Spreadsheets and programming

Dominic Gehring

Department of Computer Science, University of Warwick, Coventry, CV4 7AL

Abstract

Spreadsheets provide powerful but simple control constructs. Possibly the most important source of control in a spreadsheet is the maintenance of dependencies between cell values. In research at Warwick this principle of automatically maintaining dependencies has been used in a number of more general modelling notations. This paper suggests that using this spreadsheet principle as the basis of a programming paradigm would be useful for prototype generation, user-interface design and end user programming. A generalisation of dependency maintenance is introduced to allow the construction of *higher-order* dependency structures, such as dependencies between cell formulae. The introduction of higher-order dependency means that more dynamic systems can be made whilst remaining within the framework, and takes us nearer to a more general programming paradigm.

Introduction

In this paper I discuss how spreadsheets may suggest better ways of programming. In particular I put forward a programming paradigm based on the principle of dependency maintenance. I begin by asking if we need more ways of programming and how we can test the merits of different approaches. Through empirical studies of users we can see that spreadsheets possess some properties that are desirable for certain programming tasks. I will introduce *definitive* (definition-based) notations, in which dependencies are represented by scripts of definitions, and describe my own contribution to this work. I conclude by assessing the advantages and disadvantages of dependency-based systems, and suggest areas where they could be usefully applied.

Background or “Oh no, not another paradigm”

There are many programming paradigms, indeed some people would say too many. Each paradigm has its own focus and may claim advantages over other approaches. This begs two questions. Firstly, is there a single *best* paradigm? This would certainly be desirable as it would give a unified framework for all software development. However the position that I adopt is that no paradigm can effectively cover the full spectrum of programming tasks: from throw-away prototypes quickly produced by novices, to large-scale fault-tolerant systems developed and maintained over many years by teams of experts. I believe that there will always remain a fringe of programming tasks too varied to be effectively tackled using the same approach.

Secondly, do we need any more programming paradigms? Well we know that programming remains a difficult process where improvements need to be made. Many advances can be made within the existing paradigms, such as improving their programming notations. However there is still scope to look for more effective ways to program, particularly for the fringe of programming tasks, such as end user programming and interface design.

The notion of being able to program something *effectively* is purposely vague: the wide spectrum of programming uses means there are many criteria by which a programming process can be judged. How do we know if a programming paradigm satisfies some criteria? Although we will have our intuitions, we will not know if they are true, or the extent of their truth. Therefore we use applied psychology to check the truth of our intuitions and possibly get results that are not immediately obvious to experts in the field [Green 1980]. In my work I have taken the results of this kind of research to suggest better ways to program: i.e. empirical studies of users show that spreadsheets possess many desirable properties.

Spreadsheets

The spreadsheet was the first application to give the power of computing to the masses. It gave non-programmers a way of working with numbers without needing to learn FORTRAN, Basic or even simple ideas such as iteration and control. A spreadsheet is a grid of cells, that can be thought of as variables. A cell may store a string, a numeric value or a formula returning a numeric value. Formulae express relations between cell values by referencing the values of other cells. If we change the value of a cell referenced by a formula, then that formula will be automatically recalculated. In this way a formula establishes a *dependency* between scalar values, i.e. the value returned by the formula depends on the values it references.

Much of the spreadsheet's advantage comes from the high-level task specific nature of their programming primitives. These operations are accessible to the novice user because they are already part of the user's problem domain. The primitives are less flexible or general than those of a general programming language, but the basic operations do tasks that the users want to do. Nardi found that spreadsheet users normally use less than ten functions in their models, yet are highly-productive (cf. functions in C) [Nardi 1993]. Nardi also notes that it is important from a motivation standpoint for users to attain early success and that spreadsheets enable this.

The spreadsheet provides a good marriage of text and graphics. Complex formulae can be written very compactly in the text-based formula language, leaving most of the screen to display program state. The two-dimensional grid is natural for working with numbers: rows and columns represent the main parameters for the model and cell ranges can represent subparts of the model. The grid gives the user a 'blank canvass' on which they can structure, format and segment their model, thus aiding comprehension and navigation.

Each cell in a spreadsheet represents an entity in the real world phenomenon being modelled. Therefore the spreadsheet model can be understood with only knowledge of the problem domain, and because the values of cells are maintained the spreadsheet can always be verified against the phenomenon (cf. imperative programming). Furthermore, dependency maintenance allows the user to do "what-if" experiments with the system throughout its development and in uncircumscribed ways.

"Many empirical studies of novice programmers have found that beginning users often have great difficulty understanding control constructs in programming languages" [Nardi 1993]. Spreadsheets avoid this problem by having greatly simplified control constructs that require little programming effort. The most important source of control in spreadsheets is the maintenance of dependencies. The flow of control is given implicitly in the interrelation of users' formulae. The spreadsheet takes care of the mechanics of updating cell values, leaving the user to concentrate on the domain-related problem of describing the dependency structure.

Other aspects of control are also simplified in spreadsheets. A conditional statement in a spreadsheet is simple to understand because its effects are local to the cell, for example IF(B12>100,B12,100) defines the cell to be the value B12 if greater than 100, and return the value 100 otherwise. To iterate a command the user need only specify the range of cells to iterate over, for example SUM(C1..C8) would sum cells 1 to 8 in column C. Another form of iteration can be achieved by copying a formula over a range of cells, so that the cells referenced by copies of the formula change relative to position.

The key to the spreadsheet's success is its task-specific nature. The principles of having simple primitives, a grid structure and dependency maintenance all combine beautifully to produce the perfect tool for end users who are working with numbers. The problem is, can we apply the same recipe to other domains? In some cases we can, for example [Levoy 1994], where a spreadsheet-like framework is used to maintain dependencies between images. However in general it will be inappropriate to use the same combination of principles. Instead I believe we can learn a lot from spreadsheets by applying the individual principles to other domains, for example using simple task-specific primitives where generality is not needed. In the next section I describe how our work at Warwick uses another of the spreadsheet's principles.

Definitive notations

Definitive (definition-based) notations use the spreadsheet principle of dependency maintenance to model real-world systems [Beynon 1985]. We begin modelling by identifying things to observe in the system. In a definitive script, a variable is introduced to represent each of these observables. A variable can either be given an explicit value or defined in terms of other variables (like a formula in a spreadsheet). The definition of a variable corresponds to establishing a relationship between that observable and those referenced in its definition. These relationships form uni-directional dependencies which are maintained, as in a spreadsheet. In this way we can create a metaphorical representation of an external state. Definitive notations are a more general way of modelling than spreadsheets because they are not constrained by the grid interface and data type: indeed a spreadsheet can be thought of as a definitive notation. They are generally task-specific, for example we have a notation for line drawings, with a data type and interface appropriate to that domain.

In my mind there is a blur between modelling and programming. For example it is often stated that all programming is modelling [Kent 1978, Meyer 1988]. Therefore I am interested in seeing how well this modelling method can be applied to more general forms of programming than line drawings and accounts. Indeed some moves towards this have already been made. For example, EDEN is a hybrid programming language supporting both definitive and procedural language elements [Yung 1994]. It was originally intended to be an implementation tool for the development of definitive notations, but in practice has been used for more general programming tasks.

The hybrid nature of EDEN means that many of the advantages of having dependency maintenance are lost through use of the procedural language elements. It is desirable to have a purely definitive general programming language (with no procedural elements), however traditionally the dependency structures that we use are fixed, limiting the models we can make. The focus of my work has been to introduce the notion of higher-order dependency to our modelling method, and thereby allow the construction of more dynamic models.

Higher-order dependency

A good introduction to higher-order dependency is the spreadsheet-like *tables* notation that I have developed. Tables use the spreadsheet's rectangular grid of variables and the property of relative change to enable the construction of dependencies between dependencies. A higher-order dependency in a table is similar to copying a formula in a spreadsheet, except that the copy is maintained.

When you copy a formula from one cell to another the cells referenced within the formula change relative to their destination.

	A	B	C
1	5		
2	6	15	
3	A1+A2 =11	37	
4		B2+B3 =52	

In the example above the formula in A3 has been copied to B4. In B4 you can see that the expression is still the same, i.e. the sum of two things, but the cells referenced within it have changed relative to their position. In a sense there is an underlying *template* within a formula. In the example above this template formula would be *sum the two values directly above me*. When the formula was copied from A3 to B4, we could say that B4 depended on the formula in A3. It is through maintaining this kind of dependency that we can establish higher-order dependencies. For example, if A3 were to be changed to A1*A2 then that change would be propagated, and B4 would become B2*B3.

As motivation for higher-order dependencies we can imagine a spreadsheet in which at the bottom of many columns we want to calculate the same aggregation of the column's values. We could set it up by entering the formula at the bottom of the first column and copying it to the others. If at a later time we wanted to refine the formula we would change it in the first column but must then manually copy it to each of the others again. By creating a dependency rather than copying, we could better represent what we really mean, i.e. that the aggregation formulae should all be consistent and follow the definition of the first one. In this way we can set up dependency-based models that are more dynamic in their behaviour.

Detailed discussion of higher-order dependencies is beyond the scope of this paper, however I shall hint at the key ideas. In the existing definitive notations, when a variable is referenced by some definition, the value of the variable is used. However in the example above the variable's formula is used. I have showed that by referencing a variable's formula, rather than value, we can introduce a higher-level of definition. By this principle we can always reference a variable at its highest-level of definition and thereby introduce a level above this and therefore arbitrarily many levels. We can see that values, dependencies and higher-order dependencies are similar in many ways but just at different levels of abstraction. For example we can use arithmetic operators to combine references to formulae, just as we can do this with references to values.

The tables notation relies on a grid structure, and more specifically the property of relative change, to make higher-order definitions. Relative change gives an essential variation in the copied formula: essential because if the formula did not vary, it would just evaluate to give the same value. Outside of a grid structure we can supply the variation explicitly and therefore still establish higher-order dependencies in a script-based language.

The advances in dependency maintenance described above are matched by advances in our modelling methodology. Our modelling method began by identifying a set of observables, monitoring their values and describing the patterns of change between these values as dependencies. I suggest that our modelling method be extended so that we then consider dependencies as higher-level observables, i.e. we monitor these dependencies and describe patterns of change between them as dependencies at a level above. We would then continue this process at higher-levels, until we believe that our model faithfully represents the phenomenon being modelled.

When modelling a phenomenon there will in general be many ways to represent it using different numbers of levels. The way in which the model is constructed reflects the modeller's perspective. The role of higher-order definitions is to reflect the high-level observables perceived by the modeller. The problem solving process encouraged by this approach is to understand a specific problem instance, then generalise to account for the relationship between different instances, and so on until the system has the desired behaviour.

Assessment of definitive notations

To recap it would be useful to see what the advantages and disadvantages of definitive notations are. Firstly we can see that they retain many of the advantages of spreadsheets, namely:

- Through dependency maintenance users need only concentrate on the high-level task of describing the dependency structure, and can ignore the management of value updates.
- The model can be understood and verified through reference to the real world phenomenon being modelled.
- The model can be developed interactively (cf. compiling C code).
- The user can experiment with the model in uncircumscribed ways.
- The effects of changing a definition are localised and so easier to comprehend.
- When definitive notations are applied to specific problem domains they can retain the advantage of simple primitives.
- Definitive notations can retain an intuitive interface, however this depends on the problem domain (it would be difficult to match the success of the spreadsheet's interface).

Finally it would be most impressive to claim that dependencies are a natural way to model. This is hard to say, not least because all paradigms tout the naturalness of their approach. Although my intuition is that in many problem domains our understanding comes through establishing dependencies between entities, the truth and implications of this statement would need to be proved. (Can anyone help?)

I have described the many advantages of dependency-based systems, however there are also many disadvantages. These disadvantages weed-out many potential application domains, however several uses of definitive notations remain. The overhead of dependency maintenance means that definitive systems are relatively inefficient, a problem shared by all declarative approaches. Most applications in which speed is important could be better solved using optimised sequential code.

A definition is similar to an object in OOP, in that a definition is a program module with a real world counterpart. A definition is at a lower-level of abstraction than an object. Although this gives the advantage that no tedious sequential code is needed beneath this level of abstraction, the simplicity of the abstraction gives several disadvantages. The fine-grain modularity that definitions provide mean that it is difficult to get a more global view. Also in current notations the dependency structure is fixed.

A definitive system is very open, in that any variable can be related to any other. The principle of propagating change through the dependencies can also have the disastrous effect of propagating errors and changes in specification. This is what OO methods try so hard to avoid [Meyer 1988]. Hence debugging in definitive notations is notoriously difficult, although tools to visualise the dependency structure could help.

Furthermore I believe that large-scale models and distributing work among programmers is precluded by the openness of our current definitive notations, another serious disadvantage.

Conclusion

The section above shows that although definitive notations have many points in their favour, their use is constrained by the problems of dependency maintenance. In my opinion their use is limited to only relatively small models with at most a few developers, so this approach would not be appropriate for most software engineering tasks. Efficiency is another constraint, meaning that most programming tasks could be more efficiently implemented using optimised sequential code. An interesting exception is programming user-interfaces, an area that commonly uses different techniques from conventional software engineering. In this area dependencies between the internal representation and visualisation on screen are very important. We would expect the expensive overhead of dependency maintenance to be an issue too, however, in this case all values must be kept up-to-date to be displayed on screen, and hence few optimisations could be made.

The advantages of using dependency maintenance include improved simplicity, interaction and experimentation. End user programming is a natural use for definitive notations. In the same way that spreadsheets are designed for professionals to work with numbers, we could develop other task specific definitive notations that possess similar advantages. The properties of interaction and experimentation are extremely useful in *Empirical Modelling*, a flexible method of modelling novel phenomena that is based on experimental experience. Dependency maintenance is also helpful when rapidly prototyping systems, because the efficiency and the restricted size of models are less important than the speed with which the system can be implemented and changed. Through the examples given above we can see how lessons learned from spreadsheets can help us in some programming tasks.

References

[Beynon 1985] Beynon W. M. “Definitive Principles For Interaction”, in Proc hci'85 CUP Sept 1985

[Green 1980] Green T. R. G. “Programming as a cognitive activity”, in Human Interaction With Computers, edited by Smith H. T. and Green T. R. G., Academic Press, London, 1980.

[Kent 1978] W. Kent “Data and Reality”, North-Holland Publishing Company, Amsterdam, 1978.

[Levoy 1994] Levoy, M “Spreadsheets for images”, Technical Report CSL-TR-94-607, Stanford University, 1994.

[Meyer 1988] Meyer B. “Object-Oriented Software Construction”, Prentice-Hall, New York, 1988.

[Nardi 1993] Nardi, B. A. “A Small Matter of Programming: Perspectives on End User Computing”, The MIT Press, Cambridge, MA, 1993.

[Yung 1994] Yung Y. W. “EDEN Handbook” Department of Computer Science, Warwick University, reprinted 1994