Algebraic specification and verification with CafeOBJ

Part 2 – Advanced topics

Norbert Preining

ESSLLI 2016

Bozen, August 2016
Solution to the exercises
Exercises

- Implement \( \text{factorial}(n) = n! \)

- Implement \( \text{fib}(n) \), \( n \)-th Fibonacci number, where \( \text{fib}(0) = 0 \), \( \text{fib}(1) = 1 \), and \( \text{fib}(n) = \text{fib}(n - 2) + \text{fib}(n - 1) \) 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.
- They are declared by either one of `mod!` `mod*` `mod`.

The difference of the three is the models that are considered:

- `mod!`: initial models
- `mod*`: all models
- `mod`: undecided

The 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)
Modules

- Modules are the basic building blocks of CafeOBJ specifications, corresponding to (order-sorted) algebras.
- The are declared by either one of \texttt{mod!} \texttt{mod*} \texttt{mod}
- Difference of the three are the models that are considered:
  - \texttt{mod!}: initial models
  - \texttt{mod*}: all models
  - \texttt{mod}: undecided
**Modules**

- Modules are the basic building blocks of CafeOBJ specifications, corresponding to (order-sorted) algebras.
- They are declared by either one of `mod!` `mod*` `mod`
- The 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
definition of sorts and order
operator constant 0
normal prefix operator
infix operator
variable declaration
equation/axioms
another equation
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(0))))) : Nat
(0.000 sec for parse, 4 rewrites(0.000 sec), 7 matches)
%PNAT> close
CafeOBJ>

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:

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

Q What will happen?
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)) .
}

What will happen?
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)) .
}

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
Definition

- pair of terms $\ell \rightarrow r$ is rewrite rule if $\ell \notin V$ & $\text{Var}(r) \subseteq \text{Var}(\ell)$
- term rewrite system (TRS) $R$ is set of rewrite rules
**Term Rewrite System (TRS)**

**Definition**

- pair of terms $\ell \rightarrow r$ is **rewrite rule** if $\ell \notin V$ & $\text{Var}(r) \subseteq \text{Var}(\ell)$
- **term rewrite system** (TRS) $R$ is set of rewrite rules
- rewrite step: $s \rightarrow_R t$ if
  \[ s = C[\ell\sigma] \text{ and } t = C[r\sigma] \]
  for some substitution $\sigma$, context $C$, and rule $\ell \rightarrow r \in R$

**NOTATIONS**

$V$ stands for set of all variables and $\text{Var}(t)$ for variables in $t$
**Example of TRS**

TRS $R$

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{mul}(0, y) & \rightarrow 0 \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \\
\text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y))
\end{align*}
\]
**Example of TRS**

TRS $R$

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{mul}(0, y) & \rightarrow 0 \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \\
\text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y))
\end{align*}
\]

rewrite sequence

\[
\begin{align*}
\text{mul}(s(0), s(0)) & \rightarrow_R \text{add}(s(0), \text{mul}(s(0), 0)) \\
& \rightarrow_R \text{add}(s(0), 0) \\
& \rightarrow_R s(\text{add}(0, 0)) \\
& \rightarrow_R s(0)
\end{align*}
\]
Example of TRS

TRS $R$

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{mul}(0, y) & \rightarrow 0 \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \\
\text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y))
\end{align*}
\]

rewrite sequence

\[
\begin{align*}
\text{mul}(s(0), s(0)) & \rightarrow_R \text{add}(s(0), \text{mul}(s(0), 0)) \\
& \rightarrow_R \text{add}(s(0), 0) \\
& \rightarrow_R s(\text{add}(0, 0)) \\
& \rightarrow_R s(0)
\end{align*}
\]

Definition

$t$ is **normal form** if $t \rightarrow_R u$ for no $u$
Example of TRS

TRS $R$

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{mul}(0, y) & \rightarrow 0 \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \\
\text{mul}(s(x), y) & \rightarrow \text{add}(y, \text{mul}(x, y))
\end{align*}
\]

rewrite sequence

\[
\begin{align*}
\text{mul}(s(0), s(0)) & \rightarrow_R \text{add}(s(0), \text{mul}(s(0), 0)) \\
& \rightarrow_R \text{add}(s(0), 0) \\
& \rightarrow_R s(\text{add}(0, 0)) \\
& \rightarrow_R s(0)
\end{align*}
\]

Definition

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

Definition

$R$ is **terminating** if there is no infinite sequence $t_1 \rightarrow_R t_2 \rightarrow_R \cdots$
Uniqueness of Normal Forms
UNIQUENESS OF NORMAL FORMS

Definition

- $t \rightarrow^*_R u$ if $t \rightarrow_R \cdots \rightarrow_R u$ (possibly no step)
- $t \rightarrow^!_R u$ if $t \rightarrow^*_R u$ 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

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

(conditional) TRS $R$

\[
\begin{align*}
f(x, y) &\rightarrow x + y & \text{if } x \geq 50 \\
f(x, y) &\rightarrow 0 & \text{if } y < 50
\end{align*}
\]

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

\[
100 \leftarrow^!_R f(70, 30) \rightarrow^!_R 0
\]
Uniqueness of Normal Forms

Definition

- \( t \rightarrow^*_R u \) if \( t \rightarrow_R \cdots \rightarrow_R u \) (possibly no step)
- \( t \rightarrow^!_R u \) if \( t \rightarrow^*_R u \) and \( u \) is normal form
- \( t \downarrow_R \) denotes normal form of \( t \) if there is exactly one normal form of \( t \)

(conditional) TRS \( R \)

\[
\begin{align*}
\text{f}(x, y) & \rightarrow x + y & \text{if } x \geq 50 \\
\text{f}(x, y) & \rightarrow 0 & \text{if } y < 50
\end{align*}
\]

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

\[
100 \leftarrow^!_R \text{f}(70, 30) \rightarrow^!_R 0
\]

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 $R$

append(nil, $ys$) $\rightarrow$ $ys$

append($x : xs$, $ys$) $\rightarrow$ $x :$ append($xs$, $ys$)

e.g.
append(1 : 2 : 3 : nil, 4 : 5 : nil) $\rightarrow$ 1 : 2 : 3 : append(nil, 4 : 5 : nil) $\rightarrow$ 1 : 2 : 3 : 4 : 5 : nil
Quiz

TRS \( R \)

\[
\begin{align*}
    \text{append}(\text{nil}, ys) & \rightarrow ys \\
    \text{append}(x : xs, ys) & \rightarrow x : \text{append}(xs, ys)
\end{align*}
\]

e.g.

\[
\begin{align*}
    \text{append}(1 : 2 : 3 : \text{nil}, 4 : 5 : \text{nil}) & \rightarrow 1 : \text{append}(2 : 3 : \text{nil}, 4 : 5 : \text{nil}) \\
    & \rightarrow 1 : 2 : \text{append}(3 : \text{nil}, 4 : 5 : \text{nil}) \\
    & \rightarrow 1 : 2 : 3 : \text{append}(\text{nil}, 4 : 5 : \text{nil}) \\
    & \rightarrow 1 : 2 : 3 : 4 : 5 : \text{nil}
\end{align*}
\]
Quiz

TRS $R$

append(nil, ys) $\rightarrow$ ys
append($x \colon xs$, ys) $\rightarrow$ $x \colon append(xs, ys)$

e.g.
append(1 : 2 : 3 : nil, 4 : 5 : nil) $\rightarrow$ 1 : append(2 : 3 : nil, 4 : 5 : nil)
$\rightarrow$ 1 : 2 : append(3 : nil, 4 : 5 : nil)
$\rightarrow$ 1 : 2 : 3 : append(nil, 4 : 5 : nil)
$\rightarrow$ 1 : 2 : 3 : 4 : 5 : nil

is $R$ terminating?
More on CafeOBJ
Built-in data types

[ NzNat < Nat < NzInt < Int < NzRat < Rat ]
Built-in data types

\[
\text{[ NzNat} < \text{Nat} < \text{NzInt} < \text{Int} < \text{NzRat} < \text{Rat } \text{]}
\]

\[
\text{[ Triv Bool Float Char String ]}
\]
**BUILT-IN DATA TYPES**

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

[ Triv Bool Float Char String ]

[ 2Tuple 3Tuple 4Tuple ]
**Built-in data types**

\[
\text{[ NzNat < Nat < NzInt < Int < NzRat < Rat ]}
\]

\[
\text{[ Triv Bool Float Char String ]}
\]

\[
\text{[ 2Tuple 3Tuple 4Tuple ]}
\]

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**

```plaintext
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 $f : \text{Nat NzNat} \rightarrow \text{Nat}$.
**Operator definitions**

**prefix** (default)

\[
\text{op } f : \text{Nat NzNat } \rightarrow \text{Nat}.
\]

**mixfix** (useful, but can be dangerous)

\[
\text{op } _+_- : \text{Int Int } \rightarrow \text{Int}.
\]
**Operator Definitions**

**prefix** (default)

\[
\text{op } f : \text{Nat NzNat} \rightarrow \text{Nat}.
\]

**mixfix** (useful, but can be dangerous)

\[
\text{op } _+_- : \text{Int Int} \rightarrow \text{Int}.
\]

\[
\text{op } \ll\ldots\rr : \text{Nat Nat Nat} \rightarrow \text{Nat}.
\]
**Operator definitions**

- **prefix** *(default)*
  
  \[
  \text{op } f : \text{Nat NzNat} \rightarrow \text{Nat} .
  \]

- **mixfix** *(useful, but can be dangerous)*
  
  \[
  \text{op } _+_- : \text{Int Int} \rightarrow \text{Int} .
  \]
  \[
  \text{op } \langle\langle\cdots\rangle\rangle : \text{Nat Nat Nat} \rightarrow \text{Nat} .
  \]
  \[
  \text{op } \text{if}_\text{then}_\text{else}_\text{fi} : \text{Bool Nat Nat} \rightarrow \text{Nat} .
  \]

---

Warning: **mixfix** operators can create difficult to parse terms, sometimes proper qualification of terms is necessary.
**Operator definitions**

**prefix** (default)

\[ \text{op } f : \text{Nat NzNat } \rightarrow \text{Nat} . \]

**mixfix** (useful, but can be dangerous)

\[ \text{op } _+_: \text{Int Int } \rightarrow \text{Int} . \]
\[ \text{op } \llcorner\llcorner\lrcorner\lrcorner: \text{Nat Nat Nat } \rightarrow \text{Nat} . \]
\[ \text{op if_then_else_fi : Bool Nat Nat } \rightarrow \text{Nat} . \]
\[ \text{eq if } \ldots = ? \]
**Operator definitions**

**prefix** (default)

\[
\text{op } f : \text{Nat NzNat } \rightarrow \text{Nat} .
\]

**mixfix** (useful, but can be dangerous)

\[
\text{op } _+_: \text{Int Int } \rightarrow \text{Int} .
\]

\[
\text{op } <<<>>>: \text{Nat Nat Nat } \rightarrow \text{Nat} .
\]

\[
\text{op if}_\text{then}_\text{else}_\text{fi} : \text{Bool Nat Nat } \rightarrow \text{Nat} .
\]

\[
\text{eq if } ... = ?
\]

**Warning**

Mixfix operators can create difficult to parse terms, sometimes proper qualification of terms is necessary.
EQUATIONAL THEORY ATTRIBUTES

associativity, commutativity, identity, idempotence

\[
\text{op } \_\&\_ : \text{Bool} \text{ Bool } \rightarrow \text{Bool} \{ \text{assoc comm idem id: true} \}
\]
**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 \_\&\_ : \text{Bool} \times \text{Bool} \rightarrow \text{Bool} \{ \text{assoc comm idem id: true} \}

\[
\text{mod*}\ \text{GROUP} \{ \\
[ G ] \begin{align*}
op \ 0 & : \rightarrow G . \\
op \_+\_ & : G \times G \rightarrow G \{ \text{assoc} \} . \\
op \_-\_ & : G \rightarrow G . \\
\text{var} \ X & : G . \\
eq[0\text{left}] & : 0 + X = X . \\
eq[neg\text{inv}] & : (- X) + X = 0 . 
\end{align*}
\}
\]

\[\boxdash\text{inherited}\]
**PARSING ATTRIBUTES**

precedence, associativity

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

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

precedence, associativity

\begin{align*}
op \_+\_ & : \text{Int} \ \text{Int} \to \text{Int} \ \{ \text{prec}: 33 \} . \\
op \_\times\_ & : \text{Int} \ \text{Int} \to \text{Int} \ \{ \text{prec}: 31 \} .
\end{align*}

effect: $\times$ binds stronger than $+$.

\begin{align*}
op \_+\_ & : \text{S} \ \text{S} \to \text{S} \ \{ \text{l-assoc} \} .
\end{align*}

reduces $X + X + X$ to $(X + X) + X$. 

Importing modules imports the declarations.

Three different modes:

**protecting (pr)**  
\[ \text{pr}(\text{NAT}) \]
all intended models are preserved as they are

**extending (ex)**  
\[ \text{ex}(\text{BOOL}) \]
models can be inflated, but cannot collapse

**including (inc)**  
\[ \text{inc}(\text{INT}) \]
no restrictions on models

**using (us)**  
\[ \text{us}(\text{FLOAT}) \]
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

$$L ::= \text{nil} \mid x \mid L \quad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list
Lists

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

$$L ::= \text{nil} \mid x \cdot L \quad (x \in \mathbb{N})$$

- We assume right associativity of $\cdot$

Example

- $\text{nil} - \text{list}$
- $1 \cdot (3 \cdot (2 \cdot \text{nil})) - \text{list}$
Lists

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

$$L ::= \text{nil} \mid x \mid L \quad (x \in \mathbb{N})$$

- Empty list
- Cons

- We assume right associativity of $\mid$

Example

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

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

\[
L ::= \text{nil} \mid x \mid L \quad (x \in \mathbb{N})
\]

- empty list
- cons

- We assume right associativity of $\mid$

Example

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

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

$$L ::= \text{nil} \mid x \mid L \quad (x \in \mathbb{N})$$

- we assume right associativity of $\mid$

Example

- nil — list
- $1 \mid (3 \mid (2 \mid \text{nil}))$ — list
- $1 \mid 3 \mid 2 \mid \text{nil}$ — list
- $1 \mid 3 \mid 2$ — not list
- $(1 \mid 3) \mid 2 \mid \text{nil}$ — not list
**Lists**

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

$$ L ::= \text{nil} \mid x \mid L \quad (x \in \mathbb{N}) $$

  - empty list
  - cons

- we assume right associativity of $\mid$

**Example**

- nil — list
- $1 \mid (3 \mid (2 \mid \text{nil}))$ — list
- $1 \mid 3 \mid 2 \mid \text{nil}$ — list
- $1 \mid 3 \mid 2$ — not list
- $(1 \mid 3) \mid 2 \mid \text{nil}$ — not list
- $1 \mid \text{true} \mid 3 \mid \text{nil}$ — not list
Lists in CafeOBJ

Lists can be defined as sorted terms over constructor symbols:

\[
\text{nil} : \text{NatList} \quad \text{and} \quad \_|\_ : \text{Nat} \times \text{NatList} \rightarrow \text{NatList}
\]

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

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

\[
\begin{align*}
\text{len}(\text{nil}) &= 0 \\
\text{len}(3 \mid \text{nil}) &= 1 \\
\text{len}(2 \mid 3 \mid \text{nil}) &= 2 \\
\text{len}(1 \mid 2 \mid 3 \mid \text{nil}) &= 3
\end{align*}
\]

op \(\text{len}: \text{NatList} \rightarrow \text{Nat}\)

eq \(\text{len}(\text{nil}) = ?\)

\(\text{eq len}(E:\text{Nat} \mid L:\text{NatList}) = ?\)
**APPEND**

\[
\begin{align*}
\text{nil}@ (3 \mid 4 \mid \text{nil}) &= 3 \mid 4 \mid \text{nil} \\
(2 \mid \text{nil})@ (3 \mid 4 \mid \text{nil}) &= 2 \mid 3 \mid 4 \mid \text{nil} \\
(1 \mid 2 \mid \text{nil})@ (3 \mid 4 \mid \text{nil}) &= 1 \mid 2 \mid 3 \mid 4 \mid \text{nil}
\end{align*}
\]

\[
\text{mod}^* \ \text{NATLIST} @ \{
\begin{array}{l}
\text{pr} (\text{NATLIST}) \\
\text{var} \ E : \text{Nat} \\
\text{vars} \ L1 \ L2 : \text{NatList} \\
\text{op} \ _@_ : \text{NatList} \ \text{NatList} \to \text{NatList}
\end{array}
\]

\[
\begin{align*}
\text{eq} \text{nil} \ @ \ L2 &= ? \\
\text{eq} \ (E \mid L1) \ @ \ L2 &= ?
\end{align*}
\]
Reusing data
Association Lists

association lists are

- lists of pairs: \((x_1, y_1) \mid ... \mid (x_n, y_n) \mid \text{nil}\)
Association Lists

association lists are

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

\[
l = ("Kanazawa", 921) \mid ("Nomi", 923) \mid \text{nil}
\]

lookup("Kanazawa", l) = 921
lookup("Nomi", l) = 923
lookup("Hakusan", l) = not-found
Association Lists

Association lists are

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

\[
l = ("Kanazawa", 921) \mid ("Nomi", 923) \mid \text{nil}
\]

\[
\text{lookup}("Kanazawa", l) = 921
\]
\[
\text{lookup}("Nomi", l) = 923
\]
\[
\text{lookup}("Hakusan", l) = \text{not-found}
\]

Q

- what would be the signature of data constructors and \text{lookup}?
- how would one define \text{lookup}?
- implementation?
**Parametrized Modules**

- \( \text{mod! } M(\overline{X} :: \overline{C}, ...) \{ \cdots f.X \cdots \} \)

- Parametrized module

---

Algebraic specification and verification with CafeOBJ [5pt]Part 2 - Advanced topics
**Parametrized Modules**

- **variable module constraint**
  - \[ \text{mod! } M(\tilde{X} \:: \tilde{C}, ...) \{ \cdots f.X \cdots \} \]
  - parametrized module

- **view module instantiation**
  - \[ M(N\{\text{sort A -> B, op f -> g, \ldots}\}) \]
  - module instantiation

---

Algebraic specification and verification with CafeOBJ [5pt]Part 2 - Advanced topics
PARAMETRIZED MODULES

variable module constraint

- mod! \( M(X :: C, ...) \{ \cdots f.X \cdots \} \) \hspace{1cm} \text{parametrized module}

- \( M(N\{\text{sort } A \rightarrow B, \text{op } f \rightarrow g, \ldots \}) \) \hspace{1cm} \text{module instantiation}

mod* C {
  \[A\]
  \text{op add : } A A \rightarrow A .
}

Algebraic specification and verification with CafeOBJ [5pt]Part 2 - Advanced topics
**Parametrized Modules**

- **Parametrized Module**
  
  ```
  mod! M(X :: C, ...) {⋯ f.X ⋯}  
  ```

- **Module Instantiation**
  
  ```
  M(N{sort A -> B, op f -> g, ...})
  ```

---

**Example**

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

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

---

Algebraic specification and verification with CafeOBJ [5pt]Part 2 - Advanced topics
**Parametrized Modules**

variable module constraint

- mod! $M(\tilde{X} :: \tilde{C}, \ldots) \{ \cdots f.X \cdots \}$ parametrized module

  view

- $M(N\{\text{sort } A \rightarrow B, \text{op } f \rightarrow g, \ldots\})$ module instantiation

mod* $C \{ $

[A]

  op add : A A \rightarrow A .

} 

mod! $\text{TWICE}(X :: C) \{$

  op twice : A.X \rightarrow A.X .

  eq twice(E:A.X) = add.X(E,E) .

} 

open $\text{TWICE}(\text{NAT} \{ \text{sort } A \rightarrow \text{Nat}, \text{op } \text{add} \rightarrow _+\_ \})$

  red twice(10) . \rightarrow 10 + 10 \rightarrow 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

\texttt{TRIV} consists of only sort \texttt{Elt}; see \texttt{show TRIV}
Parametrized Lists

\texttt{TRIV} consists of only sort \texttt{Elt}; see \texttt{show TRIV}

\texttt{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**

```plaintext
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)) }, or
mod! NATLIST { pr(LIST(NAT)) } +

**Elt** is automatically identified if module contains only one sort
**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

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

```plaintext
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 })
}

The isomorphism renames <List, nil, |> ↦ <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

Algebraic specification and verification with CafeOBJ [5pt]Part 2 - Advanced topics
Renaming of instances cont.

```plaintext
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

\(<\text{List}, \text{nil}, |> \mapsto <\text{NatList}, \text{natnil}, ||>:\)
The isomorphism renames

\(<\text{List}, \text{nil}, |>| \mapsto <\text{NatList}, \text{natnil}, ||>|:

\%SUPERMODULE>\ parse 3 || 4 || 7 || 1 || \text{natnil} .
(3 || (4 || (7 || (1 || \text{natnil})))):\text{NatList}
\%SUPERMODULE>\ parse 3 | 4 | 7 | 1 | \text{nil} .
(3 | (4 | (7 | (1 | \text{nil})))):\text{IntList}
\%SUPERMODULE>\ parse 3 | 4 | 7 | 1 | \text{natnil} .
[\text{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 .
  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?

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

Assume we have

0 + 𝑎 = 𝑎

(1)

−𝑎 + 𝑎 = 0

(2)

𝑎 + −𝑎 = 0 + 𝑎 + −𝑎

by (1) right-to-left

= −−𝑎 + −𝑎 + 𝑎 + −𝑎

by (2) right-to-left

= −−𝑎 + 0 + −𝑎

by (2)

= −−𝑎 + −𝑎

by (1)

= 0

by (2)

Why did it not work in CafeOBJ?
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

\[ 0 + a = a \quad (1) \]
\[ -a + a = 0 \quad (2) \]

\[ a + -a = 0 + a + -a \quad \text{by (1) right-to-left} \]
\[ = -a + -a + a + -a \quad \text{by (2) right-to-left} \]
\[ = -a + 0 + -a \quad \text{by (2)} \]
\[ = -a + -a \quad \text{by (1)} \]
\[ = 0 \quad \text{by (2)} \]
Group theory cont.

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

\[ 0 + a = a \] (1)
\[ -a + a = 0 \] (2)

\[ a + -a = 0 + a + -a \quad \text{by (1) right-to-left} \]
\[ = -a - a + a + -a \quad \text{by (2) right-to-left} \]
\[ = -a + 0 + -a \quad \text{by (2)} \]
\[ = -a + -a \quad \text{by (1)} \]
\[ = 0 \quad \text{by (2)} \]

Why did it not work in CafeOBJ?
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?
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 $(x + y) + z = x + (y + z)$ for all $x$, $y$, and $z$?

Proof by induction:

**Induction base**
Show that $(0 + y) + z = 0 + (y + z)$
Assume that $0 + y = y$ and $s(x) + y = s(x + y)$ for all $x$ and $y$.
How do we show that $(x + y) + z = x + (y + z)$ 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

```ocaml
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
```

Algebraic specification and verification with CafeOBJ [5pt]Part 2 - Advanced topics 50/81
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

<table>
<thead>
<tr>
<th>Cloud</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td></td>
<td>$n$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC-1</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td>$k$</td>
<td></td>
</tr>
<tr>
<td>tmp</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC-2</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td>$l$</td>
<td></td>
</tr>
<tr>
<td>tmp</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>PC-$n$</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td>$m$</td>
<td></td>
</tr>
<tr>
<td>tmp</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
**CloudSync in images**

The image depicts a state transition diagram for CloudSync. The state transitions are driven by the `gotvalue` event.

### Cloud State

<table>
<thead>
<tr>
<th>State</th>
<th>Busy Stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n$</td>
</tr>
</tbody>
</table>

### PC-1 State

<table>
<thead>
<tr>
<th>State</th>
<th>Busy Stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>$k$</td>
</tr>
<tr>
<td>Stamp</td>
<td></td>
</tr>
<tr>
<td>Temp</td>
<td>$n$</td>
</tr>
</tbody>
</table>

### PC-2 State

<table>
<thead>
<tr>
<th>State</th>
<th>Busy Stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>$l$</td>
</tr>
<tr>
<td>Stamp</td>
<td></td>
</tr>
<tr>
<td>Temp</td>
<td>0</td>
</tr>
</tbody>
</table>

### PC-n State (General)

<table>
<thead>
<tr>
<th>State</th>
<th>Busy Stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>$m$</td>
</tr>
<tr>
<td>Stamp</td>
<td></td>
</tr>
<tr>
<td>Temp</td>
<td>0</td>
</tr>
</tbody>
</table>
CloudSync in images

<table>
<thead>
<tr>
<th>Cloud</th>
<th>state</th>
<th>busy</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td>$k$</td>
<td></td>
</tr>
</tbody>
</table>

transition: update assuming $k \geq n$

<table>
<thead>
<tr>
<th>PC-1</th>
<th>state</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td>$k$</td>
<td></td>
</tr>
<tr>
<td>tmp</td>
<td>$k$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC-2</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td>$l$</td>
<td></td>
</tr>
<tr>
<td>tmp</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>PC-n</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>stamp</td>
<td>$m$</td>
<td></td>
</tr>
<tr>
<td>tmp</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
**CloudSync in Images**

<table>
<thead>
<tr>
<th></th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud</td>
<td>state</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>stamp</td>
<td>$k$</td>
</tr>
</tbody>
</table>

transition: gotoidle

<table>
<thead>
<tr>
<th>PC-1</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>stamp</td>
<td>$k$</td>
</tr>
<tr>
<td></td>
<td>tmp</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC-2</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>stamp</td>
<td>$l$</td>
</tr>
<tr>
<td></td>
<td>tmp</td>
<td>0</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>PC-n</th>
<th>state</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>stamp</td>
<td>$m$</td>
</tr>
<tr>
<td></td>
<td>tmp</td>
<td>0</td>
</tr>
</tbody>
</table>
**SPECIFICATION**

ClLabel: \{idlecl, busy\}

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

Algebraic specification and verification with CafeOBJ [5pt]Part 2 - Advanced topics
**SPECIFICATION**

ClLabel: \(\{\text{idlecl, busy}\}\)  
PcLabel: \(\{\text{idlepc, gotvalue, updated}\}\)

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

ClLabel: \{idlecl, busy\}
PcLabel: \{idlepc, gotvalue, updated\}
ClState: ClLabel × ℕ

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

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

\text{mod! PCSTATE} \{ 
\text{pr}(3\text{TUPLE}(\mathbb{N}, \mathbb{N},
\text{PCLABEL}\{\text{sort Elt} \rightarrow \text{PcLabel}\}))*
\text{pr}(3\text{Tuple} \rightarrow \text{PcState})
\}
**SPECIFICATION**

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

\[
\text{mod! PCSTATES } \{ \\
\quad \text{pr(MULTISET(PCSTATE\{\text{sort Elt -> PcState}\})^*)} \\
\quad \{\text{sort MultiSet -> PcStates}\}) \\
\}
\]
SPECIFICATION

ClLabel: \{idlecl, busy\}
PcLabel: \{idlepc, gotvalue, updated\}
ClState: ClLabel \times \mathbb{N}
PcState: PcLabel \times \mathbb{N} \times \mathbb{N}
PcStates: \text{MultiSet}(PcState)
State: ClState \times 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
GetValue: if PC and Cloud is idle, fetch Cloud value

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

```plaintext
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

```plaintext
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!)

\texttt{mod! CLOUD \{} \texttt{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!)

```plaintext
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
  
  \[
  \text{initial} : \text{State} \rightarrow \text{Bool}
  \]
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
  \[ \text{initial} : \text{State} \rightarrow \text{Bool} \]
- define a set of predicates
  \[ \text{invariant} : \text{State} \rightarrow \text{Bool} \]
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
  \( \text{initial} : \text{State} \rightarrow \text{Bool} \)
- define a set of predicates
  \( \text{invariant} : \text{State} \rightarrow \text{Bool} \)
- show for all states
  \( \forall S : \text{initial}(S) \rightarrow \text{invariant}(S) \)
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
  initial : State → Bool
- define a set of predicates
  invariant : State → Bool
- show for all states
  ∀S : initial(S) → invariant(S)
- show for all states
  ∀S : invariant(S) → invariant(S')
where S ↦ 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

```plaintext
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

\[
\text{op pcs-are-idle-name : } \rightarrow \text{PredName}.
\]

\[
\text{eq[pcs-are-idle] : apply(pcs-are-idle-name,S:State) = zero-gotvalue(S) \text{ and zero-updated}(S)}.
\]
**Initial Predicates**

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

```plaintext
mod! INITIALSTATE {
pr(INITPRED)
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

\begin{verbatim}
op inv-condition : State State -> Bool .
eq inv-condition(S, SS) =
  (not ( 
    S =(*,1)=>+ SS
    suchThat
    (not
      ((invariant(S) implies invariant(SS))
        == true)
    )
  )
).
\end{verbatim}
induction step

red \text{inv-condition}(s_1, SS) . -- OK
red \text{inv-condition}(s_2, SS) . -- OK
red \text{inv-condition}(s_3, SS) . -- OK
red \text{inv-condition}(t_1, SS) . -- OK

\text{--> The following condition does not reduce directly to true, we will deal with it later on}
red \text{inv-condition}(t_2, SS) . -- BAD
red \text{inv-condition}(t_3, 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
Given a sorted list \( \ell \), the function \( \text{insert}(x, \ell) \) computes the sorted version of \( x \mid \ell \). For instance,

\[
\begin{align*}
\text{insert}(5, 2 \mid 4 \mid 6 \mid \text{nil}) &= 2 \mid 4 \mid 5 \mid 6 \mid \text{nil} \\
\text{insert}(7, 2 \mid 4 \mid 6 \mid \text{nil}) &= 2 \mid 4 \mid 6 \mid 7 \mid \text{nil}
\end{align*}
\]

Implement \( \text{insert} : \text{Nat NatList} \rightarrow \text{NatList} \).

- use \( \text{insert} \) to implement the \( \text{insertion sort} \) algorithm (\( \text{isort} \)).
  
  Hint:

\[
\text{isort}(3 \mid 2 \mid 1 \mid \text{nil}) = \text{insert}(3, \text{insert}(2, \text{insert}(1, \text{nil}))) = 1 \mid 2 \mid 3 \mid \text{nil}
\]