Understanding old assembly code using formal transformations

Doni Pracner

Department of mathematics and informatics
Faculty of Sciences
University of Novi Sad

22 – 27 August 2011
11th Workshop “Software Engineering Education and Reverse Engineering” Ohrid, Macedonia
Presentation organisation

1. Introduction
   - Software Evolution
   - WSL – Wide Spectrum Language

2. Our transformation process
   - Asm2wsl
   - Transf.wsl

3. Examples
   - Programs

4. Summary
   - Results and open questions
Presentation organisation

1. Introduction
   - Software Evolution
   - WSL – Wide Spectrum Language

2. Our transformation process
   - Asm2wsl
   - Transf.wsl

3. Examples
   - Programs

4. Summary
   - Results and open questions
Introduction – Software Ageing

- Old software can be very problematic for maintenance:
  - Obsolete (or no) documentation
  - Source code not available
  - Old technologies
  - Incompatible hardware, etc.

- Software does not degrade with time on its own, the environment changes

- Two main types of aging (Parnas)
  - Lack of Movement
  - Ignorant surgery
Software Evolution

- Software Evolution is the dynamic behavior of programming systems as they are maintained and enhanced over their life times.
- Software Evolution is (largely) repeated reengineering.
- Our aim is to make old, low level, assembly code easier to understand, and hopefully restructure it.
WSL – *Wide Spectrum Language*

- Developed by Martin Ward (since 1989)
- Strong mathematical core
- Formal transformations
- Wide spectrum: from abstract specifications to low level program code
- MetaWSL – operations on WSL code
- Successfully used in migrating legacy assembly code to maintainable C/COBOL code
- Implemented as Fermat program transformation system
Presentation organisation

1. Introduction
   - Software Evolution
   - WSL – Wide Spectrum Language

2. Our transformation process
   - Asm2wsl
   - Transf.wsl

3. Examples
   - Programs

4. Summary
   - Results and open questions
Our transformation process

- Two steps:
  - Asm2wsl – translate the assembly code to WSL
  - Trans.wsl – Automated transformations
- Possible manual transformations
- Main goal is to get a high level version of the original program.
Asm2wsl application

- Translates a subset of x86 assembly to WSL
  - Mostly presumes 80286 for simplicity
- Implemented in Java
- Basically a line by line translator
- Focus is on translating all aspects, not optimization (at this stage)
- We work with a “virtual” processor
The “processor”

- All processor registers are local variables
  - Low and High parts of registers implemented with additional operations
- Flags are variables too
- Overflow variable, needed for 8/16 bits
- Labels – Action system names
- Stack – a list
- Some special macros are recognized and translated directly
- Procedures – nested Action systems
Program parameters of Asm2wsl

Assembler to WSL converter. v 0.78, 2010, by Donny

usage:

    asm2wsl {--option[+-]} filename

options: (def value in parenthesis)

-oc : original code in comments (-)
-c : translate comments (+)
-dump : add memory dump commands to end (+)
# Command translation

<table>
<thead>
<tr>
<th>Assembly</th>
<th>WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ax, dx</td>
<td>ax := dx</td>
</tr>
<tr>
<td>xchg ax, dx</td>
<td>&lt;ax := dx, dx := ax &gt;</td>
</tr>
<tr>
<td>add dx, ax</td>
<td>overflow := 65536; dx := dx + ax;</td>
</tr>
<tr>
<td></td>
<td>IF dx &gt;= overflow THEN</td>
</tr>
<tr>
<td></td>
<td>dx := dx MOD overflow;</td>
</tr>
<tr>
<td></td>
<td>flag_o := 1; flag_c := 1;</td>
</tr>
<tr>
<td></td>
<td>ELSE flag_o := 0; flag_c := 0;</td>
</tr>
<tr>
<td></td>
<td>FI;</td>
</tr>
</tbody>
</table>
## Command translation (cont.)

<table>
<thead>
<tr>
<th>Assembly</th>
<th>WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ah,n</td>
<td>t_e_m_p := n;</td>
</tr>
<tr>
<td></td>
<td>ax := (ax MOD 256) + t_e_m_p * 256;</td>
</tr>
<tr>
<td>add al,12</td>
<td>overflow := 256;</td>
</tr>
<tr>
<td></td>
<td>t_e_m_p := (ax MOD 255) + 12;</td>
</tr>
<tr>
<td></td>
<td>IF t_e_m_p &gt;= overflow THEN</td>
</tr>
<tr>
<td></td>
<td>t_e_m_p := t_e_m_p MOD overflow;</td>
</tr>
<tr>
<td></td>
<td>flag_o := 1; flag_c := 1;</td>
</tr>
<tr>
<td></td>
<td>ELSE flag_o := 0; flag_c := 0;</td>
</tr>
<tr>
<td></td>
<td>FI;</td>
</tr>
<tr>
<td></td>
<td>ax := (ax DIV 256)*256 + t_e_m_p;</td>
</tr>
</tbody>
</table>
## Translation of special macro names

<table>
<thead>
<tr>
<th>Assembly</th>
<th>WSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_str x</td>
<td>PRINT(x);</td>
</tr>
<tr>
<td>print_num x</td>
<td></td>
</tr>
<tr>
<td>read_str x</td>
<td>x := @Read_Line(Standard_Input_Port);</td>
</tr>
<tr>
<td>read_num x</td>
<td></td>
</tr>
</tbody>
</table>
Automatic transformations – transf.wsl

- A small script to call the existing transformations
- Main transformations:
  - Collapse Action Systems
  - Transform DO ... OD loops
  - Constant propagation
  - Remove Redundant
  - Flag removal
Introduction

Software Evolution
WSL – Wide Spectrum Language

Our transformation process

Asm2wsl
Transf.wsl

Examples

Programs

Summary

Results and open questions
Examples

**GCD**  – greatest common divisor

**SumN**  – A simple program with a call to a procedure that sums the top of the stack
Assembly version of the GCD program

```asm
model small
.code
    mov ax,12
    mov bx,8
compare:
    cmp ax,bx
    je theend
    ja greater
    sub bx,ax
    jmp compare
greater:
    sub ax,bx
    jmp compare
theend:
    nop
end
```
GCD – translated to an *Action system*

```
ACTIONS A_S_start:
A_S_start ==
   Ax := 12; bx := 8;
   CALL compare END
compare ==
   IF ax = bx THEN flag_z := 1 ELSE flag_z := 0 FI;
   IF ax < bx THEN flag_c := 1 ELSE flag_c := 0 FI;
   IF flag_z = 1 THEN CALL theend FI;
   IF flag_z = 0 AND flag_c = 0 THEN CALL greater FI;
   IF bx = ax THEN flag_z := 1 ELSE flag_z := 0 FI;
   IF bx < ax THEN flag_c := 1 ELSE flag_c := 0 FI;
   bx := bx - ax;
   CALL compare;
   CALL greater END
greater ==
   IF ax = bx THEN flag_z := 1 ELSE flag_z := 0 FI;
   IF ax < bx THEN flag_c := 1 ELSE flag_c := 0 FI;
   ax := ax - bx;
   CALL compare;
   CALL theend END
theend ==
   CALL Z END
ENDACTIONS
```
model small
.code
  mov ax,12
  mov bx,8
compare:
  cmp ax,bx
  je theend
  ja greater
  sub bx,ax
  jmp compare
greater:
  sub ax,bx
  jmp compare
theend:
  nop
diagram

GCD program – diagram
GCD – Remove flags

ACTIONS A_S_start:
A_S_start == ax := 12; bx := 8; CALL compare END
compare ==
  IF ax = bx
    THEN IF ax < bx THEN CALL theend ELSE
        CALL theend FI
    ELSE IF ax >= bx THEN CALL greater FI FI;
  bx := bx - ax;
  CALL compare;
  CALL greater END
greater ==
  ax := ax - bx; CALL compare; CALL theend END
theend == CALL Z END ENDACTIONS
GCD – **Collapse Action System**

\[
\begin{align*}
ax & := 12; \\
bx & := 8; \\
\text{DO IF } ax &= bx \text{ THEN} \\
& \quad \text{IF } ax < bx \text{ THEN EXIT(1) ELSE EXIT(1) FI} \\
& \quad \text{ELSE IF } ax \geq bx \text{ THEN } ax := ax - bx \text{ ELSE} \\
& \quad \quad bx := bx - ax \text{ FI} \\
\text{FI OD}
\end{align*}
\]
ax := 12;
bx := 8;
WHILE ax <> bx DO
    IF ax >= bx THEN
        ax := ax - bx
    ELSE
        bx := bx - ax
    FI
OD
Assembly version of the SumN procedure

sumn proc
; if n is on top of the stack
; sum the next n top elements of the stack
    pop cx
    mov bx, 0
    mov ax, 0
    mov dx, 0
theloop:
    pop ax    ; get next from stack
    cmp bx,cx ; is it the final one?
    je endp  ; skip to end if ti is
    add dx, ax ; array sum is in dx
    inc bx
    jmp theloop
endp:
    push dx    ; result
    ret
sumn endp

SumN – translated to an *Action system*

**ACTIONS**

A_S_start:

A_S_start ==
stack := < num1 > ++ stack;
stack := < num2 > ++ stack;
stack := < num3 > ++ stack;
stack := < n > ++ stack;
CALL sumn;
rez := HEAD(stack);
stack := TAIL(stack);
PRINT(rez);
CALL end1

end1 ==
SKIP;
CALL Z; SKIP END

sumn ==
**ACTIONS**
  dummysys: dummysys ==
  cx := HEAD(stack);
  stack := TAIL(stack);
  bx := 0;
  ax := 0;
  dx := 0;
  CALL theloop
  END
  theloop ==
  ax := HEAD(stack);
  stack := TAIL(stack);
  IF bx = cx THEN
    flag_z := 1
  ELSE
    flag_z := 0
  FI;
  ............... 
  bx := bx + 1;
  CALL theloop;
  CALL endp
  END
  endp ==
  stack := < dx > ++ stack;
  CALL Z;
  SKIP END
ENDACTIONS

ENDACTIONS;
Sumn – transformed program

VAR < ax := 0, bx := 0, cx := 0, dx := 0, stack := < > >:
VAR < rez := 0 >:
stack := <12> ++ stack;
stack := <8> ++ stack;
stack := <22> ++ stack;
stack := <3> ++ stack;

cx := HEAD(stack);
stack := TAIL(stack);
ax := HEAD(stack);
stack := TAIL(stack);
WHILE bx <> cx DO
  dx := ax + dx;
  IF dx >= 65536 THEN dx := dx MOD 65536 FI;
  bx := bx + 1;
  ax := HEAD(stack);
  stack := TAIL(stack) OD;
stack := <dx> ++ stack ENDVAR ENDVAR

num1, num2, num3, n were constants in the original program and are replaced adequately
Presentation organisation

1. Introduction
   - Software Evolution
   - WSL – Wide Spectrum Language

2. Our transformation process
   - Asm2wsl
   - Transf.wsl

3. Examples
   - Programs

4. Summary
   - Results and open questions
Summary

Translating assembly programs to WSL gives us options to:

- Generate call diagrams for easier understanding of original code;
- Automatically transform the code to much simpler versions;
- Optionally to manually tweak the results with more transformations

Automated transformations show more than 30% improvement of code in weighted Structure metric. Improvements are noted in other metrics as well.

A lot of space for improvements

- More options in the assembler translation system
- More automatic transformations
- Expanding existing transformations for specific examples
Thank you for your attention

Questions?