CS405 Introduction to Empirical Modelling - 2010-11

Reflecting on Lab 1: CADENCE in context ...

Recall the observation that:

> **Empirical Modelling ("EM")** is a body of principles and tools concerned with computing activity that is based on observation and experiment (hence 'empirical') ...
>
> ... the principal theme of CS405 is alternative ways to think about "computing-in-the-wild" - with special attention to computer **programming** and computer **science**

Will explore this further by looking closely at the concept that has probably been most influential in framing the science of computing - the notion of "a computer program". One particular aim is to show why "programming" / modelling in the CADENCE environment (as introduced in Lab 1) is radically different from traditional programming. This is a first step towards tackling the agenda suggested by Lab 1.

What is a program?

"The classical answer": *A program is a recipe for action that computes an input-output relationship*

- Turing's abstract view of a computation
- checking for palindromes as a simple illustrative example ...

A **procedural** program explicitly expresses a recipe as a sequence of actions ...

- each action assigns a value to a variable, or invokes another procedure
- transfer of control ("sequence of execution and invocation") determined by logic conditions. These are framed using constructs like **`while`** and **`if ... then ... else ...`**

A problem with procedural programs

- variable has a value that can be elusive e.g. when debugging
- always changing, possibly in ways that can't easily be tracked

Declarative programming approaches offer a potential solution

- Program (e.g.) by specifying the required input-output relation in a mathematical form:
  ```
  out = f(in)
  ```
  This is called **functional programming ("FP")**.
- FP exploits a special-purpose interpreter that can compute the function `f`
- FP uses very powerful operators ("λ - calculus") in order to frame the function `f`

[Aside: Another form of declarative programming is **logic programming**, in which the I/O relation is specified by a predicates framed using special logical constructions and interpretation is automated deduction.]

Key "virtues" of declarative programming:

- it hides internal states of the computation
- have **referential transparency**
- frame computational problems in terms of the external domain, not the computer

BUT

Declarative programming doesn't make **interaction** easy ... ... have to introduce mechanisms to support interaction

- One technique that can be introduced in FP is 'lazy evaluation', which gives opportunities for user intervention
- A data-flow approach gives more flexibility, by making the internal steps in the computation of f more explicit

Another aspect of declarative programming that is challenging is supporting the input and output

---

The legacy of the TM concept of computation has been *a highly abstract conception of programming*

This doesn't chime well with "emerging computing" ...

Some issues

- non-standard peripheral devices and modes of interaction
- non-terminating programs maintaining relationships over time
- the transition from "one-person programming" to "programming of reactive systems" [Harel]
- the relevance of real-time, distributed computing and concurrency concerns
- new processes for developing software e.g. with team work and user participation
- to accommodate the continuously evolving nature of software products [cf. Manny Lehman])

To understand software development in these broader domains, need

- to think more broadly about computing with many interacting objects
- to consider design and interaction from many perspectives
- to construct software in such a way that it is comprehensible and manipulable potentially even by the non-specialist

Some techniques that can contribute to addressing these goals:

- object-orientation
- agent-based analysis and conception of systems
- design patterns
- spreadsheet principles

The CADENCE environment offers scope for interpretation reflecting the influence of all of these programming idioms and features Some prototype-based object-oriented code:

```
this sgobjects puddle

        primitive = cube
        width = 3.3
        height = 3.3
        depth = 0.1
        visible = true

        position = (new x=0.0 y=0.0 z=-3.0
                z is { @stargate position z }
        )

        orientation = (new x=0.0 y=2.5 z=0.0
                y is { @stargate orientation y }
        )
;
```

Some data-flow in the gate.dasm script:

```
#How fast the hole appears
holespeed = 0.8

#The hole animation definition
hole = 1.0
hole := {
        if (.ready) {
                if (..active) -1.0 else {
                        ..hole - (..holespeed * (@root itime))
                }
        } else 1.0
}

active = false
active is { .hole < -0.9999 }
```

A "real-time" ingredient:

```
match = false
ready = false
locked = false
rotspeed = -0.3
rotation = 0.0
rotation := {
        if (.dial and (.match not or (.locked)) and (.ready not)) {
                ..rotation + (..rotspeed * (@root itime))
        } else {
                ..rotation
        }
}
```

Examples of spreadsheet-style dependencies

```
position = (new x=0.0 y=0.0 z=-3.0
        z is { @stargate position z }
)

orientation = (new x=0.0 y=2.5 z=0.0
        y is { @stargate orientation y }
)
```

Going beyond classical programming ...

As mentioned in the introduction ...

> *Characteristics of tools to be introduced in the module ... they are concerned with modelling in which we*
>
> - *observe meaningful things*
> - *adopt a **constructivist** stance*
> - *exploit an **empirical** approach*
>
> *that we wish to reconcile / can be reconciled with the more abstract, rationalist, theoretical framework that characterises classical computer science*

The key move in reconceptualising programming is **introducing the human dimension** ...

... key shift in emphasis towards questions of the following kind:

> ? what is the *experience* of the people engaging with Turing computation, procedural programs, functional programs etc.

Consider people's experience (whether they are 'programmers', 'users', 'modellers' or 'analysts' etc.) with reference to

- What are the significant things that they **observe**?
- How are they able to **interact** and **manipulate**?
- What is the **context** for their interaction and interpretation?

when they are engaged in some variety of programming / model-building activity.

For the present, we describe these informally in natural language ...

[**Aside:** part of the point of EM is to develop ways in which we can convey these characteristics in a more effective way - though (we claim!) we can never expect to express them *formally* in the spirit of traditional theoretical computer science. Compare the discussion in Peter Naur's paper *Intuition in software development*, and the commentary in EM paper #105.]

Experience in relation to programs for a Turing machine

- The **context** for observation is some desired transformation of state ...
    - Programmer and user are concerned with how a symbolic rule-based process achieves a specific family of transformations of state, and have to understand and trust the reliable machine that can execute the rules. Only the programmer needs to understand the rule-based process intimately, but both need to have trust in the machine
    - Both programmer and user are concerned with the fact that and how abstract symbolic representations correspond to external referents (e.g. "how is a number represented on the tape?" "what indicates that the computation has terminated?" "what is on the tape on termination when the number is / isn't prime?)
- What do the human agents engaged in Turing computation **observe**?
    - The programmer also has to pay attention to highly abstract rules + processes they define
    - Both programmer and user have to appreciate the potentially subtle and complex interpretation of the I/O representation
- How do the human agents engaged in Turing computation **interact** and **manipulate** state?
    - In developing a TM program, there is characteristic "geeky" code generation for the programmer - framing rules
    - The user cannot influence the rules - can only specify the initial state of the tape
    - Neither the programmer nor the user can intervene in the automatic execution of the rules, unless the development environment is such that new rules can be introduced even when the Turing machine is executing

It's possible to ask the same question of programming in different paradigms: e.g.

- ? what's the experience of the functional programmer
- ? what's the experience of a dataflow programmer
- ? what's the experience of spreadsheet user

Doing this in detail is beyond the scope of this lecture (!), but it's good for you to reflect on these questions yourself.

What is readily clear is that human experience of programming can be very diverse, according to which programming paradigm is used. For instance:

- The overarching concern in programming a Turing machine, and in traditional procedural programming, is with how a process achieves a specific family of transformations of state
- Classical procedural programming is very similar, but the machine execution and interpretation of the rules in the program is more complex and prone to misconception / error
- In FP, we are (ideally) not concerned with observing operations and recipes on the computer at all.
- In spreadsheets, we don't typically have an I/O relationship of the classical programming variety in mind, and observation of situations external to the computer (e.g. the marks that students achieved in exams etc) have a much more direct role in observation and interaction.

What is quite apparent is ...

There is a major problem of incoherence in interpretation if these paradigms are used in conjunction ...

How does CADENCE aspire to address this?

There is a human experiential activity of creating a computer model of something ...

... this model develops incrementally through representing key observables and fashioning the changes they undergo

... the changes may relate to design decisions (e.g. specifying the dimensions of the Stargate) or to characteristics that are subject to change continuously (e.g. the orientation of the Stargate when it's rotating)

... imagine the whole activity of building up the Stargate from its components, fixing values by design, testing the behaviour of the prototype, formulating relationships between its components that are useful in making the design ...

... can regard this as a blend of programming and using that takes place in one continuous stream of experience of the evolving, emerging Stargate model

... the computer representation used is so generic, primitive and anodyne ('computation is navigation') that it is enormously flexible and can carry all these interpretations throughout the history of the model-building, from initial conception to final form

... the computer is a reliable finely tuned instrument that delivers rich visual experience in an efficient manner through a continuous process of updating state from one moment to the next according to rules that can themselves be adapted on-the-fly moment-by-moment

Some exceptional qualities of DOSTE:

- modelling/tracking continuous / "analogue" quantities
- managing change of context, reference and content
- recording histories of a modelling trajectory

Useful analogy with creating artefacts and interactive experiences using a potter's wheel

Has to do with a certain kind of **construal** of experience, that arguably can be more or less appropriate in any particular context. Issues (limitations?) possibly relate to

- the correspondence between the model and its referent?
- where the referent is itself in the process of being identified
- where the context for the human modelling activity is unstable

- where the model-building is not taking place on a conventional reliable computer

The theme of this lecture is 'from programming to construal'. In much of the module, we'll be concerned with going in the opposite direction 'from construal to programming'. This is where other aspects of the modelling tool become relevant.

References

Peter Naur, Intuition in Software Development, TAPSOFT, volume 2, 60-79, 1985
Meurig Beynon, Russell Boyatt, Zhan En Chan, Intuition in Software Development Revisited, Proceedings of 20th Annual PPIG conference, September 2008. [EM paper #105 as indexed in the publication list on the EM website].
David Harel, Biting the Silver Bullet: as referenced in Lecture 13, CS405-0708
Manny Lehman, see e.g. http://www.doc.ic.ac.uk/~mml/feast2/papers/pdf/556.pd