Reimplementing the Wheel: Teaching Compilers with a Small Self-Contained One

Daniil Berezun    Dmitry Boulytchev

Saint-Petersburg State University
JetBrains Research

10th International Workshop on Trends in Functional Programming in Education
February 16, 2021
Online
Background
Background

Yandex
An education track in programming languages & tools:

- Semantics of programming languages;
- Metacomputations;
- Logic & relational programming;
- ...

PL Program
An education track in programming languages & tools:

- **Programming languages and compilers;**
- Semantics of programming languages;
- Metacomputations;
- Logic & relational programming;
- ...

**PL Program**
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.

Course outline:
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.

Course outline:

- Source language = implementation language.
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.

Course outline:

- Source language = implementation language.
- No parser-centricity.
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.

Course outline:

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
Compiler Construction as an Introductory Course

Prerequisites (soft):
- Functional programming.
- Formal grammars, languages and automata.

Course outline:
- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.

Course outline:

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.
- Self-contained codegenerator (no animals are hurt heavy infrastructure is involved).
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.

Course outline:

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.
- Self-contained codegenerator (no animals are hurt heavy infrastructure is involved).
- Functional programming techniques.
Compiler Construction as an Introductory Course

Prerequisites (soft):

- Functional programming.
- Formal grammars, languages and automata.

Course outline:

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.
- Self-contained code generator (no heavy infrastructure is involved).
- Functional programming techniques.
The $\lambda M a$ Programming Language
The $\lambda Ma$ Programming Language

$$\lambda Ma = \lambda + \text{ALGOL}$$
The \( \lambda M^a \) Programming Language

\[ \lambda M^a = \lambda + \text{ALGOL} \]

fun reverse (l) {
  (fix $ fun (rec) {
    fun (acc, l) {
      case l of
      | {} → acc
      | x : xs → rec (x : acc, xs)
      esac
    ))) ({{}, l)
  }
}
The Structure of the Compiler
The Structure of the Compiler

Source Code
The Structure of the Compiler

Source Code \rightarrow \text{AST} \quad \text{Parser}
The Structure of the Compiler

Source Code → Parser → AST → SM Compiler → SM Code
The Structure of the Compiler

Source Code → AST → SM Code → Native Code

Parser → SM Compiler → Native Compiler

Component LOC

Compiler (OC AML) 3000
Runtime (C+GAS) 1000
Standard library (λ a M a) 900

x86-32
The Structure of the Compiler

Source Code \( \rightarrow \) AST \( \rightarrow \) SM Code \( \rightarrow \) Native Code

Parser \( \downarrow \)

Source Interpreter

SM Compiler

Native Compiler

\( \lambda \) a M

Component LOC

Compiler (OCAML) 3000

Runtime (C+GAS) 1000

Standard library (\( \lambda \) a M) 900

x86-32
The Structure of the Compiler

Source Code → AST → SM Code → Native Code

- Parser
- SM Compiler
- Native Compiler

Source Interpreter
SM Interpreter

Component LOC
Compiler (OC, AML) 3000
Runtime (C+GAS) 1000
Standard library (λ, M, a) 900

x86-32
The Structure of the Compiler

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler (OCAML)</td>
<td>3000</td>
</tr>
<tr>
<td>Runtime (C+GAS)</td>
<td>1000</td>
</tr>
<tr>
<td>Standard library ($\lambda a.Ma$)</td>
<td>900</td>
</tr>
</tbody>
</table>
A Stack of Languages with “Vertical” Homework Assignments
A Stack of Languages with “Vertical” Homework Assignments

Simple straight-line programs made of assignments and sequential composition;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + all control constructs treated as expressions;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + all control constructs treated as expressions;
4. + local definitions, scopes and functions;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + all control constructs treated as expressions;
4. + local definitions, scopes and functions;
5. + arrays and builtin functions;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + all control constructs treated as expressions;
4. + local definitions, scopes and functions;
5. + arrays and builtin functions;
6. + fixednum arithmetics;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + all control constructs treated as expressions;
4. + local definitions, scopes and functions;
5. + arrays and builtin functions;
6. + fixednum arithmetics;
7. + S-expressions;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + all control constructs treated as expressions;
4. + local definitions, scopes and functions;
5. + arrays and builtin functions;
6. + fixednum arithmetics;
7. + S-expressions;
8. + pattern-matching;
A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + all control constructs treated as expressions;
4. + local definitions, scopes and functions;
5. + arrays and builtin functions;
6. + fixednum arithmetics;
7. + S-expressions;
8. + pattern-matching;
9. + first-class functions (this point actually have never been reached within one semester).
Deep Embedding vs. Syntax Analyser

```plaintext
infix + at (+ (l, r) {Binop ("+", opnd (l), opnd (r))})
infix - at (- (l, r) {Binop ("-", opnd (l), opnd (r))})
infix * at (* (l, r) {Binop ("*", opnd (l), opnd (r))})
infix / at (/ (l, r) {Binop ("/", opnd (l), opnd (r))})
infix == at (== (l, r) {Binop ("==", opnd (l), opnd (r))})
```

...
Deep Embedding vs. Syntax Analyser

No syntax analyzer initially.
Deep Embedding vs. Syntax Analyser

*No syntax analyzer initially.*

```plaintext
infix + at + (l, r) {
    Binop ("+", opnd (l), opnd (r))
}

infix - at - (l, r) {
    Binop ("-", opnd (l), opnd (r))
}

infix * at * (l, r) {
    Binop ("*", opnd (l), opnd (r))
}

infix / at / (l, r) {
    Binop ("/", opnd (l), opnd (r))
}

infix == at == (l, r) {
    Binop ("==", opnd (l), opnd (r))
}

...
```
Deep Embedding vs. Syntax Analyser

No syntax analyzer initially.

```
#include <iostream>

using namespace std;

typedef int BinopType;

int main() {
    BinopType x = 10;
    BinopType y = 20;
    BinopType z;

    if (x + y > z) {
        cout << z = x + y;
    }

    if (x - y > z) {
        cout << z = x - y;
    }

    if (x * y > z) {
        cout << z = x * y;
    }

    if (x / y > z) {
        cout << z = x / y;
    }

    if (x == y) {
        cout << z = x == y;
    }

    if (x >= y) {
        cout << z = x >= y;
    }

    if (x > y) {
        cout << z = x > y;
    }

    return 0;
}
```
Syntax Analysis with Parser Combinators

OSTAP—a library of monadic parser combinators is CPS and memoization [Johnson, 1995; Izmaylova, Afroozeh, van der Storm, 2015].

Embedded DSL for \( \lambda \):
syntax (kSkip {Skip} | x=lident s["=":] e=exp {Assn (x, e)} | kRead x=inbr[s("(")], lident, s(")"] {Read (x)} | kWrite e=inbr[s("(")], exp, s(")"] {Write (e)} | kWhile e=exp b=inbr[kDo, stmt, kOd] {While (e, b)})
Syntax Analysis with Parser Combinators

OSTAP — a library of monadic parser combinators is CPS and memoization [Johnson, 1995; Izmaylova, Afroozeh, van der Storm, 2015].
OSTAP — a library of monadic parser combinators is CPS and memoization [Johnson, 1995; Izmaylova, Afroozeh, van der Storm, 2015].

Embedded DSL for $\lambda^M a$:

```plaintext
syntax (kSkip           {Skip}  | 
x= lidsent s["="] e= exp    {Assn (x, e)}  | 
kRead x= inbr[s("(") , lidsent , s(")")] {Read (x)}  | 
kWrite e= inbr[s("(") , exp , s(")")] {Write (e)}  | 
kWhile e= exp b= inbr[kDo, stmt, kOd] {While (e, b)}
```
Operational Semantics

fun eval (c@⟨σ, w⟩, stmt) {
  case stmt of
    | While (e, b) →
      if evalExpr (σ, e) then eval (eval (c, b), stmt) else c fi

...
Operational Semantics

\[ \sigma \xrightarrow{e, n \neq 0} \langle \sigma, w \rangle \xrightarrow{S} c' \quad c' \xrightarrow{\text{while } e \text{ do } S} c'' \]

\[ \langle \sigma, w \rangle \xrightarrow{\text{while } e \text{ do } S} c'' \]

\[ \sigma \xrightarrow{e, 0} \]

\[ \langle \sigma, w \rangle \xrightarrow{\text{while } e \text{ do } S} \langle \sigma, w \rangle \]
fun eval (c@[s, w], stmt) {
    case stmt of
        ...
    | While (e, b) -> if evalExpr (s, e)
        then eval (eval (c, b), stmt)
        else c
        fi
    ...
}
Operational Semantics (SM)
Operational Semantics (SM)

\[
\frac{\langle (x \oplus y)s, c \rangle \xrightarrow{p} c'}{\langle yxs, c' \rangle \xrightarrow{\text{BINOP } \otimes \ p} c'}
\]
Operational Semantics (SM)

\[
\frac{\langle(x \oplus y)s, c \rangle}{p} \rightarrow c'
\]

\[
\langle yxs, c \rangle \xrightarrow{[\text{BINOP } \otimes]} p \rightarrow c'
\]

```haskell
fun eval (c@[st, s, w], insns) {
  case insns of
    {}         \rightarrow c
  | i : insns \rightarrow
    eval (  
      case i of
        ...  
        | BINOP (op) \rightarrow
          case st of
            x : y : st \rightarrow [evalOp (op, y, x) : st, s, w]
            esac
        ...
      }
    }
}
```
Operational Semantics (Static)
ref $x : \text{Ref}$  
$x : \text{Val}$  
ignore $x : \text{Void}$  
$x \in \mathcal{X}$
Operational Semantics (Static)

\[
\text{ref } x : \text{Ref} \quad x : \text{Val} \quad \text{ignore } x : \text{Void} \quad x \in \mathcal{X}
\]

\text{syntax} \quad (x=\text{lident} \quad \{ \text{fun} \ (a) \ {\{ \text{case} \ a \ \text{of} \}

\begin{align*}
\text{Ref} & \rightarrow \text{Ref} \ (x) \\
\text{Void} & \rightarrow \text{Ignore} \ (\text{Var} \ (x)) \\
\text{Val} & \rightarrow \text{Var} \ (x)
\end{align*}

\text{esac}

\}

\}

\}

\ldots
Codegeneration with Symbolic Interpreters
Codegeneration with Symbolic Interpreters

Idea: symbolic interpreter which operates on *locations* instead of data values can be used for codegeneration.
Codegeneration with Symbolic Interpreters

Idea: symbolic interpreter which operates on *locations* instead of data values can be used for codegeneration.

<table>
<thead>
<tr>
<th>Stack before</th>
<th>Stack machine instruction</th>
<th>Stack after</th>
<th>Machine instruction emitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>CONST 1</td>
<td>{%eax}</td>
<td>movl $1, %eax</td>
</tr>
<tr>
<td>{%eax}</td>
<td>LD x</td>
<td>{%eax, %ebx}</td>
<td>movl $x, %ebx</td>
</tr>
<tr>
<td>{%eax, %ebx}</td>
<td>BINOP +</td>
<td>{%eax}</td>
<td>addl %ebx, %eax</td>
</tr>
<tr>
<td>{%eax}</td>
<td>ST y</td>
<td>{}</td>
<td>movl %eax, $y</td>
</tr>
</tbody>
</table>
Codegeneration with Symbolic Interpreters

| CONST (n) → |
| [n : st, cst, s, w] |

| LD (x) → |
| [lookup (s, x) : st, cst, s, w] |

| ST (x) → |
| let n : _ = st in |
| [st, cst, assign (s, x, n), w] |

| CONST (n) → |
| let [s, env] = env.allocate in |
| [env, code <+ Mov (L (box $ n), s)] |

| LD (x) → |
| let [s, env] = env.allocate in |
| [env, code <+> move (env.loc (x), s)] |

| ST (x) → |
| let [s, env] = env.allocate in |
| [env, code <+> move (env.peek, env.loc (x))]|
Organization Trivia

- The course has been taught since 2016 in OCaml; since the spring of 2020 — in λMLa itself.
- 80+ students each semester.
- Homework assignment each week.
- Continuous integration (TRAVISCI via GITHUB).
- “Lightning” division: a questionnaire of 100+ items for grade C (3/5), no homework.
Students’ Feedback

The vast majority qualified the course material as *new* for them (42% — completely new, 58% — mostly new);

42% qualified the material as potentially *irrelevant* to their future professional activity; 25% as relevant, and the rest as partially relevant;

An essential fraction complained about the lack of a type system in $\lambda a\mathcal{M}a$ (prior to the spring of 2020 — about the type system in OCAML).

“Writing a compiler for $\lambda a\mathcal{M}a$ in $\lambda a\mathcal{M}a$ was a terrible thing when you had no experience with neither $\lambda a\mathcal{M}a$ nor its relative language OCAML.”

“A very pleasant thing was that $\lambda a\mathcal{M}a$ was developed specifically for the course and was truly convenient for compiler implementation, especially if one had no prior experience with OCAML.”
Conclusions and Future Work

- Not very mature, not very efficient.
+ Self-contained, small, good for introduction purposes.
+ With diversity of constructs.
+ A “tower” of sublanguages.
+ With compiler-oriented DSLs.

Future:

- Multiple backends (IA64? ARM? WebAssembly? JVM? LLVM?)
- Static semantics (type system?)
- Better codegeneration (but still within symbolic interpreter model).