Suggestions for Graduated Exposure to Programming Concepts Using Fading Worked Examples

Simon Gray
College of Wooster

Caroline St. Clair
North Central College

Richard James
Rollins College

Jerry Mead
Bucknell University

ITiCSE'06 Working Group

- Jerry Mead, Bucknell University
- Simon Gray, College of Wooster
- John Hamer, University of Auckland
- Dick James, Rollins College
- Juha Sorva, Helsinki University of Technology
- Caroline St. Clair, North Central College
- Lynda Thomas, University of Wales

Plan

- Motivation
- Conjecture
- Cognitive Load Theory
- Fading Worked Examples
- Samples
- Moving Forward
- Q & A

Motivation

- The problems students have mastering programming skills is well-documented - basis of ITiCSE'07 working group
- What is the problem? What can we do about it?
- Cognitive load requires attention

Conjecture

- Part of the reason students don't acquire the desired programming skills is that cognitive overload results in the development of near-transfer
- We'll get a better result if we do a better job of controlling for cognitive load
- Fading worked examples may be a way to control for cognitive load

Cognitive Load Theory

- John Sweller 1988

  → defined a model of memory that could be used to understand how the load on memory resources during problem-solving impacts learning

Cognitive Architecture

- Model of memory
  - Sensory memory
  - Working (short-term) memory – *limited!*
  - Long-term memory
- Forms of data
  - Raw sensory data
  - Encoded data in short-term memory
  - Schema in long-term memory

Schema

- Constructed in working memory by integrating new stuff with old stuff
- Can be come quite abstract, representing a more widely applicable piece of knowledge
- Takes up only a single slot in working memory

Cognitive Load

- *Intrinsic cognitive load* – reflects inherent difficulty of the material
- *Germane cognitive load* – amount of working memory resources required by other data needed for schema formation
- *Extraneous Cognitive Load* – the amount of working memory resources needed for instruction

So…

- Working memory is a gateway and bottleneck to learning

- Given this understanding of cognitive architecture and cognitive load, how can we design instruction to improve/enhance learning?

“Traditional” Approach

- Solving problems with specific goals from scratch is not effective for *novices*
- “means ends analysis” – at each step the student must be aware of the current state of the solution and the state of the problem goal
- Sweller: this approach leads to solutions, but not learning

Some CLT-Based Alternatives

- *The goal-free effect* – the student works a problem without specific goals
- *Worked example effect* – working from a solved example, the student solves an unworked problem
- *Partially worked example effect* – the student completes partially worked solutions

Fading Worked Examples

- Start with a completely worked example to serve as a model and a *process* for creating the solution
- Follow this with a sequence of problems each of which contains one fewer worked step than its predecessor
- Conclude with a solve-from-scratch problem
- Caution: expertise reversal effect

Forward versus backward fading

Lorem ipsum dolor sit amet.
Vivamus ac nibh a gravida.
Phasellus et leo eleifend.
Praesent eros in urna.

Lorem ipsum dolor sit amet.
Vivamus ac nibh a gravida.
Phasellus et leo eleifend.
____

Lorem ipsum dolor sit amet.
Vivamus ac nibh a gravida.
____
____

Lorem ipsum dolor sit amet.
____
____
____

____
____
____
____

Facets of Programming

- Programming is multi-faceted
- Facets that guide the construction of FWEs
  - Design
  - Implementation
  - Semantics
    - Asserts
    - Execution
    - Verification

Creating FWE Sequences

- For each facet, identify a sequence of steps a student will follow
- Identify a sequence of problems to fade
- Create the completely worked solution for one problem
- Provide faded solutions for the others
- Leave at least one problem completely unworked

Selection : design facet

Design produces a matrix:

rows correspond to the cases in the selection

columns correspond to the conditions and actions associated with the cases.

Step 1: From the problem description, identify the cases of the selection and determine if there is a default case.

Step 2: For each case, determine an appropriate condition; enter it in the appropriate matrix row.

Step 3: For each case, determine the action to be taken when the case is selected. If there is a default, identify the associated action. Enter the actions into the matrix in the corresponding rows.

Problem Statement

A grade school categorizes its students as being “pre-school” for ages between 4 and 5 (inclusive), and “grade school” for ages between 6 and 11 (inclusive). A program is to input a student’s name and age and print a report with the student’s name and category. Design an appropriate selection statement that when executed will print the required output. If the student’s age does not fit into one of the categories, then an error message will be displayed.

Step 1: Identify Cases

case	condition	action
pre-school		
grade school		
default		

Step 2: Identify conditions

case	**condition**	**action**
pre-school	4 <= age < 6	
grade school	6 <= age < 12	
default	age < 4 OR age >= 12	

Step 3: Identify actions

case	condition	action
pre-school	4 <= age < 6	print name is in ‘pre-school’
grade school	6 <= age < 12	print name is in ‘grade school’
default	age < 4 OR age >= 12	print Error: bad age

Subsequent Problems…

… would be different

- The first would have the last step missing
- The second would have the last *two* steps missing
- The third would have all the steps missing, but possibly with the empty steps still labeled
- Final problem is completely empty

Selection : implementation facet

Step 1: If the first case has a condition $cond_1$ and action $action_1$ write

if ($cond_1$) $action_1$

Step *n*: For each subsequent case add to the end of the growing statement the following.

else if ($cond_n$) $action_n$

Step last: If there is a default with action $action_d$, add the following at the end of the statement.

else $action_d$

Repeat the problem statement (not shown here)

Step 1: Review the selection matrix.

case	condition	action
small	50 <= weight < 60	print "small"
Large	60 <= weight < 70	print "large"
extra large	70 <= weight	print "extra large“

Step 2: Using steps provided for implementation of a selection statement and the worked example as a model, complete the translation of the design (selection matrix) into the target language.

```
if ( 50 <= weight && weight < 60 )
  cout << "Egg weight (g): " << weight << " => small";
else if ( _______________ )
    cout << _______________ << endl;
else
   ____________________
```

Instructor chooses how much of the implementation is faded

And so on…

Selection *semantics* – *verification FWEs*

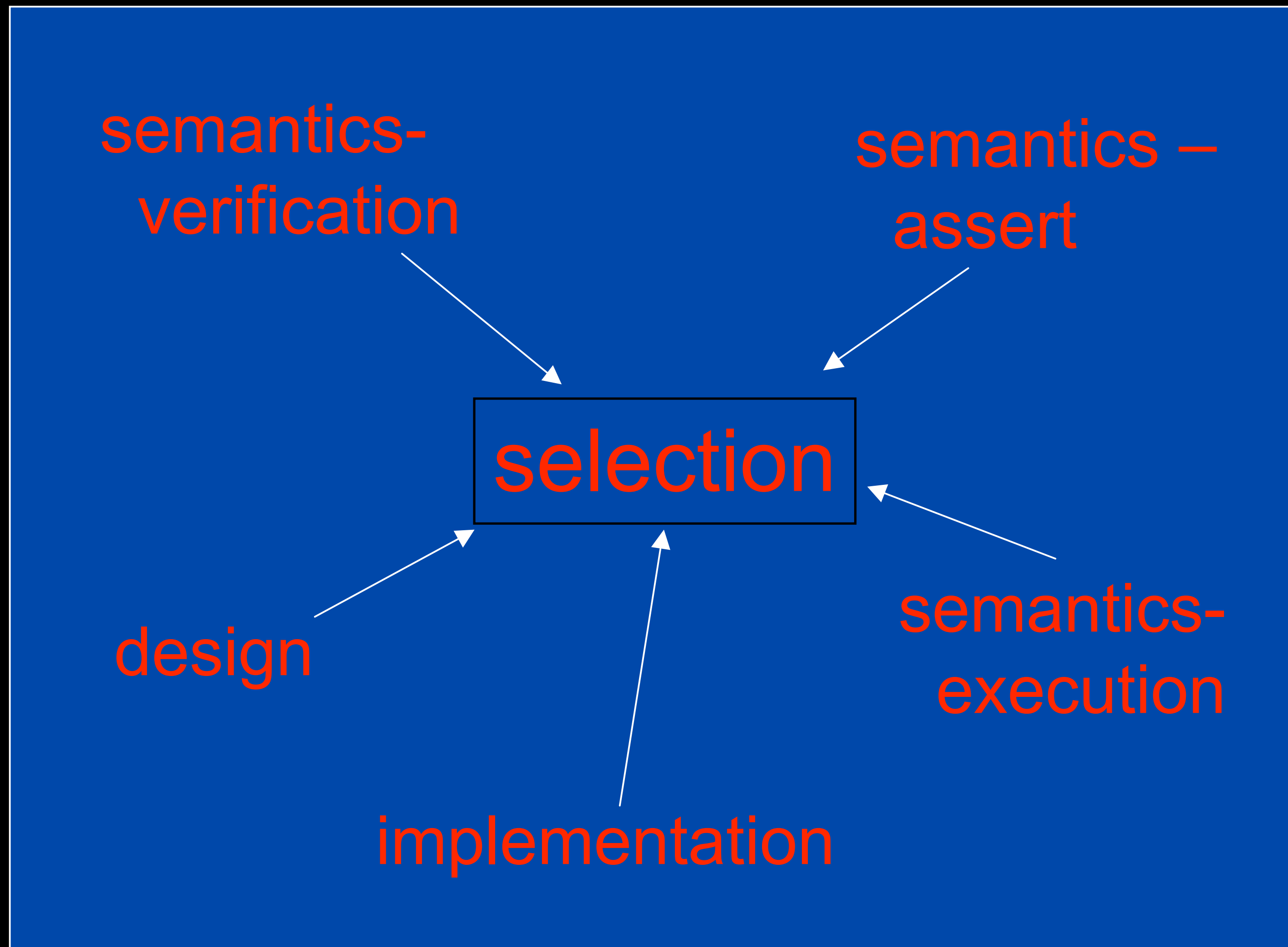
Identify test cases describing expected behavior

Selection *semantics* – *assert FWEs*

Explicitly state semantics of components

Selection *semantics* – *execution FWEs*

Trace execution given some inputs


semantics-
verification
semantics –
assert
selection
design
semantics-
execution
implementation

Conclusion

- Learning to program is complex, demanding and prone to cognitive overload
- FWEs provide *graduated* and *repeated* exposure to the facets of programming through working a *variety* of problems in an area and address the issue of cognitive load

Moving Forward

- Will this work?
- How can we evaluate the effectiveness of this approach?
- Are the problem solving skills developed transferable to other domains?

Questions?

Comments?