CASM: Implementing an Abstract State Machine based programming language

Roland Lezuo, Gergö Barany, Andreas Krall
{llezuo,gergo,andi}@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

ATPS 2013
February 26, 2013
• CASM: programming language based on Abstract State Machines
• Interpreted/compiled (to C++), supports symbolic execution
• Our application: verified instruction set simulation
### Definitions

**Abstract State Machines**

- **State:** arbitrary data
- **Rules:** specify data values in next state

**Rules**

- Pure
- Independent
- Allow parallel execution
Example: Parallel Swap

function x : -> Int
function y : -> Int

rule swap = {
    x := y
    y := x
}

- Functions: state data
- Map locations (argument tuples) to values
Example: Parallel Swap

```
function x : -> Int
function y : -> Int

rule swap = {
    x := y
    y := x
}
```

- Rules: specify updates
- Independent evaluation of updates
- Update set captures all effects
Example: Parallel Swap

function x : -> Int
function y : -> Int

rule swap = {
    x := y
    y := x
}

- State transition: atomic application of all updates
Example: Parallel Swap

function x : -> Int
function y : -> Int

rule swap = {
    x := y
    y := x
}

Example

Initial state: \{x = 3, y = 2\}
Update set: \{(x, 2), (y, 3)\}
New state: \{x = 2, y = 3\}
Example: Naïve Fibonacci

```plaintext
function i : -> Int initially { 2 }

function fib : Int -> Int initially { 0 -> 0, 1 -> 1 }

rule nextfib = {
  i := i + 1
  fib(i) := fib(i-1) + fib(i-2)
}
```

Evaluations of `fib(...)`: lookups, not calls; Memoization built in!

Execution model: Repeat top-level rule until explicit termination.
Example: Naïve Fibonacci

function i : -> Int initially { 2 }
function fib : Int -> Int initially { 0 -> 0, 1 -> 1 }

rule nextfib = {
    i := i + 1
    fib(i) := fib(i-1) + fib(i-2)
    if i >= 100 then
        program(self) := undef /* terminate */
}

if rule: empty update set if condition false
rule hundred_fibs =

    seqblock
        fib(0) := 0
        fib(1) := 1
        i := 2

    endseqblock

seqblock: Compute subrules sequentially.

Track intermediate update sets.
rule hundred_fibs =
  seqblock
    fib(0) := 0
    fib(1) := 1
    i := 2
  iterate
    if i < 100 then {
      i := i + 1
      fib(i) := fib(i-1) + fib(i-2)
    }
  endseqblock

seqblock: Compute subrules sequentially.
iterate: Repeat subrule until update set empty.
Track intermediate update sets.
Other language constructs

Parallel loop:

\[
\text{forall } i \text{ in } [0..3] \text{ do }
\]

\[
square(i) := i \ast i
\]
Parallel loop:

\[
\text{forall } i \text{ in } [0..3] \text{ do}
\]
\[
\text{square}(i) := i \times i
\]

Derived expressions ("function definitions"):

\[
\text{derived } d(a: \text{Int}, b: \text{Boolean}) = (a \geq 3) \text{ and } b
\]
\[
\ldots
\]
\[
\text{foo}(y) := d(x, \text{true})
\]
Parallel loop:

```
forall i in [0..3] do
    square(i) := i * i
```

Derived expressions ("function definitions"):

```
derived d(a: Int, b: Boolean) = (a >= 3) and b
...
foo(y) := d(x, true)
```

Subrule invocation:

```
rule simulation = {
    call time_update()
    call system_update()
    call environment_update()
}
```
Static monomorphic type system with simple type inference.

Primitive types: Int, Boolean, ...
Type constructors: List(...), Tuple(..., ...)
Symbols: enum MyEnum = { one, two, three }

No support (yet?) for algebraic datatypes.
Symbolic execution

function i : -> Int initially { 2 }
function fib : (symbolic) Int -> Int

rule nextfib = {
    i := i + 1
    fib(i) := fib(i-1) + fib(i-2)
}

Symbolic trace of successive states:

{(i, 2)}
Symbolic execution

function i : -> Int initially { 2 }
function fib : (symbolic) Int -> Int

rule nextfib = {
    i := i + 1
    fib(i) := fib(i-1) + fib(i-2)
}

Symbolic trace of successive states:

- \{ (i, 2) \}
- \{ (i, 3), (fib(0), s_0), (fib(1), s_1), (fib(2), s_0 + s_1) \}
Symbolic execution

```
function i : -> Int initially { 2 }
function fib : (symbolic) Int -> Int

rule nextfib = {
    i := i + 1
    fib(i) := fib(i-1) + fib(i-2)
}
```

Symbolic trace of successive states:

- \{ (i, 2) \}
- \{ (i, 3), (fib(0), s_0), (fib(1), s_1), (fib(2), s_0 + s_1) \}
- \{ (i, 4), (fib(0), s_0), (fib(1), s_1), (fib(2), s_0 + s_1), (fib(3), (s_0 + s_1) + s_1) \}
- ...
Implemented MIPS instruction set simulator in CASM. Example instruction: and-immediate

```
rule andi(instr: Int) =
let rs = PARG(instr, FV_RS) in
let rt = PARG(instr, FV_RT) in
let imm = PARG(instr, FV_IMM) in
let imm_ex = BVZeroExtend(imm, 16, 32) in
  if rt != 0 then
    GPR(rt) := BVAnd(32, GPR(rs), imm_ex)
```

Instruction set specification: ~ 600 lines
Execution model (simplified):

```
rule run_program =
  seqblock
    /* execute instruction at PC */
    call(PMEM(PC))(PC)
    /* update PC for next instruction */
    if BRANCH = undef then
      PC := PC + 4
    else {
      PC := BRANCH
      BRANCH := undef
    }
  endseqblock
```

Simple simulator model executes at $\sim 1$ MHz
Verified more complex simulator implementations:

- Specified pipelined execution models (1500 LOC instruction descriptions, 400 LOC execution models)

Instructions in different pipeline stages execute independently

Symbolic execution trace

Equivalence proof of simple and pipelined semantics
Summary

- CASM: interpreted/compiled/symbolic programming language based on abstract state machines
- Pure (or: disciplined?), statically typed
- Efficient enough for real-world instruction set simulation
CASM: interpreted/compiled/symbolic programming language based on abstract state machines

- Pure (or: disciplined?), statically typed
- Efficient enough for real-world instruction set simulation

Thank you for your attention!

This work is supported by the Austrian Research Promotion Agency (FFG) under contract 827485, *Correct Compilers for Correct Application Specific Processors*, and Catena DSP GmbH.