Lecture M2: Introduction to definitive notations

M2.1. Background and History

A *definitive notation*

= a simple formal language in which to express definitions

A set of definitions is called a *definitive script*

Definitive notations different according to

types of the variables that appear on the LHS of definitions

operators that can be used in formulae on the RHS.

These are termed the *underlying algebra*. for the notation.

Use of definitive notation concept

- Codd relational algebra query language ISBL
- Brian Wyvill's interactive graphics language
- spreadsheets
- style definition in word processors

The term "definitive notation" was first introduced by Beynon.

What does *definitive* mean?

definition has a technical meaning in this module

definitive means "definition-based"

"definitive" means

more than informal use of a particular programming technique.

Definitive notations are

a means to *represent state* by definitive scripts

and *how* scripts are interpreted is highly significant.

For instance

Miranda *can* be viewed as a definitive notation

over an underlying algebra of functions and constructors

but this interpretation emphasises

program design as a state-based activity

rather than

declarative techniques for *program specification*.

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

Donald = a definitive notation for 2-d line-drawing

underlying algebra has 6 primary data types:
integer, **real**, **bool**ean, **point**, **line**, and **shape**

A **shape** = 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:
they can be added: p+q
and multiplied by a scalar factor: p*k

A line [p,q] is a line segment that joins two points p and q

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
```

To supplement the definitions in the donald "file1", use

```
cat  file1 - | donald | eden -n
```

For donald demonstration files, use:
"demo.donald" command in ~wmb/public/demo.

Source files for the demos are in ~wmb/public/demo/DONALD.

Elementary DoNaLD use: writing a script

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 points p and q, and om as the line from 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:

{0,0} bottom-left to {1000,1000} top-right.

[Can specify extent of donald display explicitly using SCOUT]

Defining shapes in DoNaLD

Two kinds of shape variable in DoNaLD:

these are declared as **shape** and **openshape**

An openshape variable S is defined componentwise

as a collection of points, lines and subshapes

given explicit identifiers typically declared

" within the context of the openshape S ", thus:

```
openshape S
within S {
        # donald prompt changes from D> to D:S>
        # 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 {
                # donald prompt changes from D:S> to D:S/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
        }
        ...
}
```

Other mode of definition of shape in DoNaLD is

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

– illustrated in definition of vehicle in cruise.s.

DoNaLD and EDEN

The donald translator is called **donald**

donald generates eden output that is then piped to eden.
This can be inspected by teeing the output of donald into a file:

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

Some features of donald are only accessible via eden

For example:

the dashed line for the cable in the donald room demo
is defined by changing its attributes (A_cable) in eden.

When running the pipeline donald | eden, you can type

```
%eden
```

to send input to eden directly, **bypassing** the donald translator.

To embellish the donald script by adding eden definitions, you need to know how donald definitions are represented in eden.
[The donald-eden interface will be discussed later [ref M3]]

Using donald | eden creates a default donald screen.
NB the screen doesn't appear
until the first declaration or definition is introduced.

[When donald is used with scout, scout handles donald windows]

M2.3. Significant Features / Uses of Definitive Notations

M2.3.1. Reference and Moding

A definition creates a relationship between reference and value

reference = *value*

definitive variable differs from

- traditional procedural variable

  meaningful only during the execution of a program

- declarative variable

  statically defined, independent of program execution.

Definitive variables aim to support references as in real-world use.

Reference = a concept of identity

For instance, a double point on a self-intersecting curve is difficult to describe satisfactorily using formal mathematical variables.

A definitive script may have distinct variables with same values:

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

each with a different "identity" and significance.

Modes of definition

mode of definition

= the way in which reference and value are associated

form of a definitive notation

not determined by the underlying algebra alone

possible modes of definition also important

..... there are many ways to define a complex structure

e.g

- define a list in its entirety:

  list1 = reverse(list2)

- give a list of recipes to define the components of the list:

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

- use a recipe that combines the two modes of definition:

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

Issues for mode of definition

Related issue:

can list components be treated as independent variables?: cf

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

many different modes of definition => potential inconsistency.

For instance:

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

involves two independent definitions of list1[1]

Different modes of definition represented

- in shape and openshapes in DoNaLD
- in EDEN, components of a list can't be defined via l[1] = ...
- in ARCA the mode of definition of a variable is itself specified in a definitive notation over an underlying algebra of modes.

M2.3.2. Agents and semantics

Archetypal use of definitive notation: human-computer interaction

Variables in a definitive script represent

- the values that the user can observe
- the parameters that the user can manipulate
- the way in which these are linked indivisibly in change

=> definitive script can model physical experiments

cf the role of spreadsheets in describing and predicting

A script supplies an environment rather than a document.

In a document:
meaning of a symbol has to be represented in a stateless fashion
the **reader** animates it by studying the contexts in which it occurs

In a definitive script:
explore significance of symbols via experiment and observation.

Definitive methods for concurrent systems modelling
= generalising definitive principles for the user-computer interface
to modelling the relationship between all interacting agents

Each agent-system interface is treated as a domain for experiment

M2.3.3. Objects vs observations

A definitive script

represents the atomic transformations of a geometric symbol

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.

Thesis:

set of atomic transformations of a symbol captures its semantics

cf Klein's view of a geometry

The digit eight vs the floor-plan of a filing cabinet: a geometric pun

Is the DoNaLD room an object in the OOP sense?

Can view each room transformation as a method for the object.

BUT definitive script is an object specification only if
set_of_transformations_performed_on_room **circumscribed**

Circumscription creates objects

BUT
definitive modelling merely records observed transformations

Comprehending an object = knowing everything we can do with it

BUT
definitive script doesn't circumscribe transformations we can apply

The distinction is between *an object* and *an observation*

Observation is the more primitive concept:
needs fewer preconceptions about what might be observed

"Definitive scripts neutral wrt agent's views & privileges"

definitive script differs from an object:

can express different agent views and privileges to transform

What architect can do to the room layout (e.g. relocate the door)

vs what the room user can do (e.g. open/shut the door).

=> significance of script relative to view of possible transformations

M2.3.4. Variable values, observations and state

Definitive variables

- correspond to observations of real-world objects and processes external to the computer system
- are defined by having an identity and a value that changes according to the circumstances of observation.

The term *state* refers to what we understand by

sets of observations made at the same time

the current state =

what I deem to be simultaneous observations of the world

The concept of state is

- relative to the observer ("observing agent")
- relative to focus of attention and mode of observation

A definitive script can represent many different states at once

Broad objective:

use definitive scripts as primitive device to represent a whole range of abstractions in PL design and development.

Examples:

- an object can be viewed in terms of observations and transformations that can be modelled by agent protocols

- a data type is complex when "there are several ways in which the object can be observed"

  For instance, we regard the value of a list as an entity, or think about the values of constituent elements.

  Aim to express via "states within states" by extracting subsets of variables from a script cf Miranda script + evaluated function

Problems of use of reference arise in mathematics also:

exposition of proof has never been formalised

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?