DRAFT

Definitions for the specification of educational software

Meurig Beynon
Keith Halstead
Steve Russ

Introduction

This brief paper outlines a novel approach to the problem of achieving greater portability of educational software. It is primarily concerned with exploring a method of describing such software at a high level of abstraction in a way that can assist its implementation on different machines, and in different environments and programming languages. Following the best contemporary practice in commercial software development, our approach emphasises abstract very high-level specification, and may be contrasted with approaches that focus upon prescribing standards for software and hardware, and on developing translation tools to ameliorate specific problems concerning incompatible machine codes and capabilities. The portability problem we address is not of course confined to educational software, but the novel ideas we intend to apply are particularly promising in that context. To be more specific, our methods are probably best-suited for "specification of relatively small programs for user-centred abstraction and portability" rather than "specification of large programs for automatic verification".

General background

It has long been recognised that the satisfactory abstract representation of computer programs is an essential problem that must be solved to achieve software portability. Clearly, what is required is some method of describing programs that makes it possible to capture generic characteristics at many different levels of abstraction. For instance, at some level we should like to be able to regard two programs that perform the same task - such as displaying an image of a steam engine and simulating the motion of its pistons - as equivalent, but at a lower level of abstraction distinguish between a program that incrementally updates the graphical display from a program that repeatedly redraws the entire screen.

Many of the problems of translating programs written in conventional languages, such as Basic, C or Pascal, derive from the fact that they make use of a *procedural* programming paradigm. That is to say, the emphasis is upon spelling out in detail the precise steps needed to perform a computational task: upon *how* a computation is performed rather than upon *what* is being computed. A procedural program characteristically prescribes the transition from each computational state to the next, each basic transition corresponding to the assignment of a new value to a variable. An important step in understanding how to achieve a more abstract specification of programs was taken by *functional* programming. A functional program in essence consists of a set of formal expressions that represent functions in the traditional mathematical sense, together with an appropriate function evaluation. For instance, the sequence of images required to simulate a steam engine can be specified as a function S of parameters such as the dimensions of the piston rods, and the diameter of the wheels etc, and the simulation itself is then the "value" returned when this function S is evaluated. Notice that this sequence of images is viewed as a time-invariant abstract result of an evaluation; there is no place in pure functional programming for the concept of computational state.

The advantages of using functional rather than procedural principles in software specification for portability may not be immediately apparent. One particularly important consideration is that pure functions can be easily implemented and checked for correctness *in isolation*. Unlike procedures, that frequently operate through side-effects, functions have no hidden meaning other than to specify

what result will be returned given a specified set of input parameters. Once implemented, functions can be freely re-used in other contexts, so that it becomes possible in principle to build up libraries of basic building blocks. The steam engine simulation, for instance, could be written in such a way as to incorporate basic functions such as are required to relate the position of a point on a wheel to the wheel motion.

Functional programming principles play a very important role in formal specification techniques. The sophistication of modern functional programming environments is such that very complex functions can be routinely specified in an appropriate abstract mathematical notation, and automatically evaluated by powerful interpreters. An important point to note is that - in principle - the programmer need never be concerned with how the evaluation of a specified function is carried out: in this sense, the goal of abstracting away from the particular mode of execution ("how the program works") to express only the essential objective of the program ("what is to be computed") is then achieved.

Despite its merits, there remain many problems with the functional programming approach. Some of these will certainly be alleviated over time by advances in software and hardware technology, but others may reflect more fundamental difficulties. To apply functional programming principles to the generation of portable software, the obvious path is to adopt a functional programming language as a specification language, and to equip all machines with suitable software to interpret such a specification as a program. There is no doubt that - educational policies apart! - the software needed to support sophisticated functional progamming systems will be routinely within the range of the computer hardware that will be found in schools over the next decade. What is more contentious is whether the aspirations of the functional programming school can be fully realised on technical grounds.

The problem of replacing "how to do" by "what is to be done" is that it abstracts away from the resources available to perform a task. It seems fanciful to suppose that an automatically generated function evaluation can be ideally adapted to exploit the special characteristics of particular machines and environments, and functional programming has so far yet to satisfactorily address the more modest problems of guaranteeing good practical performance under all conditions. What is more, the suitability of functional programming methods for applications involving interaction and concurrency is yet to be proven. At the root of this difficulty is the problem of bridging the gap between functional specifications and traditional programs and protocols that implicitly depend heavily upon manipulation of computational state. In practice, pure functional specifications appear best suited to sequential non-interactive applications, and support abstraction from programs by "function performed" far more effectively than abstraction based upon procedural similarity between programs.

Principles of the definitive (definition-based) programming paradigm

The definitive programming paradigm was initially conceived as a means of modelling user-computer interaction. The central idea behind this model is that the state of a suspended dialogue can be very effectively represented by a system of variable definitions. The simplest example of such a use of variable definitions is a spreadsheet from which the screen interface has been removed: in such a system the variables correspond to screen locations and the definitions to the functional relationships between variables. For instance, the profit **p** of a company may be defined by the difference between income **i** and expenses **e**, and the functional relationship between the integer variables **p**, **e** and **i** specified by the definition **p=i-e**. Note that this definition is very different in kind from a procedural assignment **p:=i-e** such as might occur in PASCAL; if the values of **i** or **e** are subsequently reassigned, then the value of **p** is also changed.

The development of a programming paradigm based on definitions has been in progress in the Computer Science at Warwick since 1981. Over this period, the significant problems of generalisation that are needed to extend definitive principles to support interaction in which the

variables represent much more complex entities, such as graphical objects, display interfaces, data base relations etc have been solved. The first step towards this generalisation is the adoption of an underlying value algebra - analogous to arithmetic ie the algebra of integers for the spreadsheet - that contains complicated explicit data structures, such as recursive **list**s or configurations of **point**s and **line**s in the plane, together with operators, such as concatenating lists, or rotating geometric configurations, that act on these values. The operators of this algebra are no more than pure functions, just as in functional programming - the significant difference between definitive programming and functional programming being in the way in which these functions are used.

In functional programming, every program has to be performed through a function evaluation; the nature of the functions used is accordingly generally rather complex. This is quite satisfactory from a mathematical perspective, since there is a most elegant logical theory - the lambda calculus - that makes it possible to specify the most convoluted functions, including higher-order functions - resembling integration or differentiation in the calculus - that take a function as an argument and return a function as a result. In describing the steam engine simulation, higher order functions would almost certainly play a significant role. In a definitive program, the functions that are employed are necessarily more primitive: they correspond either to basic operators in the underlying algebra, or to derived operators specified by the user. In a spreadsheet, basic operators are addition and multiplication; a derived user-defined operator a function such as might return the interest payable on a sum of money invested at a particular interest rate.

The merits of definitive programming as a vehicle for describing interaction are by now reasonably well-understood. The first non-trivial illustrations of its use are to be found in prototype graphical applications that have been developed: the one a line-drawing package, the other an environment intended for representing group-graphs ("Cayley diagrams") and examining their properties. What is not immediately apparent is how - if at all - definitive principles can be applied to general-purpose programming. The solution to this problem is conceptually quite simple: systems of variable definitions have to be used to describe the intermediate states through which an abstract machine passes in carrying out a computation. In effect, we imagine that the programmer instructs the computer to carry out a computation as if it were conducting a dialogue, or - to be more precise - as if it were allowing agents to act upon a system of definitions according to an appropriate protocol in such a way that each changes the current values of parameters just as the user manipulates a spreadsheet. A pleasing consequence of adopting this style of programming is that the representation of the intermediate state of a suspended computation is in principle transparent to the user: there is no plethora of values to be analysed, but a system of variable definitions.

Characteristic of the distinction between functional and definitive programming is the fact that definitive programming provides strong support for the representation of specific knowledge. A definitive program may include an abstract function that computes interest on investments at given rates, but there may also be variables to represent the specific interest on specific investments at specific rates. It is the cumulative power of systems of variable definitions to represent a context for interaction that gives definitive programming its expressive range. In describing the steam engine simulation, for instance, the functional relationships between the positions of the wheels and pistons can be represented by a system of variable definitions in which the algebraic operators used are quite elementary: the complexity of the model derives from the fact that there will be variables to represent each of the explicit components of the steam engine, and that the values of these variables are subtly interdependent.

It is instructive at this point to reconsider the philosophical considerations that motivated the development of functional programming ie the desire to escape from the tiresome analysis of anarchic systems of variables whose values are frequently reassigned in intricate ways. Notice that, in principle, there is nothing to prevent a "definitive programmer" from writing a steam engine simulation that is essentially a BASIC program. After all, a definitive program incorporates variables whose values can be explicitly defined and redefined: to describe the steam engine, it suffices to invent enough parameters to represent the precise state of all its components at any given

time, and to consider transitions between states such as respect the appropriate functional relationships. That is to say, if the variable that represents the point of contact of the first driving wheel is displaced by a distance x, then so also is the point of contact of the second driving wheel etc. The fact is that to represent the steam engine in such a manner within a definitive framework is rather absurd: there are functional relationships between the values of parameters, and these can be explicitly described. Within the definitive programming framework, "good programming style" is synonymous with identifying and exploiting the appropriate semantic relationships.

The latter point deserves more careful examination. In their zeal, the advocates of functional programming have deemed the use of assignments to procedural variables (such as **p:=i-e**) unnecessary and inappropriate. For the steam engine simulation, the effect of such a discipline is to render the representation of the current disposition of the wheels and pistons problematical, though the relationships between these components can be very satisfactorily described. In procedural programming, the converse is true: the current values of all the parameters that describe the steam engine captures its current state precisely, but fails to give explicit expression to the functional relationships between these parameters. It is unsurprising that the focus of much recent research in functional programming has been upon elegant ways in which state information can be infiltrated into pure functional specifications, and that historically the emphasis of much research into procedural methods been concerned with techniques, such as the use of invariants or object-oriented methods, that provide support for systematic updating of procedural variables in such a way as to preserve functional relationships. Definitive programming supports both the representation of functional relationships and of state information: good definitive programming style involves identifying where it is appropriate to use functional relationships, and where to use explicitly defined parameters.

The development of the programming tools needed to support definitive programming is still at a relatively early stage, but the main principles are by now quite clear. The focus of current research is on applying definitive principles to the modelling and simulation of concurrent systems, and to the development of support environments for CAD [2,3]: both challenging programming problems for which conventional programming paradigms have either proved inadequate (or - perhaps - have yet to be proved adequate). In both contexts, definitive programming has offered a significant new perspective from which to address well-established problems, and suggested new ways to deal with interaction and concurrency in particular. It seems likely that its application to issues of software portability will provide equally interesting insights. A particular attraction of addressing the issues of software specification for portability using definitive principles is that it permits functional and procedural concepts to co-exist amicably within a single framework. By using implicit variable definition, and sophisticated operators, a high degree of abstraction is achieved (as in a functional specification); by introducing many independently defined variables, it becomes possible to abstractly describe procedural activity in as much detail as is required.

Applying definitive principles to software specification for portability

As explained above, our approach to portability will focus upon abstract specification of programs. Since we are primarily interested in educational software, we shall work within a general framework in which the computer is in some way primed with "knowledge about a problem domain" and - at any given time during an interaction - with specific knowledge about particular problems, and there is a protocol by which the user is allowed to view and manipulate this internal knowledge. It will be difficult to describe the nature of the knowledge and the form of the protocol in general, since this will be strongly influenced by the particular application, but we shall seek a programming language independent way of representing the information about the problem domain, how it is to be viewed by the user, and how it can be manipulated by the user. The aim will be to use variable definitions of the sort introduced above for this purpose. Indeed, we intend to use such definitions to represent all the principal aspects needed to specify particular programs (eg information about the abstract problem domain, the nature of the display interface, the status of the user-program interaction). To do this, we shall make use of principles similar to those proposed

for a CAD support system [3], incorporating special purpose definitive notations that can be used to express definitions with different roles to play within the model. For instance, in software that is intended to teach pupils about linkages, we should need to be able to define eg geometric relationships between the wheels and pistons of a steam engine, the current status of the display, the present status of the interaction, the profile of the user.

A simple example of a definitive specification is given below. Definitions have the virtue of supporting modularity, and allow easy separation of concerns. For instance, it would not be difficult to exchange one system of screen textual display definitions for another to describe a different display interface component. On the one hand, we might use a single variable to represent the entire content of the screen, on the other, we might introduce a system of variables to represent specific areas on the screen that can be independently addressed and updated. Separation of concerns is conceptually simple primarily because sets of definitions can generally be juxtaposed and split into independent categories without affecting the meaning of a particular definition. An important feature of a system of variable definitions is that the dependencies and interrelationships between variables are made explicit, so that potential interference between definitions can be readily identified.

At the heart of the abstract specification of a program is a set of core definitions that captures the basic concepts and relationships in the problem domain. This system of definitions in effect describes the semantic framework within which the interaction takes place. In effect, if at any stage the user-computer interaction is terminated, the core definitions represent the internal problem-specific information that is to be stored. We can distinguish three fundamental ingredients in the way in which the user interacts, whether in response to the problem-specific state as represented by the core definitions, or in such a way as to affect this state:

input - what mechanisms the user employs to modify the core definitions: eg what textual input is expected, what mouse gestures are significant

output - how the information represented in the core definitions is conveyed to the user: the nature of the graphical and textual displays

control - what protocol governs the interaction between the user and the program.

The technical details of the representations we shall use are beyond the scope of this brief paper, but are illustrated to some extent by the example below. Both the input and output aspects can be satisfactorily described by systems of definitions. The control aspects require a more sophisticated framework: we propose to use a process control notation based upon the use of definitions and resembling LSD [1]. Within this model, we can capture the most important abstract features of the interaction as seen by the user: viz the user's oracle variables (what he/she knows about the core state through the output channels), and state variables (what he/she can conditionally change within the core state, subject to the constraints of the specified protocol.)

To summarise: the advantages we anticipate from the use of definitive specification of educational software for portability are:

(1) support for separation of concerns, allowing easy transposition of one system of definitions for another to change the characteristics of eg the display interface

(2) scope to describe both abstract functional relationships and specific procedural features within the same abstract framework

(3) the representation of software in terms of functional ingredients that have the merits of pure functions over procedures, but are much simpler than complete functional programs

(4) a context within which libraries of standard functions can be linked to an automatic program generator to create efficient prototypes from a definitive specification.

Illustrative example

As a simple illustrative example, we consider the program JUGS: a simple simulation in which the user is presented with two jugs of given integral capacities and asked to determine whether it is

possible by systematically filling and emptying jugs to realise a particular target quantity. The definitions required to specify the simulation can be split into three classes: core definitions, describing the characteristics of the jugs and their current status, display definitions describing the way in which the form of the screen display reflects the current status of the interaction, and definitions of control parameters that determine the user protocol.

The core definitions for the problem are:
capA/B - capacities of the jugs,
contentA/B - content of the jugs,
target - quantity required
- these have explicit values that can be changed by the user subject to a protocol, and
fullA/B - boolean variables that are defined by the current content and capacity of the jugs.
Thus fullA = contentA==capA.

The display definitions describe the three ingredients of the screen display: a display showing the current status of the jugs (a picture showing how full each of the jugs is at any given stage), a menu informing the user of what options are available (eg that it is possible to fill A provided that A is not full), and a field to indicate the current status of the interaction (waiting for user input, registering an invalid input and awaiting input, registering successful attainment of the target). In a naive interface model, such as is appropriate for instance when the screen comprises an array of characters, and all screen updating has to be done by refreshing the entire screen, the values of the display variables will be taken to be arrays of characters, and there will be standard operators - such as "juxtapose display A and display B left to right" - to compose such arrays into displays. For this purpose, the appropriate variable definitions are:
jugA/B - definitions describing how the jugs are displayed,
menuform - a definition given the format of the menu display,
status - a definition to describe the state of the interaction.
These definitions will be formulated in terms of problem specific operators, that are needed to describe just what a jug should look like for instance, but there may be more general purpose operators such as describe a general menu display, in which the fields that are highlighted as valid options are specified as parameters to the display.

When both core definitions and display definitions are in place, it becomes possible for the user to interact directly with the program in such a way as to change the key parameters and see the corresponding changes to the display as expressed by the functional relationship between the internal status and the display interface. Changing the content of the jug A immediately changes the level of liquid in jug A as depicted on the screen. Not all the changes that the user can make in this way are appropriate; it is possible for the user to set parameters in bizarre ways, so eg that the content of a jug exceeds its capacity. To complete the specification of the JUGS program, it remains to introduce control information so as to establish the appropriate protocol for the user-computer interaction.

In a definitive programming context, the specification of control depends upon enriching the current state of the interaction through the introduction of appropriate control variables. In the JUGS program, we must distinguish between the situation where the program is awaiting further input and when the target has been attained for instance. As briefly explained above, a definitive program is conceptually a dialogue between several participating agents, each of which - like the user - acts according to a specified protocol. The control variables required will normally be explicit parameters set by the agents. The LSD notation [1] (a notation for specifying concurrent systems developed in collaboration with British Telecom) provides a methodology for describing the control information. In effect, each agent is able to observe certain variable values (its "**oracles**") and to act conditionally upon other parameters (its "**states**"). For the JUGS program, the user has as an **oracle** a control variable "updating" that indicates whether or not input is currently awaited, and has a **state** variable "input". The user exercises conditional control over the value of the variable "input"; provided that "updating" is false, the user can set the value of input to indicate the selection

of a menu option. The complementary agent is an updating agent that has the variable "input" as an **oracle**, and in the appropriate context updates the core parameters in accordance with a selected menu option whilst setting a flag via its **state** variable "updating".

An executable version of the JUGS program, written for the EDEN interpreter - developed at the University of Warwick as an "Evaluator for Definitive Notations" - is attached. This program is rather less abstract than a definitive specification might be; the control in particular is specified in an ad hoc fashion, rather than written in the LSD notation. Most of the main features described above are nevertheless easy to distinguish from the program: the core definitions (target, capA etc), the display definitions (jugA **is** jugdisplay(height, capA, widthA, contentA) and status **is** ...etc), the control parameters (error, updating), the problem specific user-defined operators (**func** jugline, jugdisplay etc), and the generic operators (**func** displayright etc). (For convenience, the generic operators are listed in a separate file from the problem specific operators, to indicate the kind of distinction to be expected between special-purpose functions and library functions.) Some points are of particular interest: the simple nature of the core definitions - comprising just 5 parameters; the relationship between the display definitions that present the values of the variables capA/B and contentA/B to the user, and the variables menuform and status that give cues for the interaction; the way in which the user can only act indirectly to modify the core definitions, according to the protocol between the user() and the update() agents.

Further comments and directions

The idea behind the definitive approach to specification we have described is that the issues of portability can be addressed by implementing a library of fairly simple general purpose operators, and writing abstract specifications

(a) that can be tailored to the specific characteristics of the particular medium for implementation that is available, and the techniques to be used

(b) from which a program can be derived simply by translation of an appropriate set of problem-specific functions that are relatively simple in form.

To justify (a), we should like to demonstrate that abstract specifications can be written for a specific educational program so that these can be targetted for different machine and software tool environments. To justify (b), we propose to develop a software tool - a relatively simple modification of the EDEN interpreter - that can translate a definitive specification into a skeleton program in a target language such as ISO PASCAL or BBC BASIC in such a way that only the appropriate operators remain to be explicitly written.

References

1. M Beynon, "The LSD notation for communicating systems", UW/CS RR#87, 1986
2. M Beynon, "Definitive principles for interactive graphics", NATO ASI Series F:40, 1083-1097
3. M Beynon, "A definitive programming approach to the implementation of CAD software: 1"
(to appear in Intelligent CAD Systems 2: Implementational Issues, Springer Verlag)

```
func max { return $1<$2 ? $2 : $1; };

target=1;
capA = 5;
capB = 7;
Afull is capA==contentA;
Bfull is capB==contentB;
contentA = 0;
contentB = 0;

/* Specifying the jugs display as a list of strings */

height is max(capA,capB)+2;

func jugline    /* specifying a line of a jug display */
{
        /* $1 is the line number of the display, $2 is height of jug */
        /* $3 is the width of the jug, $4 is the content of the jug */
        /* line number 0 corresponds to the base of the jug */
        auto c,r,s,t;
        r = repchar(' ',$3-1);
        s = repchar((($1>$4)||($1<0))?' ':(($1==0)?'|':'*'),$3);
        c = (($1>$2)||($1<0))?" ":"|";
        t = r // c // s // c // r;
        return t;
}

func jugdisplay
{
        /* $1 is height of display, $2 is height of the jug */
        /* $3 is the width of the jug, $4 is the content of the jug */
        /* func returns list of strings representing the display of the jug */

        auto s,i;
        s = [];
        for (i=$1; i>=0; i--)
        {
                append s,jugline(i-1, $2,$3,$4);
        }
        return s;
}

proc display : jugA, jugB, menuform, stat, targt
{
        auto s;
        s = displayright(jugA, displayright(jugB, displayright(targt,stat)));
        display_list(displayabove(s, menuform));
}

widthB = 5;
widthA = 5;
jugA is jugdisplay(height, capA, widthA, contentA);
jugB is jugdisplay(height, capB, widthB, contentB);
menu is ["Fill A","Fill B","Empty A","Empty B","Pour A to B","Pour B to A"];
menustatus is [valid1, valid2, valid3, valid4, valid5, valid6];
menuform is menudisplay([menu,menustatus]);
valid1 is !Afull;
valid2 is !Bfull;
valid3 is contentA != 0;
valid4 is contentB != 0;
valid5 is valid3 && valid2;
valid6 is valid4 && valid1;

/* specifying the control information */

error=0; updating=0;
```

```
finish is ((contentA==target)||(contentB==target))&&!updating;

func avail
{
        /* indicates whether the menu option with parameter $1 is open */
        auto t;
        t = menustatus[$1];
        return t;
}

proc update : input
{
        error = 0; steps = 0; updating =1;
        while (avail(input)) {
                switch (input) {
                        case 1: contentA++; steps++; break;
                        case 2: contentB++; steps++; break;
                        case 3: contentA--; steps++; break;
                        case 4: contentB--; steps++; break;
                        case 5: contentA--; contentB++; steps++; break;
                        case 6: contentB--; contentA++; steps++; break;
                }
        }
        if (steps==0) {
                error = 1; updating = 0;
        }
        else updating = 0;
}

status is (error)?"invalid option": ((updating)?"updating":"awaiting input");
totstat is (finish) ? "Success!" : status;
targ is ("Target is " // str(target) // " : ");
stat is messdisplay(height,totstat);
targt is messdisplay(height,targ);
```

```
proc display_list
{
        /* display a display_list $1 */
        auto i;
        for (i=1; i<=$1#; i++)
        {
                writeln($1[i]);
        }
}

func menudisplay
{
        /* construct display list for menu $1[1] with associated status $1[2] */
        auto i,s,l;
        s = "";
        for (i=1; i<=$1[1]#; i++)
        {
                s = s // "  " // (($1[2][i])?$1[1][i]:("<" // $1[1][i] // ">"));
        }
        l = [s];
        return l;
}


func displayabove
{
        /* two display lists $1 and $2 -> display list for $1 above $2 */
        auto s,i;
        s = $1;
        for (i=1; i<=$2#; i++) {
                append s, $2[i];
        }
        return s;
}

func displayright
{
        /* two display lists $1 and $2 -> display list for $1 to right of $2 */
        auto s,i;
        s = [];
        for (i=1; i<=$1#; i++) {
                append s, ($1[i] // $2[i]);
        }
        return s;
}

func messdisplay
{
        /* display text line in a specified vertical position */
        /* $1+1 is the height  of the display, $2 is the message */
        auto i,s;
        s=[];
        for (i=0; i<=$1; i++) {
                append s, ((i==($1/2))?$2:repchar(' ',$2#));
        }
        return s;
}


func repchar
{
        /* $1=char, $2=number of repetitions */
        auto s,c,i;
        s = "";
```

```
        for (i=1; i<=$2; i++)
        {
                c = $1;
                s = s // c;
        }
        return s;
}
```