Algebraic specification and verification with CafeOBJ

Part 2 – Advanced topics

Norbert Preining

ESSLLI 2016 Bozen, August 2016

Solution to the exercises

EXERCISES

- Implement $\mathsf{factorial}(n) = n!$
- Implement $\mathsf{fib}(n)$, n-th Fibonacci number, where $\mathsf{fib}(0) = 0$, $\mathsf{fib}(1) = 1$, and $\mathsf{fib}(n) = \mathsf{fib}(n-2) + \mathsf{fib}(n-1)$ otherwise

Modules

- modules are the basic building blocks of CafeOBJ specifications, corresponding to (order-sorted) algebras

Modules

- modules are the basic building blocks of CafeOBJ specifications, corresponding to (order-sorted) algebras
- the are declared by either one of `mod!` `mod*` `mod`

Modules

- modules are the basic building blocks of CafeOBJ specifications, corresponding to (order-sorted) algebras
- the are declared by either one of `mod!` `mod*` `mod`
- difference of the three are the models that are considered:
  - `mod!`: initial models
  - `mod*`: all models
  - `mod`: undecided

Modules

- modules are the basic building blocks of CafeOBJ specifications, corresponding to (order-sorted) algebras
- the are declared by either one of `mod!` `mod*` `mod`
- difference of the three are the models that are considered:
  - `mod!`: initial models
  - `mod*`: all models
  - `mod`: undecided
- body of a module contains a specification of the algebra with axioms:
  - sorts and order on sorts
  - operators and their arity
  - variables and their sorts
  - equations (with or without conditions)

Anatomy of a module

start of a module and name	`mod! PNAT {`
definition of sorts and order	`[Nat]`
operator constant 0	`op 0 : -> Nat .`
normal prefix operator	`op s : Nat -> Nat .`
infix operator	`op _+_ : Nat Nat -> Nat .`
variable declaration	`vars X Y : Nat`
equation/axioms	`eq 0 + Y = Y .`
another equation	`eq s(X) + Y = s(X + Y) .`
end of the module	`}`

```
mod! PNAT {
  [Nat]
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat
  eq 0 + Y = Y .
  eq s(X) + Y = s(X + Y) .
}
```

Defining the first module

```
CafeOBJ> mod! PNAT {
  [Nat]
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat
  eq 0 + Y = Y .
  eq s(X) + Y = s(X + Y) .
}

-- defining module! PNAT
[....]
CafeOBJ>
```

Reducing a term

```
CafeOBJ> open PNAT .
-- opening module PNAT.. done.
%PNAT> red s(s(s(0))) + s(s(0)) .
-- reduce in %PNAT : (s(s(s(0))) + s(s(0))):Nat
(s(s(s(s(s(0)))))):Nat
(0.000 sec for parse, 4 rewrites(0.000 sec), 7 matches)
%PNAT> close
CafeOBJ>
```

Reducing a term

```
CafeOBJ> open PNAT .
-- opening module PNAT.. done.
%PNAT> red s(s(s(0))) + s(s(0)) .
-- reduce in %PNAT : (s(s(s(0))) + s(s(0))):Nat
(s(s(s(s(s(0)))))):Nat
(0.000 sec for parse, 4 rewrites(0.000 sec), 7 matches)
%PNAT> close
CafeOBJ>
```

Q How did this happen?

Trace a reduction

```
CafeOBJ> set trace whole on
CafeOBJ> open PNAT .
-- opening module PNAT.. done.
%PNAT> red s(s(s(0))) + s(s(0)) .
-- reduce in %PNAT : (s(s(s(0))) + s(s(0))):Nat
[1]: (s(s(s(0))) + s(s(0))):Nat
---> (s((s(s(0)) + s(s(0))))):Nat
[2]: (s((s(s(0)) + s(s(0))))):Nat
---> (s(s((s(0) + s(s(0)))))):Nat
[3]: (s(s((s(0) + s(s(0)))))):Nat
---> (s(s(s((0 + s(s(0))))))):Nat
[4]: (s(s(s((0 + s(s(0))))))):Nat
---> (s(s(s(s(s(0)))))):Nat
(s(s(s(s(s(0)))))):Nat
(0.000 sec for parse, 4 rewrites(0.000 sec), 7 matches)
%PNAT> close
CafeOBJ>
```

More on rewriting

Rewriting can be used in funny ways:

More on rewriting

Rewriting can be used in funny ways:

```
MOD! FOO {
  [ Elem ]
  op f : Elem -> Elem .
  var x : Elem
  eq f(x) = f(f(x)) .
}
```

More on rewriting

Rewriting can be used in funny ways:

```
MOD! FOO {
  [ Elem ]
  op f : Elem -> Elem .
  var x : Elem
  eq f(x) = f(f(x)) .
}
```

Q What will happen?

REWRITING FOO

```
CafeOBJ> open FOO .
%FOO> set trace whole on
%FOO> red f(3) .
-- reduce in %FOO : (f(3)):Nat
[1]: (f(3)):Nat
---> (f(f(3))):Nat
[2]: (f(f(3))):Nat
---> (f(f(f(3)))):Nat
[3]: (f(f(f(3)))):Nat
---> (f(f(f(f(3))))):Nat
[4]: (f(f(f(f(3))))):Nat
---> (f(f(f(f(f(3)))))):Nat
...
```

Term Rewriting & Termination

Term Rewrite System (TRS)

Definition

- pair of terms $\ell \to r$ is rewrite rule if $\ell \notin \mathcal{V}$ & $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(\ell)$
- **term rewrite system** (TRS) $\mathcal{R}$ is set of rewrite rules

Term Rewrite System (TRS)

Definition

- pair of terms $\ell \to r$ is rewrite rule if $\ell \notin \mathcal{V}$ & $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(\ell)$
- **term rewrite system** (TRS) $\mathcal{R}$ is set of rewrite rules
- rewrite step: $s \to_{\mathcal{R}} t$ if

  $$s = C[\ell\sigma] \text{ and } t = C[r\sigma]$$

  for some substitution σ, context C, and rule $\ell \to r \in \mathcal{R}$

NOTATIONS

$\mathcal{V}$ stands for set of all variables and $\mathcal{V}\mathsf{ar}(t)$ for variables in t

Example of TRS

TRS R

$$\begin{array}{rlrl}
\mathsf{add}(0, y) & \to y & \mathsf{mul}(0, y) & \to 0 \\
\mathsf{add}(\mathsf{s}(x), y) & \to \mathsf{s}(\mathsf{add}(x, y)) & \mathsf{mul}(\mathsf{s}(x), y) & \to \mathsf{add}(y, \mathsf{mul}(x, y))
\end{array}$$

Example of TRS

TRS R

$$\begin{aligned} \mathsf{add}(0,y) &\to y & \mathsf{mul}(0,y) &\to 0 \\ \mathsf{add}(\mathsf{s}(x),y) &\to \mathsf{s}(\mathsf{add}(x,y)) & \mathsf{mul}(\mathsf{s}(x),y) &\to \mathsf{add}(y,\mathsf{mul}(x,y)) \end{aligned}$$

rewrite sequence

$$\begin{aligned} \mathsf{mul}(\mathsf{s}(0),\mathsf{s}(0)) &\to_R \mathsf{add}(\mathsf{s}(0),\mathsf{mul}(\mathsf{s}(0),0)) \\ &\to_R \mathsf{add}(\mathsf{s}(0),0) \\ &\to_R \mathsf{s}(\mathsf{add}(0,0)) \\ &\to_R \mathsf{s}(0) \end{aligned}$$

Example of TRS

TRS R

$$\begin{aligned} \mathsf{add}(0, y) &\to y & \mathsf{mul}(0, y) &\to 0 \\ \mathsf{add}(\mathsf{s}(x), y) &\to \mathsf{s}(\mathsf{add}(x, y)) & \mathsf{mul}(\mathsf{s}(x), y) &\to \mathsf{add}(y, \mathsf{mul}(x, y)) \end{aligned}$$

rewrite sequence

$$\begin{aligned} \mathsf{mul}(\mathsf{s}(0), \mathsf{s}(0)) &\to_R \mathsf{add}(\mathsf{s}(0), \mathsf{mul}(\mathsf{s}(0), 0)) \\ &\to_R \mathsf{add}(\mathsf{s}(0), 0) \\ &\to_R \mathsf{s}(\mathsf{add}(0, 0)) \\ &\to_R \mathsf{s}(0) \end{aligned}$$

Definition

t is **normal form** if $t \to_R u$ for no u

Example of TRS

TRS R

$$\begin{aligned} \mathsf{add}(0, y) &\to y & \mathsf{mul}(0, y) &\to 0 \\ \mathsf{add}(\mathsf{s}(x), y) &\to \mathsf{s}(\mathsf{add}(x, y)) & \mathsf{mul}(\mathsf{s}(x), y) &\to \mathsf{add}(y, \mathsf{mul}(x, y)) \end{aligned}$$

rewrite sequence

$$\begin{aligned} \mathsf{mul}(\mathsf{s}(0), \mathsf{s}(0)) &\to_R \mathsf{add}(\mathsf{s}(0), \mathsf{mul}(\mathsf{s}(0), 0)) \\ &\to_R \mathsf{add}(\mathsf{s}(0), 0) \\ &\to_R \mathsf{s}(\mathsf{add}(0, 0)) \\ &\to_R \mathsf{s}(0) \end{aligned}$$

Definition

t is **normal form** if $t \to_R u$ for no u

Definition

R is **terminating** if there is no infinite sequence $t_1 \to_R t_2 \to_R \cdots$

Uniqueness of Normal Forms

Uniqueness of Normal Forms

Definition

- $t \to_R^* u$ if $t \to_R \cdots \to_R u$ (possibly no step)
- $t \to_R^! u$ if $t \to_R^* u$ and u is normal form
- $t{\downarrow_R}$ denotes normal form of t if there is exactly one normal form of t

Uniqueness of Normal Forms

Definition

- $t \to_R^* u$ if $t \to_R \cdots \to_R u$ (possibly no step)
- $t \to_R^! u$ if $t \to_R^* u$ and u is normal form
- $t{\downarrow_R}$ denotes normal form of t if there is exactly one normal form of t

(conditional) TRS R

$$\begin{aligned} \mathsf{f}(x, y) &\to x + y && \text{if } x \geqslant 50 \\ \mathsf{f}(x, y) &\to 0 && \text{if } y < 50 \end{aligned}$$

$\mathsf{f}(70, 30){\downarrow}$ is not well-defined:

$$100 \;{}_R^!{\leftarrow}\; \mathsf{f}(70, 30) \to_R^! 0$$

Uniqueness of Normal Forms

Definition

- $t \to_R^* u$ if $t \to_R \cdots \to_R u$ (possibly no step)
- $t \to_R^! u$ if $t \to_R^* u$ and u is normal form
- $t{\downarrow_R}$ denotes normal form of t if there is exactly one normal form of t

(conditional) TRS R

$$\begin{aligned} \mathsf{f}(x, y) &\to x + y && \text{if } x \geqslant 50 \\ \mathsf{f}(x, y) &\to 0 && \text{if } y < 50 \end{aligned}$$

$\mathsf{f}(70, 30){\downarrow}$ is not well-defined:

$$100 \; {}_R^!\!\!\leftarrow \mathsf{f}(70, 30) \to_R^! 0$$

REMARK

well-definedness requires uniqueness of normal forms

Two warnings

Termination

CafeOBJ does not check whether the generated rewrite system is terminating.

TWO WARNINGS

Termination

CafeOBJ does not check whether the generated rewrite system is terminating.

Confluence

CafeOBJ does not check for confluence.

Quiz

TRS R

$$\mathsf{append}(\mathsf{nil}, ys) \to ys$$
$$\mathsf{append}(x : xs, ys) \to x : \mathsf{append}(xs, ys)$$

Quiz

TRS R

$$\begin{aligned} \mathsf{append}(\mathsf{nil}, ys) &\to ys \\ \mathsf{append}(x : xs, ys) &\to x : \mathsf{append}(xs, ys) \end{aligned}$$

e.g.

$$\begin{aligned} \mathsf{append}(1 : 2 : 3 : \mathsf{nil}, 4 : 5 : \mathsf{nil}) &\to 1 : \mathsf{append}(2 : 3 : \mathsf{nil}, 4 : 5 : \mathsf{nil}) \\ &\to 1 : 2 : \mathsf{append}(3 : \mathsf{nil}, 4 : 5 : \mathsf{nil}) \\ &\to 1 : 2 : 3 : \mathsf{append}(\mathsf{nil}, 4 : 5 : \mathsf{nil}) \\ &\to 1 : 2 : 3 : 4 : 5 : \mathsf{nil} \end{aligned}$$

Quiz

TRS R

$$\begin{aligned} \mathsf{append}(\mathsf{nil}, ys) &\to ys \\ \mathsf{append}(x : xs, ys) &\to x : \mathsf{append}(xs, ys) \end{aligned}$$

e.g.

$$\begin{aligned} \mathsf{append}(1 : 2 : 3 : \mathsf{nil}, 4 : 5 : \mathsf{nil}) &\to 1 : \mathsf{append}(2 : 3 : \mathsf{nil}, 4 : 5 : \mathsf{nil}) \\ &\to 1 : 2 : \mathsf{append}(3 : \mathsf{nil}, 4 : 5 : \mathsf{nil}) \\ &\to 1 : 2 : 3 : \mathsf{append}(\mathsf{nil}, 4 : 5 : \mathsf{nil}) \\ &\to 1 : 2 : 3 : 4 : 5 : \mathsf{nil} \end{aligned}$$

Q is R terminating?

More on CafeOBJ

Built-in data types

```
[ NzNat < Nat < NzInt < Int < NzRat < Rat ]
```

Built-in data types

```
[ NzNat < Nat < NzInt < Int < NzRat < Rat ]

[ Triv Bool Float Char String ]
```

Built-in data types

```
[ NzNat < Nat < NzInt < Int < NzRat < Rat ]

[ Triv Bool Float Char String ]

[ 2Tuple 3Tuple 4Tuple ]
```

Built-in data types

```
[ NzNat < Nat < NzInt < Int < NzRat < Rat ]

[ Triv Bool Float Char String ]

[ 2Tuple 3Tuple 4Tuple ]
```

plus records

Number tower examples

```
open NAT .
 red 10 + 20 .
 red 32 * 57 .
 -- operator precedence, see later
 red 2 + 3 * 4 .
 -- what will we get here?
 red 7 - 3 .
close
```

Number tower examples

```
open INT .
 red 7 - 3 .
 red 3 - 9 .
 -- operator precedence (see later)
 red 3 + 5 * 7 .
 -- what will we get here?
 red 3 / 5 .
close
```

Number tower examples

```
open RAT .
 parse 3 / 5 .
 red 3 / 5 + 1 / 2 .
 -- what will we get here?
 red sqrt(2) .
close
```

Operator definitions

prefix (default)

```
op f : Nat NzNat -> Nat .
```

Operator definitions

prefix (default)

```
op f : Nat NzNat -> Nat .
```

mixfix (useful, but can be dangerous)

```
op _+_ : Int Int -> Int .
```

Operator definitions

prefix (default)

```
op f : Nat NzNat -> Nat .
```

mixfix (useful, but can be dangerous)

```
op _+_ : Int Int -> Int .
op <<___>> : Nat Nat Nat -> Nat .
```

Operator definitions

prefix (default)

```
op f : Nat NzNat -> Nat .
```

mixfix (useful, but can be dangerous)

```
op _+_ : Int Int -> Int .
op <<___>> : Nat Nat Nat -> Nat .
op if_then_else_fi : Bool Nat Nat -> Nat .
```

Operator definitions

prefix (default)

```
op f : Nat NzNat -> Nat .
```

mixfix (useful, but can be dangerous)

```
op _+_ : Int Int -> Int .
op <<___>> : Nat Nat Nat -> Nat .
op if_then_else_fi : Bool Nat Nat -> Nat .
eq if ... = ?
```

Operator definitions

prefix (default)

```
op f : Nat NzNat -> Nat .
```

mixfix (useful, but can be dangerous)

```
op _+_ : Int Int -> Int .
op <<___>> : Nat Nat Nat -> Nat .
op if_then_else_fi : Bool Nat Nat -> Nat .
eq if ... = ?
```

Warning

mixfix operators can create difficult to parse terms, sometimes proper qualification of terms is necessary

Equational theory attributes

associativity, commutativity, identity, idempotence

```
op _&_ : Bool Bool -> Bool { assoc comm idem id: true }
```

Equational theory attributes

associativity, commutativity, identity, idempotence

```
op _&_ : Bool Bool -> Bool { assoc comm idem id: true }
```

```
mod* GROUP {
 [ G ]
 op 0 : -> G .
 op _+_ : G G -> G { assoc } .
 op -_ : G -> G .
 var X : G .
 eq[0left] : 0 + X = X .
 eq[neginv] : (- X) + X = 0 .
}
```

Equational theory attributes

associativity, commutativity, identity, idempotence

```
op _&_ : Bool Bool -> Bool { assoc comm idem id: true }
```

```
mod* GROUP {
 [ G ]
 op 0 : -> G .
 op _+_ : G G -> G { assoc } .
 op -_ : G -> G .
 var X : G .
 eq[0left] : 0 + X = X .
 eq[neginv] : (- X) + X = 0 .
}
```

!!! inherited

Parsing attributes

precedence, associativity

```
op _+_ : Int Int -> Int { prec: 33 } .
op _*_ : Int Int -> Int { prec: 31 } .
```

effect: $*$ binds stronger than $+$.

Parsing attributes

precedence, associativity

```
op _+_ : Int Int -> Int { prec: 33 } .
op _*_ : Int Int -> Int { prec: 31 } .
```

effect: $*$ binds stronger than $+$.

```
op _+_ : S S -> S { l-assoc } .
```

reduces `X + X + X` to `(X + X) + X`.

Modules import

Importing modules imports the declarations.
Three different modes:

protecting (pr) `pr(NAT)`
: all intended models are preserved as they are

extending (ex) `ex(BOOL)`
: models can be inflated, but cannot collapse

including (inc) `inc(INT)`
: no restrictions on models

using (us) `us(FLOAT)`
: allows for total destruction (redefinition)

Modules import

Importing modules imports the declarations.
Three different modes:

protecting (pr) `pr(NAT)`
: all intended models are preserved as they are

extending (ex) `ex(BOOL)`
: models can be inflated, but cannot collapse

including (inc) `inc(INT)`
: no restrictions on models

using (us) `us(FLOAT)`
: allows for total destruction (redefinition)

Most use cases: `pr(NAT)` or `ex(NAT)`.

Lists

Lists

- lists over $\mathbb{N}$ are terms given by BNF

$$L \ ::= \ \underbrace{\mathsf{nil}}_{\text{empty list}} \ \mid \ \underbrace{x \mid L}_{\text{cons}} \qquad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list

Lists

- lists over $\mathbb{N}$ are terms given by BNF

$$L \ ::= \ \underbrace{\mathsf{nil}}_{\text{empty list}} \quad \mid \ \underbrace{x \mid L}_{\text{cons}} \qquad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list
- $1 \mid (3 \mid (2 \mid \mathsf{nil}))$ — list

Lists

- lists over $\mathbb{N}$ are terms given by BNF

$$L \ ::= \ \underbrace{\mathsf{nil}}_{\text{empty list}} \ \mid \ \underbrace{x \mid L}_{\text{cons}} \qquad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list
- $1 \mid (3 \mid (2 \mid \mathsf{nil}))$ — list
- $1 \mid 3 \mid 2 \mid \mathsf{nil}$ — list

Lists

- lists over $\mathbb{N}$ are terms given by BNF

$$L \;::=\; \underbrace{\mathsf{nil}}_{\text{empty list}} \;\mid\; \underbrace{x \mid L}_{\text{cons}} \qquad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list
- $1 \mid (3 \mid (2 \mid \mathsf{nil}))$ — list
- $1 \mid 3 \mid 2 \mid \mathsf{nil}$ — list
- $1 \mid 3 \mid 2$ — not list

LISTS

- lists over $\mathbb{N}$ are terms given by BNF

$$L \ ::= \ \underbrace{\mathsf{nil}}_{\text{empty list}} \ \mid \ \underbrace{x \mid L}_{\text{cons}} \qquad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list
- $1 \mid (3 \mid (2 \mid \mathsf{nil}))$ — list
- $1 \mid 3 \mid 2 \mid \mathsf{nil}$ — list
- $1 \mid 3 \mid 2$ — not list
- $(1 \mid 3) \mid 2 \mid \mathsf{nil}$ — not list

LISTS

- lists over $\mathbb{N}$ are terms given by BNF

$$L \; ::= \underbrace{\mathsf{nil}}_{\text{empty list}} \;\mid\; \underbrace{x \mid L}_{\text{cons}} \qquad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list
- $1 \mid (3 \mid (2 \mid \mathsf{nil}))$ — list
- $1 \mid 3 \mid 2 \mid \mathsf{nil}$ — list
- $1 \mid 3 \mid 2$ — not list
- $(1 \mid 3) \mid 2 \mid \mathsf{nil}$ — not list
- $1 \mid \mathsf{true} \mid 3 \mid \mathsf{nil}$ — not list

Lists in CafeOBJ

lists can be defined as sorted terms over $\overbrace{\text{constructor symbols}}^{\text{functions as values}}$:

$$\mathsf{nil} : \mathsf{NatList} \quad \text{and} \quad \mathsf{\_|\_} : \mathsf{Nat} \times \mathsf{NatList} \to \mathsf{NatList}$$

```
mod! NATLIST {
 pr(NAT)
 [ NatList ]
 op nil : -> NatList {constr} .
 op _|_ : Nat NatList -> NatList {constr} .
}

open NATLIST .
 red 1 | 2 | 3 | 4 | nil .
close
```

Length

$$
\begin{aligned}
\mathsf{len}(\mathsf{nil}) &= 0\\
\mathsf{len}(3 \mid \mathsf{nil}) &= 1\\
\mathsf{len}(2 \mid 3 \mid \mathsf{nil}) &= 2\\
\mathsf{len}(1 \mid 2 \mid 3 \mid \mathsf{nil}) &= 3
\end{aligned}
$$

```
op len : NatList -> Nat

eq len(nil) = ?
eq len(E:Nat | L:NatList) = ?
```

APPEND

$$\mathsf{nil}@(3 \mid 4 \mid \mathsf{nil}) = 3 \mid 4 \mid \mathsf{nil}$$
$$(2 \mid \mathsf{nil})@(3 \mid 4 \mid \mathsf{nil}) = 2 \mid 3 \mid 4 \mid \mathsf{nil}$$
$$(1 \mid 2 \mid \mathsf{nil})@(3 \mid 4 \mid \mathsf{nil}) = 1 \mid 2 \mid 3 \mid 4 \mid \mathsf{nil}$$

```
mod* NATLIST@ {
 pr(NATLIST)
 var E : Nat
 vars L1 L2 : NatList
 op _@_ : NatList NatList -> NatList

 eq nil @ L2 = ?
 eq (E | L1) @ L2 = ?
}
```

Reusing data

Association Lists

association lists are

- lists of pairs: $(x_1, y_1) \mid \cdots \mid (x_n, y_n) \mid \mathsf{nil}$

Association Lists

association lists are

- lists of pairs: $(x_1, y_1) \mid \cdots \mid (x_n, y_n) \mid \mathsf{nil}$
- equipped with lookup function

$$\mathsf{l} = (\texttt{"Kanazawa"}, 921) \mid (\texttt{"Nomi"}, 923) \mid \mathsf{nil}$$

$$\begin{aligned} \mathsf{lookup}(\texttt{"Kanazawa"}, \mathsf{l}) &= 921 \\ \mathsf{lookup}(\texttt{"Nomi"}, \mathsf{l}) &= 923 \\ \mathsf{lookup}(\texttt{"Hakusan"}, \mathsf{l}) &= \texttt{not-found} \end{aligned}$$

Association Lists

association lists are

- lists of pairs: $(x_1, y_1) \mid \cdots \mid (x_n, y_n) \mid \mathsf{nil}$
- equipped with lookup function

$$\mathsf{l} = (\texttt{"Kanazawa"}, 921) \mid (\texttt{"Nomi"}, 923) \mid \mathsf{nil}$$

$$\begin{aligned}
\mathsf{lookup}(\texttt{"Kanazawa"}, \mathsf{l}) &= 921 \\
\mathsf{lookup}(\texttt{"Nomi"}, \mathsf{l}) &= 923 \\
\mathsf{lookup}(\texttt{"Hakusan"}, \mathsf{l}) &= \texttt{not-found}
\end{aligned}$$

Q

- what would be the signature of data constructors and lookup?
- how would one define lookup?
- implementation?

Parametrized Modules

- `mod!` $M(\overbrace{X}^{\text{variable}} \ \texttt{::}\ \overbrace{C}^{\text{module constraint}}, \ldots)\ \{\cdots\ f.X\ \cdots\}$ parametrized module

Parametrized Modules

- mod! $M(\overbrace{X}^{\text{variable}} :: \overbrace{C}^{\text{module constraint}}, \ldots)\ \{\cdots\ f.X\ \cdots\}$ parametrized module
- $M(N\overbrace{\texttt{\{sort A -> B,op f -> g,}\ldots\texttt{\}}}^{\text{view}})$ module instantiation

Parametrized Modules

- $\texttt{mod!}\ M(\overbrace{X}^{\text{variable}} :: \overbrace{C}^{\text{module constraint}}, \ldots)\ \{\cdots\ f.X\ \cdots\}$ parametrized module
- $M(N\overbrace{\{\texttt{sort A -> B, op f -> g}, \ldots\}}^{\text{view}})$ module instantiation

```
mod* C {
  [A]
  op add : A A -> A .
}
```

Parametrized Modules

- $\texttt{mod!}\ M(\overbrace{X}^{\text{variable}} :: \overbrace{C}^{\text{module constraint}}, \ldots)\ \{\cdots\ f.X\ \cdots\}$ parametrized module
- $M(N\overbrace{\{\texttt{sort A -> B,op f -> g,}\ldots\}}^{\text{view}})$ module instantiation

```
mod* C {
  [A]
  op add : A A -> A .
}
mod! TWICE(X :: C) {
  op twice : A.X -> A.X .
  eq twice(E:A.X) = add.X(E,E) .
}
```

Parametrized Modules

- mod! $M(\overbrace{X}^{\text{variable}} :: \overbrace{C}^{\text{module constraint}}, \ldots) \{\cdots f.X \cdots\}$ parametrized module
- $M(N\overbrace{\{\texttt{sort A -> B, op f -> g}, \ldots\}}^{\text{view}})$ module instantiation

```
mod* C {
  [A]
  op add : A A -> A .
}

mod! TWICE(X :: C) {
  op twice : A.X -> A.X .
  eq twice(E:A.X) = add.X(E,E) .
}

open TWICE(NAT { sort A -> Nat, op add -> _+_ })
  red twice(10) . - -> 10 + 10 -> 20
close
```

Views and Module Instantiations

all are same:

- `open TWICE(NAT { sort A -> Nat, op add -> _+_ })`
- ```
  view C2NAT from C to NAT {
    sort A -> Nat
    op add -> _+_
  }
  ```

  `open TWICE(C2NAT)`
- `open TWICE(X <= C2NAT)`
  ```

Views and Module Instantiations

all are same:

- `open TWICE(NAT { sort A -> Nat, op add -> _+_ })`
- ```
  view C2NAT from C to NAT {
    sort A -> Nat
    op add -> _+_
  }
  ```

  `open TWICE(C2NAT)`
- `open TWICE(X <= C2NAT)`

All describe a homomorphism from the parameter algebra to the instantiation algebra
  ```

Views and Module Instantiations

all are same:

- `open TWICE(NAT { sort A -> Nat, op add -> _+_ })`
- ```
  view C2NAT from C to NAT {
    sort A -> Nat
    op add -> _+_
  }
  ```

  ```
  open TWICE(C2NAT)
  ```
- `open TWICE(X <= C2NAT)`

All describe a homomorphism from the parameter algebra to the instantiation algebra

WARNING That is a homomorphism of multi-sorted algebra, thus sorts and operators have to be translated.
  ```

Parametrized Lists

`TRIV` consists of only sort `Elt`; see `show TRIV`

Parametrized Lists

`TRIV` consists of only sort `Elt`; see `show TRIV`

```
mod! LIST(X :: TRIV) {
```

Parametrized Lists

TRIV consists of only sort Elt; see show TRIV

```
mod! LIST(X :: TRIV) {
  [List]
  op nil : -> List              {constr}
  op _|_ : Elt.X List -> List {constr}
  op _@_ : List List -> List
```

Parametrized Lists

TRIV consists of only sort Elt; see show TRIV

```
mod! LIST(X :: TRIV) {
  [List]
  op nil : -> List              {constr}
  op _|_ : Elt.X List -> List {constr}
  op _@_ : List List -> List

  var E : Elt.X
  vars L1 L2 : List
```

Parametrized Lists

TRIV consists of only sort Elt; see show TRIV

```
mod! LIST(X :: TRIV) {
  [List]
  op nil : -> List              {constr}
  op _|_ : Elt.X List -> List {constr}
  op _@_ : List List -> List

  var E : Elt.X
  vars L1 L2 : List

  eq nil @ L2 = L2 .
  eq (E | L1) @ L2 = E | (L1 @ L2) .
}
```

Parametrized Lists

TRIV consists of only sort Elt; see show TRIV

```
mod! LIST(X :: TRIV) {
  [List]
  op nil : -> List              {constr}
  op _|_ : Elt.X List -> List {constr}
  op _@_ : List List -> List

  var E : Elt.X
  vars L1 L2 : List

  eq nil @ L2 = L2 .
  eq (E | L1) @ L2 = E | (L1 @ L2) .
}
```

USAGE

- mod! NATLIST { pr(LIST(NAT {sort Elt -> Nat})) }, or

PARAMETRIZED LISTS

TRIV consists of only sort Elt; see show TRIV

```
mod! LIST(X :: TRIV) {
  [List]
  op nil : -> List               {constr}
  op _|_ : Elt.X List -> List {constr}
  op _@_ : List List -> List

  var E : Elt.X
  vars L1 L2 : List

  eq nil @ L2 = L2 .
  eq (E | L1) @ L2 = E | (L1 @ L2) .
}
```

USAGE

- `mod! NATLIST { pr(LIST(NAT {sort Elt -> Nat})) }`, or
- `mod! NATLIST { pr(LIST(NAT)) }`
  ☞ Elt is automatically identified if module contains only one sort

Renaming of instances

Assume

```
mod! SUPERMODULE {
 pr(LIST(NAT {sort Elt -> Nat}))
 pr(LIST(INT {sort Elt -> Int}))
}
open SUPERMODULE .
 check regularity
...
```

Renaming of instances

Assume

```
mod! SUPERMODULE {
 pr(LIST(NAT {sort Elt -> Nat}))
 pr(LIST(INT {sort Elt -> Int}))
}
open SUPERMODULE .
 check regularity
...
```

Why? –

Renaming of instances

Assume

```
mod! SUPERMODULE {
 pr(LIST(NAT {sort Elt -> Nat}))
 pr(LIST(INT {sort Elt -> Int}))
}
open SUPERMODULE .
 check regularity
...
```

Why? – Instantiation is a homomorphism from `C` to target module. But the "generated module" is called in both cases `LIST`.

Renaming of instances

Assume

```
mod! SUPERMODULE {
 pr(LIST(NAT {sort Elt -> Nat}))
 pr(LIST(INT {sort Elt -> Int}))
}
open SUPERMODULE .
 check regularity
...
```

Why? – Instantiation is a homomorphism from `C` to target module.
But the "generated module" is called in both cases `LIST`.
Solution: Add another "renaming" isomorphism at the end.

Renaming of instances cont.

```
mod! SUPERMODULE {
 pr(LIST(NAT {sort Elt -> Nat})
            * { sort List -> NatList,
               op nil -> natnil,
               op _|_ -> _||_ })
 pr(LIST(INT {sort Elt -> Int})
            * { sort List -> IntList })
}
```

Renaming of instances cont.

```
mod! SUPERMODULE {
 pr(LIST(NAT {sort Elt -> Nat})
          * { sort List -> NatList,
              op nil -> natnil,
              op _|_ -> _||_ })
 pr(LIST(INT {sort Elt -> Int})
          * { sort List -> IntList })
}
```

The isomorphism renames
`<List, nil, |>` $\mapsto$ `<NatList, natnil, ||>`:

Renaming of instances cont.

```
mod! SUPERMODULE {
 pr(LIST(NAT {sort Elt -> Nat})
           * { sort List -> NatList,
              op nil -> natnil,
              op _|_ -> _||_ })
 pr(LIST(INT {sort Elt -> Int})
           * { sort List -> IntList })
}
```

The isomorphism renames
`<List, nil, |>` $\mapsto$ `<NatList, natnil, ||>`:

```
%SUPERMODULE> parse 3 || 4 || 7 || 1 || natnil .
(3 || (4 || (7 || (1 || natnil)))):NatList
%SUPERMODULE> parse 3 | 4 | 7 | 1 | nil .
(3 | (4 | (7 | (1 | nil)))):IntList
%SUPERMODULE> parse 3 | 4 | 7 | 1 | natnil .
[Error] no successful parse
...
```

Association Lists Revisited

`2TUPLE(X1 :: TRIV, X2 :: TRIV)` is parametrized module for pairs

Association Lists Revisited

`2TUPLE(X1 :: TRIV, X2 :: TRIV)` is parametrized module for pairs

Quiz

```
mod! ALIST(K :: TRIV, V :: TRIV) {
  pr(LIST(2TUPLE(K, V) {sort Elt -> 2Tuple}))
  [         ?         ]
  op not-found : -> NotFound .
  op lookup : Elt.K List -> Value&NotFound .
  vars X1 X2 :     ?     .
  var Y : Elt.V .
  var L : List .
  eq lookup(X1, nil) = not-found .
  eq lookup(X1, « X2 ; Y » | L) =
    if X1 == X2 then Y else lookup(X1, L) fi .
}
```

Proving

PROOF SCORES

- proofs of properties by reducing them to `true` (e.g.)
- usually written between `open` and `close`
  statements between the two are temporary and are lost after the `close` (temporary module)
- usually several modules plus several blocks of open-close

Proof scores

- proofs of properties by reducing them to `true` (e.g.)
- usually written between `open` and `close`
  statements between the two are temporary and are lost after the `close` (temporary module)
- usually several modules plus several blocks of open-close

Examples

- $x + (-x) = 0$ in group theory
- Associativity of $+$ in PNAT

Group theory

group-theory1.cafe

```
mod* GROUP {
  [ G ]
  op 0 : -> G .
  op _+_ : G G -> G { assoc } .
  op -_ : G -> G .
  var X : G .
  eq[0left] : 0 + X = X .
  eq[neginv] : (- X) + X = 0 .
}
open GROUP .
  op a : -> G .
  red a + ( - a ) .
close
```

Group theory

group-theory1.cafe

```
mod* GROUP {
 [ G ]
 op 0 : -> G .
 op _+_ : G G -> G { assoc } .
 op -_ : G -> G .
 var X : G .
 eq[0left] : 0 + X = X .
 eq[neginv] : (- X) + X = 0 .
}
open GROUP .
 op a : -> G .
 red a + ( - a ) .
close
```

…would be nice – but does not work

Group theory cont.

Why?

Group theory cont.

WHY? Let us try to give a proof – can you do it?

Group theory cont.

Why? Let us try to give a proof – can you do it? Assume we have

$$0 + a = a \tag{1}$$

$$-a + a = 0 \tag{2}$$

$$\begin{aligned}
a + -a &= 0 + a + -a && \text{by (1) right-to-left} \\
&= --a + -a + a + -a && \text{by (2) right-to-left} \\
&= --a + 0 + -a && \text{by (2)} \\
&= --a + -a && \text{by (1)} \\
&= 0 && \text{by (2)}
\end{aligned}$$

Group theory cont.

WHY? Let us try to give a proof – can you do it? Assume we have

$$0 + a = a \tag{1}$$

$$-a + a = 0 \tag{2}$$

$$\begin{aligned} a + -a &= 0 + a + -a && \text{by (1) right-to-left} \\ &= --a + -a + a + -a && \text{by (2) right-to-left} \\ &= --a + 0 + -a && \text{by (2)} \\ &= --a + -a && \text{by (1)} \\ &= 0 && \text{by (2)} \end{aligned}$$

Why did it not work in CafeOBJ?

Group theory – better proof score

group-theory2.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (0) .
 apply -.neginv with X = - a at [1] .
 apply reduce at term .
close
```

Group theory – better proof score

group-theory2.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (0) .
 apply -.neginv with X = - a at [1] .
 apply reduce at term .
close
```

Still not there – why?

Group theory – even better proof score

group-theory3.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (1) .
 apply -.neginv with X = - a at [1] .
 apply +.neginv with X = a at [2 .. 3] .
 apply reduce at term .
close
```

Group theory – even better proof score

group-theory3.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (1) .
 apply -.neginv with X = - a at [1] .
 apply +.neginv with X = a at [2 .. 3] .
 apply reduce at term .
close
```

Where can we go from here?

Group theory – even better proof score

group-theory3.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (1) .
 apply -.neginv with X = - a at [1] .
 apply +.neginv with X = a at [2 .. 3] .
 apply reduce at term .
close
```

Where can we go from here?
Prove that 0 is also right inverse

0 IS RIGHT INVERSE

group-theory4.cafe

```
open GROUP .
 op a : -> G .
 -- we have proven the following equation
 -- so we can add it
 eq[invneg] : a + ( - a ) = 0 .
 start a + 0 .
 apply -.neginv with X = a at (2) .
 apply +.invneg at [1 .. 2] .
 apply reduce at term .
 -- and we get a, so (a + 0) = a
close
```

Associativity of $+$ in PNAT

ASSOCIATIVITY OF +

Recall PNAT

```
mod! PNAT {
 [Nat]
 op 0 : -> Nat .
 op s : Nat -> Nat .
 op _+_ : Nat Nat -> Nat .
 vars X Y : Nat
 eq 0 + Y = Y .
 eq s(X) + Y = s(X + Y) .
}
```

Mathematical proof

Assume that $0 + y = y$ and $s(x) + y = s(x + y)$ for all x and y.
How do we show that $(x + y) + z = x + (y + y)$ for all x, y, and z?

Mathematical proof

Assume that $0 + y = y$ and $s(x) + y = s(x + y)$ for all x and y.
How do we show that $(x + y) + z = x + (y + y)$ for all x, y, and z?
Proof by induction:

Induction base

Show that $(0 + y) + z = 0 + (y + z)$

Mathematical proof

Assume that $0 + y = y$ and $s(x) + y = s(x + y)$ for all x and y.
How do we show that $(x + y) + z = x + (y + y)$ for all x, y, and z?
Proof by induction:

Induction base

Show that $(0 + y) + z = 0 + (y + z)$

Induction step

Show that if $(x + y) + z = x + (y + z)$, then also
$(s(x) + y) + z = s(x) + (y + z)$.

Formal proof in CafeOBJ

```
mod ADD-ASSOC {
 pr(PNAT)
 -- theorem of constants, denote arbitrary values
 ops x y z : -> Nat .
 op addassoc : Nat Nat Nat -> Bool .
 vars X Y Z : Nat
 eq addassoc(X,Y,Z) = ((X + Y) + Z == X + (Y + Z)) .
}
```

Formal proof in CafeOBJ

```
mod ADD-ASSOC {
 pr(PNAT)
 -- theorem of constants, denote arbitrary values
 ops x y z : -> Nat .
 op addassoc : Nat Nat Nat -> Bool .
 vars X Y Z : Nat
 eq addassoc(X,Y,Z) = ((X + Y) + Z == X + (Y + Z)) .
}
```

Induction base

```
open ADD-ASSOC .
 red addassoc(0,y,z) .
close
```

Checking induction base

```
CafeOBJ> set trace whole on
CafeOBJ> open ADD-ASSOC .
%ADD-ASSOC> red addassoc(0,y,z) .
-- reduce in %ADD-ASSOC : (addassoc(0,y,z)):Bool
[1]: (addassoc(0,y,z)):Bool
---> (((0 + y) + z) == (0 + (y + z))):Bool
[2]: (((0 + y) + z) == (0 + (y + z))):Bool
---> ((y + z) == (0 + (y + z))):Bool
[3]: ((y + z) == (0 + (y + z))):Bool
---> ((y + z) == (y + z)):Bool
[4]: ((y + z) == (y + z)):Bool
---> (true):Bool
(true):Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 12 matches)
%ADD-ASSOC> close
CafeOBJ>
```

Checking induction step

```
CafeOBJ> set trace whole off
CafeOBJ> open ADD-ASSOC .
%ADD-ASSOC> red addassoc(x,y,z) implies
              addassoc(s(x),y,z) .
-- reduce in %ADD-ASSOC : (addassoc(x,y,z) implies addassoc(s
    (x),y,z)):Bool
(true):Bool
(0.000 sec for parse, 11 rewrites(0.000 sec), 50 matches)
%ADD-ASSOC> close
CafeOBJ>
```

End of the proof

Observational Transition Systems

System specification with OTS

- describe the system as state machine (automaton)

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states
- find an invariant of transitions that guarantees the target property

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states
- find an invariant of transitions that guarantees the target property
- base case of induction
  - find a finite set of covering state descriptions
  - show for those that if a state is initial then the invariant property holds

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states
- find an invariant of transitions that guarantees the target property
- base case of induction
  - find a finite set of covering state descriptions
  - show for those that if a state is initial then the invariant property holds
- step case of induction
  - find again a finite set of covering state descriptions for the left hand sides of the transitions
  - show that if the lhs of the transition satisfies the invariant condition, then also the rhs.

CloudSync

CloudSync in images

Cloud	state	idle
	stamp	n

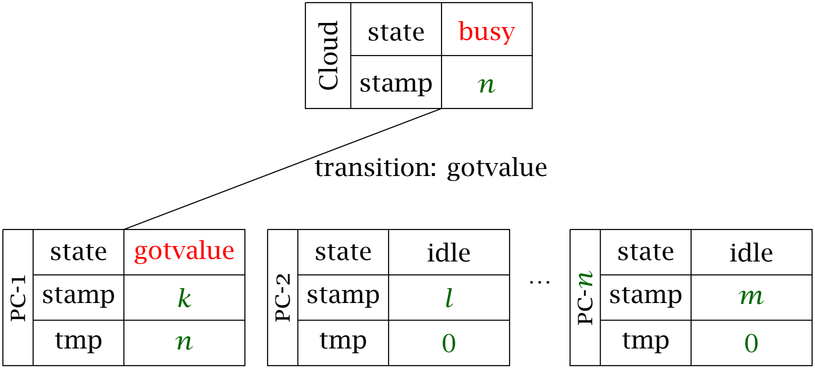
PC-1	state	idle
	stamp	k
	tmp	0

PC-2	state	idle
	stamp	l
	tmp	0

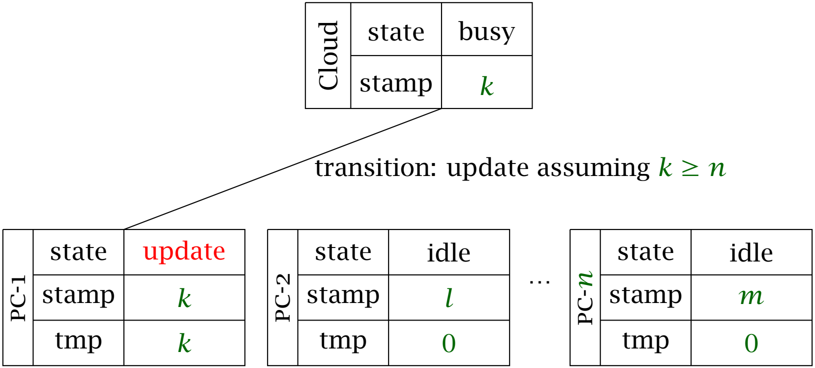
…

PC-n	state	idle
	stamp	m
	tmp	0

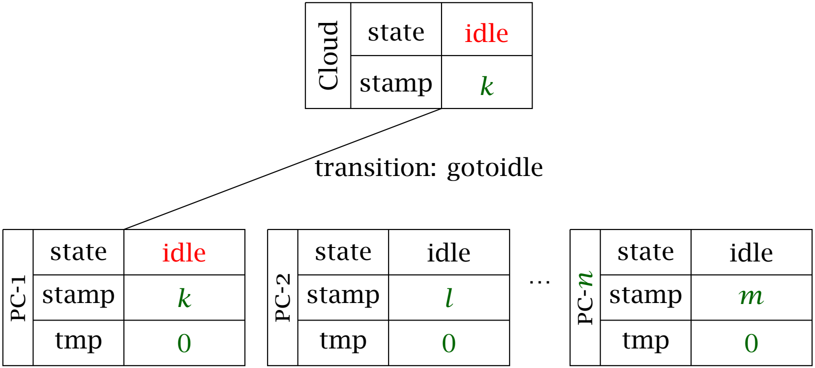
CloudSync in images



CloudSync in images



CloudSync in images



SPECIFICATION

ClLabel: $\{\text{idlecl}, \text{busy}\}$

```
mod! CLLABEL {
 [ClLabelLt < ClLabel]
 ops idlecl busy : -> ClLabelLt {constr} .
 eq (L1:ClLabelLt = L2:ClLabelLt) = (L1 == L2) .
}
```

Specification

ClLabel: $\{idlecl, busy\}$
PcLabel: $\{idlepc, gotvalue, updated\}$

```
mod! PCLABEL {
  [PcLabelLt < PcLabel]
  ops idlepc gotvalue updated : -> PcLabelLt {constr} .
  eq (L1:PcLabelLt = L2:PcLabelLt) = (L1 == L2) .
}
```

Specification

ClLabel: $\{\text{idlecl}, \text{busy}\}$
PcLabel: $\{\text{idlepc}, \text{gotvalue}, \text{updated}\}$
ClState: $\text{ClLabel} \times \mathbb{N}$

```
mod! CLSTATE {
 pr(PAIR(NAT, CLLABEL{sort Elt -> ClLabel})*{
      sort Pair -> ClState, op fst -> fst.clstate,
      op snd -> snd.clstate })
}
```

Specification

ClLabel: $\{\text{idlecl}, \text{busy}\}$
PcLabel: $\{\text{idlepc}, \text{gotvalue}, \text{updated}\}$
ClState: $\text{ClLabel} \times \mathbb{N}$
PcState: $\text{PcLabel} \times \mathbb{N} \times \mathbb{N}$

```
mod! PCSTATE {
 pr(3TUPLE(NAT, NAT,
           PCLABEL{sort Elt -> PcLabel})*
             {sort 3Tuple -> PcState})
}
```

Specification

ClLabel:	$\{\text{idlecl}, \text{busy}\}$
PcLabel:	$\{\text{idlepc}, \text{gotvalue}, \text{updated}\}$
ClState:	$\text{ClLabel} \times \mathbb{N}$
PcState:	$\text{PcLabel} \times \mathbb{N} \times \mathbb{N}$
PcStates:	$\text{MultiSet}(\text{PcState})$

```
mod! PCSTATES {
 pr(MULTISET(PCSTATE{sort Elt -> PcState})*
     {sort MultiSet -> PcStates})
}
```

Specification

ClLabel:	$\{\text{idlecl}, \text{busy}\}$
PcLabel:	$\{\text{idlepc}, \text{gotvalue}, \text{updated}\}$
ClState:	$\text{ClLabel} \times \mathbb{N}$
PcState:	$\text{PcLabel} \times \mathbb{N} \times \mathbb{N}$
PcStates:	$\text{MultiSet}(\text{PcState})$
State:	$\text{ClState} \times \text{PcStates}$

```
mod! STATE {
 pr(PAIR(CLSTATE{sort Elt -> ClState},PCSTATES
    {sort Elt -> PcStates})*{sort Pair -> State})
}
```

Transitions

GetValue: if PC and Cloud is idle, fetch Cloud value

TRANSITIONS

GetValue: if PC and Cloud is idle, fetch Cloud value

```
mod! GETVALUE { pr(STATE)
 trans[getvalue]:
   <
    < ClVal:Nat , idlecl > ,
    ( <<PcVal:Nat; OldClVal:Nat; idlepc>> S:PcStates)
   > =>
   <
    < ClVal , busy > ,
    ( <<PcVal; ClVal; gotvalue>> S)
   > .
}
```

Transitions

GetValue:	if PC and Cloud is idle, fetch Cloud value
Update:	update Cloud/PC according to larger value

Transitions

GetValue: if PC and Cloud is idle, fetch Cloud value
Update: update Cloud/PC according to larger value

```
mod! UPDATE { pr(STATE)
 trans[update]:
 <
  < ClVal:Nat , busy > ,
  (<<PcVal:Nat;GotClVal:Nat;gotvalue>> S:PcStates)
 > =>
 if PcVal <= GotClVal then
   < <ClVal,busy> ,(<<GotClVal;GotClVal;updated>> S)>
 else
   < <PcVal,busy> , (<< PcVal;PcVal;updated >> S) >
 fi .
}
```

Transitions

GetValue:	if PC and Cloud is idle, fetch Cloud value
Update:	update Cloud/PC according to larger value
GotoIdle:	both PC and Cloud go back to idle

Transitions

GetValue:	if PC and Cloud is idle, fetch Cloud value
Update:	update Cloud/PC according to larger value
GotoIdle:	both PC and Cloud go back to idle

```
mod! GOTOIDLE {pr(STATE)
 trans[gotoidle]:
  <
   < ClVal:Nat ,busy > ,
   ( <<PcVal:Nat;OldClVal:Nat;updated >> S:PcStates)
  > =>
  < <ClVal, idlecl> , ( <<PcVal; OldClVal; idlepc>> S) > .
}
```

CloudSync

Final specification is combination of the three transitions
(included modules are shared!)

```
mod! CLOUD {
 pr(GETVALUE + UPDATE + GOTOIDLE)
}
```

CloudSync

Final specification is combination of the three transitions
(included modules are shared!)

```
mod! CLOUD {
 pr(GETVALUE + UPDATE + GOTOIDLE)
}
```

Goal

CloudSync

Final specification is combination of the three transitions (included modules are shared!)

```
mod! CLOUD {
 pr(GETVALUE + UPDATE + GOTOIDLE)
}
```

Goal

If PC is in updated state, then the values of the Cloud and the PC agree.

Verification

Hoare style proof

VERIFICATION

Hoare style proof

1) show invariant for all initial states

Verification

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

VERIFICATION

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  initial : State $\mapsto$ Bool

VERIFICATION

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  initial : State $\mapsto$ Bool

- define a set of predicates

  invariant : State $\mapsto$ Bool

Verification

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  $\text{initial} : \text{State} \mapsto \text{Bool}$
- define a set of predicates

  $\text{invariant} : \text{State} \mapsto \text{Bool}$
- show for all states

  $\forall S : \text{initial}(S) \to \text{invariant}(S)$

VERIFICATION

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  $\text{initial} : \text{State} \mapsto \text{Bool}$
- define a set of predicates

  $\text{invariant} : \text{State} \mapsto \text{Bool}$
- show for all states

  $\forall S : \text{initial}(S) \to \text{invariant}(S)$
- show for all states

  $\forall S : \text{invariant}(S) \to \text{invariant}(S')$

where $S \mapsto S'$ is any transition

How to prove $\forall S$

Question

How to prove a statement like

$$\forall S : \text{initial}(S) \rightarrow \text{invariant}(S)$$

?

How to prove $\forall S$

Question

How to prove a statement like

$$\forall S : \text{initial}(S) \rightarrow \text{invariant}(S)$$

?

Answer

Show it for any element of a covering set of state expressions.

Covering set

most general: S (state variable) – every state is an instance of S

Covering set

most general: S (state variable) – every state is an instance of S

more general $\{S_1, \ldots, S_n\}$ such that

$$\forall S \exists S_i : S = \sigma(S_i)$$

i.e., every state term is an instance of one of the elements of the covering set

Proving with covering sets

Requirements for proving Hoare style

all transitions and predicates have to be *applicable* to terms of the covering set

Covering set

```
ops s1 s2 s3 s4 t1 t2 t3 t4 : -> State .
ops M N K : -> Nat . var PCS : PcStates .
eq s1 = < < N, idlecl > , ( << M; K; idlepc >> PCS ) > .
eq s2 = < < N, idlecl > , ( << M; K; gotvalue >> PCS ) > .
eq s3 = < < N, idlecl > , ( << M; K; updated >> PCS ) > .
eq t1 = < < N, busy > , ( << M; K; idlepc >> PCS ) > .
eq t2 = < < N, busy > , ( << M; K; gotvalue >> PCS ) > .
eq t3 = < < N, busy > , ( << M; K; updated >> PCS ) > .
```

Initial predicates

cl-is-idle: Cloud is initially idle

```
op cl-is-idle-name : -> PredName .
eq[cl-is-idle] : apply(cl-is-idle-name,S:State) =
              ( snd(fst(S)) = idlecl ) .
```

Initial predicates

cl-is-idle: Cloud is initially idle
pcs-are-idle: all PCs are initially idle

```
op pcs-are-idle-name : -> PredName .
eq[pcs-are-idle] : apply(pcs-are-idle-name,S:State) =
   zero-gotvalue(S) and zero-updated(S) .
```

Initial predicates

cl-is-idle:	Cloud is initially idle
pcs-are-idle:	all PCs are initially idle
init:	cl-is-idle & pcs-are-idle

```
mod! INITIALSTATE {
 pr(INITPREDS)
 op init-name : -> PredNameSeq .
 eq init-name = cl-is-idle-name pcs-are-idle-name .
 pred init : State .
 eq init(S:State) = apply(init-name, S) .
}
```

Invariant predicates

goal: all PCs in updated state agree with Cloud

Invariant predicates

goal: all PCs in updated state agree with Cloud

if Cloud is idle then all PCs, too

only at most one PC is out of the idle state

all PCs in gotvalue state have their tmp value equal to the Cloud value

if Cloud is in busy state, then the value of the Cloud and the gotvalue of the Pcs agree

Hoare style in term reduction

initial step

```
red init(s1) implies invariant(s1) . -- OK
red init(s2) implies invariant(s2) . -- OK
red init(s3) implies invariant(s3) . -- OK
red init(t1) implies invariant(t1) . -- OK
red init(t2) implies invariant(t2) . -- OK
red init(t3) implies invariant(t3) . -- OK
```

Hoare style in term reduction

induction step search predicate

```
op inv-condition : State State -> Bool .
eq inv-condition(S, SS) =
  (not (
       S =(*,1)=>+ SS
       suchThat
       (not
         ((invariant(S) implies invariant(SS))
         == true)
       )
     )
  ) .
```

Hoare style in term reduction

induction step

```
red inv-condition(s1, SS) . -- OK
red inv-condition(s2, SS) . -- OK
red inv-condition(s3, SS) . -- OK
red inv-condition(t1, SS) . -- OK
--> The following condition does not reduce directly
--> to true, we will deal with it later on
red inv-condition(t2, SS) . -- BAD
red inv-condition(t3, SS) . -- OK
```

Hoare style in term reduction

induction step

```
red inv-condition(s1, SS) . -- OK
red inv-condition(s2, SS) . -- OK
red inv-condition(s3, SS) . -- OK
red inv-condition(t1, SS) . -- OK
--> The following condition does not reduce directly
--> to true, we will deal with it later on
red inv-condition(t2, SS) . -- BAD
red inv-condition(t3, SS) . -- OK
```

Rest of the invariant condition with case distinctions

Lab Time

- Given a sorted list ℓ, the function $\mathsf{insert}(x, \ell)$ computes the sorted version of $x \mid \ell$. For instance,

$$\begin{aligned}\mathsf{insert}(\mathbf{5}, 2 \mid 4 \mid 6 \mid \mathsf{nil}) &= 2 \mid 4 \mid \mathbf{5} \mid 6 \mid \mathsf{nil} \\ \mathsf{insert}(\mathbf{7}, 2 \mid 4 \mid 6 \mid \mathsf{nil}) &= 2 \mid 4 \mid 6 \mid \mathbf{7} \mid \mathsf{nil}\end{aligned}$$

  Implement `insert : Nat NatList -> NatList`.

- use `insert` to implement the *insertion sort* algorithm (isort).
  Hint:

$$\mathsf{isort}(3 \mid 2 \mid 1 \mid \mathsf{nil}) = \mathsf{insert}(3, \mathsf{insert}(2, \mathsf{insert}(1, \mathsf{nil}))) \quad = 1 \mid 2 \mid 3 \mid \mathsf{nil}$$