CafeOBJ

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

CafeOBJ

- CafeOBJ: an algebraic specification language/system.
  - 1995–: Japan Institute of Advanced Science and Technology (JAIST).
    Kokichi Futatsugi (JAIST), Razvan Diaconescu (IMAR institute, Romania), et al.
- A member of the OBJ language family.
  - Since 1970s: Josef Goguen (Univ. of California at San Diego), et al.
  - From http://www.cse.ucsd.edu/users/goguen/sys/obj.html:
    The OBJ languages are broad spectrum algebraic programming and specification languages, based on order sorted equational logic, possibly enriched with other logics (such as rewriting logic, hidden equational logic, or first order logic), and providing the powerful module system of parameterized programming.
- Current Version (2008): 1.4.8
  - Open source implementation in Common Lisp.
  - Binaries for Linux, Windows, Mac OS provided.

http://www.ldl.jaist.ac.jp/cafeobj

Starting CafeOBJ

```
> cafeobj
-- loading standard prelude
; Loading /usr3/cafeobj-1.4/prelude/std.bin

           -- CafeOBJ system Version 1.4.7b5(PigNose0.99) --
                  built: 2007 Jan 27 Sat 23:48:16 GMT
                         prelude file: std.bin
                                  ***
                      2008 Jan 22 Tue 7:27:38 GMT
                            Type ? for help
                                  ***
                  -- Containing PigNose Extensions --
                                  ---
         built on International Allegro CL Enterprise Edition
                 8.0 [Linux (x86)] (Jan 28, 2007 8:48)
CafeOBJ>
```

Defining Tight Modules

- module! STACK
  - Introduce a "tight module": a named specification with initial (executable) semantics.
- protecting (NAT)
  - Import another specification preserving its model.
- signature { [*sorts*] *opns* }
- axioms { var *vars* *equns* }
  - Pattern matching on the left hand side of each equation.
  - Note the period after each equation (preceded by a blank)!

Executable specifications.

```
-- stack of natural numbers
module! STACK
{
  protecting (NAT)
  signature
  {
    [ Stack ]
    op empty : -> Stack
    op push : Nat Stack -> Stack
    op top : Stack -> Nat
    op pop : Stack -> Stack
  }
  axioms
  {
    var N : Nat
    var S : Stack
    eq top(push(N, S)) = N .
    eq pop(push(N, S)) = S .
  }
}
```


Predefined Modules

CafeOBJ provides a library of predefined modules.

- Some modules are automatically imported.
  - BOOL: sort Bool, ops. true, false, not, and, or, xor, implies.
- Other modules require explicit import.
  - NAT: sort Nat, number literals, operations 0, s, 1, +, *, <, <=, ...
  - INT: sort Int, literals and operations as for NAT extended by -.
  - RAT: sort Rat, literals and operations as for INT extended by /.
  - CHARACTER: sort Character with various operations.
  - STRING: sort String with various operations.
  - ...

See subdirectory lib of CafeOBJ installation for module names, use command "show *Module*" for viewing module contents.

Showing Module Contents

```
CafeOBJ> show NAT
; Loading /usr3/cafeobj-1.4/lib/nat.bin
...
sys:mod! NAT principal-sort Nat
{
  ...
  signature {
    op s _ : Nat -> NzNat { demod }
    pred _ >= _ : Nat Nat  { demod }
    pred _ > _ : Nat Nat  { demod }
    pred _ <= _ : Nat Nat  { demod }
    pred _ < _ : Nat Nat  { demod }
    op _ * _ : Nat Nat -> Nat { assoc comm idr: 1 demod r-assoc }
    op _ + _ : Nat Nat -> Nat { assoc comm idr: 0 demod r-assoc }
    op sd : Nat Nat -> Nat { comm demod }
    op _ quo _ : Nat NzNat -> Nat { demod }
    op _ rem _ : Nat NzNat -> Nat { demod l-assoc }
    pred _ divides _ : NzNat Nat  { demod }
    op p _ : NzNat -> Nat { demod }
  }
  ...
}
```


Reading Modules from Files

```
CafeOBJ> input Stack.cobj
processing input : /usr2/schreine/.../Examples/Stack.cobj
-- defining module! STACK
-- reading in file  : nat
; Loading /usr3/cafeobj-1.4/lib/nat.bin
-- defining module! NAT
-- reading in file  : nznat
;   Loading /usr3/cafeobj-1.4/lib/nznat.bin
-- defining module! NZNAT......._..............._.........* done.
-- done reading in file: nznat
..._................_...............* done.
-- done reading in file: nat
........_..* done.
CafeOBJ> show STACK
module! STACK
  ...
```

Command input reads file with module definitions.

Evaluating Terms

```
CafeOBJ> open STACK
-- opening module STACK.. done.
%STACK> reduce top(pop(push(2, push(1, empty)))) .
-- reduce in %STACK : top(pop(push(2,push(1,empty))))
1 : NzNat
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
%STACK> reduce top(pop(push(1, empty))) .
-- reduce in %STACK : top(pop(push(1,empty)))
top(empty) : Nat
(0.000 sec for parse, 1 rewrites(0.000 sec), 2 matches)
%STACK> close
CafeOBJ>
```

Commands `open`/`close` enter/leave the context of a module; command `reduce` evaluates terms (note the period preceded by a blank).

Tracing Evaluations

```
%STACK> set trace on
%STACK> reduce top(pop(push(2, push(1, empty)))) .
-- reduce in %STACK : top(pop(push(2,push(1,empty))))
1>[1] rule: eq pop(push(N:Nat,S:Stack))
   = S
    { N:Nat |-> 2, S:Stack |-> push(1,empty) }
1<[1] pop(push(2,push(1,empty))) --> push(1,empty)
1>[2] rule: eq top(push(N:Nat,S:Stack))
   = N
    { N:Nat |-> 1, S:Stack |-> empty }
1<[2] top(push(1,empty)) --> 1
1 : NzNat
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
```

Command `set trace on` shows rules applied in the reduction.

Tracing Evaluations (Contd)

```
%STACK> set trace whole on
%STACK> reduce top(pop(push(2, push(1, empty)))) .
-- reduce in %STACK : top(pop(push(2,push(1,empty))))
1>[1] rule: eq pop(push(N:Nat,S:Stack))
   = S
    { N:Nat |-> 2, S:Stack |-> push(1,empty) }
1<[1] pop(push(2,push(1,empty))) --> push(1,empty)
[1]: top(pop(push(2,push(1,empty))))
---> top(push(1,empty))
1>[2] rule: eq top(push(N:Nat,S:Stack))
   = N
    { N:Nat |-> 1, S:Stack |-> empty }
1<[2] top(push(1,empty)) --> 1
[2]: top(push(1,empty))
---> 1
1 : NzNat
(0.000 sec for parse, 2 rewrites(0.010 sec), 2 matches)
```

Identifiers

Almost arbitrary strings may denote names of sorts and operators.

- Identifier `x-value`
  - Difference term `x - value`
- Identifier `x+value`
  - Sum term `x + value`
- Identifier `x*value`
  - Sum term `x * value`
- ...

Always use blanks around infix/mixfix operators.

Modules

- Every module introduces a name space.
  - Only entities declared in a module can be directly referenced within the module.
    - By their unqualified name.
  - Entities of other modules can be referenced by qualified names.
    - *name.module*: entity *name* in *module*.
- Other modules may be imported.
  - Remote entities become visible.
    - Can be referenced like local entities.
    - Ambiguities can be resolved by qualification with module name.
  - Imported modules are *not* duplicated.
    - Multiple imports of a module share the same model.

```
-- comments
module! name
{
  imports
  signature
  {
    sorts
    operators
  }
  axioms
  {
    equations
  }
}
```


Sorts

A signature may introduce one or more sorts.

```
[ sort1 sort2 ... ]
```

- Sequence of sort names separated by blanks.
  - Note the blanks after `[` and before `]`.
- Sorts may be partially ordered:

```
[ Nat < Int < Rat, Int < Float ]
```

  - *Subsort* `<` *Supersort*
  - Sort order is interpreted as set inclusion.

    The type checker considers values of the subsort also as values of the supersort.

The use of subsorts may simplify specifications considerably.

Operators

A signature may introduce "operators" (operations/constants).

```
op name : argument sorts -> result sort
op name : -> result sort
```

- Note the blanks around the tokens ":" and "->".
- Operators may be declared as infix/mixfix by the use of "_".

```
op _+_ : Nat Nat -> Nat
op _<_ : Nat Nat -> Bool
op if_then_else_fi: Bool Nat Nat -> Nat

2 + 3, 2 < 3, if N < M then N else M fi
```

- Multiple operators may be declared with the same arity.

```
op (_+_) (_*_) : Nat Nat -> Nat
```

- Operator names may be overloaded.

```
op _+_ : Nat Nat -> Nat -- addition
op _+_ : Set Set -> Set -- union
```

Predicates

Predicates are operators with target sort `Bool`

```
op   name : argument sorts -> Bool
pred name : argument sorts
```

- `pred` can be used as a shorthand for predicate declarations.

```
pred _<_ : Nat Nat
```

- The (in)equality predicate is implicitly defined on each sort.

```
pred _==_  : S S
pred _=/=_ : S S
```

  - Equality is defined in terms of evaluation.
    - `(`t `==` t'`)` `=` `true` iff t and t' evaluate to a common term.
  - Works correctly iff term rewriting system is Noetherian and confluent.

CafeOBJ considers predicates just as normal operators.

Axioms

Axioms declare variables and (conditional) equations.

```
var  name : sort
vars name1 name2 ... : sort
eq  term = term .
ceq term = term if boolean-term .
```

- Syntax pitfalls:
  - Note the blanks around the tokens ":" and "=".
  - Note the period "." preceded by a blank.
- Equations may be labeled:

```
var N : Nat
eq [ right-id ] :  N+0 = N
```

  - Labels are printed in reduction traces.

Equations of arbitrary shape are allowed but only especially constrained equations are used as reduction rules (to be discussed later).

Example

```
module! GCD
{
  protecting (INT)
  signature
  {
    op gcd : Int Int -> Int
  }
  axioms
  {
    vars N M : Int
    eq  gcd(N, 0) = N .
    eq  gcd(0, M) = M .
    ceq gcd(N, M) = gcd(N - M, M) if N >= M and M > 0 .
    ceq gcd(N, M) = gcd(N, M - N) if M >= N and N > 0 .
  }
}

%GCD> reduce gcd(15,12) .
-- reduce in %GCD : gcd(15,12)
3 : NzNat
(0.000 sec for parse, 45 rewrites(0.010 sec), 95 matches)
```


Context Variables

```
CafeOBJ> open GCD
-- opening module GCD.. done.
%GCD> let a = 15 .
-- setting let variable "a" to :
    15 : NzNat
%GCD> let b = 12 .
-- setting let variable "b" to :
    12 : NzNat
%GCD> show let
[bindings]
b = 12
a = 15
%GCD> reduce gcd(a,b) .
-- reduce in %GCD : gcd(15,12)
3 : NzNat
(0.000 sec for parse, 45 rewrites(0.000 sec), 95 matches)
```

Command `let` to bind variables in current module context.

Local Bindings

Unfortunately CafeOBJ does not support local bindings in a term.

- Abstract specification:

  $f(x, y) = \textbf{let}\ z = x * x\ \textbf{in}\ x + y * z$
- CafeOBJ:

```
eq f(x, y)      = f0(x, y, x * x)
eq f0(x, y, z) = x*y*z
```

Use auxiliary operators as a substitute for local bindings

Operator Attributes

There is a shorthand notation for some special axioms.

`op` *`name`* `:` *`argument sorts`* `->` *`result sort`* `{` *`attributes`* `}`

- Example: `op _+_ : S S -> S { assoc comm idem id:n }`

  Predicate == considers these operation attributes.
  - `assoc(iative)`: $x + (y + z) = (x + y) + z$
  - `comm(utative)`: $x + y = y + x$
  - `idem(potent)`: $x + x = x$
  - `id:n`: $x + n = x$
- Constructor attribute `constr`:

  Unused (treated as comment) by CafeOBJ.

```
op nil : -> List { constr }
op _<_ : List List -> List { constr }
```


Evaluating Terms

A tight module defines a term rewriting system.

- (Conditional) equations define (conditional) rewrite rules.
  - `eq` t `=` t' `.` defines $t \rightarrow t'$.
  - `eq` t `=` t' `if` b `.` defines $t \rightarrow t'$ with condition b.
- Also the rewrite rules of the imported modules are included.
  - Rewrite rules of module `BOOL` are always included.
- Equations must satisfy two constraints to become rewrite rules.
  1. Every variable on the righthand side of the equation (or in the condition) must occur on the left-hand side.
  2. The lefthand side must not be a single variable.

The term rewriting system is not necessarily Noetherian and confluent (i.e. reductions need not terminate, different reduction strategies may give different results).

Showing Rules

```
CafeOBJ> open STACK
-- opening module STACK.. done.
%STACK> show rules
 -- rewrite rules in module : %STACK
  1 : eq top(push(N,S)) = N
  2 : eq pop(push(N,S)) = S
%STACK> show all rules
 -- rewrite rules in module : %STACK
  1 : eq top(push(N,S)) = N
  2 : eq pop(push(N,S)) = S
  3 : eq [:BDEMOD] : sd(M:Nat,N:Nat) = #! (abs (- m n))
  4 : eq [:BDEMOD] : M:Nat + N:Nat = #! (+ m n)
  5 : eq [:BDEMOD] : N:Nat * 0 = 0
  6 : eq [:BDEMOD] : M:Nat quo NN:NzNat = #! (truncate m nn)
  7 : eq [:BDEMOD] : M:Nat rem NN:NzNat = #! (rem m nn)
 ...
```

Commands `show rules` and `show all rules`.

Evaluation Strategy

CafeOBJ supports various evaluation strategies.

- Default strategy: when evaluating a term $f(\ldots, a_i, \ldots)$,
  - first evaluate every a_i, for which there is a rewrite rule $f(\ldots, t_i, \ldots) \to \ldots$ with a non-variable term t_i in the position of a_i.
  - then evaluate the whole term $f(\ldots)$.
- Alternative strategy may be specified by attribute `strat: (`*`ints`*`)`
  - *ints* is a list of integers denoting argument positions.
  - Positive number denotes eager evaluation on corresponding argument.
  - Negative (or missing) number denotes lazy evaluation on argument.
  - 0 denotes evaluation of the the whole term.

```
op if_then_else_fi : Bool Int Int -> Int  { strat: (1 0) }
op _+_             : Int Int      -> Int  { strat: (1 2 0) }
op cons            : Elem List    -> List { strat: (0) }
```

The chosen strategy may affect the result/termination of the evaluation.

More Advanced Features

Further features of CafeOBJ.

- Term rewriting commands.
  - CafeOBJ may be used for term rewriting/induction proofs (see chapters 9 and 10 of the manual).
- Behavioral operators and behavioral equations.
  - Modeling object methods: an operator may have a special argument describing an "object" whose state is modified by the method.
- Transitions.
  - Non-symmetric relations between terms.
- Generalized module expressions:
  - Modules may be renamed and combined.
  - Modules may be parameterized.
  - Parameterized modules may be instantiated.

A powerful module concept is crucial for "specifying in the large".

Parameterized Modules

- A "loose module" is a named specification with loose semantics.

```
module* ELEM { signature { [ Elem ] } }
```

- May be used as the "type" of a parameter in a tight module.

```
module! STACK(E :: ELEM) {
  signature {
    [ Stack ]
    push : Elem.E Stack -> Stack
  }
}
```

- The parameter may be instantiated by a matching tight module.

```
view NATELEM from ELEM to NAT { sort Elem -> Nat }
module! NATSTACK
{
  -- introduces natural number stacks
  protecting (STACK(E <= NATELEM))
}
```

We are now going to present the theory of CafeOBJ-like specifications.