Chapter 3

Theoretical Issues

3.1 Introduction

In the previous chapter, I have discussed reasons why empirical modelling is good for geometry, and why strong geometry is beneficial for empirical modelling. In this chapter, the technical issues faced in trying to develop the relationships between these two types of modelling are described, in terms of both conceptual issues and the limitations of current tools that support empirical modelling. It is important that the representation of geometric data and dependency between that data in the implementation of such a tool is clearly consistent with the geometry itself. Solving the problems posed by the technical issues is the motivation for the research presented in the chapters that follow.

Interaction with geometric models reveals the integrity of the geometry. It is often the case that what is apparently one piece of geometry is actually an assembly of several component parts. A mechanism is required to manage the components and their dependencies so that interaction, reference and redefinition of the subcomponent parts are consistent with the overall definition of the assembled geometry in a definitive script. The way in which interaction with a geometric assembly affects a copy needs to be appreciated by the modeller. An overview of the issues discussed in this chapter is introduced in the following itemized list.

- The copying of geometric assemblies is a desirable feature in an interactive notation for geometric modelling. What is the status of the internal subcomponent definitions of a copied assembly and its dependency on the original assembly?

- Dynamic instantiation and removal of geometric entities is a desirable feature in some applications, for instance, drawing graph paper with scales that depend on the contents of particular data sets. If component geometry is at one level of definition in a script, are there higher-levels of abstraction in definitions in a script dynamically to control the lower-level geometry consistently with some high-level model?

- Where dependencies exist between geometric entities, these entities can be defined in many different modes. A mode describes the level in the data structure of an entity at which its defining parameters are given by definitions or by construction with constructors. How does this process affect a modeller's interaction with geometry in a modelling environment containing dependency?

- Structure in data can be considered as a form of dependency between values of different types. What is the relationship between dependency and data structure?

- Geometric data is often of a continuous rather than a discrete nature. A straight line can be defined by two end points and it is also a point set defined by a set membership condition of the set of points that lie directly between the end points. A straight line can also have infinite length, where it is parametrised by a vector and a point through which is passes. Dependency between geometric entities may need to be defined in terms of point sets rather than defining parameters. By what methods is it possible to represent geometric *continuous* data in an environment containing dependency on a discrete computer system?

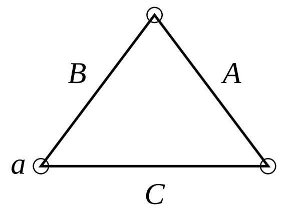
- Geometric data can be defined by parameters, is represented by point sets and is often associated with many attributes for data such as colour, texture, transformation, line width and so on. How is it possible to associate attributes with geometric entities where there is dependency?

Each of these questions is considered in separate sections of this chapter. Firstly, issues relating to empirical modelling tools that capture aspects of geometry are considered Section 3.2. This is followed in Section 3.3 by discussion of the relationship between data structure and dependency, with motivating examples relating to geometry. Secondly, issues relating to integration of good geometry with empirical modelling are considered. Material in Section 3.4 examines issues of data representation, the creation of good computer-based representations for geometric data that can be beneficial to a modeller and consistent with real world experience of geometry. Finally, Section 3.5 presents the basis for a solution to the problems posed by the technical issues raised. This is the basis for the practical work presented in chapters 4, 5, 6, 7 and 8.

Where possible, examples of existing methods for handling the technical problems that arise from the technical issues are presented. Often these methods are only partial solutions and not suitable for the representation of complex geometry. In particular, Section 3.4.1 examines the issues of representing geometric data within the EDEN generic dependency maintainer tool [YY88, Yun90].

3.2 Definitions and Dependency with Geometry

Dependency in geometry cannot be observed through static inspection. It is interaction with artefacts that reveals the integrity of their geometry. A polygon is a geometric entity made up of interdependent lines, where the end point of every line is also the end point of another. This relationship is a form of dependency that is common in geometry. Empirical modelling principles can be applied to represent this dependency in a script of definitions. A polygon can be considered as an *assembly* of integrated subcomponent lines with dependencies between them. An *assembly* is defined as a grouping of definitions in a definitive script that combine to construct an entity that contains subcomponent dependencies. The polygon and its component lines can each be uniquely identified and the indivisible relationships between the lines expressed as definitions in a definitive script.



```
B = line(a, 1 @ pi/3)

A = line(B.2, 1 @ -pi/3)

C = line(A.2, a)
```

Dependency relationships in empirical modelling are acyclic and therefore there is a need to define a starting point on which all the lines depend. Consider the triangle T shown above. With definitions in definitive scripts, it is not possible to define the start of line A to be the end of line B, the start of C to be the end of line A and the start of B to be the end of line C, because this is a cyclic dependency. With the introduction of one more definition for point a it becomes possible to define the start of line A to depend on a and the end of line C to also depend on a. The loop of cyclic dependency is removed in this way.

Triangle T is an assembly that integrates straight line subcomponents. The construction of assemblies may take the form of one definition for the whole assembly, for example "$T = \mathit{triangle}(a, ||A||, ||B||, ||C||)$"[1] or a grouping of separate definitions for each separate line[2]. This section of the chapter examines the issues for the definition and subsequent reference to or redefinition of the subcomponents of an assembly definition.

The first issue discussed in this section (Section 3.2.1) concerns the status of a copy of assembled geometry. In a graphical drawing tool such as *xfig*, when an object is copied its new instance is like a photograph of the original in the state that the original is in at the exact moment of the copy operation. The copy is completely independent of any subsequent changes to the original. If S is a triangle defined by indivisible relationship to be a *copy*[3] of T, i.e. a definition of the form "$S = \mathit{copy}(T)$", the subsequent states of S and T are in some way connected. Dependency introduces ambiguity into the concept of *copy* and a modeller needs to be aware of the subtle qualities of the different kinds of copy possible by definition. Ambiguity can be introduced in respect of three types of activity:

- redefinition of T or its subcomponents;
- referencing subcomponents of T;
- privilege to make redefinitions of subcomponents of S.

The redefinition of T or its subcomponents, making reference to the subcomponents of S and the permission to make redefinitions of the subcomponents of S can all have more than one possible interpretation in terms of the future state of the script and may not match a modeller's expectation of the effect of the copy.

[1]The length of a line x is given by "$||x||$".

[2]The second grouping method is similar to the DoNaLD *openshape* notation construct.

[3]Copying is a general term that admits the possibility of operations such as rotation, scaling and translation of shapes in a geometric modelling environment.

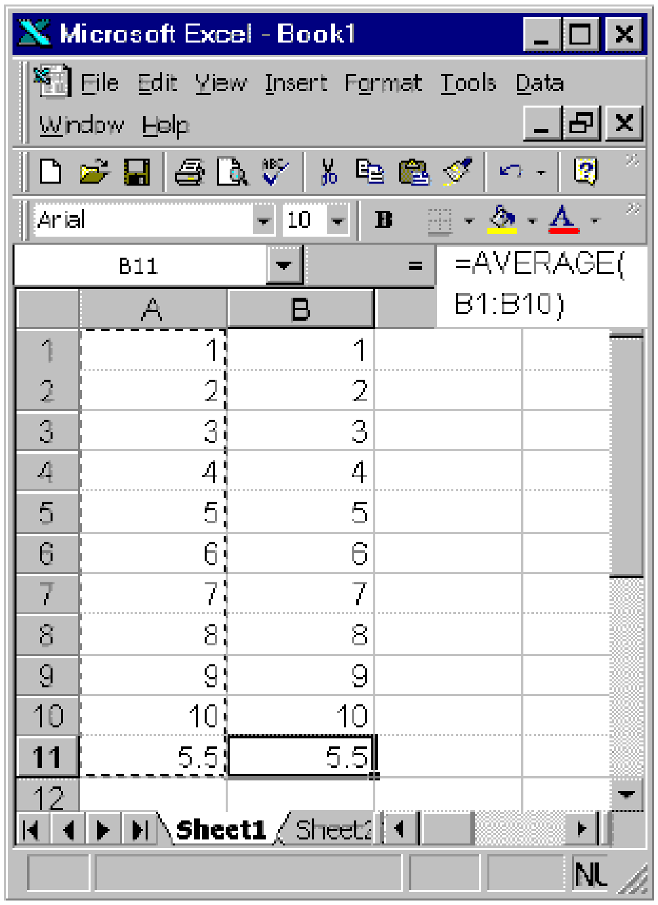
Dependency introduces ambiguity into the concept of *copy* and a modeller needs to be aware of the different kinds of equivalent copy possible by definitions.

In Section 3.2.2, the levels of abstraction for the definition of assemblies are demonstrated. At a low-level of abstraction, a script may have operators and data types for geometry such as triangles, rectangles, hexagons, circles and so on. At a higher-level, an operator such as *polygon* may be available to construct assemblies of lines to represent a generic polygon. In the high-level notation, a polygon is defined to depend on a parameter representing its number of sides. Changing the value of this parameter directly influences the number of component lines in the assembly of the geometry at the lower-level. If definitions in the low-level script depend on the lines that represent a high-level polygon, what happens if the value defining the number of sides is redefined? This issue often creates a complex definition management problem, especially if the number of sides is reduced. The references that low-level definitions depended on for their evaluation may no longer exist.

The *polygon* operator is an example of *higher-order dependency*. Higher-order dependency and the different kinds of copy are compared in Section 3.2.3.

3.2.1 Definitions for Copying Assemblies

The process of making a copy of an object in a definitive script is very different from the style of *cut and paste* copying provided by tools such as word processors and drawing packages. The reason for this is that there is the need to consider the effect of dependency maintenance between the state of original assembly and its copied instance. An example of this in an application that contains dependency maintenance is the use of the *copy and paste* system for a block of cells in a spreadsheet. The default method for copying cells is one of many possible paths by which a copy could be carried out. Subsequent alteration to the values in these cells by a user may cause the update of other cells in a way that the user is not expecting.



B11 = =AVERAGE(B1:B10)

	A	B
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	10	10
11	5.5	5.5
12		

Figure 3.1: Copying a region of cells in an *Excel* spreadsheet.

Close examination of the traditional copying operation in a spreadsheet application illustrates some of the issues. Consider the spreadsheet shown in Figure 3.1. The first step in its creation was to enter the values in cells **A1** up to **A10**. Cell **A11** is then defined by a formula to be the average of these cells. The next step was to highlight cells **A1** through to **A11**, copy them and then paste them into column **B**. Cell **A1** contains only a number and not a formula, so only it's explicit value is copied. Any subsequent change to **A1** will not then update **B1**.

Cell **B11**, a copy of the formula in cell **A11**, is copied in a particular way. The formula from **A11** is not copied into **B11** verbatim, character by character. Instead, the range for the evaluation of the average formula is substituted with cells **B1** through to **B10**. Cell **B11** remains independent of cells **A1** to **A11**, even if the users intention for the copy is that cell **B11** is equal to the average of **A1** through **A10**. This would be the effect of an exact character by character copy of the formula. Many spreadsheet applications provide some form of a *special paste* that allow a user to set the future behaviour of a paste operation to suit subsequent interaction with a spreadsheet model[4].

A definition for an observable is either explicit of implicit. An *explicit definition* is of the form "*identifier* = *value*", where the identifier on the left-hand side is associated with the explicit value on the right-hand side. An *implicit definition* is of the form "*identifier* = *function*(*arguments*)", where the identifier on the left-hand side is associated with a value that is evaluated by applying the function to the sequence of arguments.

The process of copying a single definition that is for an explicit value, has only two possible outcomes in term of definitions to represent that copy in future states of the same script. Consider the example definitions "$a = 3$" and "$b = copy(a)$".

[4]The spreadsheet used for this example is Microsoft's *Excel 97* [KDS96, Jac97], which includes a "Paste special" operation to give a user additional control over copying of cells.

The definition of b is a new definition that a user passes to a tool for dependency maintenance that can implement the *copy* process for a in one of two ways, as shown in the two cases below. To name the different kinds of copy, analogies are drawn to mechanisms used to copy images in the real world.

Photographic copy The value associated with identifier b is set to be equal to the value of a at the moment of the copy. In this case, the definition "$b = copy(a)$" cannot remain in the script and should be replaced by "$b = 3$". This is similar to a photograph where the image in the picture remains exactly the same as the image viewed by a camera lens the instant that the shutter was open.

Mirror copy The value associated with identifier b depends on the value of a and subsequent change of the value of a changes the value of b. The definition of b is unchanged. This is similar to the way in which the image in a mirror continuously reflects the image of objects placed in front of it, including the motion of these objects.

When copying a single definition that is implicitly defined, there are at least three possible outcomes in terms of definitions to represent that copy. Consider the example definitions "$a = c + d$" and "$b = copy(a)$". Photographic and mirror copies for the implicit definition are similar in the list below to the types of copy for explicit definitions in the list above. Reconstruction copy is a more general case than the first two and can take more than one form. Reconstruction of an image involves the use of the same or apparently identical components of an original to make a copied instance.

Photographic copy The value associated with identifier b is set to be equal to the value of a in the state of the script at the point of the definition of b. For a value of $c + d$ of 4, the definition "$b = copy(a)$" should not remain in the script and is replaced with "$b = 4$". Subsequent change to the values of a, c or d does not affect the value of b.

Mirror copy The value associated with identifier b depends exactly on the value of a and any subsequent change to the value of a changes the value of b. The definition of b should remain unaltered.

Reconstruction copy The value associated with identifier b is defined by the same implicit formula as the value of a. The definition of b is replaced by the definition "$b = c+d$". Subsequent redefinition of a will not effect the value of b. Further interpretations of the copy of a are possible in this case, where implicit and explicit definitions are mixed in the right-hand side of the formula. For example, if the current value of d is 2 then the definition of b can be replaced by "$b = c + 2$". In this mixed argument example, the value of b is effected by subsequent change to c but not by subsequent change to d.

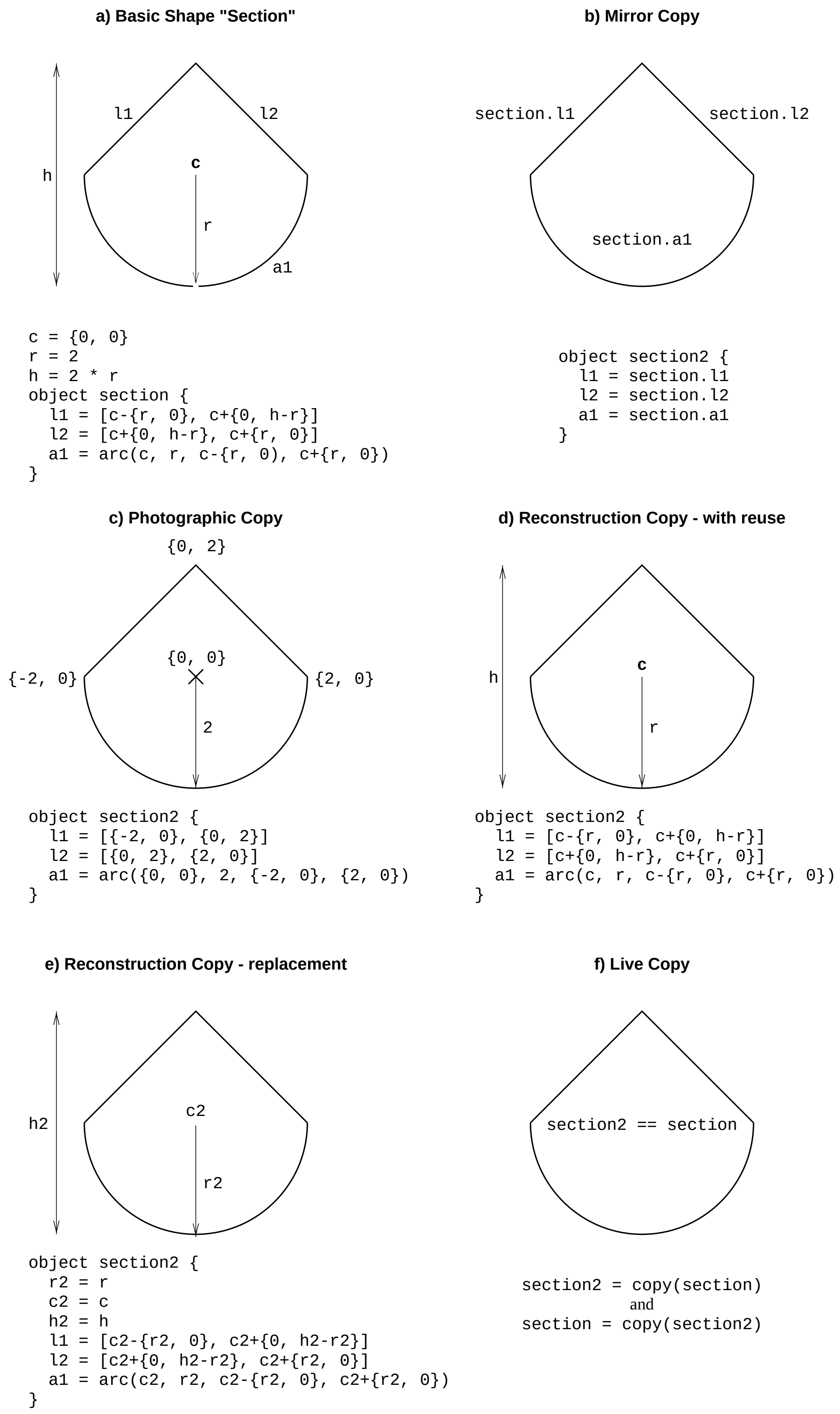
All the cases above are representations of copying processes that define the value of the copy to be identical to the value of the original definition at the moment that the *copy* definition for b is introduced into the script. The process by which the copy is represented in the creation of new definitions in a script determines the effect that subsequent redefinitions have in the propagation update of the current state of the values in the script.

With an assembly of geometry, there are several more cases possible than those listed for one single definition. If T is an assembly of some geometry, the definition "$S = copy(T)$" can have many interpretations for the definition of the subcomponents of the assembly. For every component definition of an assembly that is implicit, there can be any of the three types listed for single implicit definitions above. The effect of subsequent redefinition of the initial subcomponents on the copy is determined by the procedure for the creation of the copied instance by a tool for dependency maintenance. In general, it is better if every subcomponent definition is handled in exactly the same way to avoid ambiguity. It may, however, be necessary to consider mixtures of the procedures presented for copying subcomponent definitions in certain applications.

Figure 3.2 is split into six parts. Each part contains an identical piece of geometry defined by an assembly of definitions and the script to represent that geometry. A brief synopsis of the script notation used in the figure is presented in Table 3.1. The diagram of the geometry is labelled with parametrisations that correspond with the associated script of definitions and highlight the differences between different scripts the represent the same piece of geometry. The differences relate to the dependencies between the subcomponents of the assembly and other parameters in the script. In Figure 3.2a, the basic shape "`section`" represents the assembled geometry that is considered as the original geometry. This is copied in the other parts of the figure. This original geometric assembly consists of two lines ("`l1`" and "`l2`") and an arc ("`a1`"). The defining parameters of the geometry are a centre "`c`" and a radius "`r`". The subcomponent geometry of the *arcline* assembly depends on these parameters.

Each part of Figure 3.2 other than the top left hand corner represents a copy of "`section`" that is called "`section2`". These parts correspond to possible interpretations of the definition "`section2 = copy(section)`" and the script shown in each of these parts details the resulting definitions that persist in a script to represent the copy. The following list describes each part of the figure in more detail.

b - Every single subcomponent definition in the *mirror copy* defines the value of the associated identifier in the copy to be equal and dependent on the value of the identifier with the same name in the original assembly. Any subsequent



```
c = {0, 0}
r = 2
h = 2 * r
object section {
  l1 = [c-{r, 0}, c+{0, h-r}]
  l2 = [c+{0, h-r}, c+{r, 0}]
  a1 = arc(c, r, c-{r, 0), c+{r, 0})
}
```

```
object section2 {
  l1 = section.l1
  l2 = section.l2
  a1 = section.a1
}
```

```
object section2 {
  l1 = [{-2, 0}, {0, 2}]
  l2 = [{0, 2}, {2, 0}]
  a1 = arc({0, 0}, 2, {-2, 0}, {2, 0})
}
```

```
object section2 {
  l1 = [c-{r, 0}, c+{0, h-r}]
  l2 = [c+{0, h-r}, c+{r, 0}]
  a1 = arc(c, r, c-{r, 0}, c+{r, 0})
}
```

```
object section2 {
  r2 = r
  c2 = c
  h2 = h
  l1 = [c2-{r2, 0}, c2+{0, h2-r2}]
  l2 = [c2+{0, h2-r2}, c2+{r2, 0}]
  a1 = arc(c2, r2, c2-{r2, 0}, c2+{r2, 0})
}
```

```
section2 = copy(section)
         and
section = copy(section2)
```

Figure 3.2: Different procedures for the copy of an assembly of geometric definitions.

Expression	Description
id = *expr*	Definition of identifier *id* to be equal to the right-hand side expression *expr*.
{p, q}	A point on the drawing plane defined by scalars p and q.
[m, n]	A line on the drawing plane with end points m and n.
arc(c, t, p1, p2)	An arc on the drawing place with centre c, radius r, start point p1 and end point p2.
object o { ... }	An assembly of definitions called o constructed from a group of subcomponent definitions.
x.y	Reference to subcomponent y of assembly x.

Table 3.1: Synopsis of the script notation used in Figure 3.2.

redefinition of the original l1, l2 and a1 will propagate to effect the values associated with the copy.

c - The *photographic copy* defines the assembly section2 to be equal to the values associated with the definitions in the original section in the state as they are in at the instance of the copy. Subsequent changes to the original shape or its defining parameters will not propagate through to update the copy instance.

d - For the *reconstruction with reuse copy*, subsequent redefinition of the original l1, l2 and a1 parameters does not propagate to the copy. However, the change of the value through redefinition of the defining parameters c, r and h will.

Other possible interpretations of a copy are illustrated in Figure 3.2e and Figure 3.2f. The *reconstruction with replacement copy* of Figure 3.2e is a variant of the *reconstruction with reuse copy.* The right-hand side of the definitions for l1, l2 and a1 of section2 are similar to the right-hand side definitions for section with the parameters c, r and h replaced by c2, r2 and h2 respectively. At the exact moment of the copy, c2 is defined to be equal to c, r2 equal to r and h2 equal to `h`. Subsequent changes to the parameters `c` and `r` will effect both the original (`section`) and the copy (`section2`). The definition of the defining parameters can be redefined to break their link with the original defining parameters. The copy can become independent of the original in this way. This reconstruction copy can be viewed as a way of creating templates of dependency that can be instantiated and customised by their parameters.

Figure 3.2f illustrates a script in which the high-level definition "`section2 = copy(section)`" is the persistent definition of the copy and there is no copy of the assembly of definitions. This copy is like the image on a television set of a live broadcast, where whatever happens in front of the camera lens in the television studio is shown on the television screen. In this case, there is no redefinable subcomponent geometry for `section2` as definitions in their own right. If there were, redefinition of a subcomponent would cause `section2` to be no longer consistent with its definition as a copy. This kind of copy does not fit well in current definitive programming environments. There are several possible ways in which subsequent interaction with the subcomponents of `section` and `section2` can be handled by a dependency maintaining tool. For instance:

1. Attempts to redefine the subcomponents of `section2` by the user cause a runtime error to be reported.

2. A redefinition of the subcomponents of `section2` is considered as a redefinition of the same subcomponent of `section`. Using this procedure, the definitions "`section2 = copy(section)`" and "`section = copy(section2)`" can coexist in the same script.

In this section, several different procedures for copying assemblies of geometry with definitions have been discussed. Note that if there were no assemblies

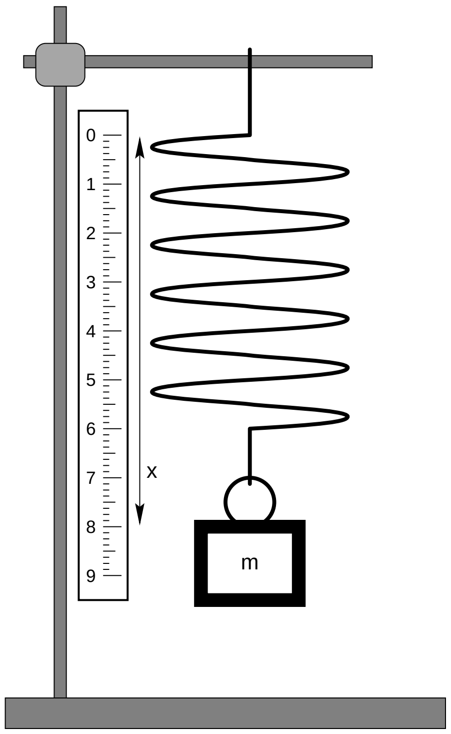


Figure 3.3: Mass on a spring experiment.

of definitions then there would be far fewer possible procedures for copying single definitions.

3.2.2 Higher-Order Dependency

The process of model making requires a period of observation of a real world phenomena and the recording of quantifiable data during a period of experimentation. This data can then be analysed, possibly through the preparation of a graph or other graphical representation, in such a way that patterns of behaviour that exist between certain observables can be determined. In the empirical modelling paradigm, this dependency between observables can be represented in a definitive script. Once the patterns between primitive observables have been represented in definitions, it may be that it is possible to observe patterns in the definitions that could be expressed in a higher-order definition.

For example, take the experiment that demonstrates Hooke's Law as shown in Figure 3.3. Increasing the mass increases the length of the spring and reducing the mass shortens the spring length. Experiments with several different springs can be carried out to measure the relationship between mass and extension. From the data set of each experiment there can be observed, within a certain margin for error, a proportional relationship between the mass and the extension. For each different spring tested, it is possible to express as a definition the observed relationship between mass and extension by finding an explicit constant for the spring concerned.

At some level of abstraction above the single spring on experiment, it can be observed that there is a common dependency to many spring experiments. Each has the same template definition describing the relationship between the length of the spring and the mass, the only difference is some constant of multiplication. This pattern between definitions can then be represented as a template definition describing all experiments involving masses suspended on springs.

This is similar to the *reconstruction with replacement* process shown in Figure 3.2 and described in Section 3.2.1 of the chapter. In this copying process, the defining parameters for a geometric assembly are identified independently of the dependency between the subcomponents and a local version of these parameters is created in the copied assembly. The equivalent in higher-order dependency is that these parameters become arguments to a high-level implicit definition, where the value associated with the left-hand side identifier of the definition is an assembly. With no original to copy from, the assembly is created by the high-level definition. The higher-order definition is in a separate definitive script notation at a level of abstraction above the representation of the low-level dependency for the geometric assemblies.

The table below shows high-level definition and the associated low-level ge-

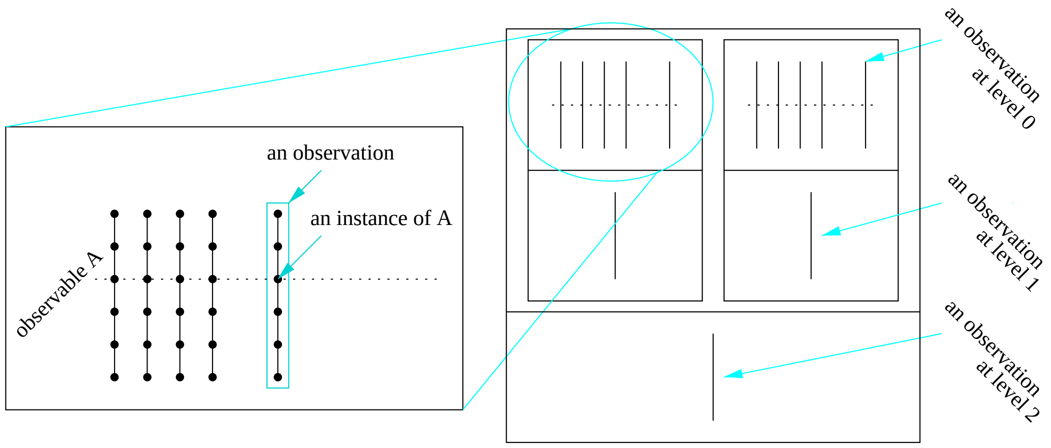

an observation
an instance of A
observable A
an observation at level 0
an observation at level 1
an observation at level 2

Figure 3.4: Levels of abstraction in observations.

ometric assembly generated by the high-level definition. In the high-level notation, the definition below of the arc and two line shapes called "*arcline*" will generate and maintain dependency for the assembly in the low-level script. The geometry of the *arcline* is the similar to the examples in figure 3.2. Redefinition of the value of r or the point c at the high-level will redefine the explicit values of `section3.r` and `section3.c` at the lower-level.

High-Level $section3 = arcline(r, c)$

Low-Level
```
object section3 {
   c  = Current value of c at high-level.
   r  = Current value of r at high-level.
   h  = 2 * r
   l1 = [c-{r, 0}, c+{0, h-r}]
   l2 = [c+{0, h-r}, c+{r, 0}]
   a1 = arc(c, r, c-{r, 0}, c+{r, 0}]
}
```

This process of observing phenomena from patterns in experimental data to form higher-level definitions at many increasing levels of abstraction is generalised in Figure 3.4. The advantage of characterising common patterns in observed real world data is that it is often more appropriate to represent these patterns rather than to model the original data sets in their entirety. The diagram shows three separate levels in the abstraction of higher-order dependency that are also listed below.

Level 0 Observation of data that may be related in some way, over the variation of one or more parameters. In the figure, a single straight line represents a snapshot of a system's state and the dots represent the value of the observables during the snapshot. In the Hooke's Law example the parameter being varied is the mass and the effect on the extension is observed to see if it is possible to establish a dependency between these parameters at level 1.

Level 1 Observation of data from level 0 shows that there is a dependency between certain observed values. This dependency is represented as one level 1 definition. In the Hooke's Law example, the mass applied and resulting extension for one spring experiment is represented as a definition.

Level 2 Observation of a number of dependencies shows that there are similarities between the definitions that represent common level 0 phenomena at level 1. These patterns of dependency also be observed and can be represented in one level 2 higher-order definition. In the Hooke's Law example, the level 2 definition represents the fact that there is a linearly proportional relationship between mass and spring extension in all such experiments.

Level n It is possible to extend the process of observing higher and higher levels of dependency between sets of definitions up to some finite level **n**. For the Hooke's Law example, the top level possible is level 2 as there is only one definition at this level.

This process, when used to construct definitions in a script, is an observation-oriented method for identifying structure in observed dependency. It is motivated by observation of phenomena that already exist in the world and can be experienced in some way, even if that experience is one of some new conceptual model that has never been realised as a real world referent[5]. In a geometric context, the process can be used as a means to identify and represent common parameters and varying parameters between geometry that has a common basis. For example, a rectangular box has a height and a width at *level 1* but consists of four lines in a geometric assembly of definitions at *level 0*.

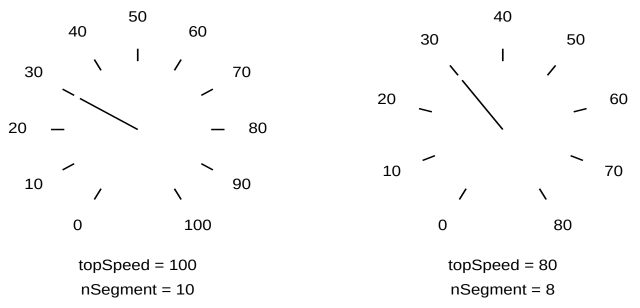
3.2.3 Comparison of Complex Dependencies

Many similarities exist between the many procedures for a copy in a tool that supports definitive scripts and the management of higher-order dependency. The two methods of structuring dependency for similar entities using assemblies and level of scripts are compared in this section. Higher-order dependency is a useful process for identifying patterns in real world observation for the construction of computer-based artefacts. Copying is a useful tool in the incremental construction of computer-based artefacts.

When a user interacts with a script, there are many ways in which a created object can be instantiated and linked to its original object component. The problems with many methods for copy, as detailed in Section 3.2.1, are associated with assemblies constructed from subcomponent definitions. Higher-order definitions allow a user to go from observation to reusable templates for making several lower-level definitions, whereas making a copied instance of an object is used as a tool in creation of objects in scripts. Tools that support empirical modelling with geometry should incorporate both support for higher-order template definitions and the ability to make copies of newly created definitions with a high degree of flexibility.

With higher-order dependencies, it is possible to create a template for geometry. This is similar to the way that a programmer creates a primitive object

[5]For further examples, see [GYC+96].



```
graph speedo
within speedo {
    real needleLength = 100.0
    real minA = 4 * pi div 3
    real maxA = - pi div 3
    real A = minA + (maxA - minA) * ~/curSpeed div ~/topSpeed
    line needle = [{0,0}, {needleLength @ A}]
    real gap1, gap2, LSpc
    gap1, gap2, LSpc = 10.0, 30.0, 50.0

    x<i> = ~/topSpeed * <i> div nSegment
    f<i> = minA + (maxA - minA) * <i> div nSegment
    nSegment = 8
    node = [
        label: label(itos(trunc(x<i>)), {(needleLength + gap2 + LSpc) @ f<i>});
        line: [{(needleLength + gap2) @ f<i>}, {(needleLength + gap1) @ f<i>}]
    ]
    segment = []
}
```

Figure 3.5: Two speedometer models and one template definition in DoNaLD that represents both models.

constructor that can be used in a drawing application, such as *xfig*. The generic template definition has a right-hand side that consists of parameters that will make the left-hand side of the definition reconfigure, where the left-hand object entity consists of assembled lower-level definitions. The example in the previous section for an "*arcline*" demonstrates how the defining parameters for the low-level dependency can depend on high-level values. The number of subcomponents of an assembly may also change depending on high-level defining parameters. Changing parameters on the right-hand side of the high-level generic definitions can cause the assembly of lower-level definitions to reconfigure.

One form of higher-order dependency is already implemented for the DoNaLD notation [ABH86], which is part of the *Tkeden* tool. It is known as the *graph abstraction*. Figure 3.5 shows generic template definitions in use in DoNaLD to describe the layout of a speedometer that has a top speed and number of division segments on the right-hand side. At the lower-level, there is a block of DoNaLD definitions representing the picture of a speedometer. Changing the number-of-segments parameter, considered as on the right-hand side of the high-level definition for the speedometer, leads to the reconfiguration of the low-level subcomponent definitions of the assembly of geometry.

The number of subcomponent definitions generated on the low-level left-hand side will change depending on the value of the special parameter "`nSegments`". The abstraction is considered as based on a two-dimensional plotted line with x values along the x-axis and $f(x)$ values along the y-axis. These are specified by definitions in the DoNaLD notation by descriptions of template definitions "`x<i>`" and "`f<i>`". Where "`<i>`" appears in these definitions, the elements of the sequence "0, …, *nSegments*" are used to create new low-level definitions for "`x_0`, …, `x_`*nSegments*" and "`f_1`, …, `f_`*nSegments*" respectively. The right-hand side of these definitions have the token `<i>` replaced by the index through the sequence for the definition. The "`node`" and "`segment`" definitions can be used to create a *mini script* of DoNaLD automatically for each segment of the graph, depending on the values of `x<i>` and `f<i>`. In the speedometer example, these mini scripts correspond to the speedometer graduation lines and labels.

The problem with the DoNaLD implementation is that if a user redefines a subcomponent definition then the graph is no longer consistent with its high-level definition. Maybe a designer wishes to change the length of the 70 miles-per-hour line segment to highlight the top speed limit on British roads. If the number of segments for the speedometer is increased, then this change will be lost and, in the worst case, there may no longer be a line segment corresponding to 70 miles-per-hour on the face of the speedometer. The order in which definitions are introduced into the script becomes important and the behaviour resulting from redefinition of parameters and the associated propagation of change introduces conflict. For one script of definitions there can be at least two possible states. The integrity of the state between the higher-level and lower-level of abstraction in scripts should be protected in some way.

The use of higher-order dependency in geometric modelling has a place once commitment can be made to the description of generic element templates through real world observations. In the creation of new geometry, these templates restrict the open-ended geometric design process. Higher-order abstraction in definitions is a tool that can be used for the constructing primitive geometric elements. Copying entities is a process that allows a user to take existing structure and reuse it without having to reconstruct of reason about dependencies between internal subcomponents at different levels of abstraction. The challenge of managing dependency between assemblies of definitions is to find a consistent and unambiguous way to support many kinds of copy and higher-order dependency in unified definitive notations.

3.3 Data Structure and Dependency

This section examines the issues of integrating arbitrary data structure and dependency into the same definitive notations. High-level programming languages provide data types for the representation of atomic data such as scalar and Boolean values. These atomic data types can be combined into arbitrary new data structures with *abstract data types* (ADTs) [AHU82] that represent data that is more complex than a variable of one atomic type can represent alone. The structure can be given an identity in its own right and mechanisms exist for accessing components of atomic or other constructed data types. Data structure is an association of component data types that can be regarded as a form of dependency. Definitions in definitive scripts represent indivisible relationships between variables, considered as observables, of any data type. In existing notations, definitions can record structural dependencies (as in the point constructor "`p = {x, y}`") but not reflect their special characteristics.

For example, consider the case-study of assemblies of geometry in Section 3.2. Data types exist in the example notation in Table 3.1 to represent scalar values (integer or floating point), two-dimensional points (two scalar values) and two-dimensional lines (start and end points). These are the data types, and every variable of the non-atomic data types has an internal structure: points are constructed from the atomic data type for scalars, lines are constructed from component points. These types are provided as an integral part of the notation and in order to represent more complex data a user must create assemblies that group definitions for values of these types. Dependency can be established between single variables of the built-in data types at different levels in their internal data structures, where the value of each component of data is either explicitly given or implicitly given by the evaluation of the right-hand side of the definition.

The level at which the definition of a variable of a non-atomic data type is constructed, rather than given by definitions, is known as its *mode*. The ARCA notation [Bey83, Bey86a] for scripts of definitions that describe Cayley Diagrams includes an implementation of *modes* for its variables. The process of assigning a mode to ARCA variables is known as *moding*. The relationship between levels in data structure and dependency is considered through the examination of moding and ARCA in Section 3.3.1.

Data structure describes the association of component values within a compound data type. Dependency in definitive scripts represents indivisible relationship between variables. In Section 3.3.2, the argument that data structure and dependency can be regarded as orthogonal concerns is presented. The possibility of representing one dependency between variables of compound types as several dependencies between their components at an atomic data type level is demonstrated. The ambiguities introduced by dependencies that are expressed by definition between variables at different component levels in data structures are discussed.

3.3.1 Moding and ARCA

ARCA was the first example of a definitive notation to be developed at Warwick [Bey83, Bey86a][6]. The data types and operators of ARCA are illustrated in Figure 3.6. This shows an ARCA representation of the symmetric group S_3 in three sections. From top to bottom in the figure, these are: a Cayley diagram for S_3, a graphical representation of its ARCA data structure and two segments of its script of definitions in the ARCA notation. The explicit definitions for the diagram can be subsequently redefined to experiment with the state of the diagram. In the tool developed by Bird that implements ARCA [Bir91], the screen image of the group reconfigures to reflect the current state of the definitions and their associated values

[6]Earlier work on definitive notations can be traced back to Brian Wyvill [Wyv75].

(see example ARCA output in Figure 2.1).

The main compound data structure in ARCA is the *diagram* that represents the coordinate location of the N component *vertices* in a diagram and the partial injection mapping from $\{1, \ldots, N\}$ to itself associated with edges of each *colour* in the diagram.

A variable (`x`) or a component of its structure (`x[1]`, `x[2]`, ..., `x[n]`) can be declared to be in either *concrete* or *abstract* mode. The significance of these modes is as follows:

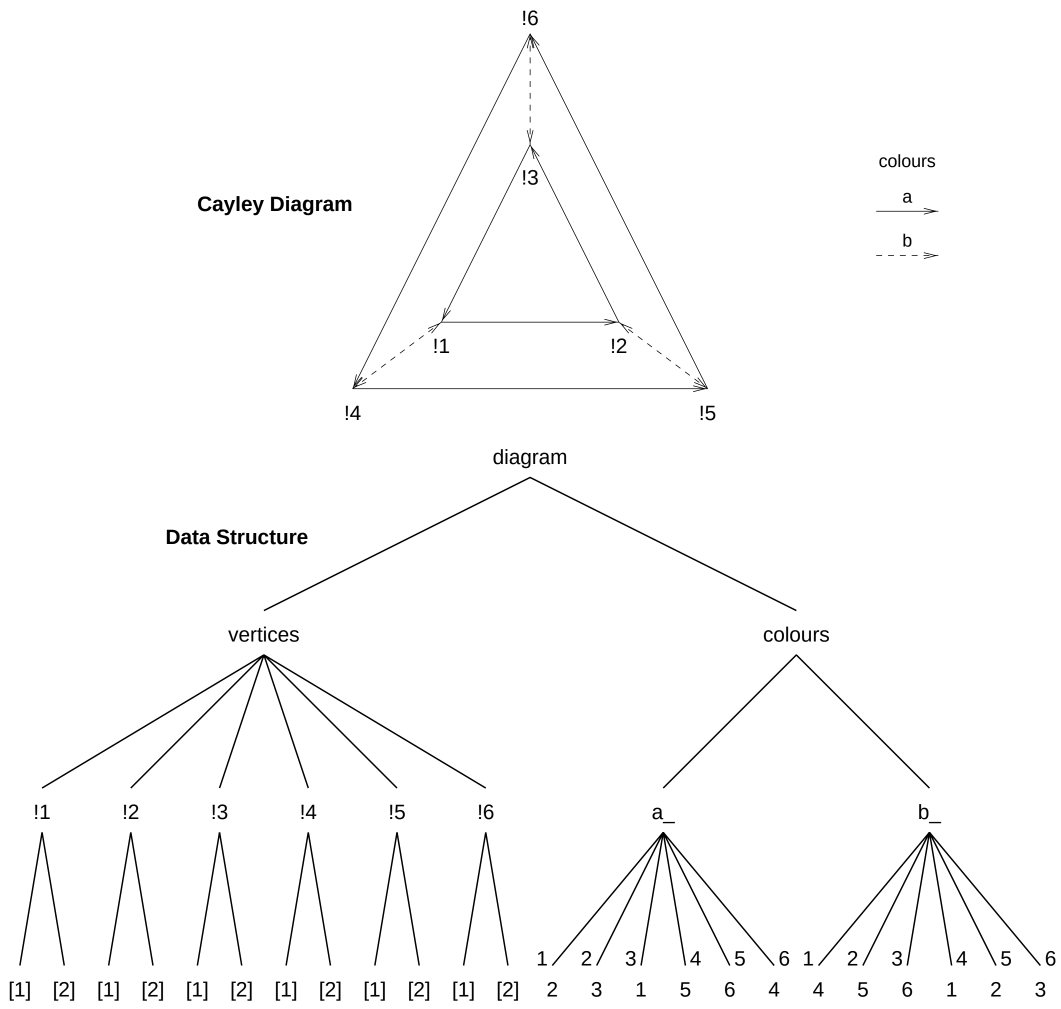
abstract mode The value of the variable or component is given by a constructor at the same level or defined by an implicit definition. It is not defined component-wise.

concrete mode The value of the variable or component is constructed from the definitions of its components.

In ARCA, if a variable is declared to be in abstract mode, then no declaration is required for the mode of its components. If a variable is declared to be in concrete mode then the mode of its components must also be declared.

The data structure for the diagram in Figure 3.6 is in concrete mode and all component values are in concrete mode down to its leaves. The start of the example script shows combined mode and type declarations for the identifiers to be defined by explicit values further through the script. A mode declaration of the form "`mode x = 'ab'-diag 6`" describes `x` as a diagram with $N = 6$ vertices and two colours `a` and `b`. Declaration of the mode of the component vertices and colours is then required.

Declaration of the dimension of each concrete ARCA vertex is of the form "`mode y = vert 3`", describing that the value of `y` is constructed from the values of its components `y[1]`, `y[2]` and `y[3]`. These components can be implicitly or



```
mode dia = 'ab'-diag 6
mode dia!1 = vert 2
mode dia!2 = vert 2
mode dia!3 = vert 2
mode dia!4 = vert 2
mode dia!5 = vert 2
mode dia!6 = vert 2
mode a_dia = col 6
mode b_dia = col 6

⋮
```

```
a_dia = {1,2,3}${4,5,6}
b_dia = {1,4}${2,5}${3,6}
dia!1 = [-200, -150]
dia!2 = [200, -150]
dia!3 = [0, 250]
dia!4 = [-400, -300]
dia!4 = [400, -300]
dia!6 = [0, 500]

⋮
```

Figure 3.6: An explicit ARCA diagram for the symmetric group S_3.

explicitly defined. The declaration of an abstract ARCA vertex takes the form "`mode z = abst vert`". This signifies that `z` is defined explicitly by a constructor that creates a vertex, or defined implicitly to be dependent on other vertices or scalar values. With this declaration for variable `z`, its value cannot be established by the definition of its components.

For every variable in ARCA, there are levels ar which its components are in abstract or concrete mode. These levels can be considered as establishing a moding template for definitions. These templates allow a user to establish dependencies between components that would otherwise be considered as cyclic dependencies (see Section 4.2.3). For example "`x[1] = x[3]`" is not a cylic dependency with concrete mode variable `x` constructed from components `x[1]` and `x[3]`. In contrast, an abstract mode variable `x` with definition "`x = [3, 2, x[1]]`" does lead to cylic dependency as the evaluation of the right-hand side depends on itself.

The ARCA notation treats the mode and type declarations as if they were definitions. In effect, the moding of variables is handled by an auxilary definitive notation. In theory, it is possible to redefine them and hence affect the mode and data structure associated with identifiers in a script through indivisible propagation of change. In practice, it is technically difficult to implement a system that can handle radical redefinition of the type associated with an identifier. For example, an implementor of the ARCA notation has to consider mechanisms for handling the effect of the interactive redeclaration of a variable of diagram type to be of vertex type, or redeclaring an existing abstract mode diagram to be concrete (so as to allow dependencies between its components to be specified).

In a high-level language, the underlying algebras over the atomic types include a look-up table for each operator. This table determines the type of the value returned by the operator, which depends on the types of the arguments. Type checking is performed during compilation of code to check that the data types declared to be associated with variable identifiers are consistent with those returned by right-hand side expressions. In ARCA, built-in operators in the notation exist that can be used in definitions to define indivisible relationships between values of compound data types. This requires that every operator has a look-up table for return type and its return mode.

Every variable in a definitive script such as ARCA has an associated mode and every expression in the script also has a mode. The reasons for moding expressions are:

- to attach a structure to values returned;
- to ensure this structure is consistent with the left-hand side variable in a definition.

For example, consider a concrete mode list `l` with three elements, declared as follows:

```
mode l = list 3
```

Consider also a general list operation `reverse` for list reversal. The mode of the expression `reverse(k)` is abstract, as the operator returns a list with a different number of component elements depending on its argument `k`. Due to the varying length of the returned list, it is not appropriate to define the three element list `l` by the definition

```
l = reverse(k)
```

or by the group of definitions

```
l[1] = (reverse(k))[1]
l[2] = (reverse(k))[2]
l[3] = (reverse(k))[3]
```

The concrete mode of the operator `reverse3` for reversing the order of three element lists is `list 3` → `list 3`. In this case, the mode definition "`l = reverse3(k)`" is appropriate as the returned value always has three components. Notice how the mode of a variable also affects the nature of a reference its value on the right-hand side of another definition. If a variable `v` is in abstract mode then it is possible to refer to the third component of the value of the variable `v` (through a projection operator `p3(v)`) but not to the value of the third component of the variable `v` (by `v[3]`).

Moding is required in a definitive notation with compound data types. The ARCA interface to moding can be improved to better support moding outside of the script notation. This improvement is hard to implement, especially to support the the degree of flexibility envisaged. The machine models developed later in this thesis handle this well and do more towards supporting user-defined data types and operators than EDEN. Existing tools are not good for implmentations that support moding, with particular problems posed by the representation of EDEN lists (discussed in Section 3.4.1). A solution is needed (such as will be described later) that not only allows user-definition of underlying algebra, but also deals with moding to accompany this.

3.3.2 Orthogonality between Data Structure and Dependency

In a definitive script, there can be dependency between components of the data structure for a variable and dependency between variables. In this section, the relationship between dependency and data structure is considered. It is appropriate to represent data structure and dependency as if they are orthogonal concerns. Every identifier and component reference in a script of definitions can have both a location in a data structure and be dependent on other data values through definition. A value is made up from component parts if it is of a compound data type and, as discussed in Section 3.3.1, can be defined in different modes.

In general, it is possible to find a way of representing all dependency given at non-atomic data type level as dependencies between component values of atomic types. If all dependency is represented at the atomic type level, then there is no ambiguity introduced as to which relationships are dependency between values and which relationships exist for data structure. All structure in dependency is between scalar values in the level of the atomic data types and all compound data types are constructed from these atomic values to form levels in data structures for high-level vatiables. In this way, data structure is considered in this thesis to be orthogonal to dependency by representing dependency at the scalar data structure level.

For example, Figure 3.7 shows a representation of the levels of data structure for a two-dimensional straight line data type. The line is represented by two component end points and each of these points is represented by a pair of scalar values. These scalar values are atomic types in the example. The figure is separated into three sections and the value of line variable `l` is the same in each. Figure 3.7a shows a line that is explicitly represented at all levels. Figure 3.7b and Figure 3.7c show the line with components defined implicitly.

Dependency between data of the same type at the same level is diagrammatically representable as embedded in that level. In Figure 3.7b and Figure 3.7c, it is possible to extract and describe a low-level of dependency between atomic scalar data types, another level of dependency for point compound data types and another for line compound data types. Dependency can also exist between data types at different levels. For example, an inner product operator maps two point arguments to a scalar value. The linking of different levels in data structure by dependency in this way leads to an ambiguity in the possible combined representation for the data values and the dependencies between them as variables in a script. The same dependency is shown represented in Figure 3.7c as in Figure 3.7b, except that it is

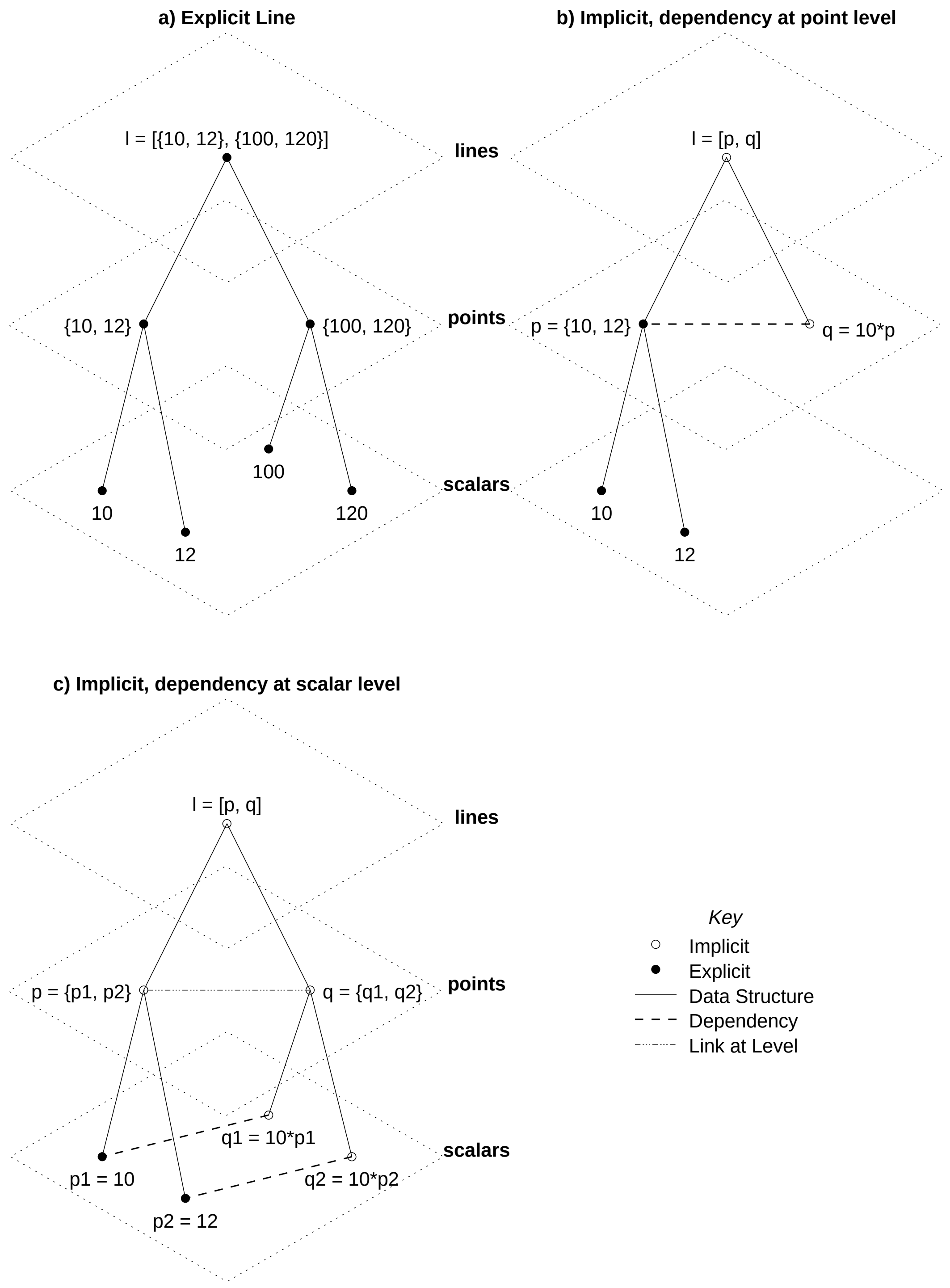


Figure 3.7: Levels in data structure and dependency for the definition of a straight line.

represented at the scalar level rather than the point level.

Data structure can be constructed by implicit definition. For example, a two-dimensional point value is represented by a data structure with component scalar values. A variable for a point can be constructed by implicit definition with a defining function `makePoint` that maps from two scalar values on the right-hand side to point variable on the left-hand side. In this case, it is not clear whether the point is a variable constructed in concrete mode or is implicitly defined. It is still possible to handle dependency between components in this case without introducing cyclic dependency. For example, consider the following definition for constructing point `p` from components `p1` and `p2`:

```
p = makePoint(p1, p2)
```

A dependency between `p1` and `p2` cab be expressed with a definition such as "`p1 = 2*p2`". A high-level definitive notation with support for moding can be represented at a lower-level by operators for constructors and dependencies between component definitions.

3.4 Data Representation and Dependency

In this section, methods for data representation for observed values that are appropriate to definitive scripts are considered, with a particular focus on geometry. To put this discussion in context, existing methods for the representation of geometric data in existing tools are presented. This is followed by a discussion of data where the parametrisation of entities is of a discrete nature but the actual representation of the value is apparently continuous and can be sampled at arbitrary values. Functions that are used on the right-hand side of definitions to create indivisible relationships between entities can operate over internal parametrisations of the entities and from a *continuous* representation of the entity (such as a function representation of its shape). Section 3.4.3 considers additional types of data that may be associated with geometric entities, such as graphical attributes, that ideally need to be represented in a definitive notation for geometry.

3.4.1 Existing Data Representations

The EDEN tool has built-in support for some atomic data types and one compound data type. The atomic data types represent integer values, floating point values, characters, strings of characters and a special type called *undefined* (represented by symbol "`@`"). The tool supports dynamic typing. The type associated with a variable is determined by interpreting the script of definitions and inferring types to be associated with identifiers on the left-hand side of explicit definitions from the string of characters on the right-hand side. For a implicit definition, there is a lookup table for each operator that determines the type associated with a variable from the arguments to the operator. The redefinition of a variable of integer type to be of string type may cause implicitly defined variables with numeric expression definitions to become undefined, the one and only value of the undefined type. Any other value that depends on an undefined value can itself become undefined.

This process is in one way very powerful as there is no need to declare variables before use to be of one particular type. The system can make sensible changes to types of variables and propagate type change through a script of definitions. A modeller using the EDEN notation benefits from this process, as they can concentrate on interactively constructing the model and identifying the best observables for a particular model without worrying about the types of the variables used to represent the observables. The redefinition of a variable that causes another implicitly defined value to become undefined is not reported to the modeller and the process of establishing where the relationships between variables cause a value to be undefined can be time consuming.

No facility exists for the creation of abstract data types in EDEN, DoNaLD or SCOUT. A *list* type exists that can be used to group integer, floating point, string, character and other list values. Elements of a list can be references and set by an index value inside of square brackets "`[]`" and the whole list can be referenced and defined by a comma separated list of implicit and explicit definitions. For example, consider the definition of the list `l1` shown below.

```
r = 10;
l1 is ["circle", 2*r, 10, 10.5];
```

The list `l1` has four elements and is used to represent the parameters of a circle shape in two-dimensional space, where the second argument is the implicitly defined radius that depends on the current value of `r` and the third and fourth elements represent the centre point. The centre point in this example is explicitly given by an integer value (`10`) and a floating point value (`10.5`). To declare that the list represents a *circle* shape, the first element of the list is set an explicit definition of the string of characters "`circle`". The EDEN parser establishes that the first element of the list is explicitly given and of string data type, that the second element is implicitly given and currently of integer data type and so on for all the elements.

What is the effect of the following redefinitions of the list `l1`? Redefinitions of lists and elements of lists may not result in the effect that a modeller expects.

1. `l1[2] is 3 * r;`
2. `l1[2] = 3 * r;`
3. `l1[3] is l1[2];`
4. `l1 is ["circle", 2*r, l1[2], 10.5];`

The first redefinition (1) fails with an error reported to the modeller because it is not possible to implicitly define only an element of a list is EDEN. The second redefinition (2) succeeds as it is an explicit definition that takes the current value of `r`, multiplies this value by three and assigns this as the explicit value of the second element of the list. This redefinition actually alters the definition of `l1` without the modeller explicitly requesting this redefinition. The third redefinition (3) fails with an error reported for the same reason as the first redefinition. It is not possible to introduce a dependency between elements of a list in this way, or in the declaration of an entire list as shown in the fourth redefinition (4). The EDEN interpreter regards the definition of a list with a dependency between its elements as a cyclic dependency.

The only way to identify the type of data in a list is by placing some marker such as the "`circle`" string in the list. Operators in EDEN are called *functions* and are defined on-the-fly by sections of interpreted procedural code. The operators to these functions have their types determined automatically. In an application that represents geometry, the use of operators over geometry represented in EDEN lists requires that the operator type checks its arguments to see if the arbitrary list passed as an argument contains the expected parameters.

It is difficult to represent the assemblies of definitions introduced in Figure 3.2 using lists in EDEN because of the problems of establishing dependency between elements of lists. One of the purposes of grouping definitions in assemblies is to represent dependencies between components of an entity, and there needs to be another way to represent these. The DoNaLD notation, implemented as part of the *Tkeden* interpreter, uses EDEN as its *back-end* dependency maintainer tool. All DoNaLD definitions are translated into EDEN and assemblies of definitions in DoNaLD known as *openshapes* are represented in EDEN by variable naming conventions.

Table 3.2 shows a DoNaLD script of definitions and some EDEN definitions that represent the DoNaLD script. The openshape construct is used to create an assembly of definitions. In the table the assembled definitions are `l` and `i2`. All DoNaLD definitions that are not part of an openshape are preceded by an underscore

DoNaLD Definitions	Translated into EDEN Definitions
`int i`	
`i = 10`	`_i is 10;`
`openshape test`	
`within test {`	
`  int i2`	
`  i2 = 20`	`_test_i2 is 20;`
`  line l`	
`  l = [{~/i, ~/i}, {i2, i2}]`	`_test_l is line(cart(_i, _i),`
`}`	`cart(_test_i2, _test_i2))`

Table 3.2: Translating DoNaLD openshapes into EDEN.

character "_" in EDEN and for an openshape called "x", all definitions are preceded by "_x_" in EDEN. In this way, dependency between assembled definitions can be represented in EDEN. However, because the current version of EDEN has no support for the automatic manipulation of this naming convention consistent with some higher-level script, the DoNaLD translator must retain a record of the EDEN names.[7].

3.4.2 Parametrised Data Sets

Conventional computer systems are finite machines that are good tools for the representation and manipulation of discrete data. When continuous parameters, often analogues of real world observables, are represented on a computer system, they need to be digitised from the measured analogue parameter and represented discretely within a margin for error. Idealised geometry in Euclidean space is of a continuous nature and can only be represented on a computer system within certain margins for error. This section examines data values represented by parametrised data sets where instead of storing discretely representable value, a membership condition for the set is used to determine the value by sampling. The modelling of geometry of a continuous nature in a definitive script relies on good data structures

[7]Pi-Hwa Sun is currently working on introducing control in EDEN for this style of grouping of agents. His method is known as *virtual agency*.

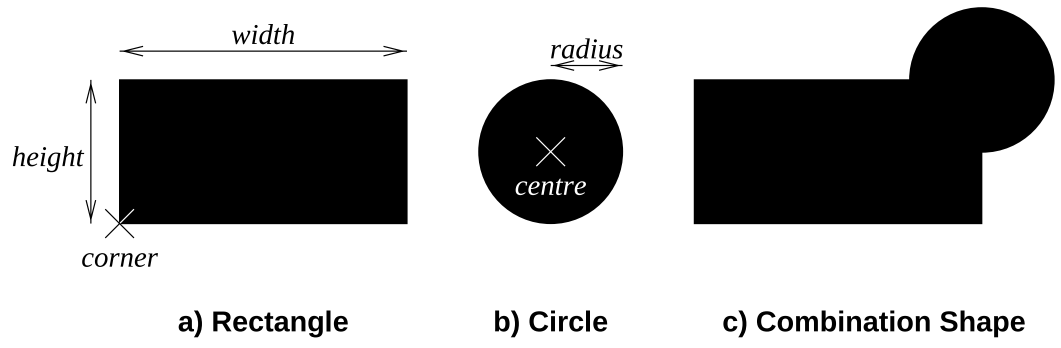


Figure 3.8: Parametrised shapes and their combination.

and mechanisms for the representation of parametrised data sets.

The solid two-dimensional geometric entities shown in Figure 3.8 are examples of sets of data of a continuous nature. Figure 3.8a is parametrised by a *height* value, *width* value and a corner point. The circle in Figure 3.8b is parametrised by a *centre* point and a *radius*. For both shapes, it is possible to implement algorithms that will draw graphical representations of the shapes on a computer screen and other procedural function code to establish dependencies between the parameters and the shape by definition.

The shape in Figure 3.8 is a combination of a solid rectangle shape and solid circle shape. One parameter-level method for representing the combination of shapes in definitive notations involves:

- describing parametrisations for combined shapes made from one rectangle and one circle shape and all combinations of shape primitives;
- implementing specific algorithms to render each of the combined shapes;
- establishing dependencies between the parameters for combined shapes and the parameters of their defining primitives.

This method allows for the construction of preconceived parametrised shape primitives but not for the arbitrary combination of any shape. The combined shape of Figure 3.8c can also be regarded as representing a point set union. A desirable feature in definitive script notations for geometric modelling is the inclusion of operators for the combination of arbitrary shapes, where the dependency is established between non discrete point sets rather than discrete defining parameters[8].

The CADNORT tool, developed as part of my third year undergraduate project [Car94b], translates scripts of definitions for two and three dimensional solid geometric shapes into EDEN definitions. For every shape description in the CADNORT notation, there is a block of EDEN definitions representing the parameters for that shape and an associated procedural function that tests for membership of the point set for the geometry. A similar naming convention to that used by DoNaLD and shown in Table 3.2 is used to represent the parameters of shapes in the EDEN model rather than a list representation.

In CADNORT, the function representation of shape [PASS95] is used as the mathematical basis for determining point set membership for the shapes. For any point in n-dimensional Euclidean space, point set membership is determined by a function f that maps from any point to a real value. To test point membership of a shape S at any point $\mathbf{p}$, the solid geometric shape is represented by function f, where

$$f(\mathbf{p}) \begin{cases} < 0 & \mathbf{p} \text{ is outside } S. \\ = 0 & \mathbf{p} \text{ is on the surface of } S. \\ > 0 & \mathbf{p} \text{ is inside } S. \end{cases} \tag{3.1}$$

Figure 3.9 shows the CADNORT graphical output for a table with a lamp placed on its top. The figure shows both front and side views of the sampled geometry at particular planar slices through the shape. The output shows only the outline of the solid shape. The generation of graphical output in CADNORT is a slow process due to the evaluation of interpreted code that represents the function

[8]DoNaLD is a notation for line drawing and does not attempt to represent filled shapes. Shapes can be displayed as filled by using attributes for the rendering of the geometry.

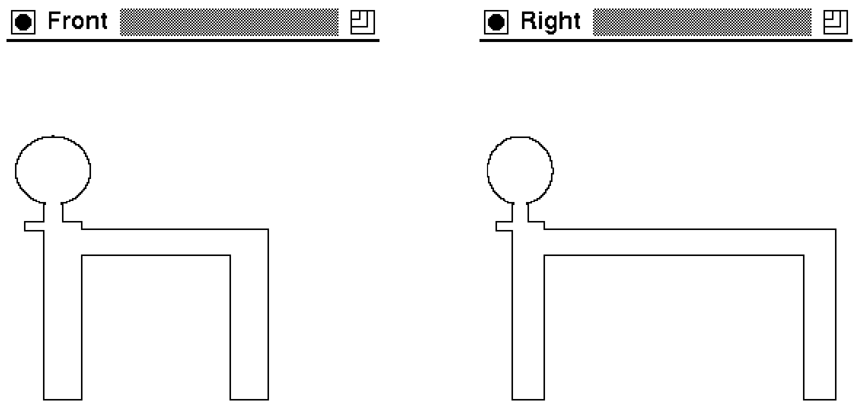


Figure 3.9: Graphical output from the CADNORT tool for a table and lamp script.

representations. The graphical output requires sampling of the function representation for the shape at every pixel. The dimensions of the graphics shown are 200 by 200 pixels and the generation of the figure was so slow in 1994 that plotted points could be observed to appear one by one.

CADNORT has some limitations. These include:

1. Managing the scripts of definitions in CADNORT and EDEN. The high-level definitive script for the table is approximately eight times shorter than the length of the EDEN script used to represent it.

2. Templates (called *generators*) representing families of geometry can be created on-the-fly. These templates cannot be redefined on-the-fly.

3. It is not possible to reference the components of shapes constructed using set-theoretic operations.

For all its limitations, CADNORT demonstrates the representation of point sets by definitions, with indivisible relationships established between point sets.

3.4.3 Geometric Data and its Associated Attributes

Large volumes of data are associated with representations of geometry on a computer system. Computer-based tools to assist a designer to model geometry typically represent geometric entities together with their defining parameters. There is a wide diversity of data that may also be associated with each model. This can be represented in additional data fields in the data representation of the geometry. The kinds of field that may need to be considered include:

Attributes Data that can describe non-geometric information associated with a geometric object. This can include the objects colour, texture, material, density, line width.

Location In addition to relative point locations within a geometric entity itself, the entity may be transformed into a different location before it is rendered. Geometric shapes are often placed in virtual computer-based worlds with other geometric shapes. The location of a shape in such as world may be different from the location where it was originally constructed.

Bounding Box Solid geometry, which has an associated point set that occupies a volume of space, may be boundable by a containing rectangle if defined in two dimensions, by a containing box in three dimensions, and so on. The bounding shape in any dimension is known as the *bounding box* and has sides that are parallel to the axis of the space. The bounding box is often required to assist algorithms for rendering geometry. Such a bounding box is particular useful for rendering shape represented by function representation.

Structure A geometric entity may itself be a child component of another entity or be the parent entity for other sub-entities. If this relationship is more than just a dependency given in a script, then it may be necessary to store information about the parent/child relationships with the data representation of a geometric entity. For example, there may be data concerning how the colours of entities appear to blend together when they are joined.

Reference A geometric entity contains information that may be useful for constructing the parameters of other entities. For example, a circle shape defined by centre and radius parameters may have an implicitly determined diameter parameter that can be used in the definition of other shapes that depend on the definition of the circle.

When creating a geometric model in a definitive script notation, it is desirable that a modeller should be able to modify not just data relating to defining shape, but also attributes, overall location, bounding box, structure and reference. The user should be able to make novel definitions between this data also, such as the colour of an object changing dependent on permissions to modify it. The challenge is to provide a data representation for geometry that not only handles defining parameters, point sets and the information in the above list, but is also able to handle the effect of the extended data representations when instantiating copies or forming high-order definitions.

Support for attribute data in existing definitive notations for geometry is poor. Attribute data in DoNaLD is not held as part of the geometric entity but instead relies on the underlying EDEN implementation of the DoNaLD interpreter to insert attribute information for use by the drawing actions of the interpreter. It is, for example, possible to link the colour of a line to its length but this has to be done at the EDEN interpreter level rather than within DoNaLD itself. Attributes cannot be applied to a group of definitions simultaneously and must be applied one by one. When making a copy of a group of definitions in an *openshape* in the form of a translation or a rotation, the attributes are not copied and it is not possible to assign different attributes to the components of the openshape.

3.5 Proposed Solution to the Technical Issues

The previous sections of this chapter have introduced some of the technical issues associated with representing data and manipulating data with scripts of definitions. Although these are general issues for empirical modelling, they pose particular problems when associated with geometry. The aim of the work presented in the chapters that follow is to develop a way of using definitive notations for geometry that overcomes these problems. An overview of the underlying approach adopted in this research follows.

The following list summarises some of the technical issues:

- The ambiguity in the status of copying definitions in a script occurs when copying an assembly of geometry rather than a single definition. Two copy procedures (photographic and mirror) exist for single explicit definitions and three classifications of copy procedures (photographic, mirror and reconstruction) exist when copying implicit definitions.
- Issues concerning the mode of definiton and the level at which dependency is defined in data structure arise when values are represented by data types other than atomic data types. Only one abstract mode exists for variables of atomic data types, which are either explicitly or implicitly defined at the same *scalar* level.
- Functions are required for definitions that represent indivisible relationships between:
  - defining parameters for geometric entities;
  - the point sets represented by geometric entities.

The above discussion motivates the introduction of a special "atomic" data type that can be used to represent all the explicit values associated with variables in a definitive script. The term *serialisation* will be used to refer to the process of transforming an explicit value of any data type into an encoded atomic value. If all values referenced in a definitive script notation (including all parameters, attributes and point inclusion algorithms and assemblies of definitions) can be represented by *serialised data types* in this way, many of the issues raised in this chapter are greatly simplified. Every value is either implicitly or explicitly defined and all explicit values are on the same data structure level. Higher-order dependency is a method for creating a lower-level script of assembly of definitions. If a value of a serialised data type can be used to represent all the information for an assembly, reference can be made to what previously would have been the subcomponent definitions of an assembly through the use of a special operator. These special operators map from a value of a serialised data type to a representation of a component its value. A low-level script is not required. For example, consider the *arcline* higher-order definition shown in Section 3.2.2. Table 3.3 shows atomic data types and operators that can be used both to create an arcline shape and make reference to its internal components. The script of definitions in Table 3.3 illustrates the construction of an arcline and reference to its component arc shape (*a1*).

Single values of serialised data types will need to be interpreted as representing a wide diversity of data, including sets. The operators that create indivisible relationships between these values will need to be able to extract and relate many different aspects of the data represented. As every value incorporates all information required for a particular type of data, interpretation of this data can include methods for determining set membership. These methods can be related to one another through special operators used in definitions in the same way as defining parameters.

Atomic Data Types	*scalar*	Floating point values.
	point	Points in two-dimensional space.
	arc	A curved section of a circle.
	arcline	A shape formed from an arc and two lines, as shown in Figure 3.2.
Operators	*makeArcline*	Define an *arcline* dependent on a scalar and a point.
	arcFromArcline	Define an *arc* to be equal to the arc of the *arcline* that is an argument to the operator.
Example Script		$r = 2$ $c = \{0,0\}$ $al = makeArcline(r, c)$ $a1 = arcFromArcline(al)$

Table 3.3: Replacing higher-order dependency with atomic data types.

In this way, all data structure, expression trees and set representation for a model can be represented by a script of definitions. The atomic types of the notations are similar to objects in object-oriented programming (OOP) [CAB$^+$94]. Every object represents a *thing* through its data fields and its relationships to other *things* through its methods. The difference from OOP is that the values are defined in interactive, open-ended scripts and communication between values occurs through indivisible relationships established by definition. In OOP, the communication between objects is given by the procedural ordering of the methods and is prescribed at the time that the code is compiled.

In the next chapter, the potential for implementing definitive notations to handle serialised data types and operations is assessed, by examining dependency in scripts that have only one data type. This concentrates on issues that are related only to efficient dependency maintenance. This model is used as the basis of an implementation of dependency maintenance that only implements one atomic data type and requires a programmer to make decisions on how to organise the data of that type in a useful way (the DAM Machine in Chapter 5). The model is also the basis for an object-oriented implementation for dependency maintenance that supports the programmer in the implementation of serialised data types and special operators (the JaM Machine API in Chapter 6). For both implementation methodologies, the case-studies presented are based on geometric modelling examples.

In Chapter 9, the technical issues described in this chapter are reviewed with resepect to the special atomic data type approach to dependency maintenance.