3XB3 L3 - Assembly Lab

Luigi Quattrociocchi (quattrl@mcmaster.ca)
Dennis Fong (fongd1@mcmaster.ca)

Avenue Group: 42

December 8, 2022

F_1: Global Variables and First Visits

Manual Translation

The following is a translation of `simple.py` into Pep/9.

```
        BR      main
x:      .BLOCK 2
main:   LDWA 3, i
        ADDA 2, i
        STWA x, d
        DECO x, d
        .END
```

The following is a translation of `add_sub.py` into Pep/9. Note that there are multiple instances of `STWA` immediately followed by `LDWA`. These pairs are redundant and can be removed.

```
        BR      main
_UNIV:    .WORD 42
variable: .WORD 3
value:    .BLOCK 2
result:   .BLOCK 2
main:   DECI value, d
        LDWA value, d
        ADDA _UNIV, d
        STWA result, d
        LDWA result, d
        SUBA variable, d
        STWA result, d
        LDWA result, d
        SUBA 1, i
        STWA result, d
        DECO result, d
        .END
```

Global vs Local variables

In python, global variables are any variables that are declared at the top level of the program (not within any function). These variables are typically referred to as "static" variables, and they are stored within the data or bss sections of the binary. Local variables in python exist at the function scope level. These variables are stored on the call stack, and they exist until the function containing them returns and the scope is destroyed.

NOP1 Instructions

The translator emits `NOP1` instructions after the entry point label for seemingly no reason. However, this instruction is necessary to avoid syntax errors due to how Pep/9 works. Every label must have an instruction on the same line, otherwise the assembler gives the error `ERROR: Must have mnemonic or dot command after symbol definition.`. Since we can't guarantee that an instruction will follow the label (the souce file could be empty), we emit a dummy instruction regardless.

Explanation of Visitors and Generators

Visitors and Generators are two concepts used in compilers after the AST is parsed. Visitors (from the Visitor design pattern) are commonly used in traversals of recursive structures (such as trees) in which the behaviour of the visitation is polymorphic and independent of the structure itself. Most compilers (notably clang) use the visitor pattern to traverse their AST. Each node in the tree is a visitor which is responsible for walking the tree and visiting its children. After the visitors are used to traverse the syntax tree and accumulate the instructions that should be emitted, the generators are used to emit said instructions. Generators are useful because if implemented correctly, cross platform support can easily be achieved. In our case, the bytecode created by our visitors are already basically in Pep/9 format, so the generators emit them verbatim.

Limitations of Current Translation

Currently the translator is very simple, leading to a couple common software engineering pitfalls. Firstly, the code is not very scalable. Everything is dumped to stdout rather than a specific file descriptor, the visitors violate the single responsibility principle by directly interpolating the string instructions to be emitted, and and there is no room for adding optimization passes without violating the open-closed principle. Secondly (and probably less importantly), the code is emitted in a very naive way. The `NOP1` instruction after the entry label, constant still stored in memory (and are not actually constant), and the translation is very literal and error prone (for instance, the length of labels is not constrained).

F_2: Allocation, Constants, and Symbols

Improvements

The first improvement of initializing global static variables with their initial value (if the expression can be evaluated at compile time) has us change allocating with `.BLOCK` to `.WORD` in order to remove redundant load and store instructions. We check if we can perform this upon every visitation of any assignment node. To determine if a variable can be initialized to a constant value, the expression must only contain constant values or other variables that can be evaluated. This is called constant propagation, and we try to do this for every assignment node that is visited at the top level of the program. If successful, we can statically initialize the memory pointer to by the variable to be the resulting value. Specifically, we

emit .WORD x where x is the result of the expression, rather than .BLOCK 2 wherever this optimization succeeds.

The next improvement changes constant variables to be textually substituted (basically macros), rather than store them in memory. It makes sense to not allocate space for these variables only if we disallow modifying them. Similar to the previous improvement, this also relies on constant propagation. If we ever visit an assignment node in which the identifier node on the left of the equal sign starts with an underscore and is capitalized, then we can simplify the generated code by hardcoding the evaluated expression as the equated value. In the assembly we emit a .EQUATE instead of a .WORD or .BLOCK directive wherever this applies. Additionally, the addressing mode must become 'i' rather than 'd', since the label now refers to an immediate value and not an address.

The last improvement fixes the susceptibility of labels to clobbering each-other if any share prefixes that are at least 8 characters long. Instead of naively naming each label based on their corresponding variable or other naming scheme, we can map every variable name to a unique alphanumeric permutation (using a symbol table). In this way, we have at least 26^8 possible label names. Whenever code that uses a variable is generated, the symbol table maps from the variable's identifier to the label used in the generated code.

Integer Overflow

When attempting to store large integers in most programming languages, we must be mindful of the width of the integer type used. In Pep/9 For example, the word size is only 2 bytes (16 bits), which means that we can only store integers between -32768 (-2^{15}) and 32767 ($2^{15} - 1$) inclusively (we represent signed integers as two's complement). Integer overflow (or underflow) occurs when a computation would result in a number too large or small to fit within the integer width. For instance our Pep/9 program cannot compute factorials larger than 7 (5040), since the result of 8 factorial (40320) would overflow a 16-bit integer. In other "real" programming languages, integer overflow is handled in various different ways. The most natural way is to take the result modulo 2^n (equivalently, the result bitwise ANDed with $2^n - 1$) where n is the number of bits in the integer. This is because at a hardware level integers are (usually) represented as two's complement, and overflowing the width simply means discarding the overflowed bits. Other languages either promote the data type to one with a larger width, or throw exceptions.

F_3: Conditionals

Manual Translation

The following is a manual translation of `gcd.py`. Due to the nature of manual translation, some optimizations have been implemented such as removing redundant labels and jumps, not loading values if they are already in the register, and reusing the result of a single comparison instruction for multiple conditions.

```
        BR      main
a:      .BLOCK 2
b:      .BLOCK 2
main:   DECI a,d
        DECI b,d
while:  LDWA a,d
        CPWA b,d
        BREQ done
        BRLE else
        SUBA b,d
        STWA a,d
        BR while
else:   LDWA b,d
        SUBA a,d
        STWA b,d
        BR while
done:   DECO a,d
        .END
```

Automation of Conditional Translation

Implementing conditionals is very similar to implementing loops, which conveniently is already done for us. To implement all of these conditionals, all we need to do is implement a visitor for `if` statements (`else` statements don't have their own visitor since they are contained in the `if`). In terms of code generation, each if/else block adds two labels and two branches so that the control flow branches into two paths, then joins back into one. The translation will always look like the following:

```
        ; Evaluate condition (usually CPWA) and branch if false
        ; if a == b:
        LDWA a,d
        CPWA b,d
        BRNE else
        ; true branch
        BR fi
else:   NOP1
        ; false branch
fi:     NOP1  ; get here by jumping (if) or falling through (else)
```

Of course, the `NOP1` instructions are included for simplicity, but they could be omitted if optimized. Luckily, an `elif` statements is equivalent to an `else` statement with the `if` statement inside of it. For this reason, the python `ast` module doesn't even recognize `elif`s, which means we don't even have to consider this case.

F_4: Function Calls

Ranking of Translation Complexity

The following is our ranking of the programs in `_samples/4_function_calls` in terms of difficulty from easiest to hardest.

`call_void.py`
`call_param.py`
`call_return.py`
`fibonacci.py`
`factorial.py`
`fib_rec.py`
`factorial_rec.py`

Arguably the easiest program to translate is `call_void.py`, since it takes no parameters and has no return value. The only difficulty comes from the change in control flow associated with jumping and returning, as well as the local variable stored on the stack. Both `call_param.py` and `call_return.py` introduce parameters which are pushed onto the stack, and then returning a value by pushing to the stack or placing it in the register before returning. For these reasons they are slightly harder to implement. By this reasoning we think that `fibonacci.py` would have a similar difficulty to `call_return.py`. `factorial.py` has two functions, one of which has two parameters, further increasing the difficulty. Lastly, `fib_rec.py` and `factorial_rec.py` are the hardest because they are incorporate all of the concepts of the previous samples, but they introduce recursion. To ensure correctness in a recursive program the stack frames must not clobber each other, and overflowing the program stack becomes a consideration.

Manual Translation

The following is a translation of `call_param.py` into Pep/9.

```
1           BR program
2  _UNIV:   .EQUATE 42
3  x:       .BLOCK 2
4  value:   .EQUATE 2  ; parameter my_func #2d
5  result:  .EQUATE 0  ; local var my_func #2d
6  my_func: SUBSP 4,i  ; push #value, #result
7           LDWA value,s
8           ADDA _UNIV,i
9           STWA result,s
10          DECO result,s
11          ADDSP 4,i  ; pop #value, #result
12          RET
```

```
program: DECI x,d
         LDWA x,d
         STWA -4,s
         CALL my_func
         .END
```

The following is a translation of `call_return.py` into Pep/9.

```
BR program
_UNIV:   .EQUATE 42
x:       .BLOCK 2
result:  .BLOCK 2
value:   .EQUATE 2 ; parameter my_func #2d
lresult: .EQUATE 0 ; local var my_func #2d
my_func: SUBSP 4,i ; push #value, #lresult
         LDWA value,s
         ADDA _UNIV,i
         STWA lresult,s
         ADDSP 4,i ; pop #value, #lresult
         RET
program: DECI x,d
         LDWA x,d
         STWA -4,s
         CALL my_func
         STWA result,d
         DECO result,d
         .END
```

The following is a translation of `call_void.py` into Pep/9.

```
         BR program
_UNIV:   .EQUATE 42
value:   .EQUATE 2  ; local var my_func #2d
result:  .EQUATE 0  ; local var my_func #2d
my_func: SUBSP 4,i  ; push #value, #result
         DECI value,s
         LDWA value,s
         ADDA _UNIV,i
         STWA result,s
         DECO result,s
         ADDSP 4,i  ; pop #value, #result
         RET
program: CALL my_func
         .END
```

Automation of Functions

Generating the code for procedures and procedure calls requires steps taken on behalf of the callee and caller. Both need to ensure that they work together to pass values correctly and maintain the stack.

On the caller side, parameters should be placed on the stack (avoiding the space that the return address will be pushed) before calling. Importantly, we rely on the fact that RBS semantics only support call-by-value parameters. This means that every integer passed to a function is copied, rather than passing a pointer to mutable memory. If we were to support call-by-reference instead, the address of the memory would have be pushed onto the stack instead of the value itself. Furthermore, since we don't have to worry about pointers, accessing the value of the parameters becomes much simpler, since we never have to dereference any pointer (or do indirect memory accesses).

Also, the caller should use the return value should be used before it is overwritten after the function returns. To keep things simple, we opted to utilize the accumulator register for the return value of the function. It's fair to assume that RBS doesn't need to support multiple return values, since we only support integers (no tuples), and of course multiple assignment is out of the question for RBS.

On the callee side, the stack frame should be created (stack pointer moves) on entry and removed (stack pointer is reset) on return, so that local variables are allocated correctly. If another function were to be called and not enough space was made for the frame, then the local variables would be clobbered. Then, all memory accesses to local variables should be done in a "stack-relative" addressing mode, since the memory is on the stack rather than global.

Stack Overflow

If the stack ever becomes so large that it would exceed some maximum size, it could clobber memory used by other parts of the program. The Pep/9 virtual machine seems to be able recognize this and detect these cases of recursion, which allows it to terminate the running program. For instance, when you call a function in java and there is no more stack memory available in the JVM, then a StackOverflow exception is thrown. In C, the stack will normally collide with other memory, and if memory sanitizers are not enabled then this will usually result in crashing due to segmentation faults (trying to access virtual memory protected by the operating system).

A common way that programs can encounter stack overflows is when executing a recursive function with an incorrect (or no) base case. If the recursive depth of the function call became very large, it would blow up the stack, and we would encounter a case of stack overflow. Another way that the stack could overflow is if a stack frame has very large local variables. For example, if a function call needed to use an array of one billion elements, that array would take up too much space in the call stack, and the program would crash.

F_5: Arrays

Automation of Global Arrays

To implement arrays as global variables, we need to add support for visiting static array initializations (a list of 0 multiplied by an integer), and subscripting variables. In terms of generating, we need to statically allocate the arrays with `.BLOCK x` where x is the size of the array times two (size of the integer type). Also, we need to generate the instructions to load value within the subscript into the index register, and add indexed addressing modes to the subsequent load of the array.

Whenever an assignment is visited, we must determine if it is either an array initialization, array access, or a normal variable access. We can do this easily by checking if the identifier ends with an underscore (as indicated by the lab handout). These identifiers represent arrays, and we can treat them accordingly by allowing assignments which subscript the array, as well as emiting instructions for indexing and changing the addressing mode from 'd' to 'x'.

Automation of Local Arrays

The process of implementing local arrays is very similar to that of global arrays, however some things must change. Firstly, the addressing mode will be 'sx' instead of just 'x'. Since local arrays are on the stack, we should access them relative to the stack pointer. Additionally, the allocation of these arrays occurs by subtracting $2x$ bytes to the stack pointer for each array within a stack frame (if x is the size of the array). Other than this, the implementation is the same as the global array implementation, and we can rely on our existing code for this.

Dynamic Arrays

Without using a non-bounded array, we would have to implement our own tpe of "non-bounded" array. To do this, we start with an array of size n. Then, if the array is full, and we try to add another item k to it, we would create a new array of size 2n, copy each element of the first array into the new array, then insert k into the new array as well. To implement this, we use a structure that has an attribute capacity denoting the amount of space allocated in the list, size to denote the number of elements currently in the list, and a pointer that points to the elements of the array, stored elsewhere (usually the heap). We do this because it is impossible to resize an array where the elements are allocated on the stack. This way, there isn't a real limit to the size of the array (aside from the limitations of the Pep/9 virtual machine), since space will be allocated accordingly when we have a lot of items. In contrast, when using a fixed array, size cannot change and the memory allocated does not either.