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
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(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 $\ell \notin V$ & $\mathsf{Var}(r) \subseteq \mathsf{Var}(\ell)$
- **term rewrite system** (TRS) R is set of rewrite rules

Term Rewrite System (TRS)

Definition

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

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

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

NOTATIONS

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

Example of TRS

TRS R

$$\mathsf{add}(0, y) \to y \qquad \mathsf{mul}(0, y) \to 0$$

$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y))$$

Example of TRS

TRS R

$$\mathsf{add}(0, y) \to y \qquad \mathsf{mul}(0, y) \to 0$$
$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y))$$

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

$$\mathsf{add}(0, y) \to y \qquad \mathsf{mul}(0, y) \to 0$$
$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y))$$

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

$$\mathsf{add}(0, y) \to y \qquad \mathsf{mul}(0, y) \to 0$$
$$\mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(y, \mathsf{mul}(x, y))$$

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

- $t \rightarrow_R^* u$ 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

- $t \to_R^* u$ 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

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

$\mathsf{f}(70, 30){\downarrow}$ 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

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

$\mathsf{f}(70, 30){\downarrow}$ 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

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

- 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 $\overbrace{\text{constructor symbols}}^{\text{functions as values}}$:

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

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

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

Length

$$\mathsf{len}(\mathsf{nil}) = 0$$
$$\mathsf{len}(3 \mid \mathsf{nil}) = 1$$
$$\mathsf{len}(2 \mid 3 \mid \mathsf{nil}) = 2$$
$$\mathsf{len}(1 \mid 2 \mid 3 \mid \mathsf{nil}) = 3$$

```
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

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

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

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

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

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

Q

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

Parametrized Modules

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

Parametrized Modules

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

Parametrized Modules

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

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

Parametrized Modules

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

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

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, |>` ↦ `<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, |>` ↦ `<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 .
}
```

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:

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