$(\text{answer}, a) \in \Im$ where $\forall 0 \leq k < i$, $(\text{number}_k, n_k), (\text{LoL}, i) \in \Im$ and

$(\sum_{k=0}^{i-1} n_k, i, a) \in /$ by (3)

Another example is to find the greatest common divisor. The program representation can be developed as follows:

The properties we are trying to preserve are:

$$gcd(x_i, y_i) = gcd(x_0, y_0) \text{ and } x_{i+1} \leq x_i, \quad y_{i+1} \leq y_i$$

and

$$x_i = m.g > 0 \text{ and } y_i = n.g > 0 \ni g = gcd(x_i, y_i)$$

These properties can be preserved as follows:

If $x_{i+1} = x_i - y_i$ if $x_i > y_i$ $= m'.g$ where $m' = m - n > 0$

$= x_i$ if $x_i < y_i$ $= m.g$

$y_{i+1} = y_i - x_i$ if $x_i < y_i$ $= n'.g$ where $n' = n - m > 0$

$= y_i$ if $x_i > y_i$ $= n.g$

then $gcd(x_{i+1}, y_{i+1}) = gcd(m.g, n.g) = gcd(x_i, y_i)$

and $x_{i+1} \leq x_i, \quad y_{i+1} \leq y_i$

From the above reasoning we can define the elements in a program as follows:

$$\begin{aligned} x_{i+1} &\stackrel{def}{=} x_i - y_i \text{ if } x_i > y_i \\ &\stackrel{def}{=} x_i \text{ if } x_i < y_i \\ y_{i+1} &\stackrel{def}{=} y_i - x_i \text{ if } x_i < y_i \\ &\stackrel{def}{=} y_i \text{ if } x_i > y_i \\ gcd &\stackrel{def}{=} x_i \text{ if } x_i = y_i \end{aligned}$$

The final example is that of an insertion sort. The reason for including it is to show that it is possible to express non-trivial problems using the experimental paradigm as well as providing a different kind of problem.

In this example *unsorted*$_i$'s are the unsorted elements, $p_{i,j}$'s are the j^{th} elements in the i^{th} pass, and *sorted* is the final sorted list. A small example of an instance in the program is:

unsorted$_0$=4	unsorted$_1$=2	unsorted$_2$=6	unsorted$_3$=3
$u_{0,0}=4$ $p_{0,0}=4$	$u_{1,0}=2$ $p_{1,0}=2$	$u_{2,0}=6$ $p_{2,0}=2$	$u_{3,0}=3$ $p_{3,0}=2$
	$u_{1,1}=2$ $p_{1,1}=4$	$u_{2,1}=6$ $p_{2,1}=4$	$u_{3,1}=3$ $p_{3,1}=3$
		$u_{2,2}=6$ $p_{2,2}=6$	$u_{3,2}=3$ $p_{3,2}=4$
			$u_{3,3}=3$ $p_{3,3}=6$
sorted$_0$=2	sorted$_1$=3	sorted$_2$=4	sorted$_3$=6

A possible representation of a program for which the above instance is a member is given below:

$$\begin{aligned} u_{i,0} &\stackrel{def}{=} unsorted_i \\ u_{i,j} &\stackrel{def}{=} u_{i,j-1} \text{ if } j <= i \\ p_{0,0} &\stackrel{def}{=} u_{0,0} \\ p_{i,0} &\stackrel{def}{=} u_{i,0} \text{ if } u_{i,0} < p_{i-1,0} \\ p_{i,i} &\stackrel{def}{=} u_{i,i} \text{ if } p_{i-1,i-1} <= u_{i,i} \\ p_{i,j} &\stackrel{def}{=} u_{i,j} \text{ if } p_{i-1,j-1} <= u_{i,j} \text{ and } u_{i,j} < p_{i-1,j} \\ &\stackrel{def}{=} p_{i-1,j} \text{ if } p_{i-1,j} <= u_{i,j} \\ &\stackrel{def}{=} p_{i-1,j-1} \text{ if } u_{i,j} < p_{i-1,j-1} \\ sorted_j &\stackrel{def}{=} p_{EoL,j} \end{aligned}$$

6 Implementation

An experimental language has been designed based the above ideas. A compiler and interpreter for the language have been implemented. A reasonable number of classroom examples have been written and tested using the current implementation.

The current implementation is limited to integer and boolean types. User-defined operators can be defined that can be used recursively. Type checking is not implemented as yet.

The most striking observation in developing the classroom examples was how easy it was to identify a fault in a program, locate the definition of the element that was incorrectly defined and correct the fault. The

whole focus of implementing a program changed to thinking about elements rather than about the order in which data got evaluated.

The paradigm encourages one to spend more time on getting one's program correct before trying to compile and test the program. The close relationship between a computational element and the definition of that element is brought home using this paradigm. This emphasizes the importance of getting this definition correct when one defines it for the first time. Any errors detected during testing highlight the need to get it right the first time.

Even with little previous experience with trying to reason formally about a program, it was possible to come up with rigorous verifications of programs. The simple semantics of the language and the fact that it relies on a relatively small background in mathematics made this possible.

7 Evaluation

This paper started out by questioning our current paradigms and claiming they are much too complex, suggesting that the operational view we have of programs makes programming inherently difficult. We are now testing this claim by seeing how removing this aspect influences programming and if it does in fact result in properties we argue a language should have. This section evaluates whether the paradigm does in fact deliver the expected results.

We accept that one's existing programming concepts make it difficult to evaluate this very different style of programming. Subjectively many are not likely to relate well to this new paradigm and are likely to be skeptical about the argument that it has simpler semantics.

The first property we considered important is that the semantics needs to be simpler. The semantics of a representation are based on straightforward mathematical algebra, universal quantification and modes ponens. The semantics describe the static relationships between elements and each element definition is independent of others in terms of its correctness. We have not had to introduce semantics to handle the way in which a program is executed such as "for looping" or recursion. The semantics do not depend on a current state or an environment and there is no need to have semantics that determines the entity we are reasoning about.

Referential transparency is advocated as one of the advantages of functional programming. This paradigm take this a step further. Each element in an instance has a unique reference. Unlike in functional programming, the reference is not dependent on the environment in which the reference is being used. The benefits of this are important in that when reasoning about an element definition it is explicit as to which elements are being referenced.

The value of the static nature of the paradigm is illustrated by our examples. One is able to start with an example and through a series of generalisations or abstractions develop a program. A program that is described as a mechanism suffers because mechanism working for one case does not necessarilary work for another. Taking into account other cases often results in have to redesign one's mechanism. The static nature of the paradigm allows one to refine the definitions of the elements to handle more cases.

The static aspect of the paradigm allow one to reason about aspects of a program independently of how the rest of the program is executed. A programmer can focus on one element definition without having to be concerned about when or in which environment it holds.

The whole paradigm focuses on a single aspect which is defining elements: elements are defined using one "construct": an element definition. The definition is stated as an algebraic expression commonly used in mathematics. Thus there are very few concepts and there is no need to have a model of how the computation takes place. The result is that we have only one syntax and semantic entity to understand: considerably less than with any other paradigm.

This paper has not gone into the details of the syntax of a language based on this paradigm, but if we look at the examples the notation required is only that required to express indices and algebraic expression. The notation used is borrowed from mathematics that requires some minor modification for text based editors.

Finally two examples are given as to how one can reason about a program. It requires no additional tools, notation or concepts to reason about a program over and above basic mathematics. There is no underlying concepts of computation that impacts on the semantics of the language. The model of computation used by the paradigm has no influence on how one reasons about a program.

8 Conclusions

This paper preents a substantially different programming paradigm, which illustrates that we have as yet not explored all the possibilities that we can use to support us in the task of programming, and in particular with more formal approaches to programming.

We are able to show that it is possible to achieve our goal of developing a paradigm that allows us to reason about a program as a static object. The importance of this is we have established that a programmer does not need to be burdened with the complexity associated with reasoning about dynamic systems.

Based on objective criteria, we are able argue that viewing a program as a static object has major benefits with attractive properties such as referential transparency, well-defined semantics etc. Our personal experience supports this argument. We accept that many may be skeptical and that more convincing empirical evidence is required. What we do claim is that there is strong evidence to support further research along these lines.

This preliminary research suggests that current programming paradigms may negatively impact on formal methods and that formal methods could benefit by exploring new paradigms that facilitate reasoning about programs. More important is that suitable paradigms would require considerable less background and possible skill to tackle programming in a formal way.

References

1. Giannesini F., Kanoui H., Pasero R., and van Canegham M., *PROLOG*, Addison-Wesley, Wokingham England, 1986, ISBN 0-201-12911-6. [2]

2. Turner D.A., 'Recursive Equations as a Programming Language', Ed Darlington J et al, *Functional Programming and its Applications.*, Cambridge Univerity press, 1982, ISBN 0 521 24503 6.