Transforming Assembly to WSL, a high-level language

Doni Pracner

Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad
Contents

- Introduction
- Software Evolution
- WSL
- Our transformation process
- Asm2wsl
- Examples
- Conclusion
Old software can be very problematic for maintenance

- Obsolete (or no) documentation
- Source code not available
- Old technologies
- Incompatible hardware, etc.

Our aim is to make old, low level, assembly code easier to understand, and hopefully restructure it.
Software Aging

- 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.

- The life cycle of software

- Reengineering has 3 phases:
  - Reverse engineering
  - Functional restructuring
  - Forward engineering

- Software Evolution is (largely) repeated reengineering.
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*
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

- 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 (problems)
Asm2wsI usage

Assembler to WSL converter. v 0.78, 2010, by Donny
usage:
asm2wsI {-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>Command</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov ax, dx</code></td>
<td><code>ax := dx</code></td>
</tr>
<tr>
<td><code>xchg ax, dx</code></td>
<td><code>&lt; ax := dx , dx := ax &gt;</code></td>
</tr>
</tbody>
</table>
| `add dx, ax`  | `overflow := 65536;
 dx := dx + ax ;
 IF dx >= overflow THEN
   dx := dx MOD overflow ;
   flag_o :=1; flag_c := 1;
 ELSE
   flag_o :=0; flag_c := 0;
FI ;` |
### Command translation (contd.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ah, n</td>
<td>t_e_m_p := n ; &lt;br&gt; ax := ( ax MOD 256) + t_e_m_p * 256;</td>
</tr>
<tr>
<td>add al, 12</td>
<td>overflow := 256; &lt;br&gt; t_e_m_p := ( ax MOD 255) + 12; &lt;br&gt; IF t_e_m_p &gt;= overflow THEN &lt;br&gt;   t_e_m_p := t_e_m_p MOD overflow ; &lt;br&gt;   flag_o :=1; flag_c := 1; &lt;br&gt; ELSE &lt;br&gt;   flag_o :=0; flag_c := 0; FI ; &lt;br&gt; ax := ( ax DIV 256) *256 + t_e_m_p ;</td>
</tr>
</tbody>
</table>
Possible solution for handling input and output:

<table>
<thead>
<tr>
<th>print_str x</th>
<th>PRINT(x);</th>
</tr>
</thead>
<tbody>
<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>
Transformation

- Collapse Action Systems
- Transform DO … OD loops
- Constant propagation
- Remove Redundant
Examples of translated programs

- GCD – greatest common divisor
- Array Sum – simple addition
- Factorial – artificial example, made to test the many features of the translator (arrays, stack, etc)
GCD - assembly

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
VAR < flag_z := 0, flag_c := 0 >:
END

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
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 END

GCD - remove flags
ax := 12;
bx := 8;
DO IF ax = bx
    THEN IF ax < bx THEN EXIT(1) ELSE EXIT(1) FI
    ELSE IF ax >= bx THEN ax := ax - bx ELSE
        bx := bx - ax FI
FI OD
ax := 12;
bx := 8;
WHILE ax <> bx DO
  IF ax >= bx THEN ax := ax - bx ELSE bx := bx - ax FI
OD
Generated with FME (Fermat Maintenance Environment)
.data
array     db  1,2,3,4,5,6,7,0
n        dw  7

.code
    mov dx, @data
    mov ds, dx
    mov bx, 0
    mov ax, 0
    mov dx, 0
mainloop:
    mov al, array[bx] ; read array member
    cmp bx, n         ; is it the n-th?
    je progend       ; if yes, go to end
    add dx, ax        ; the sum is in dx
    inc bx
    jmp mainloop
progend:
    nop
end
fl_flag1 := 0;
WHILE fl_flag1 = 0 DO
  IF bx = 7
    THEN fl_flag1 := 1
  ELSIF array[bx + 1] + dx >= 65536
    THEN dx := (array[bx + 1] + dx) MOD 65536;
        < bx := bx + 1, fl_flag1 := 0 >
  ELSE dx := array[bx + 1] + dx;
        < bx := bx + 1, fl_flag1 := 0 > FI OD
## Transformation results

<table>
<thead>
<tr>
<th>Metric</th>
<th>GCD Before</th>
<th>GCD After</th>
<th>%</th>
<th>Array Sum Before</th>
<th>Array Sum After</th>
<th>%</th>
<th>Factorial Before</th>
<th>Factorial After</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe</td>
<td>10</td>
<td>11</td>
<td>+10</td>
<td>6</td>
<td>7</td>
<td>+16</td>
<td>12</td>
<td>15</td>
<td>+25</td>
</tr>
<tr>
<td>Statements</td>
<td>52</td>
<td>41</td>
<td>-22</td>
<td>55</td>
<td>42</td>
<td>-24</td>
<td>99</td>
<td>77</td>
<td>-23</td>
</tr>
<tr>
<td>CFDF</td>
<td>82</td>
<td>48</td>
<td>-42</td>
<td>80</td>
<td>44</td>
<td>-45</td>
<td>128</td>
<td>82</td>
<td>-36</td>
</tr>
<tr>
<td>Nodes</td>
<td>302</td>
<td>218</td>
<td>-28</td>
<td>300</td>
<td>213</td>
<td>-29</td>
<td>504</td>
<td>395</td>
<td>-22</td>
</tr>
<tr>
<td>Structure</td>
<td>450</td>
<td>291</td>
<td>-36</td>
<td>483</td>
<td>337</td>
<td>-31</td>
<td>787</td>
<td>548</td>
<td>-31</td>
</tr>
</tbody>
</table>
Conclusion

- Interesting first results
  - Automated transformations show more than 30% improvement of code (weighted Structure metric)

- A lot of space for improvements
  - More options in the assembler translation system
  - More automatic transformations
  - Overall more examples
Thank you for your attention.

Questions?