Lecture M2: Introduction to definitive notations

M2.1. Background and History

A *definitive notation* is a simple formal language in which definitive scripts can be formulated. Definitive notations are distinguished by the types of the variables that appear on the LHS of definitions and the operators that can be used in formulae on the RHS. The set of types and operators associated with a definitive notation define its *underlying algebra*.

The term "definitive notation" was first introduced in [], but the essential concept was used earlier. For instance, Codd's relational algebra query language ISBL [] is a definitive notation over an underlying algebra of relations. The first use of definitive notations in interactive graphics was by Brian Wyvill [], and the principles are still being exploited in his recent research on animation. The formulae used to define the cells of a spreadsheet can also be viewed as based upon a definitive notation in which the underlying algebra is traditional arithmetic. More sophisticated spreadsheet software also extends the range of data types. Definitive principles are becoming increasingly common in everyday software applications e.g. in the style definition in word processors such as Word.

It will become apparent that our use of the term "definitive" means more than informal use of a particular programming technique. Definitive notations are a means to the representation of state by definitive scripts, and the way in which these scripts are interpreted is highly significant. For instance, a functional programming language such as Miranda *can* be viewed as a definitive notation over an underlying algebra of sophisticated functions and constructors, but this interpretation puts the emphasis on *program design* as a state-based activity, rather than on declarative techniques for *program specification*.

M2.2. DoNaLD: a definitive notation for line-drawing

The Donald notation is a definitive notation for 2-d line-drawing. Its underlying algebra has 6 primary data types: **int**eger, **real**, **bool**ean, **point**, **line**, and **shape**. A shape is a set of points and lines. A point is represented by a pair of scalar values {x,y}. Points can be treated as position vectors, in that they can be added and multiplied by a scalar factor. A line is a line segment that joins two points. The operators currently implemented in DoNaLD include:

- arithmetic operators

  +   *   div   float() if ... then ... else ...

- basic geometric operators

```
.1          .2    .x    .y    {,}    [,]    +    *
dist()  intersects()   intersect()
```

- translate()   rot()   scale()

- label() circle() ellipse()

The donald interpreter is typically invoked by typing

```
        donald | eden -n
```

Demonstration files for donald can be run by invoking the demo.donald command in ~wmb/public/demo. The donald source files for the demos are to be found in ~wmb/public/demo/DONALD.

Variables have to be declared before they can be defined.

```
#       this is a donald comment
#       the following script defines m to be the midpoint of the line l
#       joining the points p and q, and om as the line from the origin to m
point o, p, q, m
line l
l = [p,q]
m = (p+q) div 2
line om
#       new declarations can be introduced at any stage
o = {0,0}
om = [o,m]
.....
```

Default screen coordinates for a DoNaLD window are {0,0} bottom-left to {1000,1000} top-right. When donald is used in conjunction with scout, the extent of the donald display can be specified explicitly.

There are two kinds of shape variable in DoNaLD. An openshape variable S is defined componentwise, as a collection of points, lines and subshapes. These components of the openshape are given explicit identifiers typically declared " within the context of the openshape S ", thus:

```
openshape S
within S {
        # the donald prompt changes to reflect the new context
        int m
        # this is equivalent to declaring int S/m outside S
        point p, q
        openshape T
        p = {m, 2*m}
        within T {
                point p, q
                # this point has the identifier S/T/p
                p , q = ~/q, ~/p
                # a multiple definition: p = ~/q and q=~/p
                # ~/... refers to the enclosing context for T
                # viz S, so that ~/p refers to the variable S/p
                .....
                # a syntax error in here will cause an escape
                # from the "within S { ... within T {..." context
                # as if " <donald_error> } } " had been entered
        }
        ...
    }
```

The other kind of shape variable in donald is declared via

```
    shape RSQ
```

and defined by a formula that returns a value of type shape, as in

```
    shape RSQ
    RSQ = rotate(SQ)
```

This mode of shape definition occurs in defining the vehicle in cruise.s.

The donald translator generates eden output. This can be inspected by teeing the output of donald into a file:

```
    donald |tee d.output | eden -n
```

Certain features of donald, as proposed in [], such as line attributes, are not fully supported, and can only be addressed by introducing definitions in eden. To do this, it is necessary to type

```
    %eden
```

- this sends input to eden directly bypassing the donald translator. With knowledge of the conventions used to represent donald definitions in eden it is then possible to embellish the donald script by introducing additional eden definitions into the translation. The dashed line that represents the cable in the donald room demo is a simple illustrative example. The donald-eden interface will be discussed in more detail in connection with eden [ref M3].

Using donald | eden creates a default donald screen. The screen itself doesn't appear until the first declaration or definition is introduced. In contrast, when donald is used in conjunction with scout, the creation of donald windows is handled by scout [ref T2].

M2.3. Significant Features and Uses of Definitive Notations

M2.3.1. Reference and Moding

Definitive scripts create references that are different from traditional procedural variables (meaningful only during the execution of a program) and declarative variables (statically defined, like mathematical variables, independent of program execution). A definition creates a

reference = *value*

– one objective of definitive modelling is to support a use of references does more justice to real-world use. For instance, a geometer can conceive the idea of a double point on a self-intersecting curve, but this is difficult to describe satisfactorily using formal mathematical variables. In a definitive script, we may well have several distinct variables with coincident values:

```
a = b
b = 3
c = 2*a-b
....
```

but each with a different identity and significance.

The way in which reference and value are associated is a related issue. This is termed the *mode of definition*. When we define a complex structure, such as a list, we may either have in mind a recipe to define the entire list, such as

```
list1 = reverse(list2)
```

a list of recipes to define the components of the list, such as

```
list3 = [l1,l2,l3]; l2=2*l3; l1=2
```

or a recipe that combines the two modes of definition, as in

```
list4 = [list1, list3, [l5,list5]]; l5 = 7; list5 = [l5].
```

A related concern is whether the components of a list can be treated as independent variables (cf l-values), as in the sequence of definitions:

```
list4[1] = list1
list4[2] = list3
list4[3] = [l5,list5]
```

that could serve as an alternative way of defining list4 above. When there are many different modes of definition, there is potential inconsistency. For instance: the sequence of definitions

```
list1 = reverse(list2); list1[1]=3;
```

is unsatisfactory, as it involves two independent definitions of list1[1]. (The problems this dual level of definition creates can be experienced when making local amendments to globally defined line styles in Word.)

Different modes of definition are reflected in shape and openshapes in DoNaLD. The concept is even further developed in ARCA [], where the mode of definition of a variable is itself specified using a definitive notation over an underlying algebra of modes. Such use of moding underlines the additional power to express matters concerning reference that definitive notations provide.

M2.3.2. Agents and semantics

The archetypal use of definitive notations is for human-computer interaction. The variables in a definitive script represent the values that the user can observe, the parameters that the user can manipulate and the way in which these are linked indivisibly in change. In this way, a definitive script can model physical experiments, such as the relationship between load and extension in Hookes' Law, or that between potential difference, resistance and current in Ohm's Law. The role of spreadsheets in describing and predicting the consequences of commercial decisions is similar.

A script supplies an environment rather than a document. In a document, the meaning of a symbol has to be represented in a stateless fashion: it is for the reader to bring the symbol to life by exercising imagination about the meaning of the symbol from each context in which it occurs. In a definitive script, there is scope to explore the significance of symbols through experiment and observation.

Definitive methods for concurrent systems modelling apply the principles that spreadsheet-like software bring to the user-computer interface to modelling the relationship between all interacting agents. The interface between an agent and the rest of the system is treated as a domain for experiment, in much the same way as an engineer might test the characteristics of a component in isolation.

M2.3.3. Objects vs observations

A definitive script that describes a geometric symbol can be viewed as representing atomic transformations that can be applied to the symbol. The DoNaLD room can be transformed through redefinition in ways that correspond exactly to the observed patterns of change associated with opening a door, or moving a table. Our underlying thesis is that the set of atomic transformations that can be applied to a symbol capture its semantics. (Compare Klein's view of a geometry as defined by the set of properties of geometric objects invariant under a specified set of transformations.) The resemblance between the digit eight and the floor-plan of a filing cabinet is a geometric pun that illustrates the essential principle (see the Appendix).

It is tempting to regard the DoNaLD room as an object in the OOP sense. This is to view each room transformation as a method for the object. The interpretation of a definitive script as an object specification is only acceptable when the set of transformations that can be performed on the room has been circumscribed. There is an important distinction between this process of circumscription, which creates objects, and the process of definitive modelling that merely records a more primitive level of knowledge about observed transformations. Comprehending an object involves knowing everything we can do with it, but a definitive script in itself does not circumscribe the transformations we can apply. The distinction being made here is between *an object* and *an observation*, the latter being the more primitive concept, presuming fewer preconceptions about what might be observed. (Whether we can say *no* preconceptions is a talking point here!)

One important consideration that distinguishes the definitive script from an object is the way in which it lends itself to the expression of different agent views and privileges to transform. What the architect can do to the room layout (e.g. relocate the door) differs from what the room user can do (e.g. open/shut the door). This emphasises the neutrality of the script, and the way in which it acquires a different significance according to what perspective we adopt on possible transformations.

M2.3.4. Variable values, observations and state

The variables considered in this course are not like program variables, nor like mathematical variables. The variables we are interested in should correspond to observations of real-world objects and processes external to the computer system. Like these external observations, they are defined by having an identity and a value that changes according to the circumstances of observation.

The term *state* will be used here to refer to sets of observations made at the same time. So it is that today is Friday, I am at home, the sky is cloudy, I have just had lunch etc etc. These observations belong together because they are the result of what I deem to be simultaneous observations of the state of the world. The concept of state – like that of admissible transformation – is relative to the observer ("observing agent"), and even with respect to a single observer is subject to vary according to focus of attention and mode of observation. A definitive script is neutral in this respect also, in that different sets of variables can be extracted from it and viewed as defining as state.

A broad and ambitious objective that is useful to have in mind when studying definitive notations is that of finding better representations for a whole range of abstractions that have been considered in programming language design and development. For instance: an object can be viewed in terms of observations and transformations that can be modelled by agent protocols. In a similar spirit, a data type is complex when "there are several ways in which the object can be observed". For instance, we can think about the value of a list as an entity, or about the values of constituent elements. We would like to express such subtleties in terms of "states within states" obtained by extracting subsets of variables from a script. A Miranda script already illustrates similar characteristics in respect of functions: cf the three line script

```
 add x y   = x+y
   sq x    = x*x
quad x y   =  add (sq x) (sq y)
```

with the same script with the definition

```
  answer   = quad  10  23
```

appended. The first script is the definition of the abstract function quad(); the second defines quad() in conjunction with a particular evaluation. Such modes of observation of objects are commonplace in mathematics (let p be a point on the parabola ...) and engineering (when the speedometer is registering 40 mph ...).

References

WMB	Definitive Notations for Interaction
WMB et al	DoNaLD specification
WMB et al	Scientific Visualisation etc etc
Codd	ISBL: The Peterlee Relational Test Vehicle
Wyvill	
Chmilar, Wyvill	
Miranda	Manual

Follow up: T2 - SDAE - for more technical details
Programming as Modelling / Foundations
for more discussion of principles

Details

Can we make donald interpreter recognise int and integer?
Same for scout?

scout allows int m = ...;

Can donald be made to do the same?