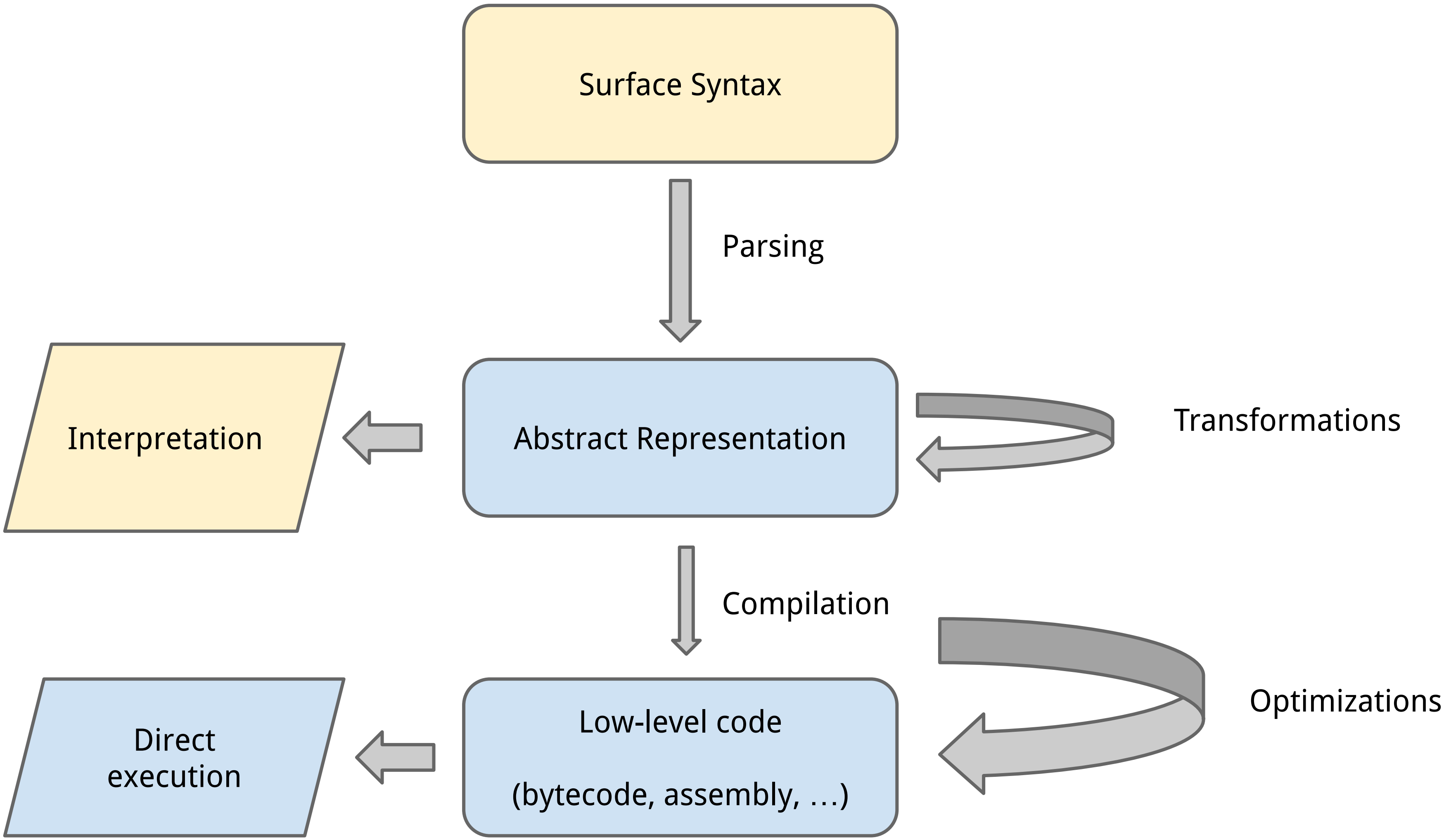
Compilation (II)

April 23, 2020

Riccardo Pucella

Compilation



Last time

We created a simple stack-based VM

We wrote a compiler to transform abstract representation into code for this VM

Today:

- We simplify the VM to try to make it easily implementable in a language that can be executed quickly

Approach

Compiler:

- Implement tail recursion
- Remove lookup by name
- Make all instructions take their args on the stack

VM:

- Use only 1 type of value (integer)
- Re-implement in C
- Need memory allocation/management

1 - Implement tail recursion

Recall what we determined when we looked at recursion:

- in our evaluator, when the last step of evaluation is simply to evaluate another expression, it needlessly grows the Scala call stack
- **Solution**: instead of making a call to evaluate, we should just "continue as" the new expression

- we have exactly that problem in our compiler

Tail recursion in our compiler

If the body of function A is a function call to B:

- we push the current environment, push the return address for B, put the arguments on the stack, and jump to B's closure. When B comes back, we pop A's return address and jump to it.
- if we don't push the return address for B and just jump to B, when B returns it also returns from A.

When expression is in non-tail position

- Use method compile()

When expression is in tail position

- Use new method compileTail()

Compiling applications

C[`EApply(f, [e1, e2, ...])`] ⇒

```
// in non-tail position
   PUSH-ENV
   PUSH-ADDR(@return)
   ...
   C[ e2 ]
   C[ e1 ]
   C[ f ]
   OPEN
   JUMP
@return:
   SWAP
   ENV
```

```
// in tail position
   ...
   C[ e2 ]
   C[ e1 ]
   C[ f ]
   OPEN
   JUMP
```

2 - Remove name lookups

Useful property of static (lexical) scoping:

You can predict the shape of the environment at any point without executing the code

Shape of the environment:

- list of environment bindings with the identifier name
- **not** the value associated with each identifier

Example

```
(let ((a 10))
  (let ((b 20))
    ((fun (c) (+ a (* b c))) 100)))
```

When you evaluate `a` in the body of `fun` the environment must look like:

```
[ c → ?, b → ?, a → ? ]
```

When evaluating, instead of doing a lookup by name, we can do a lookup by position in the environment

Static analysis

This is called **static analysis**

- take a pass over the code and compute information that will be useful for interpretation or compilation
- it is a whole field of research to define static analyses for various programming languages
  - e.g. discover dead code that can never be reached
- you can think of type checking as a simple form of static analysis

Method analyzeIds()

Exp method `analyzeIds()` associates with every identifier expression the index of the identifier in the environment where that identifier will be evaluated

For other expressions, it just visits every node in the abstract representation, passing the structure of the environment

New Env methods

```
class Env[A] (val content: List[(String, A)]) {
  ...

  def findIndex (id:String) : Int = {
    var idx = 0
    for (entry <- content) {
      if (entry._1 == id)
        return idx
      idx = idx + 1
    }
    runtimeError("unknown identifier " + id)
  }

  def lookupByIndex (idx: Int): A = content(idx)._2
}
```

EId

```
class EId (val id: String) extends Exp {
  ...

  var index = -1

  def analyzeIds (env: Env[Unit]): Unit =
    index = env.findIndex(id)
}
```

EFunction

```
class EFunction (val recName: String,
                 val params: List[String], val body: Exp)
                                                extends Exp {
  ...

  def analyzeIds (env: Env[Unit]): Unit = {
    var newEnv = env.push(recName, ())
    for (p <- params)
      newEnv = newEnv.push(p, ())
    body.analyzeIds(newEnv)
  }
}
```

Change instructions of VM

In the VM:

- LOOKUP takes an integer and not a string
- ADD-ENV doesn't need to pass the name of what it adds to the environment
- ADD-ENV obviously needs to work in the same order as the environment construction during analyzeIds()

In the compiler:

- EId.compile() generates LOOKUP(index)

3 - Simplify VM instructions

Change instructions to take their "arguments" from the stack

- ADD-ENV
- LOOKUP
- PRIM-CALL
- Idea : want to represent instructions by a single integer

We treat PUSH specially

- represented by two integers
- first is push code
- second is the integer to push

Don't distinguish addresses from integers

Opcodes

STOP	stop and return value on top of stack
PUSH(i)	push i on the stack
PUSH-ENV	push ENV on the stack
JUMP	pop a and jump to a
JUMP-TRUE	pop a and v and jump to a if v <> 0
CLOSURE	pop a and push closure (a, ENV)
OPEN	pop (a, e), set ENV = e, push a
ENV	pop e, set ENV = e
ADD-ENV	pop v, add v to front of ENV
LOOKUP	pop i, lookup v = ENV(i), push v
PRIM-CALL	pop op, pop args, call op, push result
NOP	do nothing
SWAP	swap top two values on the stack

Compiling rules

C[`EInteger(n)`] ⇒

```
PUSH(n)
```

C[`EBoolean(b)`] ⇒

```
PUSH(1) if b = true
PUSH(0) if b = false
```

C[`EId(idx)`] ⇒

```
PUSH(idx)
LOOKUP
```

Compiling rules

C[`EIf(c, t, e)`] ⇒

```
  C[ c ]
  PUSH(@then)
  JUMP-TRUE
  C[ e ]
  PUSH(@done)
  JUMP
@then:
  C[ t ]
@done:
  NOP
```

Compiling functions

C[`EFunction(self, [p1, p2, ...], body)`] ⇒

```
  PUSH(@fun)
  CLOSURE
  PUSH(@after)
  JUMP
@fun:
  PUSH(@fun)
  CLOSURE
  ADD-ENV
  ADD-ENV
  ADD-ENV
  ...
  C[ body ]
  SWAP
  ENV
  SWAP
  JUMP
@after:
  NOP
```

Compiling applications

C[`EApply(f, [e1, e2, ...])`] ⇒

```
  PUSH(@return)
  PUSH-ENV
  ...
  C[ e2 ]
  C[ e1 ]
  C[ f ]
  OPEN
  JUMP
@return:
  NOP
```

This is the non-tail call version

4 - New VM implementation

- Program code = array of integers (opcodes)
- All values are **integers**
- Stack = array of integers
- Environments:
  - an env is a pair (value, rest-of-env) [think linked list]
  - allocated in an env array
  - env value = integer (index into env array)
- Closures:
  - A closure is a pair (addr, env)
  - addr = integer, env = integer
  - allocated in a closure array
  - closure value = integer (index into closure array)
- Primitives: index into an array of primitives

Sample code

- VM in Scala
  - running directly on the compiled code in memory

- VM in C
  - running by reading file produced by `compileFile()`