Implementação de Linguagens (CC4023)

Language Implementation

Pedro Vasconcelos Luis Lopes

DCC/FCUP

February 16, 2024

Objectives

- Introduction to the implemention of functional and object-oriented languages
- Focus on interpretation and compilation techniques using virtual machines

Pre-requisites

- Knowledge of a FP language (Haskell, ML, OCaml, F#)
- Knowledge of a OO language (Java, C#)
- Knowledge of the C language (including dynamic memory and pointers)
- Principles of conventional compilers:
  - lexical analysis
  - syntatical analysis (*parsing*)
  - abstract syntax trees
  - code generation

Schelude

- Two modules:

  FP languages Pedro Vasconcelos (until 11th April)
  OO languages Luis Lopes (from 18th April)
- Two lectures per week: T 1.5 h + TP 2h

Assessment

Two practical projects:

- each graded for 4 marks out of 20

Final exam:

- 12 marks out of 20
- Minimal 40% mark in the exam

Principal bibliography

- *Foundations for Functional Programming*, Lawrence C. Paulson, `http://www.cl.cam.ac.uk/~lp15/`
- *The Implementation of Functional Languages*, Simon Peyton Jones,
- The Garbage Collection Handbook: The Art of Automatic Memory Management Richard Jones, Antony Hosking, Eliot Moss

Functional Programming

- Imperative programming: express computation by execution of instructions that mutate state
- Funcional programming: express computation as the application of functions to immutable values
- Functional languages support and encourge programming in a functional style
- Examples: Scheme, ML, O'Caml, F#, Haskell, Scala

Statically Typed Functional Languages

- Functions are first class, i.e. may be:
  - Passed as arguments
  - Returned as results from functions
  - Stored in data stuctures
- Algebraic Data Types
  - Enumerations (sums), tuples (products), recursion
  - Can be used to describe lists, trees, etc...
  - Decompositions using pattern matching
  - Automatic memory management
- Static type system
  - Every program fragment has a type assigned at compile-time
  - Type checking and inference
- Suport for programming in the large:
  - Modules (ML, O'Caml)
  - Type classes (Haskell)
  - Objects (O'Caml, F#, Scala)

State of the art

- Lots of academic research over the last 30 years, both theory and implementations
- Industrial-strength compilers (GHC, OCaml)
  - Performance similar to Java with a JIT [1]
  - Last 10 years: improvements in tooling (e.g. IDEs, packaging, build systems)
- Small but high-profile industrial user base
  - Jane Street, Standard Chartered, Facebook, Microsoft, Github

[1] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/haskell.html`

Overview of these lectures

1. Formalizing languages: syntax and semantics
2. Foundations for functional programming: λ-calculus
3. Strict functional languages
   - FUN: a minimal functional language
   - Formal semantics for FUN
   - The SECD machine
4. Lazy functional languages
   - Graph reduction
   - GHC's core language
   - The STG-machine

Formalizing languages

Syntax	rules for forming expressions, statements and programs
Sematics	specifying the meaning of expressions, statements and programs

Syntax can be formalized using *context free grammars*.

Sematics can be formalized in various ways; we will use *operational semantics*.

Context-free grammars

Σ set of *terminal symbols*

V set of *non-terminal symbols*

P set of *productions* of the form

$$X ::= x_1 \ldots x_n$$

where $X \in V$ and each $x_i \in (\Sigma \cup V)$ is a terminal or non-terminals.

$S \in V$ initial non-terminal symbol

Abreviation for multiple productions with identical left-hand side:

$$X ::= \text{alt}_1 \mid \text{alt}_2 \mid \ldots \mid \text{alt}_k$$

Example: arithmetic expressions

Terminals $\Sigma = \{0, 1, \ldots, 9, +, *, (,)\}$

Non-terminals $V = \{E, N, D\}$

Initial symbol E

Productions

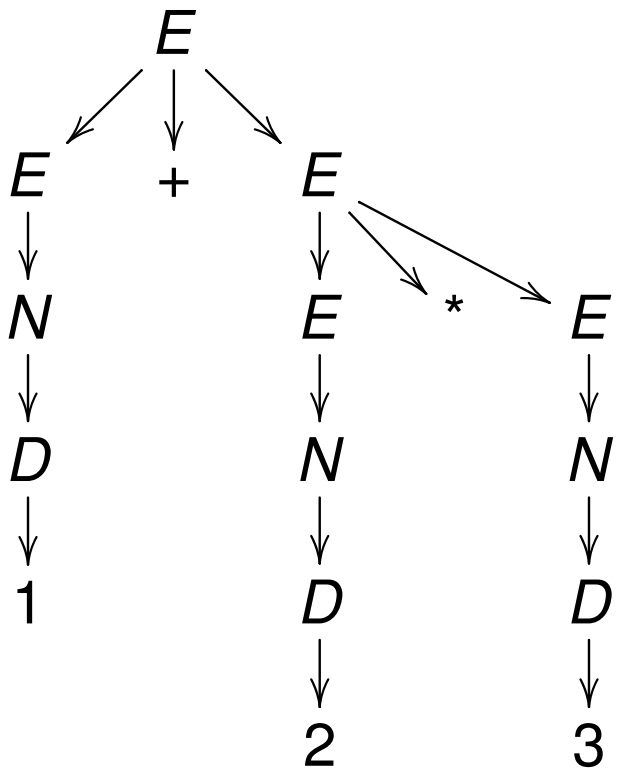
$$E ::= E + E \mid E * E \mid (E) \mid N$$
$$N ::= DN \mid D$$
$$D ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Derivations

$$
\begin{aligned}
E &\rightarrow E + E \rightarrow E + E * E \rightarrow N + E * E \rightarrow D + E * E \\
&\rightarrow 1 + E * E \rightarrow 1 + N * E \rightarrow 1 + D * E \rightarrow 1 + 2 * E \\
&\rightarrow 1 + 2 * N \rightarrow 1 + 2 * D \rightarrow \underbrace{1 + 2 * 3}_{\text{terminals}}
\end{aligned}
$$

Hence: 1+2*3 é a sentence of the language of expressions.

Derivation trees



Exercise: find another emph derivation that produces a different derivation tree [2].

[2] This gramar is ambiguous

Abstract syntax

The abstract syntax represents the structure of the derivation tree omitting unnecessary information, e.g.:

- decomposition of numerals into digits
- parenthesis

Represent it by a recursive data type in Haskell:

```
data Expr = Num Int
          | Add Expr Expr
          | Mult Expr Expr
```

Concrete vs. abstract syntax

```
1+2*3
Add (Num 1) (Mult (Num 2) (Num 3))
Mult (Add (Num 1) (Num 2)) (Num 3)

(1+2)*3
Mult (Add (Num 1) (Num 2)) (Num 3)

1+2+3
Add (Add (Num 1) (Num 2)) (Num 3)
Add (Num 1) (Add (Num 2) (Num 3))
```

Operational semantics

- A relation $\Downarrow$ between *expressions* and *values* (for this case: integers)
- Definided inductively over the shape of the abstract syntax
- Inference rules of the form

$$\frac{A_1 \quad A_2 \quad \ldots \quad A_n}{B}$$

$A_1, A_2, \ldots, A_n$ hypothesis
B conclusion
axiom when $n = 0$ (empty hypothesis)
logical reading if $A_1, \ldots, A_n$ holds then B holds
computational reading to perform B, we must perform $A_1, \ldots, A_n$.

Inference rules

$$\frac{}{\text{Num } n \Downarrow n}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{\text{Add } e_1\ e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 \times v_2}{\text{Mult } e_1\ e_2 \Downarrow v}$$

Example derivation

$$\dfrac{\dfrac{\dfrac{}{\text{Num } 1 \Downarrow 1} \qquad \dfrac{}{\text{Num } 2 \Downarrow 2}}{\text{Add (Num 1) (Num 2)} \Downarrow 3} \qquad \dfrac{}{\text{Num } 3 \Downarrow 3}}{\text{Mul (Add (Num 1) (Num 2)) (Num 3)} \Downarrow 9}$$

Interpreter in Haskell

We can implement the relation $\Downarrow$ as a recursive function:

```
eval :: Expr -> Int
eval (Num n)   = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mult e1 e2) = eval e1 * eval e2
```

Example reduction:

```
  eval (Mult (Add (Num 1) (Num 2)) (Num 3))
= eval (Add (Num 1) (Num 2)) * eval (Num 3)
= (eval (Num 1) + eval (Num 2)) * 3
= (1 + 2) * 3
= 9
```

Correction

The interpreter correctly implements the operational semantics:

$$e \Downarrow v \iff \text{eval } e = v$$

Proof: by *structural induction* on e.

A compiler for expressions

- Let us define a virtual machine for arithmetic expressions
- Compile the expression into a sequence of instructions
- Execute the sequence of instructions to evaluate the expression
- The abstract syntax is no longer needed during the evaluation
- The order of operations becomes explicit

A stack machine

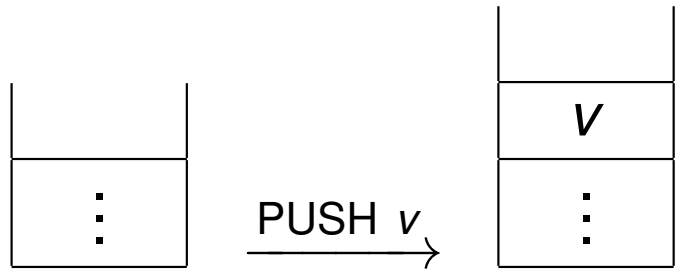
Configuration of the machine:

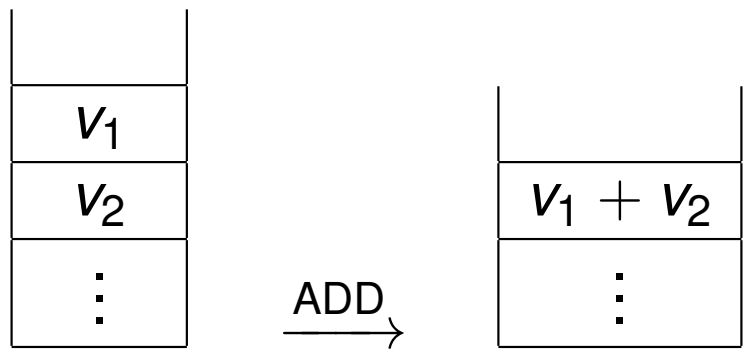
stack	a sequence of integers
code	a sequence of instructions

Machine instructions:

PUSH *n*	push the integer *n* onto the stack
ADD	remove the two values on top of the stack and push the result of the addition
MUL	remove the two values on top of the stack and push the result of the multiplication

Operations on the stack





(Similarly for MUL.)

Implementing the machine in Haskell

```
-- machine instructions
data Instr = PUSH Int
           | ADD
           | MUL

-- the stack is a list of integers
type Stack = [Int]

-- code is a list of instructions
type Code = [Instr]

-- the machine configuration
type State = (Stack, Code)
```

Compiler for expressions

```
compile :: Expr -> Code
compile (Num n) = [PUSH n]
compile (Add e1 e2)
     = compile e1 ++ compile e2 ++ [ADD]
compile (Mul e1 e2)
     = compile e1 ++ compile e2 ++ [MUL]
```

- Translates an expression into a sequence of instructions
- Same recursive structure as `eval` but the result is a list of instructions instead of a value

Invariant

Execution of `compile e` pushes the value of `e` on to the top of the stack.

Example

```
> compile (Mult (Add (Num 1) (Num 2)) (Num 3))
[PUSH 1, PUSH 2, ADD, PUSH 3, MUL]
```

Transition function

Implements the transition associated with each instruction of the virtual machine.

```
transition :: State -> State
transition (stack, PUSH v:code)
          = (v:stack, code)
transition (v1:v2:stack, ADD:code)
          = ((v1+v2):stack, code)
transition (v1:v2:stack, MUL:code)
          = ((v1*v2):stack, code)
transition (_, _)
          = error "no valid transition"
```

NB: the top of the stack is the first element of the list.

Virtual code interpreter

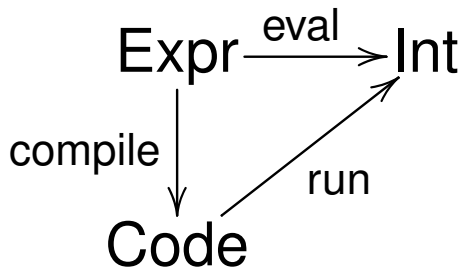
- Start with an empty stack
- Repeat the transition function until we reach the final state
- The result is the value on top of the stack

```
run :: Code -> Int
run code = runAux ([],code)

runAux :: State -> Int
runAux s | final s     = topStack s
         | otherwise = runAux (transition s)
```

(Cue demo.)

Two implementations for computing expressions



Correctness of compilation:

$$\forall e.\ \texttt{run (compile e) = eval e}$$

Prove a stronger result by structural induction on e:

$\forall e \forall s \forall c.\ \texttt{runAux}\,(s, \texttt{compile}\ e +\!+ c) = \texttt{runAux}\,(\texttt{eval}\ e : s, c)$

Conclusion

- Two implementations:
  - **eval**
    - interpreter for expressions
    - high-level specification
  - **run**
    - interpreter for virtual code
    - closer to a low-level machine
    - correctness of compiling
- What about “real” languages? Need variables and bindings
- Next lecture: reviewing the λ-calculus