Program Semantics and Analysis: Lab1

Thibault Raffaillac, 871109-A416

Overview of the program

In a first part, the compiler parses the source code and translates it to AM (Abstract Machine) code. This part was mostly already implemented in the Lab files, and thus does not need be explained.

In the second part, the compiler translates the AM code to actual assembly instructions. This part is achieved through two passes of the AM tree: the first one lists the variables used and gathers their global properties, the second one generates the instructions.

A Hashtable with the custom class Variable is used to enumerate the variables of the program and hold their properties. The class will be extended in the second Lab to hold the additional properties of the variables. Currently, the only property recorded is if the variable is first used initialized or not. This allows the compiler to add code to prompt the values of uninitialized variables at program beginning.

While writing, the compiler keeps track of the last label name used, to guarantee uniqueness of labels names in the assembly code generated. This required the creation of a mutable integer class, MutableInt, based on org.apache.commons.lang.mutable.MutableInt, to pass integers by reference.

Extension of the While language

In natural semantics: $\dfrac{\langle S_1, s\rangle \to s'}{\langle try\ S_1 catch\ S_2, s\rangle \to s'}[try^1_{NS}]$ $\dfrac{\langle S_1, s\rangle \to \bar{s}'\ \langle S_2, s'\rangle \to s''}{\langle try\ S_1 catch\ S_2, s\rangle \to s''}[try^2_{NS}]$

The AM instructions TRY(c_1, c_2) and CATCH(c_1) are added along with the abnormal machine state:

$\langle TRY(c_1, c_2), e, s\rangle \rhd \langle c_1 : CATCH(c_2), e, s\rangle$

$\langle CATCH(c_1) : c, e, \bar{s}\rangle \rhd \langle c_1 : c, e, s\rangle$

$\langle CATCH(c_1) : c, e, s\rangle \rhd \langle c, e, s\rangle$

Also, when given an abnormal state, the try statement and TRY instruction are skipped.

The transformation from Statement to code is straightforward:

$CS[\![try\ S_1 catch\ S_2]\!] = TRY(CS[\![S_1]\!], CS[\![S_2]\!])$

Derivation tree for trycatchsample.while (for convenience I noted s_n=s[x↦n] and skipped rules names):

$$\dfrac{\dfrac{\dfrac{\langle x:=7/x, s_0\rangle \to \bar{s}_0 \quad \langle x:=x+7, \bar{s}_0\rangle \to \bar{s}_0}{\langle x:=7/x; x:=x+7, s_0\rangle \to \bar{s}_0} \quad \langle x:=x-7, s_7\rangle \to s_0}{\langle S_1, s_7\rangle \to \bar{s}_0} \quad \langle x:=x-7, s_0\rangle \to s_{-7}}{\langle try\ S_1 catch\ S_2, s\rangle \to s_{-7}} \quad \langle x:=7, s\rangle \to s_7}{\langle x:=7; try\ S_1 catch\ S_2, s\rangle \to s_{-7}}$$

Execution of this program in AM language:

$$\begin{aligned}&\langle PUSH-7 : STORE-x : TRY(c_1 : c_2), e, s\rangle\\ &\rhd \langle TRY(c_1 : c_2), e, s[x \mapsto 7]\rangle\\ &\rhd \langle c_1 : CATCH(c_2), e, s[x \mapsto 7]\rangle\\ &\rhd \langle CATCH(c_2), e, \bar{s}[x \mapsto 0]\rangle\\ &\rhd \langle c_2, e, s[x \mapsto 0]\rangle\\ &\rhd \langle \varepsilon, e, s[x \mapsto -7]\rangle\end{aligned}$$

The semantics correctness is proving that $M[\![CS[\![S]\!]]\!](s)=s' \Leftrightarrow S_{NS}[\![S]\!](s)=s'$ holds for any Statement S and any States s and s'.

We first prove correctness for the try statement: Let us examine the effect of a TRY(c_1, c_2) instruction, given a normal initial state. Thanks to the first AM rule, it is directly replaced by c_1:CATCH(c_2). If c_1 ends in a normal state, the third AM rule holds and CATCH(c_2) is skipped. This results in only c_1 having been executed, which conforms to the [try^1_{NS}] rule. If c_1 had ended in an abnormal state, the second AM rule would hold, and c_2 would be executed given a normal state. This results in c_1 and c_2 being executed, as in the [try^2_{NS}] rule.

The second addition to the semantics correctness proof is the introduction of abnormal states. All statements are skipped under abnormal state, and all AM instructions except CATCH are NOPed, so they trivially verify the correctness proof.

Workflow

My first goal was a bit ambitious (and a bit crazy, but how fun!), it was to directly write machine instructions without the intermediate step of assembly code. Indeed, the opcodes are well documented in the x86 instructions list, and the PE COFF specification from Microsoft is clear and concise. In the mean time it turned out that my 32bits WinXP laptop broke down, so I decided to try the 64bits architecture and the ELF format, giving up machine instructions towards assembly instructions.

I first set a proper working environment: links and copies of the instruction set reference, AMD64 ABI (though after reading, it was ill explained), compiled nasm which has the simplest syntax of all assemblers, its manual, ld as linker, and links to a bunch of tutorials.

With this I first managed to write the assembly dumping for each AM instruction. A first test on gcd.while made an infinite loop. A function to display a number was needed. Rather than relying on glibc, the task was simple enough to be hand-coded and carried with system calls. I thus wrote the code to display a number, then the code to prompt a number, in the file samples/readNumber.asm

Since the While language had no read and write statements, there was no way to manually insert prompt and display code, so I made that uninitialized variables be prompted then displayed at program exit. All samples were modified to remove variables initializations.

I finally performed proper testing and corrected a few bugs.

I estimate having spent 5 to 6 full days on this lab, mostly on reading documentations, but the result was worth it!

Links

Here are the few valuable links I found during this Lab:

This tutorial is on 32bit architecture, thus not valid on system calls, but it was a good introduction: http://www.cin.ufpe.br/~if817/arquivos/asmtut/index.html

This tutorial is very light, but on 64bit architecture: http://vikaskumar.org/amd64/index.htm

Excellent document that sums the x86-64 ABI: http://web.cecs.pdx.edu/~apt/cs322/x86-64.pdf

Boolean arithmetic: http://webster.cs.ucr.edu/AoA/Windows/HTML/IntegerArithmetica2.html#999863

Syscall reference for x86-64: http://lxr.free-electrons.com/source/arch/x86/include/asm/unistd_64.h

Testing

The testing was carried with 4 programs, each testing 4 to 5 AM instructions:

- Test 1: FETCH, STORE, PUSH and ADD

```
i := plusOne;
i := i + 1;
plusOne := i
```

- Test 2: MULT, DIV, SUB, TRY and NOP

```
i := minusOne;
i := i - 1;
minusOne := i;
i := timesTwo;
i := i * 2;
timesTwo := i;
try
      i := divByThree;
      i := i / 3;
      divByThree := i
catch
      skip
```

- Test 3: BRANCH, LOOP, FALSE and TRUE

```
i := plusOne;
(if true then
      i := i + 1;
      (while false do
            i := i + 2)
else
      skip);
plusOne := i
```

- Test 4: AND, EQ, LE and NEG

```
i := plusFifteen;
(if 255 = 256 then
      skip
else
      i := i + 1);
(if !(0 = 1) then
      i := i + 2
else
      skip);
(if 255 <= 256 then
      i := i + 4
else
      skip);
(if !(1 <= 0) & 1 <= 2 then
      i := i + 8
else
      skip);
plusFifteen := i
```