Investigating Program Specification

Eric G. Wagner
Wagner Mathematics, 1058 Old Albany Post Road
Garrison, NY 10524, USA

October 4, 2001

Abstract

In this talk I will present some thoughts on program specification that resulted from using algebraic specification methods to specify some of the programs that I have written in recent years. The specifications were written after the programs were written and thus might better be described as documentation rather than specifications. The programs were interactive programs written in Java or JavaScript and up to 800 lines in length. The algebraic specification methodology is based on the ADJ methodology (conditional axioms, initial algebras and free functors, looser semantics by means of constraints) extended to a form of Moore automata.

Warning: These are working notes, not a final document. While I would be happy to receive comments, I want to make it clear that these documents are provided only to provide information for interested researchers, they are not provided for critics to excoriate in print.

Contents

1 Introduction

The ideas presented here developed from a desire to be able to write specifications for some of the programs that I have written in the last few years in the process of writing interactive courseware. While these programs were not all that big (e.g., 812 lines of Java) and could be designed "in the head", there was at least a need for some specification formalism to remind me what they did.

While parts of these programs could be specified as algebras, other parts could not. Many of these programs involved graphical user interfaces containing a variety of inputs and outputs such as buttons, text, and both static and animated graphics. Others involved mutable objects (i.e., object oriented programming objects with memory). Neither of these cases easily fit the standard algebraic paradigm. Also, while the algebraic methodology works reasonably well on trivial examples such as the specification of STACKS, FINITE SETS, ARRAYS, etc., it was not all that useful for complex programs even if they don't involve interfaces or mutable objects. But, in many cases, such programs can obviously modelled as automata. That is, by giving:

S, a set (of *states*)

I, a set (of *inputs*)

O, a set (of *outputs*)

$\sigma : I \times S \to S$, a total function (the *state transition function*)

$\omega : I \times S \to O$, a total function (the *output function*)

However, this is generally rather difficult to do as the sets I, O and S are very large and functions σ and ω are generally rather complex. Thus it is clear that this model is not the informal mental model that I was using while developing and implementing the programs. Furthermore, it is clear that most automata are not of interest – the automata corresponding to "practical or interesting programs" form a subset of the set of all automata. Looking at the real examples that I had in hand I realized that they had a lot of additional structure that played an important role in making them tractible even though non-trivial.

Some guiding thoughts:

- Interactive programs are not static data types
- Initial algebras are not the way to go (except for ADT)
- These programs can be viewed as automata (abstract machines) based on the Moore model of automata.
- Structuring Inputs and Outputs (as resp. coproducts and products)
- Fixed algebras used as the input and output types.
- state descriptors (hidden outputs) used to describe the features of the states.
- conditional axioms over input strings used to specify state transition and output functions. Only allow models that preserve the input and output algebras (persistency).
- the axioms should (could) lead to tests in the style of "Extreme programming"

2 Automata Specification Technique

Definition 2.1 A **Moore Automaton** is given by the following data:

I, a set, called the set of **inputs**

S, a set, called the set of **states**

O, a set, called the set of **outputs**

$\sigma : S \times I \to S$, a function, called the **transition function**

$\omega : S \to O$, a function, called the **output function**

We immediately extend σ to $S \times I^*$ by taking $\sigma(s, \lambda) = s$ and $\sigma(s, v \cdot i) = \sigma(\sigma(s, v), i)$

We immediately extend ω to $S \times I^*$ by taking $\omega(s, v) = \omega(\sigma(s, v))$.

Given two states $s, s' \in S$ we write

$$s \sim s' \text{ iff } \forall v \in I^*,\ \omega(s, v) = \omega(s', v)$$

$$s \overset{+}{\sim} s' \text{ iff } \forall v \in I^+,\ \omega(s, v) = \omega(s', v)$$

Note that $s \overset{+}{\sim} s'$ allows $\omega(s) \neq \omega(s')$. □

Our goal is to specify automata of this kind that arise as programming entities (GUI, objects, etc.). To make this feasible we supply some additional structure for I and O and introduce a means for making convenient statements attributes of the states. Proceeding informally:

A specification consists of

- **I**, a set of input names, some of which may be typed.
- **D**, a set of descriptor names, all of which are typed.
- **O**, a set of output names, all of which are typed.
- **I**, **D** and **O** are assumed to be disjoint. This is a matter of convenience rather than necessity.
- A set of axioms.

The typing will be with respect to data-types (algebras) or automata. In the case of algebras the type will correspond to a particular carrier of the algebra. For an untyped element $i \in \mathbf{I}$ we take the corresponding set to be the set $\{i\}$. The intuition is that such an untyped $i \in \mathbf{I}$ is something like a "button". In the case of automata, the type will be the set of states of the automata. For the time being we will not allow the typing to be recursive.

What we wish to specify is a pair of functions

$$\sigma : S \times I \to S$$
$$\omega : S \to O$$

where

- S consists of the set of states of the automaton.
- I consists of the union of the set of untyped input names with the disjoint union of the types corresponding to the typed input names.
- O consists of mappings from the set of output names to the sets corresponding to the typed output names. Thus O is an **O**-indexed product.

Rather than specify S directly we will make use of the descriptors. From **D** and its typing, we define as set D consisting of mappings from the set of descriptor to the sets corresponding to the typed descriptor names. Thus D is a **D**-indexed product. Then, in principle, we define a mapping

$$\delta : S \to D$$

which we immediately extend to $S \times I^*$ in the same manner as we extended ω.

Clearly, there is no mathematical difference between outputs and descriptors. The intended distinction is that the output are visible to the end-user but the descriptors are not (but they are visible to the specifier). More important, as long as "we get the right outputs in response to the right inputs" it is not necessary to fully implement the descriptors. To put it another way, the descriptors play a role similar to that played by hidden sorts and functions in initial equational algebra specifications.

The aim is to specify the functions only as much as necessary to guarantee the desired behavior. That is, we consider the specified behavior to consist of the set of all models that

1. satisfy the axioms.
2. preserve the input and output types.

The axioms use the extended versions of σ, ω and δ and can use the relations $=$, $\sim$ or $\overset{+}{\sim}$. Note that the uses of $\sim$ and $\overset{+}{\sim}$ can always be rewritten in terms of $=$. We will also allow the use of axiom schemata to describe infinite sets of axioms. Finally we will use the "compressed version" of conditional axioms that allows us to combine axioms such as

- $A \Rightarrow B$ and $A \Rightarrow C$ to $A \Rightarrow B$ & C
- $A \Rightarrow B$ and $C \Rightarrow B$ to $A \parallel C \Rightarrow B$

We will also use what we call "inertial axioms" – which are explained later.

3 Stacks as Automata

We will illustrate some of the basic ideas of program specification by giving specifications for stacks of integers.

Example 3.1 Here is the first example of a specification of a stack.
Specification
name: Stack_1

Inputs

$Arg \ : \ integer$

$push, \ pop, \ top, \ empty?$

Outputs

$Val \ : \ integer$

$Empty? : \{yes, \ no\}$

descriptors

. ("." = no descriptors)

Axioms – Stack_1
For $s \in States, \ z \ : \ Arg$

1. $Empty \in S$
2. $\omega(Empty, \ top)(Val) = 0$
3. $\omega(Empty, \ pop)(Val) = 0$
4. $\omega(Empty, \ empty?)(Empty?) = yes$
5. $\omega(s, \ z \cdot push \cdot top)(Val) = z$
6. $\omega(s, \ z \cdot push \cdot pop)(Val) = z$
7. $\omega(s, \ z \cdot push \cdot empty?)(Empty?) = no$
8. $\sigma(s, \ z \cdot push \cdot pop) \overset{+}{\sim} s$
9. $\sigma(s, \ z \cdot push \cdot top) \overset{+}{\sim} \sigma(s, \ z \cdot push)$
10. $\sigma(s, \ z \cdot push \cdot empty?) \overset{+}{\sim} \sigma(s, \ z \cdot push)$

end of specification

□

This is, unfortunately, a bigger specification than the corresponding initial algebra specification of stack. However:

- The *pop* operation capture our intuition about stacks in that it both returns the top element of the stack and removes it from the stack (changes the state of the stack).

- It has many models in which the type *int* is whatever we set it to be and which obey the axioms. That is, there is a natural behavioral semantics here.

- It is satisfied by intuitively correct models that do not obey the stricter algebraic axiom:

  $$\sigma(s,\ z \cdot push \cdot pop) = s$$

  This is, of course, a consequence of using $\stackrel{+}{\sim}$ in place of $=$, or, equivalently, because we are axiomatizing over strings of inputs rather than just individual inputs.

One objection to the above specification could be that it requires a designated state, *Empty*. This can be eliminated as follows:

Example 3.2 A variant:
Specification
name: Stack_2

Inputs

$Arg\ :\ integer$

$push,\ pop,\ top,\ empty?,\ reset$

Outputs

$Val\ :\ integer$

$Empty? : \{yes,\ no\}$

descriptors

.

Axioms – Stack_2
For $s, s' \in States,\ z\ :\ Arg$

1. $\sigma(s,\ reset) \sim \sigma(s',\ reset)$
2. $\omega(s,\ reset \cdot top)(Val) = 0$
3. $\omega(s,\ reset \cdot pop)(Val) = 0$
4. $\omega(s,\ reset \cdot empty?)(Empty?) = yes$
5. $\omega(s,\ z \cdot push \cdot top)(Val) = z$
6. $\omega(s,\ z \cdot push \cdot pop)(Val) = z$
7. $\omega(s,\ z \cdot push \cdot empty?)(Empty?) = no$

8. $\sigma(s,\ z \cdot push \cdot pop) \overset{+}{\sim} s$

9. $\sigma(s,\ z \cdot push \cdot top) \overset{+}{\sim} \sigma(s,\ z \cdot push)$

10. $\sigma(s,\ z \cdot push \cdot empty?) \overset{+}{\sim} \sigma(s,\ z \cdot push)$

end of specification

□

Example 3.3 Another Variant, with a descriptor:

Specification
name: Stack_3

Inputs

$Arg\ :\ integer$

$push,\ pop,\ top,\ empty?,\ reset$

Outputs

$Val\ :\ integer$

$Empty? : \{yes,\ no\}$

descriptors

$size\ :\ natural$

Axioms – Stack_3
For $s, s' \in States,\ z\ :\ Arg$

1. $\sigma(s,\ reset) \sim \sigma(s',\ reset)$
2. $\omega(s,\ reset \cdot top)(Val) = 0$
3. $\omega(s,\ reset \cdot pop)(Val) = 0$
4. $\delta(s,\ reset)(size) = 0$
5. $\omega(s,\ z \cdot push \cdot top)(Val) = z$
6. $\omega(s,\ z \cdot push \cdot pop)(Val) = z$
7. $\sigma(s,\ z \cdot push \cdot pop) \overset{+}{\sim} s$
8. $\sigma(s,\ z \cdot push \cdot top) \overset{+}{\sim} \sigma(s,\ z \cdot push)$
9. $\delta(s,\ z \cdot push)(size) = \delta(s,\ \lambda) + 1$

10. $\delta(s,\ top)(size) = \delta(s,\ \lambda)(size)$

11. $\delta(s,\ \lambda)(size) > 0 \Rightarrow \delta(s,\ pop)(size)$
 $= \delta(s,\ \lambda)(size) - 1$

12. $\delta(s,\ \lambda)(size) = 0 \Rightarrow \omega(s,\ empty?)(Empty) = yes$

13. $\delta(s,\ \lambda)(size) \neq 0 \Rightarrow \omega(s,\ empty?)(Empty) = no$

end of specification

□

One objection to the above specifications could be that we have made rather arbitrary choices what to have for the top of empty stacks. For example, in stack_1 we had

$$\omega(Empty,\ top)(Val) = 0$$

$$\omega(Empty,\ pop)(Val) = 0.$$

If we forget these two axioms we can still provide semantics – by looking at all the models that fit the remaining axioms. Note, however, that because of axiom 8, the value of *top* and *pop* on all stacks s, such that $s \overset{+}{\sim} Empty$, is forced to be the same. We can get around this possible difficulty by using stack_3 and replacing axioms 7 and 8 by axioms

$$\delta(s,\ \lambda)(size) > 0 \Rightarrow \sigma(s,\ z \cdot push \cdot pop) \overset{+}{\sim} s$$

$$\delta(s,\ \lambda)(size) > 0 \Rightarrow \sigma(s,\ z \cdot push \cdot top) \overset{+}{\sim} \sigma(s,\ z \cdot push)$$

Important comment It seems to be very difficult to write such specifications without making mistakes and I am almost certain that there are errors/omissions in the above specifications. The lack of a simple way to debug the specifications gives me some doubt about the efficacy of the approach. Of course it is possible to check a specification by actually programming it. But, aside from the fact that it seems backward to use programming to check specifications, it is hard to write the programs that do not make use of intuitions possibly missing from the specifications.

4 Looking at arrays as automata

Example 4.1 Specification

name: <u>Ext</u>endible<u>Arr</u>ay

Inputs

value : string,integer

index : natural

put, *get*, *length*, *arreset*

Outputs

$out \ : \ integer + error$

descriptors

.

Axioms
$i \ : \ index, \ z \ : \ value, \ s \in States, \ w \in Inputs^*$

1. $empty \in States$
2. $\omega(empty, \ i \cdot get)(output) = error$
3. $\omega(empty, \ length)(output) = 0$
4. $\sigma(s, \ arreset) = empty$
5. $i < \omega(s, \ length)(output) \ \Rightarrow \ \omega(s, i{\cdot}z{\cdot}put{\cdot}length)(output) = \omega(s, \ length)(output)$
6. $\omega(s, \ length)(output) \leq i \ \Rightarrow \ \omega(s, z \cdot i \cdot put \cdot length)(output) = i + 1$
7. $\omega(s, z \cdot i \cdot put \cdot i \cdot get)(output) = z$
8. $\omega(s, \ length)(output) \leq i \ \Rightarrow \ \omega(s, i \cdot get)(output) = error$
9. $i \cdot put \notin w \ \&\ arreset \notin w \Rightarrow \omega(s, w \cdot i \cdot get)(output) = \omega(s, i \cdot get)(output)$

end of specification

□

The last axiom in the above specification is there to guarantee that the only times that the output changes in after a *put* or *arreset* (array reset) input. While the operation appears to have a form that is not allowed in conditional axioms, there is actually no problem – see the section on "inertial axioms".

A natural question is, can we use this specification of extendible array to give a specification of stack that, in some meaningful sense, will be equivalent to one given in the file stacks.tex? The idea, of course, is to mimic the common array/pointer implementation of stacks. The result can be viewed as a refinement of the original specification.

We will refine the specification of Stack_1. Here is the original specification:

Example 4.2 Specification

name: Stack_1

Inputs

$Arg : integer$

$push,\ pop,\ top,\ empty?$

Outputs

$Val : integer$

$Empty? : \{yes,\ no\}$

descriptors

.

Axioms
For $s \in States,\ z\ :\ Arg$

1. $Empty \in States$
2. $\omega(Empty,\ top)(Val) = 0$
3. $\omega(Empty,\ pop)(Val) = 0$
4. $\omega(Empty,\ empty?) = yes$
5. $\omega(s,\ z \cdot push \cdot top)(Val) = z$
6. $\omega(s,\ z \cdot push \cdot pop)(Val) = z$
7. $\omega(s,\ z \cdot push \cdot empty?)(Empty?) = no$
8. $\sigma(s,\ z \cdot push \cdot pop) \overset{+}{\sim} s$
9. $\sigma(s,\ z \cdot push \cdot top) \overset{+}{\sim} \sigma(s,\ z \cdot push)$
10. $\sigma(s,\ z \cdot push \cdot empty?) \overset{+}{\sim} \sigma(s,\ z \cdot push)$

end of specification

□

Here is the new specification/refinement:

Example 4.3 Specification

name: Stack_4

Inputs

$Arg : integer$

$push,\ pop,\ top,\ empty?$

Outputs

$Val : integer$

$Empty? : \{yes,\ no\}$

descriptors

$Arr \ : \ ExtArr$

$Ptr \ : \ natural$

Axioms

1. $Empty = empty$

2. $\delta(s,\ \lambda)(Arr) = empty \parallel \delta(s,\ \lambda)(Ptr) = 0$
$\Rightarrow \omega(s,\ top)(Val) = 0$
& $\omega(s,\ pop)(Val) = 0$
& $\omega(s,\ empty?)(Empty?) = yes$
& $\delta(s,\ \lambda)(Ptr) = 0$

3. $\delta(s,\ z \cdot push)(Arr) = \sigma(\delta(s,\ \lambda)(Arr),\ z \cdot \delta(s,\ \lambda)(Ptr) + 1 \cdot put)$

4. $\delta(s,\ z \cdot push)(Ptr) = \delta(s,\ \lambda)(Ptr) + 1$

5. $\delta(s,\ \lambda)(Ptr) > 0$
$\Rightarrow \omega(s,\ top)(Val) = \omega(\delta(s,\ \lambda)l(Arr),\ \delta(s,\ \lambda)(Ptr) \cdot get)$
& $\omega(s,\ pop)(Val) = \omega(\delta(s,\ \lambda)l(Arr),\ \delta(s,\ \lambda)(Ptr) \cdot get)$
&$\delta(s,\ pop)(Ptr) = \delta(s,\ \lambda)(Ptr) - 1$ **&**$\delta(s,\ empty?)(Empty?) = no$

end of specification

□

Question: How hard is it to prove that this refinement is correct (or incorrect, as the case may be)?

5 The specification of canvasses

Intuitively, a canvas is something on which we can paint pictures (as in JAVA). The basic idea can be presented quite simply if we are willing to accept real numbers, and the operations thereon, as a well–understood algebra. If we insist on having to give a specification for the reals then life becomes much more difficult (but not impossible – see [5]).

Given positive real numbers h and w together with a non-empty set, C, of colors we define a canvas of *height* h, *width* w and *colors* C, to be an automaton satisfying the following

Specification
name: Canvas

Inputs

drawing : "all" partial functions from $[0,h] \times [0,w] \to C$

clear

Outputs

picture : "all" total functions from $[0,h] \times [0,w] \to C$

descriptors

.

Axioms
For $d : \mathit{drawing},\ \langle x, y\rangle,\ \langle x', y'\rangle \in [0,h] \times [0,w]$

1. $\langle x, y\rangle \in \mathit{dom}(d) \Rightarrow \omega(s,\ d)(\mathit{picture})(\langle x, y\rangle) = d(\langle x, y\rangle)$
2. $\langle x, y\rangle \notin \mathit{dom}(d) \Rightarrow \omega(s,\ d)(\mathit{picture})(\langle x, y\rangle) = \omega(s,\ \lambda)(\mathit{picture})(\langle x, y\rangle)$
3. $\omega(s,\ \mathit{clear})(\langle x, y\rangle) = \omega(s,\ \mathit{clear})(\langle x', y'\rangle)$

end of specification

The intuition is that we make a picture out of a sequence of colored drawings in which the drawings later in the sequence overlay the earlier drawings in the sequence. The motivation behind this approach is (at least) two-fold:

1. Using a sequence of overlays simplifies many specifications of specific pictures since the drawings can all be done using closed intervals (i.e., having the end-points of a line as part of the line). Otherwise it is necessary to have the ability to specify lines that do not include their endpoints (a feature found in few, if any, graphics languages). See example 5.2.
2. It provides an easy framework in which to describe pictures with parts that are state dependent since those parts can be treated as state dependent drawings thus avoiding having to specify a complete drawing for each case.

This does not describe implementations, where, from the end-user point of view, two differently colored pixels "at the same location" can be simultaneously displayed and produce a mix of the colors. If that is the level at which you wish to make your specification then you need a somewhat more complicated concept of canvas.

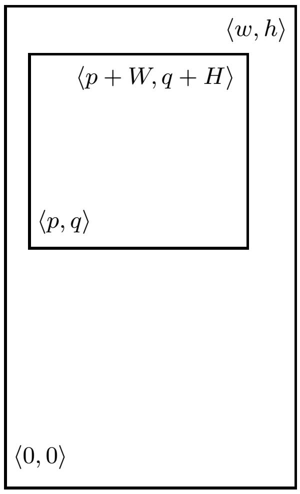
When we want to use a canvas to specify a (possibly state dependent) picture the real job is the specification of the individual drawings and the specification of their ordering within the input sequence to the canvas.

In some cases we may want to specify the drawings without being specific about their ordering in the input sequence. E.g., in the polycalculator it is not really significant which order we impose on the polynomials in M, A, B, and C.

If the order does not really matter then we can just specify the set, $Draw$, of the desired drawings and then use any linear ordering of $Draw$ as the input string. In general we will probably want at least some ordering (e.g., it might desirable to have the "border of the picture" be the last drawing in the input). To capture this notion of having "some ordering" it suffices to supply the set of drawings with a partial ordering and then specify that any linear ordering of the drawings that is compatible with the given partial ordering can be used to produce a picture satisfying the specification.

Note that the *clear* operation is under-determined: the axiom ensures is that the result will be a drawing (indeed a picture) of uniform color, but it does not specify which color. Of course it is easy to specify a drawing in which is of a given uniform color.

Example 5.1 Here is a specification of a drawing that sets a rectangle within the canvas to the color red. Let the dimensions of the rectangle be $H \times W$ inside the overall canvas of dimensions $h \times w$ with the lower left corner of the rectangle at point $\langle p,\ q\rangle$ while the lower left corner of the canvas is at point $\langle 0,\ 0\rangle$, to the color red. We assume that all these numbers are chosen so as to make the rectangle actually within the canvas so that the overall result (minus the color) is as follows:



The desired drawing is then the partial mapping with graph

$$\{\langle x,\ y,\ white\rangle \mid p \leq x \leq p + W,\ q \leq y \leq q + H\}$$

□

Example 5.2 Given the same rectangle–within–a–rectangle situation as in the previous example we can draw a shaded frame, of width m, around the inner rectangle. We can do this as one drawing or as a sequence of drawings. The single drawing version is the function with graph:

$$
\begin{aligned}
&\{\langle x,\ y,\ \mathit{darkgray}\rangle \mid p \le x \le p+m,\ q \le y \le q+H\}\\
&\cup\{\langle x,\ y,\ \mathit{black}\rangle \mid p+m < x < p+W,\ q+H-m \le y \le q+H\}\\
&\cup\{\langle x,\ y,\ \mathit{lightgray}\rangle \mid x = p+W-m,\ q+m \le y \le q+H-m\}\\
&\cup\{\langle x,\ y,\ \mathit{lightgray}\rangle \mid x = p+W,\ q \le y \le q+H\}\\
&\cup\{\langle x,\ y,\ \mathit{white}\rangle \mid p+W-m < x < p+W,\ q < y < q+H-m\}\\
&\cup\{\langle x,\ y,\ \mathit{lightgrey}\rangle \mid p+m < x < p+W-m,\ y = q+m\}\\
&\cup\{\langle x,\ y,\ \mathit{lightgrey}\rangle \mid p+m < x < p+W,\ y = q\}\\
&\cup\{\langle x,\ y,\ \mathit{white}\rangle \mid p+m < x < p+W-m,\ q < y < q+m\}
\end{aligned}
$$

The difficulty with this approach is the need to make the graph of the function single-valued. This requires careful attention to the use of $\le$ and $<$, and we do not guarantee that we have been completely sucessful. This kind of persnickety detail can be avoided by describing the shaded frame as the result of an ordered sequence of simpler drawings – the over-painting then takes care of the overlaps. The following sequence of drawings produces the same picture:

$$\text{drawing } 1 = \{\langle x,\ y,\ \mathit{white}\rangle \mid p+W-m \le x \le p+W,\ q \le y \le q+H-m\}$$

$$\text{drawing } 2 = \{\langle x,\ y,\ \mathit{white}\rangle \mid p+m \le x \le p+W-m,\ q \le y \le q+m\}$$

$$\text{drawing } 3 = \{\langle x,\ y,\ \mathit{lightgrey}\rangle \mid p+m \le x \le p+W,\ y = q\}$$

$$\text{drawing } 4 = \{\langle x,\ y,\ \mathit{lightgray}\rangle \mid x = p+W,\ q \le y \le q+H\}$$

$$\text{drawing } 5 = \{\langle x,\ y,\ \mathit{lightgray}\rangle \mid x = p+W-m,\ q+m \le y \le q+H-m\}$$

$$\text{drawing } 6 = \{\langle x,\ y,\ \mathit{lightgrey}\rangle \mid p+m \le x \le p+W-m,\ y = q+m\}$$

$$\text{drawing } 7 = \{\langle x,\ y,\ \mathit{black}\rangle \mid p+m \le x \le p+W,\ q+H-m \le y \le q+H\}$$

$$\text{drawing } 8 = \{\langle x,\ y,\ \mathit{darkgray}\rangle \mid p \le x \le p+m,\ q \le y \le q+H\}$$

Each drawing here is very similar to one of the sets used in the earlier, one-drawing specification. The difference is that all occurences of $<$ have been replaced by $\le$ and the drawing have been put in a linear ordering that (if I haven't made a mistake) provides the desired overlays to get the same picture as before.

Interestingly, neither specification gives me exactly what I had originally intended. The above specifications put the "frame" inside the rectangle, while my original intention was to put it outside the rectangle. Beware, Murphy's law ("If anything can go wrong, it will") is still in effect.

Of course we could also have simplified things by first defining some general concepts such as rectangles. □

6 Specification of a Polynomial Calculator

A calculator for manipulating and graphing polynomials with real coeficients. Designed as part of a on-line course in elementary algebra (that is, the "classical" algebra of real and complex numbers).

We are not necessarily advocating the use of such calculators by beginning students – "handwork" may be a better way to gain understanding of the subject and ability to solve problems.

6.1 Polynomials as an Abstract Data Type

Specification of Polynomials:
Sorts: $\{Nat,\ Bool,\ Real,\ Polynomial\}$

Operators:

$zeropoly\ : \rightarrow Poly$

$degree : Poly \rightarrow Nat$

$coef\!: Poly \times Nat \rightarrow Real$

$derivative : Poly \rightarrow Poly$

$add : Poly \times Poly \rightarrow Poly$

$mult : Poly \times Poly \rightarrow Poly$

$subt : Poly \times Poly \rightarrow Poly$

$div : Poly \times Poly \rightarrow Poly$

$rem : Poly \times Poly \rightarrow Poly$

$Newt : Poly \times Real \rightarrow Real$

$Eval : Poly \times Real \rightarrow Real$

Axioms:
For $i, j, k : Nat,\ n : Real,\ p, q, r, d : Polynomials$

1. $p = zeropoly \Rightarrow coef(p,\ i)0$ **&** $degree(p) = 0$
2. $i > degree(p) \Rightarrow coef(p, i) = 0$
3. $degree(p) > 0 \Rightarrow coef(p,\ degree(p)) \neq 0$
4. $coef(add(p,\ q), i) = coef\ (p) + coef\ (q)$
5. $coef\ (subt(p,\ q), i) = coef\ (p) - coef\ (q)$
6. $coef\ (mult(p,\ q), i)$
$= \sum\langle coef\ (p, k) * coef\ (q, j) \mid k + j = i\rangle$

7. $p = add(mult(d, q), r)$ & $degree(r) < degree(q)$
   $\Rightarrow div(p, q) = d$

8. $p = add(mult(d, q), r)$ & $degree(r) < degree(q)$
   $\Rightarrow rem(p, q) = r$

9. $i \geq degree(p) \Rightarrow coef\ (dervative(p), i) = 0$

10. $i < degree(p) \Rightarrow coef\ (derivative(p), i)$
    $= (i + 1) * coef\ (p, i + 1)$

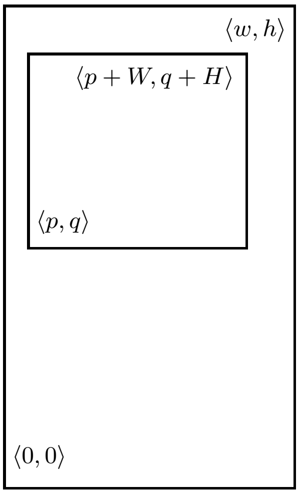
11. $Eval(p,\ n) = \sum\langle n^i * coef\ (p, i) \mid i = 0, \ldots, degree(p)\rangle$

12. $Eval(derivative(p), n) = 0 \Rightarrow Newt(p,\ n) = n$

13. $Eval(derivative(p), n) \neq 0 \Rightarrow Newt(p,\ n)$
    $= n - (Eval(p, n)/Eval(derivative(p), n)$

6.2 The Polynomial Calculator

We assume we are in some given state s with values for the descriptors. The graphs, grids, etc. are within a rectangle of dimensions $H \times W$ inside the overall canvas of dimensions $h \times w$ with the lower left corner of the graph-box at point $\langle p,\ q\rangle$ while the lower left corner of the canvas is at point $\langle 0,\ 0\rangle$.



Specification
name: Polynomial Calculator

Inputs

$Inp\ :\ polynomial$

$val\ :\ real$

$xmin,\ xmax,\ ymin,\ ymax\ : integer$

$M \rightarrow A,\ M \rightarrow B,\ M \rightarrow C,\ deriv,\ graph,$
$+,\ *,\ -,\ div,\ rem,\ solve,\ clrM,\ enter$

$grid,\ fineness,\ min/max,\ coords$

Outputs

$Inp,\ M,\ A,\ B,\ C\ :\ polynomial$

$val\ : real$

$xmin,\ xmax,\ ymin,\ ymax\ :\ integer$

$CVS\ :\ canvas$

descriptors

For $l \in \{M,\ A,\ B,\ C\}$

$Display\_l,\ gridDesc,\ coordDesc\ :\ \{yes, no\}$

$finenessDesc\ \{fine,\ course\}$

$XMIN,\ XMAX,\ YMIN,\ YMAX\ :\ real$

$graph\_l$, $grid1x$, $grid2x$, $grid1y$, $grid2y$, $coord\_x$, $coord\_y$, $legend$, box : $drawing$

Axioms
For $s \in States$, $n \in integer$, $l \in \{M, A, B, C\}$, $i\ :\ Inp$, $mop \in \{deriv,\ graph\}$, $bop \in \{+, *, -, div,\ rem\}$, and $I \in Inputs$

and letting:

1. (a) $\boldsymbol{XMIN} = \delta(s,\ \lambda)(XMIN)$,
   (b) $\boldsymbol{XMAX} = \delta(s,\ \lambda)(XMAX)$,
   (c) $\boldsymbol{YMIN} = \delta(s,\ \lambda)(YMIN)$,
   (d) $\boldsymbol{YMAX} = \delta(s,\ \lambda)(YMAX)$,
   (e) $diff\_x = \boldsymbol{XMAX - XMIN}$
   (f) $diff\_y = \boldsymbol{YMAX - YMIN}$
   (g)† $proposedUnit\_x = 10^{intPart(log_{10}(diff\_x))}$
   (h)† $proposedUnit\_y = 10^{intPart(log_{10}(diff\_y))}$
   (i)† $Unit\_y = \begin{cases} proposedUnit\_y & \text{if } roundDown(\boldsymbol{YMAX}/proposedUnit\_y) \neq roundUp(\boldsymbol{YMIN}/proposedUnit\_y) \\ & \text{and } \delta(s,\ \lambda)(finenessAttrib) = coarse \\ proposedUnit\_y/10 & \text{if } roundDown(\boldsymbol{YMAX}/proposedUnit\_y) = roundUp(\boldsymbol{YMIN}/proposedUnit\_y) \\ & \text{or } \delta(s,\ \lambda)(finenessAttrib) = fine \end{cases}$

(j)† $Unit\_x = \begin{cases} proposedUnit\_x & \text{if } roundDown(\boldsymbol{XMAX}/proposedUnit\_x) \neq \\ & roundUp(\boldsymbol{XMIN}/proposedUnit\_x) \\ & \text{and } \delta(s,\ \lambda)(finenessAttrib) = coarse \\ proposedUnit\_x/10 & \text{if } roundDown(\boldsymbol{XMAX}/proposedUnit\_x) = \\ & roundUp(\text{boldmath } XMIN/proposedUnit\_x) \\ & \text{or } \delta(s,\ \lambda)(finenessAttrib) = fine \end{cases}$

(k)† $Top = roundDown(\boldsymbol{YMAX}/Unit\_y)$

(l)† $Bot = roundUp(\boldsymbol{YMIN}/Unit\_y)$.

(m)† $Rgt = roundDown(\boldsymbol{XMAX}/Unit\_x)$

(n)† $Lft = roundUp(\boldsymbol{XMIN}/Unit\_x)$.

(o) $color(M) = red$

(p) $color(A) = green$

(q) $color(B) = magenta$

(r) $color(C) = brown$

The AXIOMS are:

1.† $\omega(s, i)(Inp) = i$

2.† $I \notin Inp \Rightarrow \omega(s,\ I)(Inp) = \omega(s,\ \lambda)(Inp)$

3.† $\omega(s,\ enter)(M) = \omega(s,\ \lambda)(Inp)$

4. $\omega(s,\ mop)(M) = mop(\omega(s,\ \lambda)(M))$

5.† $\omega(s,\ bop \cdot l)(M) = bop(\omega(s, \lambda)(M,\ l))$

6. $\omega(s,\ deriv)(M) = Derivative(\omega(s,\ \lambda)(M))$

7. $\omega(s,\ clrM)(M) = 0$

8. $I \neq enter,\ mop,\ bop,\ deriv,\ clrM$
$\Rightarrow\ \omega(s, I)(M) = \omega(s,\ \lambda)(M)$.

9. $\omega(s,\ `M \rightarrow l\text{'})(l) = \omega(s,\ \lambda)(M)$

10.† $I \neq \text{'}M \rightarrow l\text{'}$ **&** $l \in \{A, B, C\}$ $\Rightarrow \omega(s,\ I)(l) = \omega(s,\ \lambda)(l)$

11. a) $\delta(s,\ \lambda)(Display\_l) = yes$
$\Rightarrow \delta(s,\ graph \cdot l)(Display\_l) = no$

b) $\delta(s,\ \lambda)(Display\_l) = no$
$\Rightarrow \delta(s,\ graph \cdot l)(Display\_l) = yes$

12. $\omega(s,\ n : val)(val) = n$

13. $I \notin val$ & $I \neq solve,\ evalM$
$\Rightarrow \omega(s,\ I)(val) = \omega(s,\ \lambda)(val)$

14. $\omega(s, evalM)(val) = Evaluate(\omega(s,\ \lambda)(M,\ val)$

15. $\omega(s, solve)(val) = Newt(\omega(s,\ \lambda)(M,\ val))$

16. a) $\delta(s,\ \lambda)(Display\_l) = yes$
$\Rightarrow \delta(s,\ \lambda)(graph\_l) = \{\langle(x * H/diff\_x) + p,\ (Eval(P(l), x) * H/diff\_y) + q,\ color(l)\rangle \mid XMIN \leq x \leq XMAX,\ YMIN \leq Eval(p, x) \leq YMAX\}$

b) $\delta(s,\ \lambda)(Display\_l) = no$
$\Rightarrow dom(\delta(s,\ \lambda)(graph\_l) = \emptyset$

17. a) $\delta(s,\ \lambda)(gridAttrib) = yes$
$\Rightarrow \delta(s,\ \lambda)(grid1y) = \{\langle(p + (Lft - \mathbf{XMIN} + (n * Unit\_x)W/diff\_x),\ q + y,\ cyan\rangle \mid 0 \leq n * Unit\_x \leq Rgt - Lft$ and $q \leq y \leq q + H\}$
& $\delta(s,\ \lambda)grid1x) = \{p+x,\ \langle(q+(Bot - \mathbf{YMIN} + (n*Unit\_y)H/diff\_y),\ cyan\rangle \mid 0 \leq n * Unit\_y \leq Top - Bot$ and $p \leq x \leq p + W\}$

b) $\delta(s,\ \lambda)(gridAttrib) = no$
$\Rightarrow dom(\delta(s,\ \lambda)(grid1y)) = \emptyset$
& $dom(\delta(s,\ \lambda)(grid1x)) = \emptyset$

18. a) $\delta(s,\ \lambda)(gridAttrib) = yes$
$\Rightarrow \delta(s,\ \lambda)(grid2y) = \{\langle(p + (Lft - \mathbf{XMIN} + (n * Unit\_x)W/diff\_x),\ q + y,\ blue\rangle \mid Lft + n * Unit\_x = 0$ and $q \leq y \leq q + H\}$
& $\delta(s,\ \lambda)(grid2x) = \{p+x,\ \langle(q+(Bot - \mathbf{YMIN} + (n*Unit\_y)H/diff\_y),\ blue\rangle \mid Lft + n * Unit\_y = 0$ and $p \leq x \leq p + W\}$

b) $\delta(s,\ \lambda)(gridAttrib) = no$
$\Rightarrow dom(\delta(s,\ \lambda)(grid2y) = \emptyset$
& $dom(\delta(s,\ \lambda)(grid2x) = \emptyset$

19. a) $\delta(s,\ \lambda)(coordAttrib) = yes$
$\Rightarrow \delta(s,\ \lambda)(coord\_y) = \{\langle p - z,\ Bot + (n * Unit\_y * H/diff\_y),\ blue\rangle \mid 0 \leq z \leq T,\ 0 \leq n \leq Top - Bot\}$
& $\delta(s,\ \lambda)(coord\_x) = \{\langle\ Lft + (n * Unit\_x * W/diff\_x),\ q - z,\ blue\rangle \mid 0 \leq z \leq T,\ 0 \leq n \leq Rgt - Lft\}$

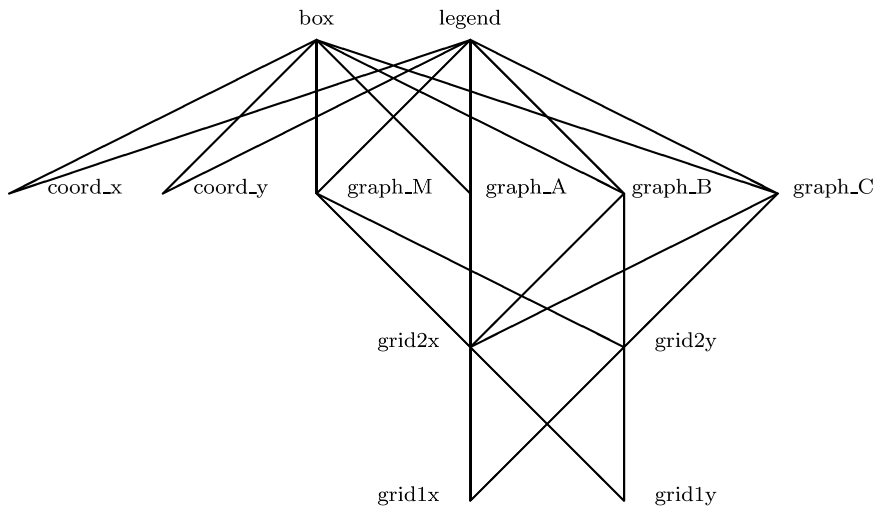
b) $\delta(s,\ \lambda)(gridAttrib) = no$
$\Rightarrow dom(\delta(s,\ \lambda)(coord\_y) = \emptyset$
& $dom(\delta(s,\ \lambda)(coord\_x) = \emptyset$

20. $\delta(s,\ \lambda)(legend) =$ Text("x times "+ $Unit\_x$+", y times "+ $Unit\_y$)

21. $\delta(s,\ \lambda)(box) =$ the set of points for the "'usual shaded box" of dimension $H \times W$ with lower left hand corner at $\langle p,\ q\rangle$. The details are left to the reader as an exercise (for one version, see the section on canvases).

22. $\omega(s,\ \lambda)(CVS) = \omega(s,\ clear \cdot w)(output)$ where w is any linear ordering of the pictures: *grid1x, grid2x, grid1y, grid2y, graph_M, graph_A, graph_B,*

graph_C, coord_x, coord_y, legend, compatible with the partial ordering given by the following diagram:



end of specification

6.2.1 Notes on the specification

re Inputs and Outputs: Note that Inp, $xmin$, $xmax$, $ymin$, $ymax$ and val appear as both inputs and outputs.

re (g) and (h): The $proposedUnit\_c$, for $c \in \{x,\ y\}$ is a preliminary value for the unit to be used to determine the grid, coordinates and legend to be displayed.

A more abstract specification of $proposedUnit\_c$, for $c \in \{x,\ y\}$ would be:

$$10^n \leq diff\_c \leq 10^{n+1} \Rightarrow Unit\_c = 10^n$$

.

re (i) and (j): This is the final determination of the unit to be used for the grid, etc. The final unit satisfies the requirement that the unit be such that both the x- and y-grids contain at least two lines.

re (k), (l), (m), (n): Top and Bot are, respectively, the position on the y-axis of the top and bottom lines; Lft and Rgt are, respectively, the positions on the x-axis of the leftmost and rightmost lines.

re 1 and 2: Together, these axioms say, in effect, that when a polynomial is written in the input (Inp) it remains there until the writing of another polynomial. Contrast this with the specification of a Stack where inputs are volatile, i.e., it does not remain after it has been pushed onto the stack.

re 3: The implementation does not literally satisfy this axiom because, in the implementation, different representations of polynomials are used in Inp and M – this is a result of limitations imposed by Java 1.1.

re 5: Note that $\omega(s,v)(o_1,\ldots,o_n)$ is an abbreviation for $\langle\omega(s,v)(o_1),\ldots,\omega(s,v)(o_n)\rangle$.

re 10: This is a schema abbreviating separate axioms for A, B, and C.

re 16: We are representing the drawing by its graph (the corresponding subset of $[0,h]\times[0,w]\times C$)

7 Imperative programming for automata

The specification of the polynomial calculator is fairly close to a program with the descriptors playing a role analogous to that played by variables in imperative programs. In this section we explore this a bit further. However, rather than describe a programming language we describe a simple calculator. I don't see any difficulty in extending this to a simple programming language such as the programming language LANG1 in my on-line notes on Universal Algebra (see www.ii.uib.no/∼wagner). When time permits I will extend this paper to cover LANG1.

To specify an actual language we would need to first specify an "environment" and then and then use it as a descriptor of a "computer". A somewhat simpler example is to specify an expression evaluator where you first enter the expression as a whole and then push an eval button that does the usual evaluation using a stack (I am assuming that the expression is in reversed Polish notation).

How do we abstractly define an environment? It seems natural to regard it as an automaton with an input for variable/value pairs (given either as literal pairs or as binary sequences) with a **put** input, and another input for variables with a **get** input and a single output that responds only to **get**s. We also need an initial state, **start** with, say, every variable set to 0 (assuming numerical values).

Example 7.1 name: Env<u>Env</u>ironment

Inputs

putdatum : $\langle variable,\ integer \rangle$

getdatum : $variable$

$put,\ get$

Outputs

$out\ :\ integer$

Descriptors

.

Axioms
For $s \in States$, $v \in Var$, $V \in val = int$, $w \in Input^*$

1. $\omega(s,\ \langle v,\ V \rangle \cdot put \cdot v \cdot get)(out) = V$
2. $(\forall V : int.\langle v,\ V \rangle \notin w\ \Rightarrow\ \omega(s,\ w \cdot v \cdot get)(out) = \omega(s,\ v \cdot get)(out)$
3. $start \in States$
4. $\omega(start,\ v \cdot get)(out) = 0$

end of specification

Remarks:

- Note that the second axiom is not a conventional conditional axiom since
  - The premise is universally quantified
  - It contains a negation

  We can replace it by:

  - an infinite conditional axiom (a conditional axiom with an infinite conjunction as its premise) if we take the carrier of sort *integer* to be the set, **Z**, of all integers.
  - The equivalent existential statement
    $(\ \exists V : int.\langle v,\ V \rangle \in w\ \Rightarrow\ \omega(s,\ w \cdot v \cdot get)(out) = \omega(s,\ v \cdot get)(out)$
  - or,
  - by the statement

    $$\text{For all } w \in (Inputs \backslash \{\langle v, V \rangle \mid V \in int\})^* \\ \omega(s,\ w \cdot v \cdot get)(out) = \omega(s,\ v \cdot get)(out)$$

    That is, we restrict the conditional to a subset of the set of all possible strings of Inputs, namely those not including any occurences of a pair $\langle v, V \rangle$. This is probably the best formulation from an intuitive point of view.

However we choose to write such axioms, their intent is to capture the idea that "the only significant changes occur under the conditions given by the other axioms". Informally we might say that the system has an inertia (as in: a body in motion will move in a straight line unless acted upon by an outside force) and refer to such axioms as **inertial axioms** (thus raising the possibility of confusion between the words "inertial" and "initial").

□

We also need to have a notion of an extendable array. That is, an array that can be lengthened. We will use the array specified in Example 4.1.

We want to give the example of a calculator into which one can enter an expression (in reversed Polish notation) and then evaluate it. We will do this in several stages: In the first stage we will define a calculator which can be stepped through the computation, i.e., it will have an input, *step*, which will perform a step in the evaluation of the expression. In the second stage we will utilize the stepping calculator to implement a calculator with an input, $eval$, which will do the complete evaluation of an already inputted expression in response to the input $eval$. The reader may observe that this is done in a manner similar to a while-do statement, *while bexp do stt odd*, where we first define a statment, *stt*, and then embed it into the while-do statement.

For the sake of easy reference, we repeat the specification of the SIMPOL calculator given in the section on stacks.

Example 7.2 We can use the example of a stack given in 3.3 as the basis of the specification of a simple calculator that evaluates arithmetic expressions written in reversed Polish notation. The stack appears as a descriptor.

name: <u>Sim</u>ple<u>Pol</u>ishCalculator

Inputs

$Val \ : \ integer$

$enter, \ add, \ mult$

Outputs

$result \ : \ integer$

Descriptors

$foo : Stack\_3$

Axioms
For any $v, w \in Inputs^*$, $n \in integer$, $s \in Stacks$

1. $\omega(s, \ n \cdot enter)(foo) = \sigma(s(foo), \ n \cdot push)$

2. $\omega(s(foo),\ \lambda)(size) > 1 \,\&\, \omega(s(foo),\ \lambda)(Val) = n \,\&\, \omega(s(foo),\ pop)(Val) = p \ \Rightarrow\ \omega(s,\ add)(foo) = \sigma(s(foo),\ pop \cdot pop \cdot n + p \cdot push)$
3. $\omega(s(foo),\ \lambda)(size) > 1 \,\&\, \omega(s(foo),\ \lambda)(Val) = n \,\&\, \omega(s(foo),\ pop)(Val) = p \ \Rightarrow\ \omega(s,\ mult)(foo) = \sigma(s(foo),\ pop \cdot pop \cdot n * p \cdot push)$
4. $\omega(s,\ \lambda)(result) = \omega(s(foo),\ \lambda)(Val)$

end of specification

□

Example 7.3 name: SteppingCalculator

Inputs

$Val\ :\ integer,\ \{+,\ *\}$

$enter,\ step,\ reset$

Outputs

$out\ :\ integer + \{error\}$

$done?\ :\ \{yes,\ no\}$

Descriptors

$cal\ :\ SimPol$ See Example 7.2

$prg\ :\ ExtArr$ See Example 4.1

$ctr\ :\ natural$

$inCtr\ :\ natural$

Axioms
For

1. $start \in States$
2. $\delta(start,\ \lambda)(ctr) = 0$
3. $\delta(start,\ \lambda)(inctr) = 0$
4. $\delta(start,\ \lambda)(arr) = empty.$
5. $\sigma(s,\ reset) = start$
6. $\delta(s, n \cdot enter)(prg) = \sigma(s(prg), n \cdot \delta(s,\ \lambda)(inCtr) \cdot put)$
7. $\delta(s, n \cdot enter)(inCtr) = \delta(s,\ \lambda)(inCtr) + 1$

8. $\delta(s,\ \lambda) < \delta(s(prg),\ length)(out)\ \Rightarrow$
   $\delta(s,\ step)(cal) = \sigma(s(cal),\ \delta(s(prg),\ \delta(s(prg), \delta(s,\ \lambda)(ctr) \cdot get)(out))$
   $\&\ \delta(s,\ step)(ctr) = \delta(s,\ \lambda)(ctr) + 1$

9. $\delta(s,\ \lambda)(ctr) \geq \delta(s(prg),\ length)(out)\ \Rightarrow$
   $\omega(s,\ \lambda)(out) = \omega(s(cal,\ \lambda)(out)$

10. $\delta(s,\ \lambda)(ctr) \leq \delta(s,\ \lambda)(inCtr)\ \Rightarrow\ \omega(s, \lambda)(done) = no$

11. $\delta(s,\ \lambda)(ctr) > \delta(s,\ \lambda)(inCtr)\ \Rightarrow\ \omega(s, \lambda)(done) = yes$

12. "The usual inertial axioms" saying that the above situations are the only ones that "really change the state".

13. WHAT ABOUT error – when we have a simpol error from insufficieent stack contents?

end of specification

□

It remains now to give a specification for the expression evaluator that has an *eval* input (button?).

Example 7.4 name: <u>Eval</u>uation<u>Calc</u>ulator

Inputs

$Val\ :\ integer,\ \{+,\ *\}$

$enter,\ eval,\ reset$

Outputs

$result\ :\ integer$

Descriptors

$stc\ :\ stepcalc$

Axioms
For $s, t \in States$, $n \in Val$

1. $\delta(s,\ n \cdot enter)(stc) = \sigma(s(stc),\ n \cdot enter)$

2. $\delta(s,\ reset)(stc) = \sigma(s(stc),\ reset)$

3. $\delta(s(stc),\ \lambda)(done) = no\ \&\ \delta(t,\ \lambda)(stc) = \sigma(s(stc),\ step)\ \Rightarrow\ \omega(s,\ eval)(out) = \omega(t,\ eval)(out)$

4. $\delta(s(stc),\ \lambda)(done) = yes\ \&\ \ \Rightarrow\ \omega(s,\ eval)(out) = \omega(s(stc),\ \lambda)(out)$

end of specification

□

8 The concept of inertial specifications

The intuition behind the inertial axioms in specifications of automata is that certain outputs and descriptors will only change under the conditions specified by the other axioms. This is loosely analogous to the concept of inertia in physics:

> A body in motion will move in a straight line unless acted upon by an external force.

An inertial axiom is always of the form that for some given set, W, of input strings, and some fixed input string, u, that

$$w \in W \Rightarrow \alpha(s,\ w \cdot u)(A) = \alpha(s,\ u)(A)$$

or

$$w \in W \Rightarrow \omega(s,\ w \cdot u)(O) = \omega(s,\ u)(0)$$

However the axioms may be given by a schema with many instances and in which the choice of W and u may be different for different instances. Furthermore, as in Example 4.1, it may be more convenient to present the axiom using the complement of W (i.e., $S^* - W = \{u \mid u \in S^*,\ u \notin W\}$.

Example 8.1 In the definition of environment (in which, say, the variables are assigned values from some set Z) we want to be able to say that the value assigned a given variable changes only when an assignment is made to the variable. That is, for a variable, v, an assignment is an input string of the form $z \cdot v \cdot put$ and the requirement is that for $W(v) = \{z \cdot v \cdot put \mid z \in Z\}$ we have

$$w \in W(v) \Rightarrow \omega(s,\ w \cdot v \cdot get)(out) = \omega(s,\ v \cdot get)(out)$$

□

The problem in proving anything about inertial specifications is, of course, to get a tight enough definition of inertial axiom so that we can make rigorous arguments.

Given the bound variables (v, w and s in the above example) we need to be able to determine

- if $w \in W(v)$
- $u(v)$

In the above example we can clearly specify functions

$$\begin{gathered} W : Var \times Inputs^* \to Bool \\ W(v,w) = \begin{cases} true & w \in W(v) \\ false & w \notin W(v) \end{cases} \\ U : Var \to Inputs^* \\ v \mapsto v \cdot get \end{gathered}$$

This is because we can clearly specify a function which "scans" a string w for an occurence of any $z \in Z$ followed by $v \cdot put$. But then the axiom reads

$$W(v, w) = false \Rightarrow \omega(s,\ w \cdot U(v))(out) = \omega(s,\ u(v))(out)$$

Note that in Example 4.1 we can reduce the axiom to a set of conventional conditional axioms. That is,
$i \cdot put \notin w$ **&** $arreset \notin w \Rightarrow\ \omega(s,\ w \cdot i \cdot get)(output) = \omega(s,\ i \cdot get)(output)$

equivalent to:

1. $i \neq j \Rightarrow \omega(s,\ j \cdot z \cdot put \cdot i \cdot get)(output) = \omega(s, i \cdot get)$
2. $\omega(s,\ j \cdot get \cdot i \cdot get)(output) = \omega(s,\ i \cdot get)(output)$
3. $\omega(s,\ length \cdot i \cdot get)(output) = \omega(s,\ i \cdot get)(output)$

The question remains, is there an easy way to handle "inertia" without having to write down so many axioms?

9 Inheritance in Automata Specifications

The word "inheritance" is used for a wide variety of vaguely related concepts. At least some of these concepts are easily adaptable to automata specifications.

9.1 "Pure Inheritance"

By "pure inheritance" we mean the ability to define a new automata from an old one in such a manner that the new automaton extends the behavior of the old one and that this is done in a manner that does not employ any knowledge of the descriptors of the old automaton but only its behavior.

Since the new automaton does not make use of the descriptors of the old one, this construction is just a special case of the use of descriptors in defining an automaton. Let the old automaton have inputs I, outputs O, transition function $\langle \sigma, t\ \omega \rangle : S \times I \to S \times I$. The specification for the new automaton is then of the form

Specification name:

Inputs I', where $I \subseteq I'$

.

Outputs O', where $O \subseteq O'$

.

Descriptors

$old \ : \ old - automaton$

possibly additional descriptors

Axioms
For $s' \in States', \ o \in O$ and for every significant input string, v,

1. $\delta'(s, \ v)(old) \overset{+}{\sim} \sigma(\delta'(s', \ \lambda)(old), v)$
2. $\omega'(s, \ v)(o) \overset{+}{\sim} \omega(\delta'(s', \ \lambda)(old), v)(o)$
3. possibly other axioms.

end of specification

Example 9.1 Given the earlier specification for an extendible array, $ExtArr$ we define a pointer array, i.e., an extendible array with a pointer that points to some admissible element of the array (i.e., an element with index less than the length of the array).

Recall that for an extendible array we had: **name:** <u>Ext</u>endible<u>Arr</u>ay

Inputs

value : string,integer

index : natural

$put, \ get, \ length, \ arrset$

Outputs

$out \ : \ integer + error$

To this we will implicitly add:

- a new descriptor $old \ : \ ExtArr$
- for each admissible input string, v for ExtArr, axioms
  - $\omega(s, \ v)(out) = \omega(\delta(s, \ \lambda)(old), \ v)(out)$
  - $\sigma(s, \ v)(old) = \omega(\delta(s, \ \lambda)(old), \ v)$

To this we will explicitly add:

- new inputs, $ValAtPtr$, up, $down$
- a new descriptor, $ptr \ : \ natural$
- axioms: For $s \in States$

1. $\omega(s,\ ValAtPtr)(out) = \omega(\delta(s,\ \lambda)(old),\ \delta(s,\ \lambda)(ptr) \cdot get)(out)$
2. $\omega(s,\ up)(ptr) = \omega(s,\ \lambda)(ptr) + 1$
3. $\omega(\delta(s,\ \lambda)(ptr) > 0 \Rightarrow \omega(s,\ down)(ptr) = \omega(s,\ \lambda)(ptr) - 1$
4. $\omega(\delta(s,\ \lambda)(ptr) = 0 \Rightarrow \omega(s,\ down)(ptr) = 0$
5. $\delta(Empty,\ \lambda)(ptr) = 0$

Remarks:

- The use of implicit structure in the above treatment of inheritance should save effort in writing specifications. This is, of course, a practical rather than a theoretical, advantage.

- We could add an axiom on the *up* operation that would keep it from "raising the pointer to high ". Without such an axiom we can get *error* values.

- The above definition of inheritance is, if I am using the lingo appropriately, sometimes called "is-a" inheritance as the new ptr-array is also an ExtArr in the sense that when restricted to ExtArr inputs it has exactly the behavior of an ExtArr. Mathematically speaking, ExtArr is a reduct of PtrArr.

□

9.2 Impure Inheritance

Impure inheritance is where descriptors are inherited in addition to the inputs and outputs. Such impurity clearly compromises encapsulation. Thus it is probably not a good idea in many cases. For example, if our specification for stack is based on PtrArr and we can get at the descriptors of PtrArr then we face the temptation of using the part of the array above the pointer as a place to keep special information.

If we go one step further, and explicitly carry over the axioms (rather than just the abstract behavior) we run into possibly severe difficulties such as having to include additional axioms to, for example, ensure that on has the desired inertial properties. For example, if we do PtrArr but with the inheritance of axioms then we will want to have an axiom of the form

$$\omega(s,\ ValAtPtr \cdot v \cdot get)(out) = \omega(s,\ v \cdot get)(out)$$

10 Related Literature

The comments here are very preliminary and subject to change upon further reflection. I would appreciate hearing about other references that should be considered.

10.1 Goguen–Meseguer: Abstract Machines

In 1985 Goguen and Meseguer, [3], introduced a notion of abstract machines and their behavior. The key definitions can be given as follows:

Given an S-sorted signature Σ, let $V \subseteq S$. Then given a Σ-algebra M let $M_V = \langle M_v \mid v \in V\rangle$. Then we can view M as a $\Sigma(M_V)$-algebra and that gives us a unique evaluation mapping (homomorphism) $\varepsilon_M : T_\Sigma(M_V) \to M$. The pair $\langle M_V, \varepsilon_M\rangle$ constitutes the behavior of the abstract machine $\langle V, \Sigma\rangle$ machine corresponding to M.

We say that two Σ-algebras, M and M' are *behaviorally isomorphic* if there exists a V-indexed bijection $\alpha : M_V \to M'_V$ such that $\alpha(\varepsilon_M(t) = \varepsilon_{M'}(\alpha^\sharp(t))$ where $\alpha^\sharp$ is the unique extension to a $\Sigma(M_V)$-homomorphism of the mapping $\eta \bullet \alpha$ where $\eta : M_V \to T_\Sigma(M_V)$ is the evident inclusion.

Here are some comments on the differences and similarities of the underlying concepts in this paper and Goguen and Meseguer's paper.

- each significant input string can be regarded as an operator. For example, in Example 3.1 the input string "$z \cdot push$" can be replaced by an operator $push : Z \times States \to States$. However we then have a problem with operators such as *pop* that both change the state and return a value. If we take the conventional approach then it is either an operator $pop : States \to States$ or an operator $pop : States \to Outputs$ but not both. This, of course, is a problem that has been around as long as the algebraic approach to data types – but it is avoided in our formulation of automata specifications.
- Our approach makes repeated use of strings of inputs (or strings of significant input strings). This corresponds to a restriction on the choice of Σ (we impose a monoid structure on the inputs). But, while we sacrifice some generality, we clearly gain in the ease of writing specifications for interactive programs (the programs that motivated our approach in the first place).
- We concern ourselves only with specifications that are "persistent".

10.2 Goguen and Malcolm: Hidden Algebras

The concept of a hidden algebra developed by Goguen et al (see [1] for a survey) seems to be closer to what we are doing but again involves a more general setting.

- In both approaches there is, what we call, a fixed algebra of primitives and the models of interest are the persistent models.
- While their terms as inputs appear superficially to be more general than our strings of input symbols, their restriction to operators having at most one variable of hidden sort produces very string-like inputs. However, since they have more than one hidden sort thay have greater syntactic control over the the set of "input strings". I conjecture that this corresponds to

defining a context-free set of input strings. But this is something we can easily do externally.

- They call an operator of the form $\rho : w \rightarrow s$ a *hidden generalized constant* if $w \in V^*$ and $s \in H$. Such "constants" are used for initialization of abstract machines. We do not have such things explicitly but they can be achieved by an axiom of the form

$$\sigma(s,\ \overline{a} \cdot \rho) \sim \sigma(s', \overline{a}w \cdot \rho)$$

  where $\overline{a} \in P^w$, and $s, s' \in States$.

- Our approach permits operators, such as the *pop* operatation for stacks, that have "side effects"; that is, they both change the state and return a result.

- Their contexts appear to be closely related to our $\sim$ and $\stackrel{+}{\sim}$ relations. More general in that they can specify any desired set of inputs, but perhaps less general in that they must apply to all states (of a given sort).

- We are using only one "hidden sort", namely $States$.

- It is not obvious that they have anything like our notion of aspects. However a comment on the bottom of page 65 of [1] says

  > "The definition also allows multiple hidden sorts, which are useful for (in the jargon of the object paradigm) *complex attributes*; these are class valued attributes."

  This jargon can be interpreted in a manner that suggests something akin to our descriptor concept. That is, descriptors can be viewed as class valued instance variables.

  I think that if we extend their framework to conditional axioms then we can get something very like descriptors. To model descriptors all we need do is think of H as consisting of a special sort for "the $States$", plus a sort for each descriptor. Then our axioms carry over to this framework.

  What is lacking then is the view of descriptors as being descriptive – i.e., in the hidden algebra approach they are implicitly all equal.

- it is not clear how they would handle concepts such as our concept of inertia since they don't have an obvious (to me) way to quantify over terms that is analogous to the way that we can quantify over strings.

- They are only using equations while we use conditional axioms – but their remark on page 75 in the last paragraph of their section 2, suggests that they also have a conditional axiom version. Indeed, the concepts, though not the theorems, of section 2 of [1] are valid in any institution (if not beyond institutions). That's not quite true, since the semantic part of the institution will vary from institution to institution so the contexts given in [1] will not suffice for the general case.

References

[1] J. Goguen and G. Malcolm. A hidden agenda. *Theoretical Computer Science*, 245:55–101, 2000.

[2] Joseph A. Goguen, J. W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, IV, Data Structuring*, pages 80–149, Prentice-Hall, 1978.

[3] J. A. Goguen and J. Meseguer. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541, Cambridge, 1985.

[4] http://www.extremeprogramming.org

[5] Till Mossakowski and Markus Roggenbach. The Datatypes REAL and COMPLEX in CASL. CoFI Note M-7, Version 0.2, 13 April 1999, available at http://www.brics.dk/Projects/CoFI/Notes/M-7

[6] Astesiano, Kreowski and Krieg-Brückner, Eds. *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, Springer (1999).