COMP304 Tutorial 6
19/08/2016

1 Programming Languages

In assignment 3 you will be implementing a programming language, a virtual machine, a compiler, and an interpreter. We'll cover those concepts in this section.

When you write a program, compile it, then execute it, the process usually happens in a sequence of steps: parsing → static analysis → compiling → execution.

1. Parsing. This takes source code e.g. a `.java` file or a `.lhs` file and turns it into an abstract syntax tree (AST). The AST is a tree representation of the structure and control flow of the program.

In the Straight-line programming language we'll be working with, source code like this:

```
1 x := 5;
2 y := x + 2;
```

Would be represented by the following AST in `Straight.lhs`:

```
1 [Asgn x (Const 5), Asgn y (Bin Plus x (Const 2))]
```

In the assignment we do **not** bother with source code or parsing; you'll start working directly with the AST representation of programs.

2. Static Analysis. We take the AST and check various safety properties. For instance, that it is well-typed (you never add booleans and ints; you never use a variable before it is declared). The version of `Straight.lhs` only has the `Int` type and does not static checking; your goal in part 2 will be to enrich the language with more types, and ensure it is type-safe and declaration-safe.

3. Compilation. Once the AST has passed our static analysis we translate it into a program in a lower-level language. The target language is sometimes called "the object language".

In the case of Java, the object-language is Java bytecode. If you're programming in C on x86 architecture, the object-language will be x86 assembly. In our assignment the object-language will be our own stack-based language. Sometimes languages compile to another language, e.g. Python to C (via Cython).

4. Execution. When the program has been compiled into object code we may wish to execute it. Execution may happen directly at the CPU (if we've e.g. compiled to assembly) or it may happen by running our code through an interpreter for the object code. The former is how programs in C would be executed; the latter is how programs in Java would be executed on a Java virtual machine.

We will be writing an interpreter for our stack-based language. Our straight-line ASTs will be compiled into object code for our stack-based language, and then executed by our interpreter.

2 Stack-Based Language

A stack-based language is one in which the majority of logic, computation, and control-flow happens through the manipulation of values on a stack. This is how most assembly languages and virtual machine languages work (including x86 asm and Java bytecode).

There are usually other notions of memory, such as the heap or registers. In our language we split memory into the stack (short-term memory) and store (long-term memory). The stack is represented as a list of `Int`; when you do part 2 you will have to generalise this to a list of e.g. `StackVal`, where a `StackVal` is either a `Bool` or an `Int`. The store is represented as a list of pairs of variables and values.

The program of a Stack-based languages is a list of commands. There are four commands in `Straight.lhs`, which are as follows.

- `LoadI x`. Pushes the integer constant `x` onto the stack.
- `Load v`. Retrieves the value `y` associated with variable `v` in the store and pushes it onto the stack.
- `Store v`. Updates variable `v` to have the value of whatever is on top of the stack. The value on top of the stack is then popped.
- `BinOp op`. Performs the arithmetic operation `op` to the two values on top of the stack. The values are then popped, and the result of the operation is pushed onto the stack.

Let's take the stack $[1, 2, 3]$ and the store $[(\text{`v'}, 5)]$. Here are some examples of how each of the commands above would affect the stack and store (note that an operation might leave the stack or store unchanged).

- `LoadI 0` would produce stack $[0, 1, 2, 3]$ and store $[(\text{`v'}, 5)]$.
- `Load v` would produce stack $[5, 1, 2, 3]$ and store $[(\text{`v'}, 5)]$.
- `Store v` would produce stack $[2, 3]$ and store $[(\text{`v'}, 1)]$.
- `Store y` would produce stack $[2, 3]$ and store $[(\text{`v'}, 5), (\text{`y'}, 1)]$.
- `BinOp Plus` would produce stack $[3, 3]$ and store $[(\text{`v'}, 5)]$.

- `BinOp Minus` would produce stack $[-1, 3]$ and store $[(\text{`v'}, 5)]$.

If you run into a problem applying a command (e.g. you load a non-existent variable) then your stack program should crash.

3 Translating Expressions

The implementation in `Straight.lhs` already translates assignments and expressions. It's worth expounding a little on how that works.

An integer constant translates directly into a `LoadI` instruction. For example, the expression 5 in a straight-line program will be turned into the following stack program:

```
1 LoadI 5
```

To translate a compound expression, like `LHS + RHS`, we must first translate the subexpressions `LHS` and `RHS`, and then add a `BinOp Plus` instruction afterwards. For example, an expression like 3+2 in source code, which would have an AST like `Bin Plus (Const 3) (Const 2)`, should translate into:

```
1 LoadI 3
2 LoadI 2
3 BinOp Plus
```

If our expression is a variable, we turn that into the appropriate `Load` instruction. Because our description of how to translate expressions is recursive, it immediately works on more complicated expressions. An expression like `3 + 2 * x` in source code, which would have an AST like `Bin Plus (Const3) (Bin Times (Const 2) x)`, would translate into the following stack program:

```
1 LoadI 3
2 LoadI 2
3 Load x
4 BinOp Times
5 BinOp Plus
```

You should read through `Straight.lhs` and verify that expressions are translated in this way.

4 Translating Statements

A straight-line program is a list of statements. The only kind of statement in `Straight.lhs` is an assignment statement. A straight-line program consisting of a single assignment like this:

```
1 x := 3
```

Would have this AST: `[Asgn x (Const 3)]`. To translate a list of statements, we translate each into its stack-language commands, and then concatenate the results into a big list of all the commands which make up our program.

To translate the statement `Asgn x (Const 3)`, we must first translate the expression and then add a `Load x` command afterwards. Our program above would turn into the following stack code:

```
LoadI 3
Store x
```

If we had straight-line source code consisting of several assignments, like so:

```
x := 5
x := 2 * x
y := x + 1
```

The straight-line AST would be: [`Asgn x (Const 5)`, `Asgn x (Bin Times (Const 2) x)`, `Asgn y (Bin Plus x (Const 1))`]. It's stack code will end up being the following (note the comments would not actually be there, they are just added by me for clarity):

```
-- this is the code for x := 5
LoadI 5
Store x
-- this is the code for x := 2 * x
LoadI 2
Load x
BinOp Times
Store x
-- this is the code for y := x + 1
Load x
LoadI 1
BinOp Plus
Store y
```

Again, you should read `Straight.lhs` and make sure you understand how the translation of assignment statements works.

5 Jumps, LessThan, NoOp

In part 1 you are to add if-statements and while-loops to the straight-line language, and add code to translate those into stack code. At the moment the four commands of our stack-based language are not rich enough to be able to express control-flow structures like loops and conditionals. In computability terms, it's not yet Turing complete (it cannot express all programs).

The way we will make our language Turing complete is to add a `Jump` command (also called `Goto`). Clever use of `Jump` is all we need to translate loops and conditionals. `Jump i` specifies that `i` should be the next command to be executed, where `i` an index/line number into the code.

Usually we have two kinds of `Jump`; one is conditional (depending on the value on top of the stack) and the other jumps no matter what. We actually only need conditional jumps, but for the sake of human readability we'll define both commands Because we need to compare integers for part 1, we'll add the binary operation `LessThan`.

- `Jump i`. Jump to line i in the commands.
- `JumpEqZ i`. If the value on top of the stack is equal to 0, jump to line i. Otherwise do nothing and move to the next line.

- `LessThan`. If the first value on the stack is less than the second value on the stack, push a 1 on the stack and throw the two values away. Otherwise push a 0 on the stack and throw the two values away.

In this way we're using `Int` values to simulate booleans, C-style. 1 is `True` and 0 is `False` (usually, any non-zero is considered `True`). In part 2 you'll be adding a true boolean type. It's up to you if you want to change how your `Jump` commands to work only on `Bool` values on the stack.

The last command we need is `NoOp`. We'll see why in a moment. `NoOp` does absolutely nothing!

6 Translating If-Statements

Imagine we had source code like this:

```
1 if (0 < 1) then { x := 1 } else { x := 0 }
```

I'm not telling you what the AST will look like (your goal is to come up with something appropriate), but the stack code should look something like this:

```
1  LoadI 0
2  LoadI 1
3  BinOp LessThan
4  JumpEqZ 8 -- if false jump to false branch
5  LoadI 1
6  Store x
7  Jump 10 -- jump to end of conditional
8  LoadI 0
9  Store x
10 NoOp
```

Lines $1-3$ are performing the $0 < 1$ check. After this, a value 1 or 0 will be on the stack. Line 4 checks that value. If it is 0/`False` then we jump to the false branch, which is lines $8-9$. Then we fall through to the end of the code (marked by `NoOp` on line 10). If the conditional jump fails, control falls through to the true branch on lines $5-7$. The jump on line 7 takes us to the end of the conditional; if we didn't have it, after executing the true branch our code would fall through to the false branch!

This is also the reason why we need a `NoOp`. If we only had lines $1-9$, and this if-statement was the last thing in our program, there'd be no line 10 to jump to. Line 7 would then be a runtime error. `NoOp` is needed to pad out the program.

Observation: since $0 < 1$ is always true we could generate more efficient code by replacing this entire expression with the false branch (since that is the branch that will always get executed). Doing this is fine, but complicates the logic of translation and not expected for the duration of the assignment.

The general strategy for translating a conditional:

1. Translate the condition

2. If result is 0, jump to false branch
3. Translate true branch
4. Add a jump to exit
5. Translate false branch
6. Add a NoOp (to signify exit)

In part 1 you're going to have to add these extra commands, implement their run-time behaviour, add an AST representation for loops and conditionals, and implement the translation of loops and conditionals into stack commands. Translation of while loops is done in a similar way to translation of if-statements (hint: you're allowed to jump backwards, not just forwards).

7 Symbolic Jumps

There's a book-keeping issue in translating conditionals: we don't know in advance how big the code for each branch will be. Therefore we aren't going to know exactly what line to jump to.

The solution for this is to allow for symbolic labels in code, and jumps to symbolic labels. This requires three extra commands.

- `Label x`. Does nothing when executed.
- `LJump l`. Jump to label `l`.
- `LJumpEqZ l`. Jump to label `l`, if the value on top of the stack is 0. Otherwise move to next line.

To solve the problem of not knowing the destination for jumps, programs are translated in two-stages. In the first stage we use symbolic jumps and labels. Once we've generated the code with symbolic jumps, we know the size of the program. So we go through and replace each `LJump` with a regular `Jump`; each `LJumpEqZ` with a regular `JumpEqZ`; and each `Label` with a `NoOp` (you need to keep the `NoOp` instructions in to keep the length of the program the same).

What this means is the final code will have **no symbolic jumps**. Symbolic jumps exist only to help us during the translation stage. By the end they're all converted to regular jumps and no-ops.

8 Translating If-Statements, Revisited

Recall the program from before.

```
1 if (0 < 1) then { x := 1 } else { x := 0 }
```

On our first pass through the AST, we would translate the if-statement into the below stack-code:

```
1  LoadI 0
2  LoadI 1
3  BinOp LessThan
4  LJumpEQ false_branch
5  LoadI 1
6  Store x
7  LJump exit
8  Label false_branch
9  LoadI 0
10 Store x
11 Label exit
```

After generating code for the whole program, we would then go through and replace each `LJump` with the appropriate `Jump`, and replace each `Label` with a `NoOp`. The code above would be turned into the following final program.

```
1  LoadI 0
2  LoadI 1
3  BinOp LessThan
4  JumpEqZ 8
5  LoadI 1
6  Store x
7  Jump 11
8  NoOp
9  LoadI 0
10 Store x
11 NoOp
```

If we used `false__label` and `exit` for every if-statement we'd run into a problem when turning symbolic jumps into regular jumps. Each if-statement needs to use a new pair of labels. We can this by using numbers to identify labels. During neighbours we keep track of which numbers are being used, perhaps with a counter. When we come to generate code for an if-statement, we generate two new labels for our own use (and increment the counter twice).

The new strategy for generating code is as follows:

1. Translate all statements using symbolic jumps.
2. Replace all `LJump`s with regular `Jump`s; replace all `Label`s with `NoOp`s.

The new strategy for generating if-statements is as follows:

1. Generate two fresh labels `i` and `j` and increment label counter by 2
2. Translate the condition
3. Add an `LJumpEqZ i` instruction
4. Translate true branch

5. Add an `LJump j` instruction
6. Add a `Label i` instruction
7. Translate false branch
8. Add a `Label j` instruction

9 Case Expressions

The logic for executing `BinOp` operations given to you in `Straight.lhs` is as below.

```
exec' (BinOp op) (x:y:stack, store) = (z:stack, store)
    where z = apply op x y

apply :: Op -> Int -> Int -> Int
apply Plus x y = x+y
apply Minus x y = x-y
apply Times x y = x*y
apply Div x y = x `div` y
```

An alternative way to implement this is using a `case` expression.

```
exec' (BinOp op) (x:y:stack, store) = (z:stack, store)
    where z = case op of
                Plus → x + y
                Minus → x - y
                Times → x * y
                Div → x `div` y
```

This lets you pattern-match in the middle of an expression. E.g.

```
let x = doSomething arg1 arg1
 in case x of
       firstPattern → firstAction
       secondPattern → secondAction
```

If the runtime cannot match an argument to any of the given patterns an error is thrown.