NaCl

Martin Rupp, Klaus Kraßnitzer

25.06.2020

The NaCl Language

The NaCl Language

NaCl...

- ...is a toy programming language
- ...can be executed from a **Python-like console** ("Read-Eval-Print-Loop")
- ...compiles down to the `.mltn` binary format which is interpreted by the **Reactor** Virtual Machine.

NaCl Language Features

- Global Execution (small programs, console execution)
- Variables and (static) Types (**float**, **int**, **bool**)
- Control flow (**if/else**, **while**)
- Functions (typed or void return)
- Multi-line comments (also nested)
- Built-in functions (currently only **print**)

NaCl Code Example

```
/* calculate the n-th fibonacci number */
fib (n:int) -> int {
    if n < 2 {
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

expected : int = 144;
actual := fib(12);
print(expected, actual);
```

Implementation

NaCl Implementation

- Fully written in **Rust** ($\sim$4700+400 lines of code)
    - Most loved programming language on GitHub
    - Fast and Reliable
    - Many unique features: borrow checker, no null values, ...
- No external dependencies (hand-written Lexer, Parser, ...)
- One program for language parsing, console and bytecode compilation (`nacl`)
- Another (separate) program for bytecode execution (`reactor`)

Error Handling (1)

Inspired by Rust, **NaCl** features sophisticated error handling for all stages (Lexer, Parser, Static Check) with error recovery in Parser and Lexer.

Example (Parser Error Handling):

```
f (x:) {
    return x;
}

f(x;
x : fl = 3.0;
if x < {
    print(x);
}
```

Error Handling (2)

```
Parser Error

    line 1, col 6: Unexpected token ")", expected:
        Type

    line 5, col 4: Unexpected token ";", expected:
        :, )

    line 6, col 5: Unexpected token "Identifier", expected:
        Type

    line 7, col 9: Unexpected token "{", expected:
        Identifier, Boolean Constant, Integer Constant, Float Constant, (

Aborting due to previous parser errors
```

Reactor VM

- Simple, stack-based machine
- Four-Stack Layout:
  - Operand Stack
  - Local Variable Stack
  - Global Variable Stack
  - Call Stack
- Variable-size instruction set
  - Operator instructions (e.g. `FAdd`, `IMod`)
  - Variable-related instructions (e.g. `StoreL`, `LoadC`)
  - Control flow instructions (e.g. `JmpU`, `Exit`)
  - Function-Related instructions (e.g. `CallF`, `Return`)

Demo

Benchmarks

Benchmarking (1)

Compare performance of **NaCl** AST-Interpreter and **Reactor** VM in simple benchmarks against:

- C, Rust (compiled)
- Java, JavaScript (JIT-Compiled)
- Python, Java, JavaScript (Interpreted)

Benchmarking Conditions:

- Test System with Intel i7 4790k @ 4.6 GHz running Linux
- Programs were benchmarked using `perf stat` with 50 repetitions (5 for AST interpreter)
- Running time in nanoseconds logged, average calculated

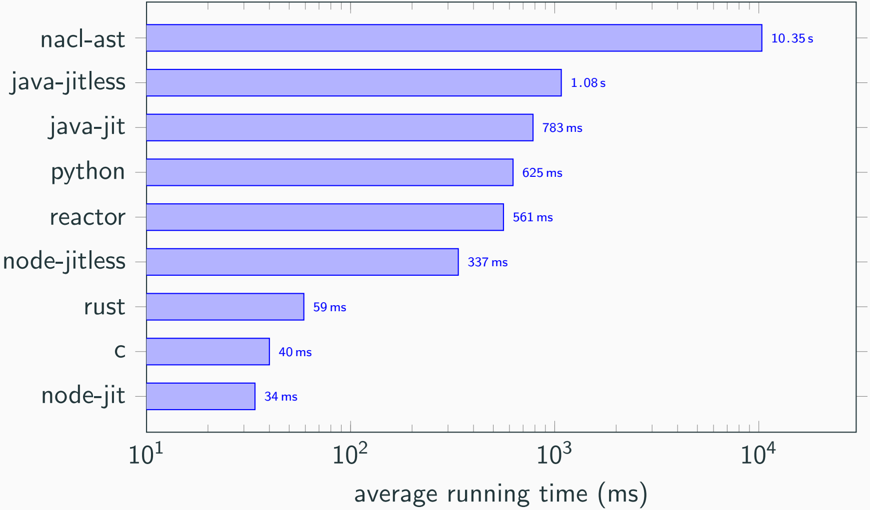
Benchmarking (2)

The following benchmarks were conducted:

Name	Benchmark	Tested Features
`fib`	40th Fibonacci-Number recursively	Function Calls
`fac-rec`	20! 1.000.000 times recursively	Function Calls
`fac-it`	20! 1.000.000 times iteratively	Loops
`pprime`	200th palindrome prime number	Function Calls, Loops
`sin`	Sine Taylor Approximation	Floats, Loops

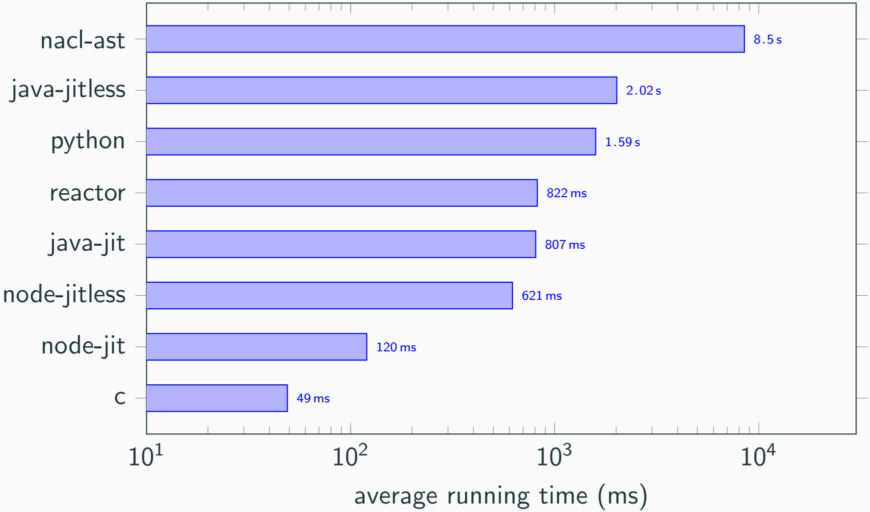
Faculty Iterative Benchmark

Running time (lower is better):



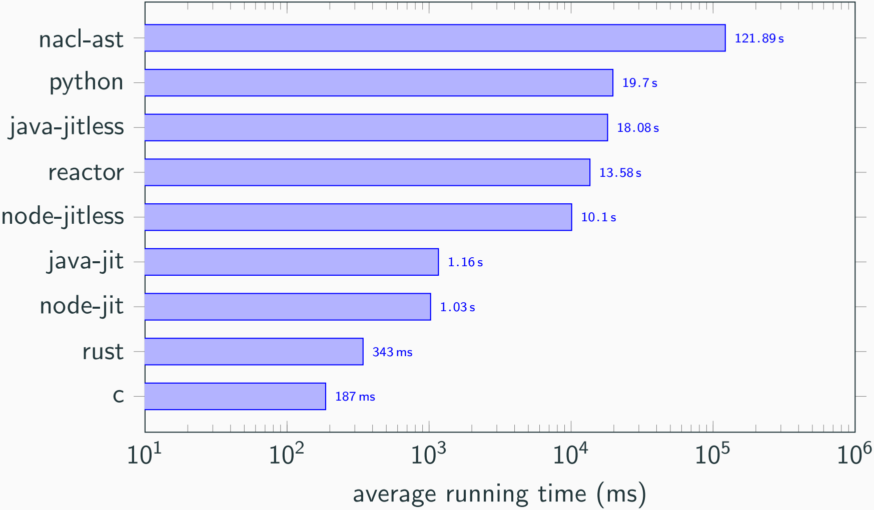
Faculty Recursive Benchmark

Running time (lower is better):



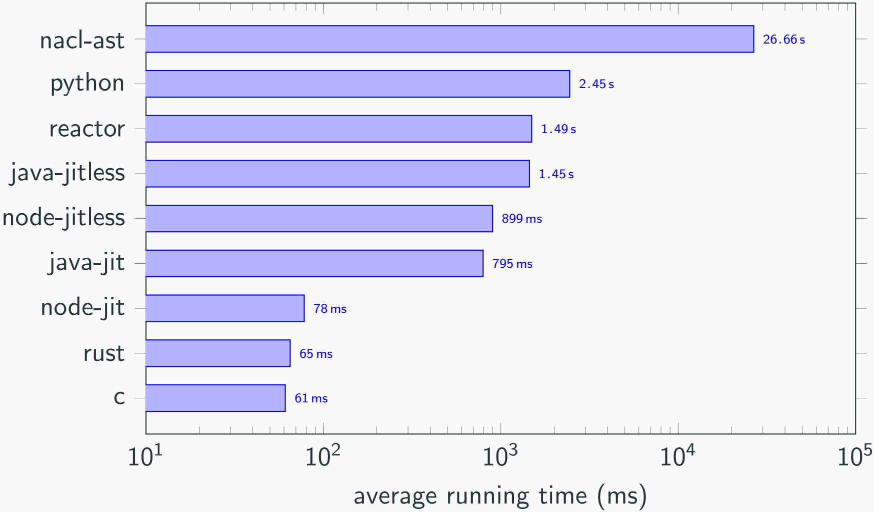
Fibonacci Benchmark

Running time (lower is better):



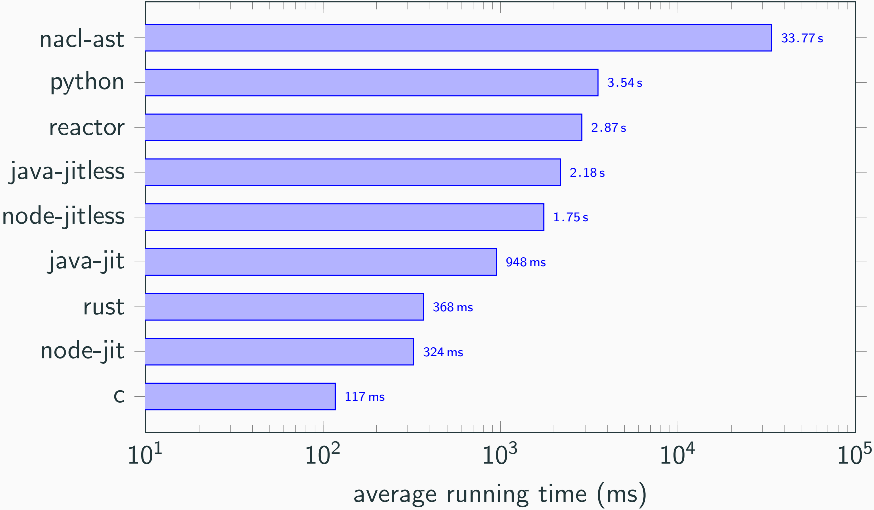
Sine Approximation Benchmark

Running time (lower is better):



Palindrome Prime Benchmark

Running time (lower is better):



Conclusion

In our benchmarks, we observed - as expected - a huge performance improvement of the **Reactor** VM over the AST interpreter.

Reactor is **comparable in performance** to common interpreted languages such as Python, JavaScript and (interpreted) Java.

In our tests, **Reactor** is on average [1]:

- 42 % faster than Python
- 24 % faster than interpreted Java (`java -Xint`)
- 53 % slower than interpreted JavaScript (`node --jitless`)
- 29 times slower than C

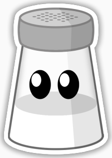
[1] average relative performance over all benchmarks

Thank you for your attention!

Questions?

Find the repositories (`nacl`, `reactor`, `nacl-bechmark`) on GitLab:

`gitlab.com/nacl-lang`



Backup Slides

Challenge: Global variable scoping and functions

```
1  f () {
2      a = a + 1;
3  }
4
5  a := 0;
6  f();
7  print(a);
8
9  if true {
10     a := 3;
11     f();
12     print(a);
13 }
14
15 print(a);
```

- Intended Behavior:
    - line 7 should print ‘1’
    - line 12 should print ‘4’
    - line 15 should print ‘1’ again
- Variable `a` is shadowed in the if statement and has therefore different global ids in the two print calls
- **Solution**: Keep track of function “dependencies” and accessed global variables, compile on function call depending on current global variable ids

Possible Future Work

- Expand instruction set (e.g. improve loop performance with Increment instruction or fast-track variable assignment)
- Make a register-based machine and compare performance to **Reactor**
- Expand **Reactor** with a JIT-Compiler

Thank you for your attention!

Questions?

Find the repositories (`nacl`, `reactor`, `nacl-bechmark`) on GitLab:

`gitlab.com/nacl-lang`