CSci 780
Advanced Software Engineering

Automated Assistance for Program Restructuring

by William D. Griswold and David Notkin

Outline

- Problem
- Proposed Solution
- Example
- Transformations
- Implementation
- Evaluation
- Discussion
- Summary

Problem

- Maintenance degrades systems over time
  - Quick and dirty fixes
  - No restructuring due to fear of breaking something
- Information hiding doesn't always help
  - Can't anticipate all changes
  - Can't always afford to implement flexibility
  - Must choose a possibly sub-optimal modularization
- Cross-module changes are necessary, and the cause of degradation

Belady and Lehman 1977 (OS/360)

"A system that is used undergoes continuing change until it becomes more economical to replace it by a new or restructured system"

Proposed Solution

- Ease maintenance by breaking it into restructuring followed by change
- But manual fixes often introduce bugs!
  - Study: 53% of bugs introduced with new feature
- Use a tool to help with the restructuring
  - Automatically transform the code using high-level operations
  - But only when the meaning will not be changed

Example

```
sub push(item,stack&) {
  insert(item,stack.head);
  return stack;
}

sub pop(stack &) {
  val = stack.head;
  stack.remove(head);
  return val;
}

push(mystack);
push(pop(mystack));
```

Transformations (1/2)

- create-variable: add a new variable
  - Can't collide with existing names
- rename-variable: rename all occurrences
  - Can't collide with existing names
- move-expr: move an expression
  - Intervening expressions can't depend on moved one
- var-to-expr: replace a var use with its definition
  - Can't have side-effects. Must satisfy move-expr

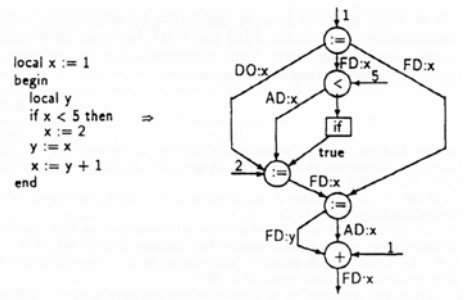
Transformations (2/2)

- extract-function: move the selected lines into a function, and replace the lines with a call
  - Need to identify all non-local variables and pass them as parameters
- scope-substitute-call: find places where code can be replaced with a function call
  - Infeasible in general! But heuristics are workable
- Also: binding-to-expr, expr-to-binding, inline-function

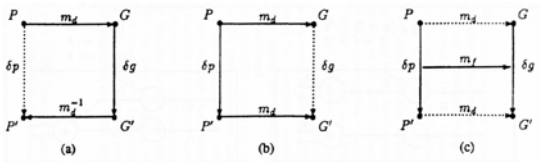
Implementation (1/2)

- Three representations
  - Syntactical: parse code into an abstract syntax tree
  - Program dependence graph (PDG): shows data relationships between statements
  - Control flow graph (CFG): shows ordering of statements
- Need to manage relationships between reps
  - Correct
  - Efficient

Program Dependence Graph



Management Strategies



Implementation (2/2)

- Performing meaning-preserving transformations
  - procedure delta_p(e)
    for u_i in uses(e) do
      compensation_trans(u_i)
    local_trans(e)
- Map compensation_trans to PDG/CFG, do graph manipulation, then map back to AST

Evaluation

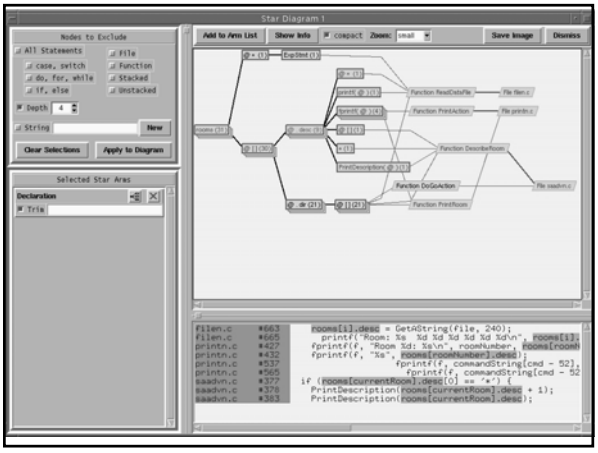
- 6 people restructuring matrix multiply by hand
  - Physical distance between changes increases errors
  - Code transformation technique should be usable
- Restructure KWIC
  - Was successful using the tool

Benefits

- Don't need to search for cross-module code related to a change
- Don't need to implement the changes manually
- Don't need to ensure that a change is correct
- No bugs will be introduced by change!

Later Work

- Restructuring data abstractions
- Restructuring using invariants
- Support for Java, Ada, C
- StarTool: shows how fragments of code relate to each other and are distributed in the system
  - Changes to the diagram change the code



Open Questions

- When should restructuring be done?
- Are operations powerful enough to be useful?
- Theoretical basis for implementing transformations?
- Is it cost effective?
- Scalability?
- High-level semantics?
- Useful?

Summary

- Show how meaning-preserving transformations can automate restructuring
- Developed a model for defining meaning-preserving source-to-source transformations
- Implemented a tool