Reverse Engineering Assembler Code

Martin Ward

martin@gkc.org.uk

http://www.cse.dmu.ac.uk/~mward

Software Technology Research Lab

De Montfort University

Leicester, UK

Outline

- Program Transformation Theory
- Data Structure Analysis
- Control Flow Restructuring
- Slicing Methods:
  - Syntactic Slicing
  - Semantic Slicing
  - Dynamic Slicing
  - Conditioned Slicing
- Raising the Abstraction Level
- Case Studies
- Combining Dynamic and Static Slicing

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.
- If *every* feature of reality were included in the model, then it would no longer be a model, but the thing itself!

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.
- If *every* feature of reality were included in the model, then it would no longer be a model, but the thing itself!
- So: the right question to ask of any mathematical model or scientific theory is not "Does it account for everything?" but "Is it useful?". In other words, "Does it account for all the features of reality that we are interested in for this purpose?".

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.
- If *every* feature of reality were included in the model, then it would no longer be a model, but the thing itself!
- So: the right question to ask of any mathematical model or scientific theory is not "Does it account for everything?" but "Is it useful?". In other words, "Does it account for all the features of reality that we are interested in for this purpose?".
- For example: In analysing a computer program for correctness, we are not interested in how long the program takes to process the input, or what sequence of internal states it goes through on the way to generating the result.

Syntax and Semantics of WSL

WSL is a Wide Spectrum Language which forms the basis for the WSL theory of program transformations and the FermaT program transformation system.

A *transformation* is any operation on a program which preserves its external input-output behaviour.

Program States

A program starts executing in some *state*. A state is a collection of variables each of which has a value. The collection of variables is called the *state space*. The state space may change during the execution of the program, with variables being added or removed.

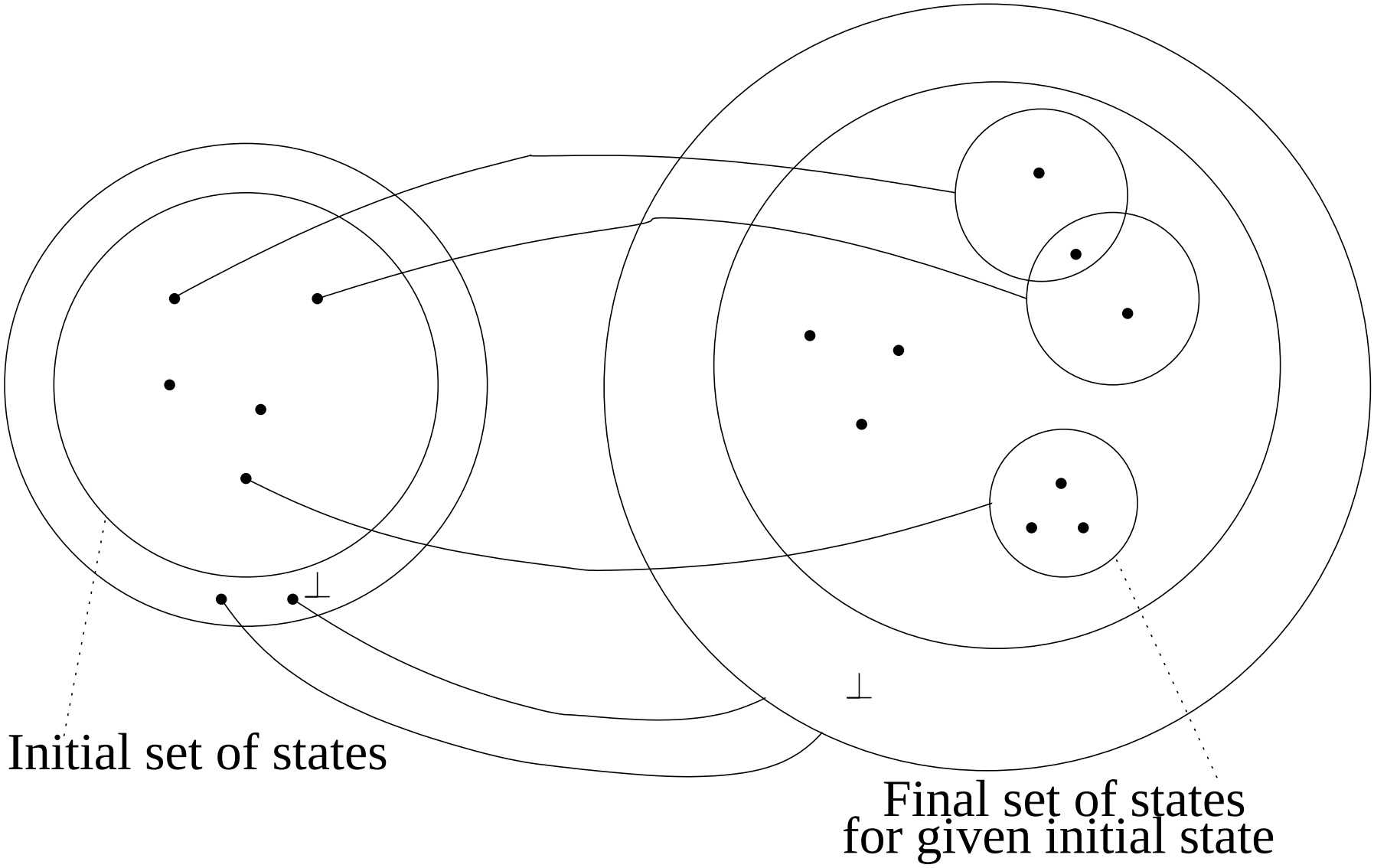
So a state can be modelled as a function from the state space to the set of values. This function returns the value of each variable in the state space.

Programs may be *non-deterministic*: for the same initial state, there may be two or more possible final states.

So we will represent the behaviour of a program by a function which maps from each initial state to the set of possible final states.

This function is called a *state transformation*.

The Semantics of WSL



The WSL Kernel Language

"The quarks of programming"

The primitive kernel statements are constructed from formulae and lists of variables.

Let **P** and **Q** be any formulae and **x** and **y** be any lists of variables:

- **Assertion:** $\{\mathbf{P}\}$ Does nothing if **P** is true, aborts if **P** is false;
- **Guard:** $[\mathbf{Q}]$ Ensures that **Q** is true by restricting previous nondeterminism;
- **Add variables:** $\mathbf{add}(\mathbf{x})$ adds the variables in **x** to the state space and assigns arbitrary values to them;
- **Remove variables:** $\mathbf{remove}(\mathbf{y})$ removes the variables in **y** from the state space.

The WSL Kernel Language

"The quarks of programming"

The compound statements are as follows; for any kernel language statements $\mathbf{S}_1$ and $\mathbf{S}_2$, the following are also kernel language statements:

- **Sequence:** $(\mathbf{S}_1;\ \mathbf{S}_2)$ executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;
- **Nondeterministic choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution;
- **Recursion:** $(\mu X.\mathbf{S}_1)$ where X appearing in the body $\mathbf{S}_1$ represents a recursive procedure call.

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

$$\mathbf{add}(\langle x \rangle);\ [x = 1]$$

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

$$\mathbf{add}(\langle x \rangle);\ [x = 1]$$

while:

$$x := x + 1$$

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

$$\textbf{add}(\langle x \rangle);\ [x = 1]$$

while:

$$x := x + 1$$

is defined as:

$$\textbf{add}(\langle x' \rangle);\ [x' = x + 1];\ \textbf{add}(\langle x \rangle);\ [x = x'];\ \textbf{remove}(\langle x' \rangle)$$

Language Extensions

The **if** statement

if B then S_1 else S_2 fi

Language Extensions

The **if** statement

$$\textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

can be implemented by a nondeterministic choice with guarded arms:

$$(([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

Language Extensions

The **if** statement

$$\textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

can be implemented by a nondeterministic choice with guarded arms:

$$(([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

Loops are defined using recursion, for example the **while** loop:

$$\textbf{while B do S od}$$

Language Extensions

The **if** statement

$$\textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

can be implemented by a nondeterministic choice with guarded arms:

$$(([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

Loops are defined using recursion, for example the **while** loop:

$$\textbf{while B do S od}$$

is defined:

$$(\mu X.((([\mathbf{B}];\ \mathbf{S});\ X)\ \sqcap\ [\neg\mathbf{B}]))$$

More Language Extensions

- **for** loops
- Dijkstra's Guarded Command Language
- Loops with multiple **exit**s
- Mutually recursive procedures (labels and **goto**s)
- Local variables
- Procedures and functions with parameters
- Expressions with side-effects
- Assembler language

The Specification Statement

WSL is a "Wide Spectrum" language. It includes both low-level programming constructs and high-level abstract specifications.

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q}$$

"Assign a new value $\mathbf{x}'$ to $\mathbf{x}$ such that $\mathbf{Q}$ is true, otherwise abort"

The formula $\mathbf{Q}$ defines the relationship between the new value $\langle x_1', x_2', \ldots, x_n' \rangle$ and the old value $\langle x_1, x_2, \ldots, x_n \rangle$

The Specification Statement

WSL is a "Wide Spectrum" language. It includes both low-level programming constructs and high-level abstract specifications.

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q}$$

"Assign a new value $\mathbf{x}'$ to $\mathbf{x}$ such that $\mathbf{Q}$ is true, otherwise abort"

The formula $\mathbf{Q}$ defines the relationship between the new value $\langle x'_1, x'_2, \ldots, x'_n \rangle$ and the old value $\langle x_1, x_2, \ldots, x_n \rangle$

For example, add 1 to x:

$$\langle x \rangle := \langle x' \rangle.(x' = x + 1)$$

The Specification Statement

WSL is a "Wide Spectrum" language. It includes both low-level programming constructs and high-level abstract specifications.

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q}$$

"Assign a new value $\mathbf{x}'$ to $\mathbf{x}$ such that $\mathbf{Q}$ is true, otherwise abort"

The formula $\mathbf{Q}$ defines the relationship between the new value $\langle x'_1, x'_2, \ldots, x'_n \rangle$ and the old value $\langle x_1, x_2, \ldots, x_n \rangle$

For example, add 1 to x:

$$\langle x \rangle := \langle x' \rangle.(x' = x + 1)$$

Swap the values of x and y:

$$\langle x, y \rangle := \langle x', y' \rangle.(x' = y \ \wedge \ y' = x)$$

The Specification Statement

Specification of a sorting program:

$$A := A'.(\mathsf{sorted}(A') \wedge \mathsf{permutation\_of}(A', A))$$

The output must be sorted and a permutation of the input.

It precisely describes *what* we want our sorting program to do without saying *how* it is to be achieved

Transformation Proof Methods

Transformation proof methods include:

Transformation Proof Methods

Transformation proof methods include:

- Translate to the kernel language and prove the transformation via:
  - Comparing the semantic functions; or
  - Comparing weakest preconditions

Transformation Proof Methods

Transformation proof methods include:

- Translate to the kernel language and prove the transformation via:
  - Comparing the semantic functions; or
  - Comparing weakest preconditions
- Prove via weakest preconditions without using the kernel language

Transformation Proof Methods

Transformation proof methods include:

- Translate to the kernel language and prove the transformation via:
  - Comparing the semantic functions; or
  - Comparing weakest preconditions
- Prove via weakest preconditions without using the kernel language
- Prove by applying a sequence of existing transformations and proof rules

Program Transformation

WSL includes both *abstract specifications* and *executable programs* within the same language. This means that:

- Refinement of a specification into an executable program, and
- Reverse engineering from a program to a specification

are both examples of program transformations.

Modelling Assembler in WSL

Our approach involves three types of modelling:

1. Complete model: Each assembler instruction is translated into WSL statements which capture all the effects of the instruction, including condition codes and registers;
2. Partial model: Branches to register are modelled by attempting to determine all possible targets of such a branch, associating a value with each target, and calling a "dispatch" routine which finds the target for the given value;
3. Self-modifying code: Some cases are detected and handled (overwriting a NOP/branch, modifying a length field etc.) but general self-modifying code requires human intervention: usually to renovate the assembler using more standard programming practices!

Modelling Assembler in WSL

- Standard opcodes
- Standard system macros for file handling etc.
- User macros
- Structured macros
- Condition Code
- BAL/BAS (Branch and Save)
- Branch to Register
- External Subroutine Calls

Modelling Assembler in WSL

- Detected Jump Tables
- EXecute Statements
- Data Declarations
- DSECT Base Register Modification
- EQUates: constants and aliases
- Self-Modifying Code
- Structured and Unstructured CICS calls
- SQL, etc.

Data Restructuring

All data declarations in the assembler module (including unnamed data items) are parsed and added to a database which records for each data element:

- Name
- Sequence number
- Base CSECT or DSECT name
- Type
- Offset
- Length
- Repeat Count
- Initial Value

Data Restructuring

The data restructuring process analyses the data layout into nested structures. Where this is not possible, (eg overlapping data declarations), structures are overlaid using:

- `union` declarations in C; or
- `REDEFINES` declarations in COBOL

Assembler DSECTs are converted to struct pointers in C and LINKAGE SECTION data structures in COBOL.

C header files and COBOL copybooks are generated automatically from the database.

Control Flow Restructuring

Unstructured control flow is represented using **Action Systems**:

actions A_1 :
$A_1 \equiv$ **S**$_1$ **end**
...
$A_n \equiv$ **S**$_n$ **end endactions**

- A collection of mutually recursive parameterless procedures
- **call** A_i is a call to action A_i
- A special statement **call** Z causes immediate termination of the whole action system
- An action system is a single statement which can appear as a component of another statement (including another action system)

Regular Action Systems

actions A_1 :

$A_1 \equiv$ **S**$_1$ **end**

...

$A_n \equiv$ **S**$_n$ **end endactions**

- If execution of each action body **S**$_i$ always leads to an action call (or **call** Z) then we have a **Regular Action System**.
- In this case, action calls are like **goto**s: no action ever returns.
- The system can only terminate via **call** Z

A Regular Action System

```
var ⟨m := 0, p := 0, last := ""⟩ :
    actions prog :
    prog  ≡ line := ""; m := 0; i := 1;
            call inhere end
    loop  ≡ i := i + 1;
            if i = n + 1 then call alldone fi;
            m := 1;
            if item[i] ≠ last
              then write(line var os);
                   line := " "; m := 0;
                   call inhere fi;
            call more end
    inhere ≡ p := number[i];
             line := item[i]; line := line ++ " " ++ p;
             call more end
    more  ≡ if m = 1
              then p := number[i];
                   line := line ++ ", " ++ p fi;
            last := item[i];
            call loop end
    alldone ≡ write(line var os); call Z end endactions end
```

Collapse Action System

```
var ⟨m := 0, p := 0, last := “”⟩ :
    line := “”;
    m := 0;
    i := 1;
    do p := number[i];
        line := item[i];
        line := line ++ “ ” ++ p;
        do if m = 1
              then p := number[i];
                   line := line ++ “, ” ++ p fi;
           last := item[i];
           i := i + 1;
           if i = n + 1
              then write(line var os); exit(2) fi;
           m := 1;
           if item[i] ≠ last
              then write(line var os);
                   line := “”;
                   m := 0;
                   exit(1) fi od od end
```

Fully Restructured Version

```
i := 1;
do line := item[i] ++ "" ++ number[i];
    do i := (i + 1);
        if item[(i − 1)] ≠ item[i] ∨ i = (n + 1)
          then exit(1) fi;
        line := line ++ ", " ++ number[i] od;
    write(line var os);
    if i = (n + 1) then exit(1) fi od
```

Assembler Restructuring

That was the easy part!

The hardest part of assembler restructuring is converting assembler **subroutines** into structured **procedures**.

Subroutine call:

A_0001 $\equiv$ r14 $:=$ 1234; **call** FOO **end**

A_0002 $\equiv$ ...

Subroutine return:

FOORET $\equiv$ destination $:=$ r14; **call** dispatch **end**

...

dispatch $\equiv$ **if** destination $=$ 0 **then call** Z
elsif ...
elsif destination $=$ 1234 **then call** A_0002
elsif ... **fi end**

Assembler Restructuring

The analyser has to do the following:

- Determine which actions belong in the body of FOO
- Prove (via dataflow analysis) that the value assigned to r14 always ends up in destination before the call to dispatch

Restructured code:

A_0001 $\equiv$ r14 := NOTUSED_1234; FOO(); **call** A_0002 **end**

A_0002 $\equiv$ ...

where

proc $FOO()$ $\equiv$ ... **end**

Assembler Restructuring

Problems:

- Multiple entry points to subroutine
- Multiple exit points from subroutine
- Branch from the middle of one subroutine into another
- Two subroutines branch to common code
- Returning directly to the caller's caller
- Increment the return address (to skip over a branch)
- Overwrite the return address with a different one
- A subroutine "return" which is actually a "call"
- . . . and many more . . .

Assembler Restructuring

Currently, over 90% of all hand-written assembler modules can be fully restructured automatically.

Program Slicing

- **Idea:** when attempting to understand a program we often need to know how variables got their values at specific points

Program Slicing

- **Idea:** when attempting to understand a program we often need to know how variables got their values at specific points
- **Informal Definition:** A program slice is a subset of a program which contains all the statements which can potentially affect the values of certain variables of interest at given positions in the program (E.g. we are interested in the value of variable x on line 232. The sliced program contains everything needed to compute x at that point)

Program Slicing

- **Idea:** when attempting to understand a program we often need to know how variables got their values at specific points
- **Informal Definition:** A program slice is a subset of a program which contains all the statements which can potentially affect the values of certain variables of interest at given positions in the program (E.g. we are interested in the value of variable x on line 232. The sliced program contains everything needed to compute x at that point)
- **More Formal Definition:** A program slice **S** is a *reduced, executable program* obtained from a program **P** by removing statements, such that **S** replicates part of the behaviour of **P**

Program Slicing

- Slicing allows one to find *semantically meaningful decompositions* of programs, where the decompositions consist of elements that are not textually contiguous
- Program slicing is a technique for visualising dependencies and restricting attention to just the components of a program relevant to evaluation of certain expressions.

Basic slicing techniques:

- Backward slicing reveals which other parts of the program the value of an expression depends on
- Forward slicing determines which parts of the program depend on a particular expression.

Program Slicing

Classes of slicing techniques:

- Static slicing
- Syntactic slicing
- Dynamic slicing
- Conditioned slicing
- Semantic slicing
- Conditioned Semantic slicing

Slicing helps programmers understand program structure, which aids program comprehension, maintenance, testing, and debugging; slicing can also assist parallelisation, integration and comparison of program versions.

Slicing as a Program Transformation

A slice is not generally a transformation of the original program because a transformation has to preserve the *whole* behaviour of the program, while in the slice some statements which affect the values of some output variables (those not in the slice) may have been deleted.

Slicing as a Program Transformation

A slice is not generally a transformation of the original program because a transformation has to preserve the *whole* behaviour of the program, while in the slice some statements which affect the values of some output variables (those not in the slice) may have been deleted.

Slicing *can* be formalised as a program transformation on a modification of the original program.

Slicing as a Program Transformation

A key insight of this formulation is that it defines the concept of slicing as a combination of two relations:

1. A syntactic relation (statement deletion) and
2. A semantic relation (which shows what subset of the semantics has been preserved).

Slicing as a Program Transformation

For example, suppose we have this program:

$x := y + 1;$

$y := y + 4;$

$x := x + z$

where we are interested in the final value of x.

Slicing as a Program Transformation

For example, suppose we have this program:

$x := y + 1;$

$y := y + 4;$

$x := x + z$

where we are interested in the final value of x.

The assignment to y on line 15 can be sliced away:

$x := y + 1;$

$x := x + z$

Slicing as a Program Transformation

For example, suppose we have this program:

$x := y + 1;$

$y := y + 4;$

$x := x + z$

where we are interested in the final value of x.

The assignment to y on line 15 can be sliced away:

$x := y + 1;$

$x := x + z$

These two programs are not equivalent, because they have different effects on the variable y. But if we append:

remove(y)

then the result is two programs which *are* equivalent.

Reduction

The syntactic relation we need for slicing is called **reduction**.

A program $\mathbf{S}_2$ is a reduction of $\mathbf{S}_1$ if it can be formed from $\mathbf{S}_1$ by replacing components of $\mathbf{S}_1$ by **skip** statements.

skip is a reduction of any program.

Every program is a reduction of itself.

Semi-Refinement

A slice does not have to be exactly equivalent to the original program. Consider the program:

$$\mathbf{S};\ x := 0$$

where we are slicing on x and **S** has no assignments to x. Clearly we want to slice away **S**.

But **S**; $x := 0$ is only equivalent to $x := 0$ on x provided **S** terminates.

We want to be able to "slice away" potentially non-terminating code. The semantic relation we need is **semi-refinement**. A semi-refinement preserves the behaviour of the original program whenever the original program terminates. Otherwise, the semi-refinement can do anything at all.

Syntactic Slice

A **Syntactic Slice** of **S** on a set X of variables is any program **S**$'$ which satisfies these two conditions:

1. **S**$'$ is a reduction of **S**
   i.e. **S**$'$ is formed from **S** by replacing statements by **skip**; and
2. **S**$'$ is a semi-refinement of **S** on X
   i.e. **S**$'$ preserves the behaviour of **S** on X whenever **S** terminates

Syntactic Slicing Example

sum $:= 0$;

prod $:= 1$;

$i := 1$;

while $i \leqslant n$ **do**

    sum := sum $+ A[i]$;

    prod := prod $* A[i]$;

    $i := i + 1$ **od**;

PRINT(“`sum = `”, sum);

PRINT(“`prod = `”, prod)

Slice with respect to the variable prod on the last line

Syntactic Slicing Example

```
sum := 0;
prod := 1;
i := 1;
while i ⩽ n do
    sum := sum + A[i];
    prod := prod * A[i];
    i := i + 1 od;
PRINT("sum = ", sum);
PRINT("prod = ", prod)
```

Slice with respect to the variable prod on the last line

These statements can be deleted

Syntactic Slicing Example

prod $:= 1;$
$i := 1;$
while $i \leqslant n$ **do**

    prod := prod $* A[i];$
    $i := i + 1$ **od**;

PRINT(“`prod = `”, prod)

Slice with respect to the variable prod on the last line

The resultant slice

Syntactic Slice

Slicing an action system:

actions A1 :

A1 $\equiv$ sum $:= 0$; **call** A2 **end**

A2 $\equiv$ prod $:= 0$; **call** A3 **end**

A3 $\equiv$ $i := 1$; **call** A4 **end**

A4 $\equiv$ **if** $i \leqslant n$ **then call** A5 **else call** B1 **fi end**

A5 $\equiv$ sum $:=$ sum $+ A[i]$; **call** A6 **end**

A6 $\equiv$ prod $:=$ prod $* A[1]$; **call** A7 **end**

A7 $\equiv$ $i := i + 1$; **call** A4 **end**

B1 $\equiv$ PRINT(“`sum =` ”, sum); **call** B2 **end**

B2 $\equiv$ PRINT(“`prod =` ”, prod); **call** Z **end**

endactions

Syntactic Slice

The slice on i is:

actions A3 :

A3 $\equiv i := 1;$ **call** A4 **end**

A4 $\equiv$ **if** $i \leqslant n$ **then call** A7 **else call** Z **fi end**

A7 $\equiv i := i + 1;$ **call** A4 **end endactions**

Here, we have unfolded and removed all actions which do nothing other than call another action.

Slicing Example

Slices can be constructed by tracking **control dependencies** and **data dependencies**.

For example:

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Which statements do not contribute to the final value of x?

Slicing Example

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Some of the control and data dependencies:

$$
\begin{aligned}
x := f &\xrightarrow{\text{ctrl}} q?(c) \\
q?(c) &\xrightarrow{\text{data}} c := g \\
x := f &\xrightarrow{\text{ctrl}} p?(i) \\
q?(i) &\xrightarrow{\text{ctrl}} p?(i) \\
p?(i) &\xrightarrow{\text{data}} i := h(i)
\end{aligned}
$$

It seems that *everything* is needed! ?

Slicing Example

Tracking all data and control dependencies will always produce a *valid* slice, but not necessarily a *minimal* slice.

What is the minimal slice?

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Slicing Example

Tracking all data and control dependencies will always produce a *valid* slice, but not necessarily a *minimal* slice.

What is the minimal slice?

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $\boxed{c := g}$ **fi**;
  $i := h(i)$ **od**

The assignment to c is redundant: once x has been assigned the value f, it does not matter whether it is assigned again, or how many times!

Semantic Slice

By dropping the syntactic relation in the definition of slicing we get a more general form of slicing.

A *semantic slice* of **S** on X is any program $\mathbf{S}'$ which satisfies the condition:

1. $\mathbf{S}'$ is a semi-refinement of **S** on X
   i.e. $\mathbf{S}'$ preserves the behaviour of **S** on X whenever **S** terminates

This allows more freedom for simplification of the code.

Note, however, that while a syntactic slice is no larger than the original program, a semantic slice could be smaller or larger.

Syntactic vs Semantic Slices

Original

if $p = q$
  then $x := 18$
  else $x := 17$ **fi**;
if $p \neq q$
  then $y := x$
  else $y := 2$ **fi**

Syntactic slice on y

if $p = q$
  then skip
  else $x := 17$ **fi**;
if $p \neq q$
  then $y := x$
  else $y := 2$ **fi**

Semantic slice on y

if $p = q$
  then $y := 2$
  else $y := 17$ **fi**

Slicing Example

A semantic slice can often give a more concise and understandable result. For example:

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f$;
      $c := g$ **fi**;
  $i := h(i)$ **od**

Slicing Example

A semantic slice can often give a more concise and understandable result. For example:

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f;$
      $c := g$ **fi**;
  $i := h(i)$ **od**

Becomes:

if $p?(i) \wedge q?(c)$ **then** $x := f$ **fi**

Slicing Example

A semantic slice on the final value of sum for this program:

sum $:= 0$;

prod $:= 1$;

$i := 1$;

while $i \leqslant n$ **do**

    sum $:=$ sum $+ A[i]$;

    prod $:=$ prod $* A[i]$;

    $i := i + 1$ **od**;

PRINT(“`sum = `”, sum);

PRINT(“`prod = `”, prod)

Slicing Example

A semantic slice on the final value of sum for this program:

sum $:= 0;$

prod $:= 1;$

$i := 1;$

while $i \leqslant n$ **do**

    sum $:=$ sum $+ A[i];$

    prod $:=$ prod $* A[i];$

    $i := i + 1$ **od**;

PRINT(“`sum = `”, sum);

PRINT(“`prod = `”, prod)

The result is:

sum $:=$ REDUCE(“+”, $A[1..n]$)

A Minimal Semantic Slice

For any program $\mathbf{S}$ and any set list of variables $\mathbf{x}$ in the final state space, the single specification statement:

$$\mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}''))$$

is a valid semantic slice on $\mathbf{x}$ for $\mathbf{S}$.

Here, WP denotes the *Weakest Precondition*.

For statements involving recursion or iteration, $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'')$ is an infinitely long formula. But the result still has practical applications for analysing program fragments.

Abstraction and Refinement

A slice over a statement containing no loops or recursion can be computed by a process of:

Abstraction and Refinement

A slice over a statement containing no loops or recursion can be computed by a process of:

1. **Abstraction** to a specification statement;

Abstraction and Refinement

A slice over a statement containing no loops or recursion can be computed by a process of:

1. **Abstraction** to a specification statement;
2. **Simplifying** the specification statement; and

Abstraction and Refinement

A slice over a statement containing no loops or recursion can be computed by a process of:

1. **Abstraction** to a specification statement;
2. **Simplifying** the specification statement; and
3. **Refining** the specification back to code again.

Abstraction and Refinement

For example, slicing this program on the final value of q:

$x := 3;$

$v := v + x;$

var $\langle x := y \rangle$:

$\quad x := x + 1;$

$\quad z_1 := x;$

$\quad x := z_2 + x;$

$\quad v := v + x;$

$\quad y := 0$ **end**;

$q := v$

Abstraction and Refinement

For example, slicing this program on the final value of q:

$x := 3;$

$v := v + x;$

var $\langle x := y \rangle$:

$x := x + 1;$

$z_1 := x;$

$x := z_2 + x;$

$v := v + x;$

$y := 0$ **end**;

$q := v$

gives:

$q := v + y + z_2 + 4$

Selective Unrolling

Another technique used in semantic slicing is Selective Unrolling of loops. For example, slicing on the final value of found_low:

$i := 1;$
while $i \leqslant n$ **do**
  if $A[i] >$ high
    then high $:= A[i];$ found_high $:= 1$ **fi**;
  if $A[i] <$ low
    then low $:= A[i];$ found_low $:= 1$ **fi**;
  sum $:=$ sum $+ A[i];$
  $i := i + 1$ **od**

Selective Unrolling

Another technique used in semantic slicing is Selective Unrolling of loops. For example, slicing on the final value of found_low:

$i := 1;$
while $i \leqslant n$ **do**
  if $A[i] >$ high
    then high $:= A[i];$ found_high $:= 1$ **fi**;
  if $A[i] <$ low
    then low $:= A[i];$ found_low $:= 1$ **fi**;
  sum $:=$ sum $+ A[i];$
  $i := i + 1$ **od**

The result is:

$i := 1;$
while low $\leqslant A[i] \wedge i \leqslant n$ **do**
  $i := i + 1$ **od**;
if $i \leqslant n$ **then** found_low $:= 1$ **fi**

Dynamic Slice

A dynamic slice of a program **S** is a reduced executable program **S**′ which replicates part of the behaviour of **S** on a particular initial state. We can define this initial state by means of an assertion $\{\mathbf{A}\}$ which specifies the initial value of each variable.

A **Dynamic Syntactic Slice** of **S** with respect to a set of variables X and an initial state, defined by the assertion $\{\mathbf{A}\}$ is any program **S**′ which satisfies these two conditions:

1. **S**′ is a reduction of **S**
   i.e. **S**′ is formed from **S** by replacing statements by **skip**; and
2. $(\{\mathbf{A}\};\ \mathbf{S}')$ is a semi-refinement of $(\{\mathbf{A}\};\ \mathbf{S})$ on X
   i.e. **S**′ preserves the behaviour of **S** on X for the initial state defined by **A**, provided **S** terminates on this state

Conditioned Slice

If we allow any initial assertion, then the result is called a *conditioned slice*:

A **Conditioned Syntactic Slice** of **S** with respect to a set of variables X and any assertion $\{\mathbf{A}\}$ is any program $\mathbf{S}'$ which satisfies these two conditions:

1. $\mathbf{S}'$ is a reduction of **S**
   i.e. $\mathbf{S}'$ is formed from **S** by replacing statements by **skip**; and
2. $(\{\mathbf{A}\}; \mathbf{S}')$ is a semi-refinement of $(\{\mathbf{A}\}; \mathbf{S})$ on X
   i.e. $\mathbf{S}'$ preserves the behaviour of **S** on X for every initial state which satisfies **A** and for which **S** terminates.

If we remove the first requirement then we have a **Conditioned Semantic Slice**.

Raising the Abstraction Level

One way to raise the abstraction level is to delete irrelevant code. If we are interested in determining the normal behaviour of a module, then all error handling code is (in this context) irrelevant.

A powerful technique for detecting and deleting error handling code makes use of FermaT transformations for **abort** statements.

The statement **abort** is equivalent to the assertion {**false**}. This, in effect, asserts that "this point in the program will not be reached". (More precisely: "I don't care what happens if processing reaches this point").

So, deleting error handling code is a form of conditioned slicing.

Raising the Abstraction Level

Transformations involving **abort**:

1. (**S**; **abort**) $\approx$ **abort**
2. (**abort**; **S**) $\approx$ **abort**
3. **if B then abort else S fi** $\approx$ $\{\neg\mathbf{B}\}$; **S**
4. **if B then S else abort fi** $\approx$ $\{\mathbf{B}\}$; **S**
5. A procedure whose body consists of a single **abort** can be unfolded and removed.

Raising the Abstraction Level

An example:

if r3 $= 0$ **then** ERR123() **fi**

...

proc ERR123() $\equiv$

  $code := 123;$

  WTO("`Error message...`");

  GENERR() **end**

...

proc GENERR() $\equiv$

  cleanup_code;

  ...;

  ABEND **end**

Raising the Abstraction Level

When we see ABEND in the assembler program, we can be certain that this *not* part of the normal processing.

So we generate an abstraction of the WSL program by inserting an **abort** statement.

Then apply the **abort** processing transformations.

Raising the Abstraction Level

When we see ABEND in the assembler program, we can be certain that this *not* part of the normal processing.

So we generate an abstraction of the WSL program by inserting an **abort** statement.

Then apply the **abort** processing transformations.

The result is:

$\{r3 \neq 0\}$

Migration Case Study (extract)

```
LAAA     B     LAB
         BAL   R10,ENDGROUP
LAB      MVI   LAAA+1,0
         MVC   WLAST,WRITEM
         ZAP   WNET,=P'0'
         BAL   R10,PROCGRP
         MVI   XSW1,X'FF'
         B     LAA
LAC      BAL   R10,PROCGRP
         MVI   XSW1,X'FF'
         B     LAA
LAD      CLI   XSW1,X'FF'
         BNE   LADA
         BAL   R10,ENDGROUP
```

Migration Case Study (extract)

```
LADA      EQU   *
          MVC   WPRT(17),=CL17'NUMBER CHANGED = '
          ED    WORKB,WCHANGE
          LA    R4,WORKB
          LA    R1,9
LADB      CLI   0(R4),C' '
          BNE   LADC
          LA    R4,1(R4)
          BCT   R1,LADB
LADC      EX    R1,WMVC1
*WMVC1   MVC   WPRT+17(1),0(R4)
          BAL   R10,WRITE1
```

Metrics

Metric	Raw WSL	Structured WSL
Statements	561	106
Expressions	1,589	210
McCabe	184	17
Control/Data Flow	520	156
Branch–Loop	145	17
Structural	6,685	751

Migrated COBOL Code

```
MOVE LOW-VALUES TO XSW1
PERFORM S0040-READ-DDIN-P
PERFORM UNTIL END-OF-FILE
  IF WLAST NOT = WRITEM THEN
    IF F-LAAA NOT = 1 THEN
      PERFORM S0050-ENDGROUP-P
    END-IF
    MOVE 0 TO F-LAAA
    MOVE WRITEM TO WLAST
    MOVE 0 TO WNET
  END-IF
  PERFORM S0080-PROCGRP-P
  MOVE HIGH-VALUES TO XSW1
  PERFORM S0040-READ-DDIN-P
END-PERFORM
IF XSW1 = HIGH-VALUES THEN
  PERFORM S0050-ENDGROUP-P
END-IF
MOVE 'NUMBER CHANGED = ' TO WPRT-X-1-17
CALL 'SMLED' USING WORKB BY VALUE 1
  BY REFERENCE CC1 WEDIT-ADDR WCHANGE BY VALUE 4
```

Typical Case Study Results

Typical results from a case study of a 442 line IBM Assembler module, taken from a large commercial system. In this case study, no manual transformations were required to get a compilable C program.

Stage	No. of Statements	McCabe Cyclometric	Control Flow /Data Flow	Branch /Loop	Structural
Initial	958	133	806	405	10,449
Data tr.	916	107	688	335	6,856
Fix Assem	336	18	222	20	2,059

Metrics

No. of Statements is the number of executable statements in the parse tree

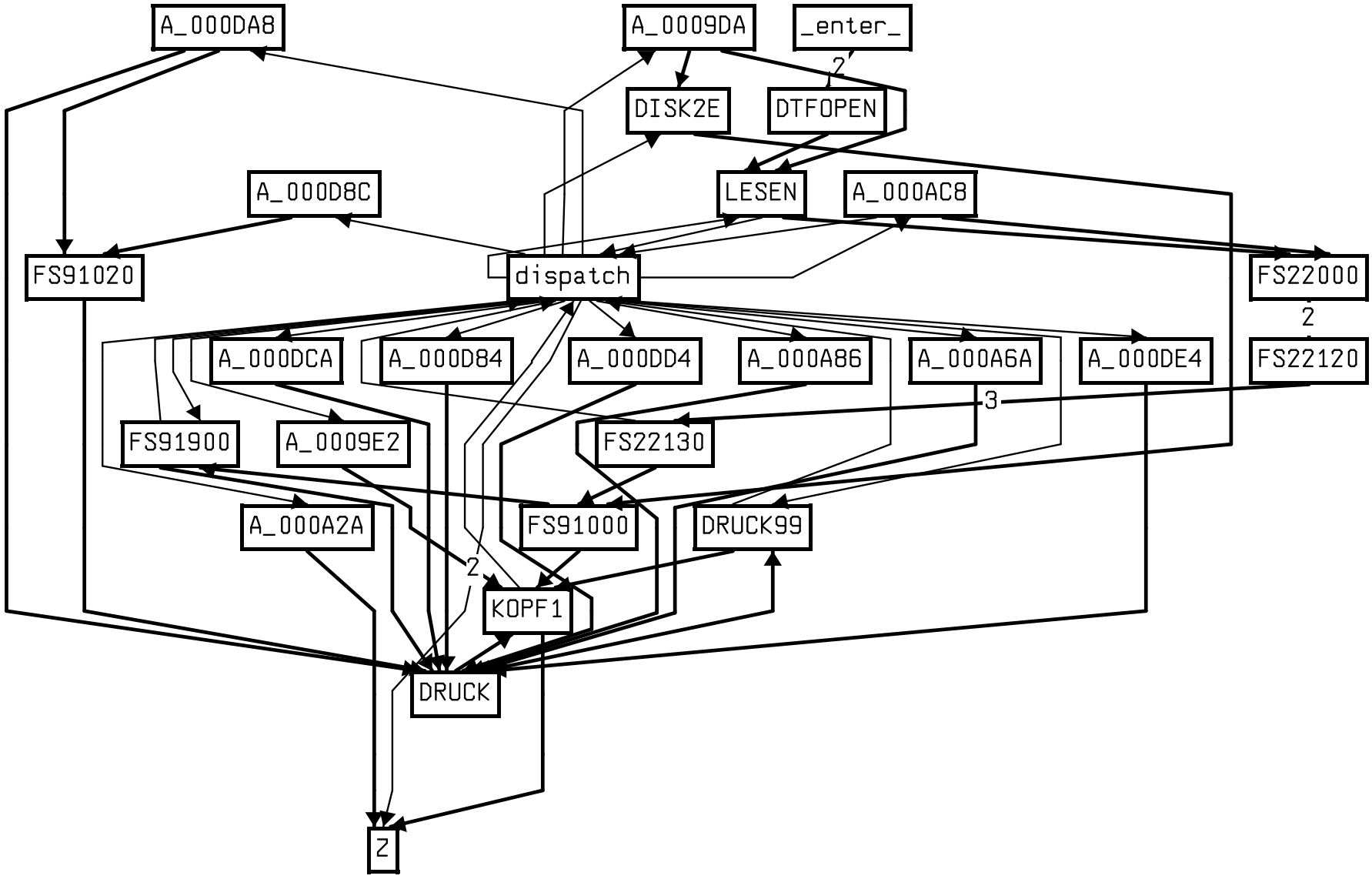
McCabe Cyclometric is the usual McCabe cyclometric complexity

Control Flow/Data Flow counts the number of control flow lines and data flow lines

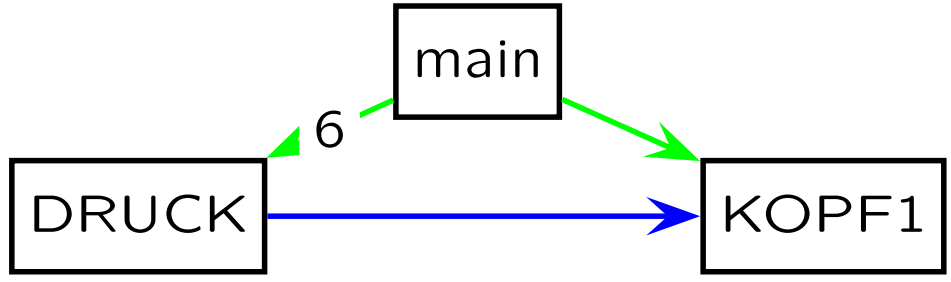
Branch/Loop is a metric which counts the size of loops

Structural is a metric which gives a weighted sum of the structural features of the program.

Call Graph: Before



Call Graph: After



Case Study 1

A complete assembler system consisting of 2,296 modules, 351 of which consisted entirely of data declarations.

	Total LOC	Per module	McCabe
Original Listings	11,959,084	6,149	—
Raw WSL	2,109,906	1,085	135
Transformed WSL	1,205,766	620	27
WSL without comments	528,440	272	27
C Code including comments	1,120,449	576	—

Over 43% of the modules (846 modules) contained *no* loops.

Case Study 1

Raising the abstraction level by deleting error handling code.

Analysis took 5 hours 10 minutes CPU time on a 2.6GHz P4 processor (under 10 seconds per module). A total of 3,876,378 transformations were applied, averaging 1,993 transformations per module and 208 transformations per second.

	Total LOC	Per module	McCabe
Original Listings	11,959,084	6,149	—
Raw WSL	2,109,704	1,085	135
Transformed WSL	513,616	264	25
Abstract WSL	256,853	132	23

Case Study 1

Raising the abstraction level by deleting error handling code.

For a programmer who needs to understand the main functions of a module, and the algorithms it implements, reading a 132 line abstract WSL program should be much simpler than trying to make sense of a 6,000 line assembler listing!

Case Study 2

A random sample of 1,905 assembler modules taken from twelve different organisations.

Over one million lines of source code.

Migration to C took 9 hours 21 minutes CPU time (19.1 seconds per module) on the same PC as the first case study.

Raising the abstraction level took 12 hours 58 minutes CPU time.

FermaT applied a total of 10,318,338 transformations, averaging 6,062 transformations per module and 221 transformations per second.

Note: this selection of modules is biased towards larger code modules.

Case Study 2

	Total LOC	Per module	McCabe
Original Listings	5,377,163	3,159	—
Raw WSL	4,047,258	2,378	373
Transformed WSL	736,816	433	62
Abstract WSL	442,764	260	50

Dynamic Slicing of Assembler

Traditional dynamic slicing algorithms incur a high runtime overhead.

Our method for recording execution paths has virtually zero overhead:

1. Insert breakpoints at the start of each basic block in the assembler module. (FermaT already computes all the potential targets of branch instructions)
2. An automatic breakpoint handler records that this basic block was executed, restores the original instruction, and branches back into the program.
3. Subsequent executions of the same basic block execute at full speed with no performance penalty.

Total overhead is proportional to the size of the program, not the execution time.

Dynamic Slicing of Assembler

If a program point is not reached during the execution(s) of interest, then we can insert an **abort** (or equivalently, the assertion $\{$**false**$\}$) at that point.

This is another form of conditioned slicing.

Combining dynamic and static slicing gives a very concise result.

Combined Slicing Case Study

`UDATECNV` is date conversion module developed by Micro Focus Ltd.

We are interested in how the variable `WRKMTH` is calculated for a particular set of input data.

- Assembler source file is 436 lines
- Syntactic Slicing on `WRKMTH` generates a 48 line WSL file
- Dynamic Slicing on the inputs of interest generates a 63 line WSL file

Combined Slicing Case Study

Applying syntactic slicing on `WRKMTH` to the output of dynamic slicing produces this result:

WRKMM := DDI0M;
DBLEWORD := pack(WRKMM);
WRKMTH := MONTHS[$3 * (\text{cvb}(\text{DBLEWORD}) - 1), 3$]

Applying semantic slicing gives:

WRKMTH := MONTHS[$3 * (\text{cvb}(\text{pack}(\text{DDI0M})) - 1), 3$]

MONTHS consists of twelve three byte strings:
"JAN", "FEB", . . . , "DEC".

Clearly, the program converts a month number DDI0M (in the range 1–12) to a three character abbreviated month name.

The FermaT Transformation System

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory
- Uses a Wide Spectrum Language, called WSL, which was developed in parallel with the development of the transformation theory

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory
- Uses a Wide Spectrum Language, called WSL, which was developed in parallel with the development of the transformation theory
- FermaT implements over 100 transformations together with their applicability conditions

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory
- Uses a Wide Spectrum Language, called WSL, which was developed in parallel with the development of the transformation theory
- FermaT implements over 100 transformations together with their applicability conditions
- Transformations are implemented in an extension of WSL, called $\mathcal{METAWSL}$

The FermaT Transformation System

- FermaT is implemented almost entirely in $\mathcal{META}$WSL
- Therefore, FermaT can transform its own source code!
- This is used on a regular bases as part of the build process

The FermaT Transformation System

- FermaT is use in commercial applications:
  - Assembler to C migration
  - Assembler to COBOL migration
  - Program Slicing
  - Program Comprehension
  - System Reengineering
- FermaT is also used in the MSc in Software Engineering course at many universities in Europe
- The core FermaT transformation engine is available under the GNU GPL from:

  http://www.cse.dmu.ac.uk/~mward/fermat.html