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)
/* 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** (~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**)

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):

```plaintext
1  f (x:) {
2      return x;
3  }
4
5  f(x;
6  x : fl = 3.0;
7  if x < {
8      print(x);
9  }
```
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
The following benchmarks were conducted:

<table>
<thead>
<tr>
<th>Name</th>
<th>Benchmark</th>
<th>Tested Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>40th Fibonacci-Number recursively</td>
<td>Function Calls</td>
</tr>
<tr>
<td>fac-rec</td>
<td>20! 1.000.000 times recursively</td>
<td>Function Calls</td>
</tr>
<tr>
<td>fac-it</td>
<td>20! 1.000.000 times iteratively</td>
<td>Loops</td>
</tr>
<tr>
<td>pprime</td>
<td>200th palindrome prime number</td>
<td>Function Calls, Loops</td>
</tr>
<tr>
<td>sin</td>
<td>Sine Taylor Approximation</td>
<td>Floats, Loops</td>
</tr>
</tbody>
</table>
Faculty Iterative Benchmark

Running time (lower is better):

- nacl-ast: 10.35 s
- java-jitless: 1.08 s
- java-jit: 783 ms
- python: 625 ms
- reactor: 561 ms
- node-jitless: 337 ms
- rust: 59 ms
- c: 40 ms
- node-jit: 34 ms
Faculty Recursive Benchmark

Running time (lower is better):

- nacl-ast: 8.5 s
- java-jitless: 2.02 s
- python: 1.59 s
- reactor: 822 ms
- java-jit: 807 ms
- node-jitless: 621 ms
- node-jit: 120 ms
- c: 49 ms
**Fibonacci Benchmark**

**Running time** (lower is better):

<table>
<thead>
<tr>
<th>Language</th>
<th>Average Running Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nacl-ast</td>
<td>121.89 s</td>
</tr>
<tr>
<td>python</td>
<td>19.7 s</td>
</tr>
<tr>
<td>java-jitless</td>
<td>18.08 s</td>
</tr>
<tr>
<td>reactor</td>
<td>13.58 s</td>
</tr>
<tr>
<td>node-jitless</td>
<td>10.1 s</td>
</tr>
<tr>
<td>java-jit</td>
<td>1.16 s</td>
</tr>
<tr>
<td>node-jit</td>
<td>1.03 s</td>
</tr>
<tr>
<td>rust</td>
<td>343 ms</td>
</tr>
<tr>
<td>C</td>
<td>187 ms</td>
</tr>
</tbody>
</table>
Sine Approximation Benchmark

Running time (lower is better):

- nacl-ast: 26.66 s
- python: 2.45 s
- reactor: 1.49 s
- java-jitless: 1.45 s
- node-jitless: 899 ms
- java-jit: 795 ms
- node-jit: 78 ms
- rust: 65 ms
- c: 61 ms

average running time (ms)
Palindrome Prime Benchmark

Running time (lower is better):

- **nacl-ast**: 33.77 s
- **python**: 3.54 s
- **reactor**: 2.87 s
- **java-jitless**: 2.18 s
- **node-jitless**: 1.75 s
- **java-jit**: 948 ms
- **rust**: 368 ms
- **node-jit**: 324 ms
- **c**: 117 ms
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

```plaintext
f () {
    a = a + 1;
}

a := 0;
f ();
print (a);

if true {
    a := 3;
f ();
    print (a);
}

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