TDDA69 Data and Program Structure
Running natively and JIT

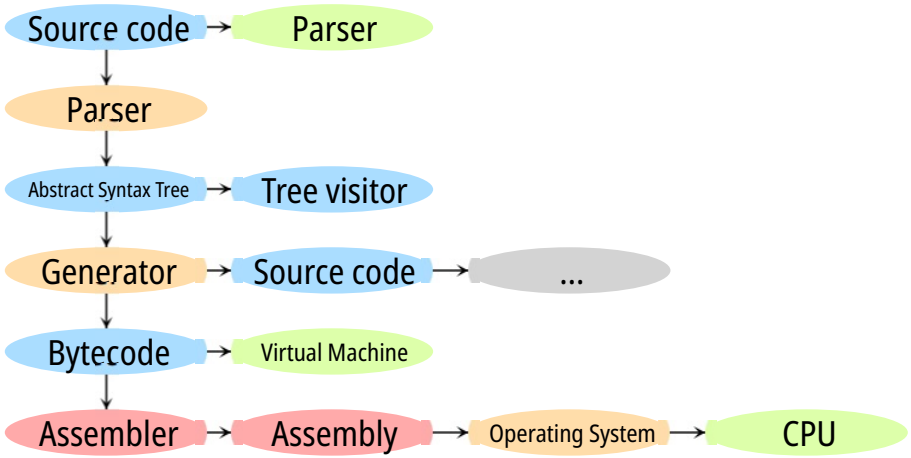
Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

1 Introduction
2 Concepts and models of programming languages
3 Declarative Computation Model
4 Declarative Programming Techniques
5 Declarative Computation Implementation
6 Declarative Concurrency
7 Message Passing Concurrency
8 Explicit State and Imperative Model
9 Imperative Programming Techniques
10 Imperative Programming Implementation
11 Shared-State Concurrency
12 Relational Programming
13 Specialized Computation Models
14 Macro
15 **Running natively and JIT**
16 Garbage Collection
17 Summary

How is a program interpreted?



Lecture content

- Type system
- Native Code Generation
  - Just-In-Time, Ahead-Of-Time
  - From Bytecode to Native code
  - V8 Case Study
  - Tracing JIT

Type system

What is a type system?

- A *type system* defines how a type is associated to a variable
- Variables are stored as bits in memory
  - 1011101010011100011...
- A type give meaning to a set of bits
- In a program, a value is associated to at least one type
  - C has void, assembly has no type

Type-Checking

- Static-type checking
  - Types are checked at compilation from static analysis
  - Usually variables have a single type
  - Polymorphism allows for *dynamicity*

```
void f(Base* b); f(new Child);
```

  - Support from downcasting

```
Child* c = static_cast<Child*>(base);
```

- Dynamic-type
  - Types are checked at runtime
  - Usually the type of a variable can change
- Gradual
  - Allow both to specify and not specify the type

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

- Unchecked
  - Example: machine code

Static-type checking

- Allow for optimization
  - No need to check for types at
- Program verification
  - Can work better than unit-

```
if(almost always true) { /* valid code */ }
else { /* type invalid code */ }
```

  - But downcasting cannot be verified with static

```
Child* c = dynamic_cast<Child*>(base);
```

Static-type vs Dynamic-type

- Trade-off
- The number of actual errors found through static typing is debatable
  - Easier polymorphism
- Dynamic typing allows for faster development and faster compilation

Implicit static-type system (1/2)

- C++, Java are explicit

```
int a = 0;
```

- Crystal is implicit

```
def shout(x)
  # Notice that both Int32 and String respond_to
`to_s`
  x.to_s.upcase
end

foo = ENV['FOO']? || 10

typeof(foo) # => (Int32 | String)
typeof(shout(foo)) # => String
```


Implicit static-type system (2/2)

```
def shout(x)
  # This fails becauses x can be Int32
  x + 'is shouting!'
end

foo = ENV['FOO']? || 10

typeof(foo) # => (Int32 | String)
typeof(shout(foo)) # => Error
```

- This approach mix benefits of dynamic and static checking

Gradual-type checking

```
def greeting(name):
    return 'Hello ' + name
greeting(10)
greeting('Cyrille')
def greeting(name: str) -> str:
    return 'Hello ' + name
greeting(10)
greeting('Cyrille')
```

Implementing static-typing

- With an AST visitor!
- Instead of returning a value, return a type

```
class StaticTypeCheckerVisitor:
  def visitValue(self, node):
    return node.type
  def checkArithmetic(self, typeA, typeB):
    if(typeA == typeB):
      return typeA
    elif(typeA.canConvert(typeB)):
      return typeB
    elif(typeB.canConvert(typeA)):
      return typeA
    else:
      throw CompilationError('Invalid types')
  def visitAddition(self, node):
    return self.checkArithmetic(node.a.accept(self), node.b.accept(self)
  def visitAssignment(self, node):
    typeVariable = environment.getVariable(node.variableName).type
    typeValue = node.value.accept(self)
    if(typeVariable != typeValue
       and !typeValue.canConvert(typeVariable)):
      throw CompilationError('Invalid types')
    return typeVariable
```


Implementing dynamic-typing

```
class EvalVisitor:
  def doArithmetic(self, op, node):
    a = node.a.accept(self)
    b = node.b.accept(self)
    if(a.type == b.type):
      return op(a, b)
    elif(a.type.canConvert(b.type)):
      return op(a.convert(b.type), b)
    elif(b.type.canConvert(a.type)):
      return op(a, b.convert(a.type))
    else:
      throw ExecutionError('Invalid types')
  def visitAddition(node):
    return self.doArithmetic(operators.add, node)
```


Strong vs Weak typing

- A language is said to be *strongly typed* when it requires explicitely casting
  - Example: ADA
- A language is said to be *weakly typed* when it allows non-explicit casting
  - For instance, from strings to integers to floating points to strings
  - Example: TCL

Native Code Generation

Performance of Virtual Machines

- Virtual Machines are faster than tree visitors interpreter...
- ...but still slower than native code
  - C++ is an average 15 times faster than Python
  - C++ is an average 5 times faster than JavaScript V8
  - Source: http://benchmarksgame.alioth.debian.org/

Just-In-Time, Ahead-Of-Time

Just-In-Time, Ahead-Of-Time

- Ahead-Of-Time
  - *classical* type of compilation
  - Code is translated to native code **before** running the application
  - GCC, VisualStudio C++, ART...
- Just-In-Time (JIT)
  - Code is translated to native code **while** running the application
  - Java VM, .NET, Dalvik...

Ahead-Of-Time

- Compiled before distribution
  - Simple
  - Examples: GCC, Visual Studio...
- Compiled at installation
  - The program is optimised for the platform
  - More complex installation
  - Examples: ART...

Just-In-Time

- Dynamic recompilation
  - The Virtual Machine recompiles part of the program *during execution*
- Adaptive optimization
  - Optimization depending on the current context
  - Profile-guided optimization
- For dynamic programming language, it can optimize for the different types

From Bytecode to Native code

Adaptive optimization

```
class A
{
  function a() { ... }
  function b() { ... a() ...}
}
class B : A
{
}
```

a() can be inlined in

Now we load:

```
class C : A
{
  function a() { ... }
}
```

a() cannot be inlined in

Stack Machine to register machine

- Use the registers as local cache
- Example:
  - Lets translate the following on a three registers machine:
  - One register contains a pointer to the stack, the other two are for arithmetic operations

PUSH 1	→ SET_A 1	[1] ≍ [A]
PUSH 2	→ SET_B 2	[1 2] ≍ [A B]
ADD	→ ADD_A_B	[3] ≍ [A]
PUSH 4	→ SET_B 4	[3 4] ≍ [A B]
PUSH5	→ MEM_SET_C_A; INC_C; SET_A_5	[3 4 5] ≍ [C[0] B A]
MUL	→ MUL_B_A	[3 20] ≍ [C[0] B]
DIV	→ DEC_C; MEM_GET_A_C; DIV_B_A	[0] ≍ [B]

Native code for static typing

- Generating native code for static typing is straight forward
- Which function to call is known

```
int a = 2;
f(a);
```

- Which operators to call is known

```
float b = 2.0;
float c = 2.0;
f(b+c);
```


Native code for dynamic typing (1/3)

- Consider:

```
function f(a)
{
  console.log('Value: ' +
a)
}
```

- What is the type of a?

Native code for dynamic typing (2/3

- Treat all variables as a pointer to an object

```
function
f(a,b)
{
  return a +
b;
}
```

```
void f(Object* a, Object*
b)
{
  Object* ret = nullptr;
  if(a->type != b->type)
cast(&a, &b);
  switch(a->type)
  {
    case Integer:
      ret = new Object(a-
>asInteger() + b-
>asInteger());
    ...
  }
}
```


Native code for dynamic typing (3/3)

- Treat all variables as a pointer to an object
- Treat all variables as 32/64bits integers
  - If the first bit is 1, then the value is an object
  - If the first bit is 0, then the value is a 31/63bits integer
- Dynamic recompilation, compile a version of function for each types when needed

V8 Case Study

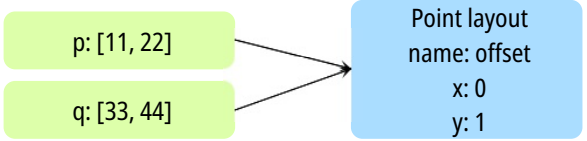
V8 Case Study

- V8 is the JavaScript engine used in Google Chrome and node.js
- Implements a JIT compiler

Object Representation in V8 (1/2)

- Does not use a dictionnary!
- Instead create a *hidden* class definition

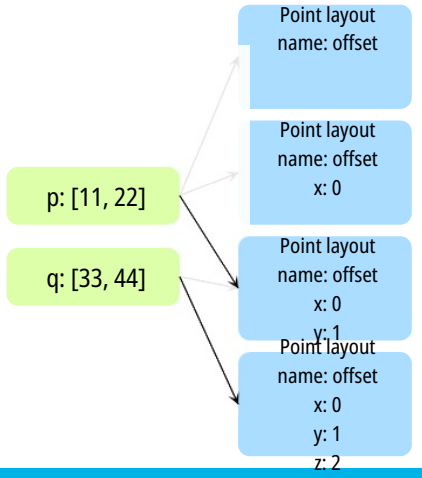
```
function Point(x, y)
{
  this.x = x;
  this.y = y;
}
var p = new Point(11,22);
var q = new Point(33,44);
```



Object Representation in V8 (1/2)

- What happen when adding a dynamic member?

```
function Point(x, y)
{
  this.x = x;
  this.y = y;
}
var p = new
Point(11,22);
var q = new
Point(33,44);
q.z = 55
```



Code generation

- Generate one native function per parameter type

```
var p = new Point(11,22);
var q = new Point(33,44);
function norm(p)
{
  return math.sqrt(p.x*p.x + p.y*p.y)
}
norm(p)
norm(q)
q.z = 55
norm(q)
```

- Will generate one native function for the two layouts
- In this case the compiler can easily guess the types of the variables based on the arguments

The problem of global variables

```
var base = 10;
function doSomething(val)
{
  doSomethingElse()
  return base + val;
}
```

- What happen if doSomethingElse() change the type of base?
- An other difficulty caused by side effects

Implementing JIT in a programming language

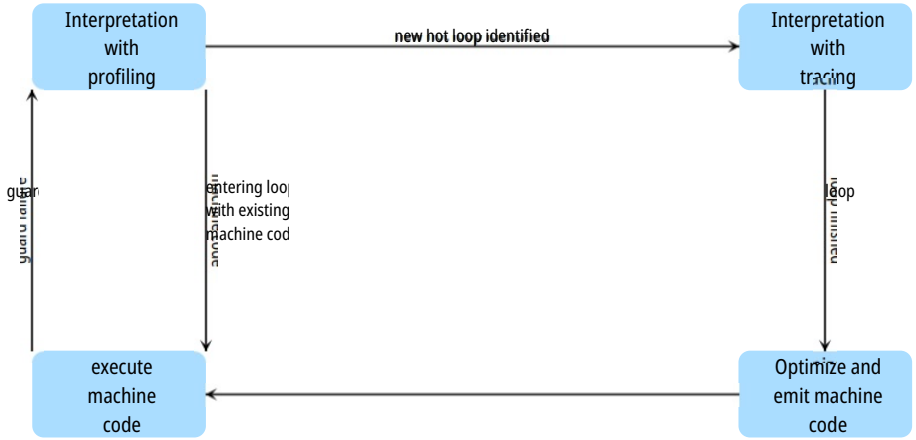
- Implementing a JIT is hard, it requires lot of low-level knowledge
- Solution, use an exist VM with JIT support, like JVM or CIL?
  - But it is generally slower than no-JIT! Source: https://pybenchmarks.org/
- Pypy propose to use meta-tracing JIT instead to build JITable VM in RPython

Tracing JIT

Tracing JIT

- Traditional JIT are *method JIT*, full methods are JITed
- *Tracing JIT* hot loops are identified and JITed
  - Remove control structure, inline function...
  - Output a linear set of instructions (no branching)
  - Add guard to check if the trace is valid

Tracing JIT - Execution Flow



Tracing JIT: example (1/2

- User Program

```
if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3
```

- Trace when x is set to 6

```
guard_type(x, int)
guard_not_less_than(x
0)
guard_type(x, int)
x = int_add(x, 2)
guard_type(x, int)
x = int_add(x, 3)
```


Tracing JIT: example (2/2

- User Program

```
if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3
```

- Optimised Trace

```
guard_type(x,
int)
guard_not_less_t
0)
x = int_add(x,
5)
```

Tracing JIT for a bytecode interpreter (1 /

- We have a hot loop:

```
def interpret(bytecode, a):
  regs = [0] * 256
  pc = 0
  while True:
    opcode = ord(bytecode[pc])
    pc += 1
    if opcode == JUMP_IF_A:
      target = ord(bytecode[pc])
      pc += 1
      if a:
        pc = target
    elif opcode == MOV_A_R:
      n = ord(bytecode[pc])
      pc += 1
      regs[n] = a
```

```
    elif opcode == MOV_R_A:
        n = ord(bytecode[pc])
        pc += 1
        a = regs[n]
    elif opcode ==
ADD_R_TO_A:
        n = ord(bytecode[pc])
        pc += 1
        a += regs[n]
    elif opcode == DECR_A:
        a -= 1
    elif opcode == RETURN_A:
        return a
```


Tracing JIT for a bytecode interpreter (2 /

- Example of program

```
MOV_A_R   0  # i = a
MOV_A_R   1  # copy of 'a'
# 4:
MOV_R_A   0  # i--
DECR_A
MOV_A_R   0
MOV_R_A   2  # res += a
ADD_R_TO_A 1
MOV_A_R   2
MOV_R_A   0  # if i!=0: goto 4
JUMP_IF_A 4
MOV_R_A   2  # return res
RETURN_A
```

- Example of trace:

```
opcode = bytecode[pc]
pc = int_add(pc, 1)
guard_equal(opcode, MOV_A_R)
n = bytecode[pc]
pc = int_add(pc, 1)
regs[n] = a
```

- What we want is a trace of the executed bytecode, not of the interpreter!
- The guards are most likely to fail after each instrucion

Meta-tracing JIT

- Pypy and RPython use a Meta-tracing JIT

Pypy/RPython JIT

```
tlrjitdriver = JitDriver(greens
= ['pc', 'bytecode'],
          reds = ['a', 'regs'])
def interpret(bytecode, a):
  regs = [0] * 256
  pc = 0
  while True:
    tlrjitdriver.jit_merge_point()
    opcode = ord(bytecode[pc])
    pc += 1
    if opcode == JUMP_IF_A:
      target = ord(bytecode[pc])
      pc += 1
      if a:
        if target < pc:
          tlrjitdriver.can_enter_jit()
        pc = target
    elif opcode == MOV_A_R:
    ... # rest unmodified
```

- *greens=...* indicates state of the interpreter
- *reds=...* indicates state of the user program
- *jit_merge_point()* indicates where to start when the guards fail
- *can_enter_jit()* indicates where JIT can be emited

Advantages/Inconvenients of Ahead-Of-Time

- Advantages
  - Program is ready to use after installation
- Inconvenients
  - Static optimization give slower code

Advantages/Inconvenients of Just-In-Time

- Advantages
  - Adaptive optimization
  - More suitable for dynamic programming language
- Inconvenients
  - Slow down during execution due to compilation
  - Debetable if the adaptive optimization gives a noticeable performance improvments for static typed programming language

Native Code Generation - Summary

- Benefits:
  - Faster to execute
- Inconvenients
  - Complexity
  - Less portable