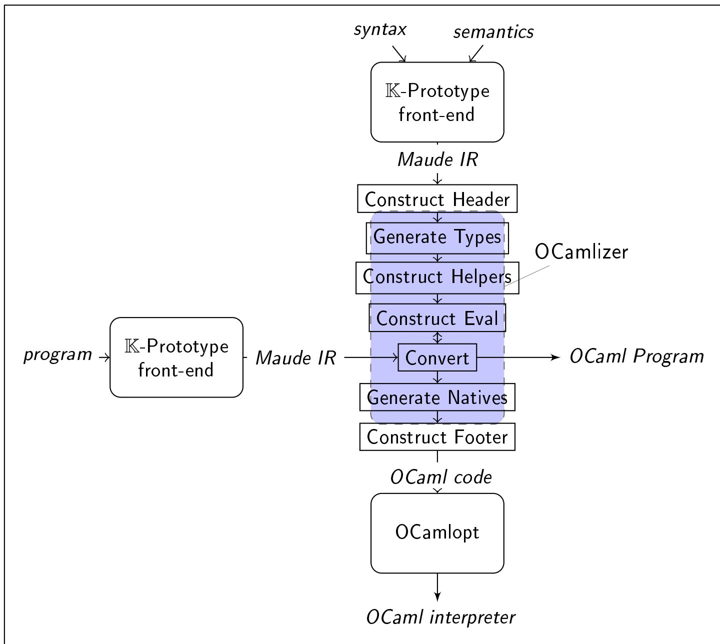
Compiling $\mathbb{K}$ Definitions into Competitive Interpreters

Michael Ilseman Chucky Ellison

Department of Computer Science
University of Illinois

August 8, 2011


syntax
semantics
𝕂-Prototype front-end
Maude IR
Construct Header
Generate Types
OCamlizer
Construct Helpers
Construct Eval
program
𝕂-Prototype front-end
Maude IR
Convert
OCaml Program
Generate Natives
Construct Footer
OCaml code
OCamlopt
OCaml interpreter

Summary of Compilation

- Header: Necessary OCaml libraries

Summary of Compilation

- Header: Necessary OCaml libraries
- Types: OCaml datatypes are built using constructors from syntax

Summary of Compilation

- Header: Necessary OCaml libraries
- Types: OCaml datatypes are built using constructors from syntax
- Helpers: Functions are constructed allowing matching/deconstruction of the configuration

Summary of Compilation

- Header: Necessary OCaml libraries
- Types: OCaml datatypes are built using constructors from syntax
- Helpers: Functions are constructed allowing matching/deconstruction of the configuration
- Eval: Rules turn into nested matching cases of a single eval function. At the bottom, OCaml specific matching rules for builtins are incorporated.

Details of Implementation

- Types are split into results and causes.

Details of Implementation

- Types are split into results and causes.
- Configurations are represented as tuples. Mappings are represented as hash-tables.

Details of Implementation

- Types are split into results and causes.
- Configurations are represented as tuples. Mappings are represented as hash-tables.
- No A or AC matching allowed.

Details of Implementation

- Types are split into results and causes.
- Configurations are represented as tuples. Mappings are represented as hash-tables.
- No A or AC matching allowed.
- Detecting a partial ordering of lookups would allow one to handle things such as

$$\begin{array}{l} \langle \mathtt{x} \curvearrowright \mathtt{k} \rangle_k \langle \mathtt{x} \mapsto \mathtt{L}, \rho \rangle_{env} \langle \mathtt{L} \mapsto \mathtt{i}, \sigma \rangle_{store} \\ \to \langle \mathtt{i} \curvearrowright \mathtt{k} \rangle_k \langle \mathtt{x} \mapsto \mathtt{L}, \rho \rangle_{env} \langle \mathtt{L} \mapsto \mathtt{i}, \sigma \rangle_{store} \end{array}$$

as opposed to just

$$\begin{array}{l} \langle \mathtt{x} \curvearrowright \mathtt{k} \rangle_k \qquad \langle \rho \rangle_{env} \langle \sigma \rangle_{store} \\ \to \langle \sigma[\rho[\mathtt{x}]] \curvearrowright \mathtt{k} \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{store} \end{array}$$

Example Rule

```
rule <k> letrec X = E in E2
        => allocate(X)
        ~> E
        ~> writeTo(X)
        ~> E2
        ~> restore(copy(Rho)) ...</k>
    <env> Rho </env>
```

```
| ((Cause (KList (
      (Cause (Apply (Cause LetrecTildeinTilde
                 :: Cause (Apply (Cause TildeEqTilde
                              :: Cause (Variable st_v_X)
                              :: vE :: [] ))
                 :: vETwo :: [] )))
   :: vRest))
  , Hash vRho, ooNEXTLOCoo, ooSTOREoo ), ooRESULToo )
-> eval ((Cause (KList(
             Cause (Allocate (Cause (Variable st_v_X) :: []))
          :: vE
          :: Cause (WriteTo (Cause (Variable st_v_X) :: []))
          :: vETwo
          :: Cause (Restore (Hash (Hashtbl.copy vRho) :: []))
          :: vRest))
         , Hash vRho, ooNEXTLOCoo, ooSTOREoo), ooRESULToo)
```

Imperative Fibonacci Comparison

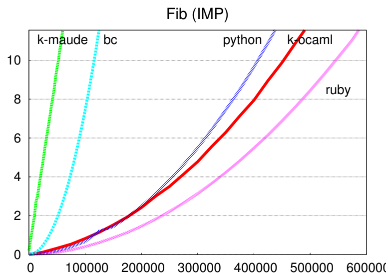
IMP:

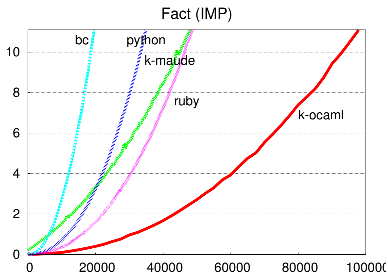
```
x := 1 ;
n := 1 ;
y := 1 ;
z := 1 ;
while ( n <= input ) (
  t := y ;
  x := y ;
  y := z ;
  z := t + y ;
  n := n + 1 ) ;
x + 0
```

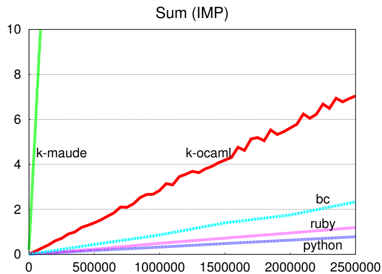
bc:

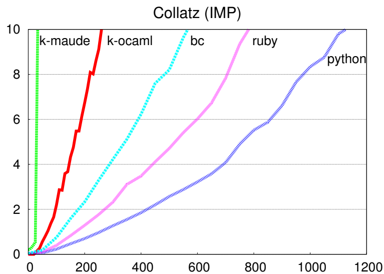
```
b = read() ;
x = 1 ;
y = 1 ;
z = 1 ;
n = 1 ;
while ( n <= b ) {
  t = y ;
  x = y ;
  y = z ;
  z = t + y ;
  n = n + 1 } ;
print x
quit
```

IMP Results









FUN Results