

I217: Functional Programming

4. Parameterized Modules

Kazuhiro Ogata, Canh Minh Do

Roadmap

- Parameterized Modules
- Views & Instantiation
- Generic Lists with Error Handling
- Generic Quicksort

Parameterized Modules

Modules can have parameters, making it possible to deal with generic data structures, such as generic lists.

```
mod! GLIST1(E :: TRIV) {
  [Nil NnList < List]
  op nil : -> Nil {constr} .
  op _|_ : Elt.E List -> NnList {constr} .
  op _@_ : List List -> List .
  var X : Elt.E .
  vars L L2 : List .
  eq nil @ L2 = L2 .
  eq (X | L) @ L2 = X | (L @ L2) .
}
```

TRIV is a built-in module declared as follows:

```
mod* TRIV { [Elt] }
```

The parameter **E** is constrained by the module **TRIV**, meaning that an actual parameter must be a module in which at least one sort is declared and replaces **Elt.E** that is the sort **Elt** in **E**.

Parameterized Modules

Declared as follows:

```
mod! MOD-NAME( $P_1 :: M_1, \dots, P_n :: M_n$ ) { ... }
```

Each P_i is a formal parameter constrained by the module M_i , such as **TRIV**. Let *sort* & *oprtr* be a sort & an operator in M_i . *sort.P_i* and *oprtr.P_i* can be used in the module *MOD-NAME* and will be replaced with a sort & an operator in an actual parameter module. Even if M_i is the same as M_j such that $i \neq j$, *sort.P_i* and *oprtr.P_i* are different from *sort.P_j* and *oprtr.P_j*.

Views & Instantiation

Parameterized modules are *instantiated* by replacing sorts (and operators if any) in each formal parameter with sorts (and operators) in an actual parameter module. The replacement is given by what is called a *view*.

```
view TRIV2NAT from TRIV to NAT {  
  sort Elt -> Nat,  
}
```

This view says that the sort **Elt** in the module **TRIV** is replaced with the sort **Nat** in the module **NAT**.

Views & Instantiation

```
mod! NATLIST1 { pr(GLIST1(E <= TRIV2NAT)) }
```

E <= TRIV2NAT replaces the sort **Elt.E** in the module **GLIST1** with the sort **Nat** in the module **NAT**.

GLIST1(E <= TRIV2NAT) is the module made by instantiating **GLIST1** with **NAT** by **TRIV2NAT**.

```
open NATLIST1 .  
  red (4 | 3 | nil) @ (2 | 1 | 0 | nil) .  
close
```

Views & Instantiation

```

mod! GLIST1(E :: TRIV) {
  [Nil NnList < List]
  op nil : -> Nil {constr} .
  op _|_ : Elt.E List -> NnList {constr} .
  op _@_ : List List -> List .
  var X : Elt.E .
  vars L L2 : List .
  eq nil @ L2 = L2 .
  eq (X | L) @ L2 = X | (L @ L2) .
}

mod! GLIST1(E <= TRIV2NAT) {
  [Nil NnList < List]
  op nil : -> Nil {constr} .
  op _|_ : Nat List -> NnList {constr} .
  op _@_ : List List -> List .
  var X : Nat .
  vars L L2 : List .
  eq nil @ L2 = L2 .
  eq (X | L) @ L2 = X | (L @ L2) .
}

```

This code fragment is just for explanation and then cannot be accepted by CafeOBJ; therefore, you do not need to type it.

Views & Instantiation

Declared as follows:

a module used to constrain
formal parameters

```

view VIEW-NAME from FPM to APM {
  sort ssort1 -> dsort1,
  ...
  sort ssortn -> dsortn,
  op soprtr1 -> doprtr1,
  ...
  op soprtrm -> doprtrm,
}

```

a module used as actual
parameters

sorts and operators declared
in *FPM*

sorts and operators declared
in *APM*

Views & Instantiation

How to instantiate *MOD-NAME* with views $V_1, \dots, V_n$:

```
mod! INSTANTIATED-MOD-NAME {
  pr(MOD-NAME( $P_1 \leq V_1, \dots, P_n \leq V_n$ ))
}

open INSTANTIATED-MOD-NAME .
...
close

open MOD-NAME( $P_1 \leq V_1, \dots, P_n \leq V_n$ ) .
...
close
```

Views & Instantiation

Anonymous views can be used to instantiate parameterized modules.

```
mod! NATLIST2 {
  pr(GLIST1( $E \leq \text{view from TRIV to NAT}$ 
           { sort  $\text{Elt} \rightarrow \text{Nat}, \}$  ) ) ) }
```

If an actual parameter module has a sort as a principal sort, such as **mod!** NAT **principal-sort** Nat { ... }, an abbreviation can be used, such as GLIST1(NAT) that is the abbreviation of **pr**(GLIST1($E \leq \text{view from TRIV to NAT}$ { **sort** $\text{Elt} \rightarrow \text{Nat}, \}$)).

```
mod! NATLIST3 { pr(GLIST1(NAT)) }
```

Generic Lists with Error Handling

```

mod! GLIST2(E :: TRIV-ERR) {
  [Nil NnList < List]
  op nil : -> Nil {constr} .
  op _|_ : Elt.E List -> NnList {constr} .
  op hd : Nil -> Err.E .
  op hd : NnList -> Elt.E .
  op hd : List -> Elt&Err.E .
  op tl : List -> List .
  op _@_ : List List -> List .
  var X : Elt.E .
  vars L L2 : List .

  eq hd(nil) = err.E .
  eq hd(X | L) = X .
  eq tl(nil) = nil .
  eq tl(X | L) = L .
  eq nil @ L2 = L2 .
  eq (X | L) @ L2 = X | (L @ L2) .
}

mod* TRIV-ERR {
  [Elt Err < Elt&Err]
  op err : -> Err .
}

```

Generic Lists with Error Handling

A view from

The same module used in lecture note 2

```

mod* TRIV-ERR {
  [Elt Err < Elt&Err]
  op err : -> Err .
}

mod! NAT-ERR { pr(NAT)
  [Nat ErrNat < Nat&Err]
  op errNat : -> ErrNat {constr} .
  ... }

view TRIV-ERR2NAT-ERR from TRIV-ERR to NAT-ERR {
  sort Elt -> Nat,
  sort Err -> ErrNat,
  sort Elt&Err -> Nat&Err,
  op err -> errNat,
}

```

Generic Lists with Error Handling

Instantiation of GLIST2 with NAT-ERR by
TRIV-ERR2NAT-ERR

```

mod! NATLIST7 {
  pr(GLIST2(E <= TRIV-ERR2NAT-ERR)
    * {sort List -> NatList,
      sort Nil -> NLNil,
      sort NnList -> NnNatList,
      op nil -> nlnil } )
}

open NATLIST7 .
red hd(nlnil) .
red hd((4 | 3 | nlnil) @ (2 | 1 | 0 | nlnil)) .
red tl(nlnil) .
red tl((4 | 3 | nlnil) @ (2 | 1 | 0 | nlnil)) .
close

```

Sorts and operators can be renamed.

Generic Lists with Error Handling

Sorts and operators in a module *MODULE* can be renamed as follows:

```

MODULE * { sort old-sort-name1 -> new-sort-name1,
  ...
  sort old-sort-namen -> new-sort-namen,
  op old-op-name1 -> new-op-name1,
  ...
  op old-op-namem -> new-op-namem, }

```

Generic Lists with Error Handling

What if GLIST2 is instantiated with NATLIST7, creating a module in which **lists of lists of natural numbers** are treated?

```
view TRIV-ERR2NATLIST7 from TRIV-ERR to NATLIST7 {
  sort Elt -> NatList,
  sort Err -> ???,
  sort Elt&Err -> ???,
  op err -> ???,
}
```

NATLIST7 does not have any sorts that can adequately replace **Err** and **Elt&Err** nor any operator that can adequately replace **err**.

Generic Lists with Error Handling

```
mod! GLIST-ERR(E :: TRIV-ERR) {
  [Nil NnList < List]
  [List ErrList < List&Err]
  op nil : -> Nil {constr} .
  op _|_ : Elt.E List -> List {constr} .
  op errList : -> ErrList {constr} .
  op _|_ : Elt&Err.E List&Err -> List&Err .
  op hd : Nil -> Err.E .
  op hd : NnList -> Elt.E .
  op hd : ErrList -> Err.E .
  op hd : List&Err -> Elt&Err.E .
  op tl : Nil -> ErrList .
  op tl : NnList -> List .
  op tl : ErrList -> ErrList .
  op tl : List&Err -> List&Err .
  op _@_ : List List -> List .
  op _@_ : ErrList List&Err -> ErrList .
  op _@_ : List&Err ErrList -> ErrList .
  op _@_ : List&Err List&Err -> List&Err .
  op if_then{ } else{ }
    : Bool List&Err List&Err -> List&Err .
}
```

Generic Lists with Error Handling

```

var X : Elt.E .
var XE : Elt&Err.E .
vars L L2 : List .
vars LE LE2 : List&Err .
eq err.E | LE = errList .
eq XE | errList = errList .
eq hd(nil) = err.E .
eq hd(X | L) = X .
eq hd(errList) = err.E .
eq tl(nil) = errList .
eq tl(X | L) = L .
eq tl(errList) = errList .

eq nil @ L2 = L2 .
eq (X | L) @ L2 = X | (L @ L2) .
eq errList @ LE = errList .
eq LE @ errList = errList .
eq if true then {LE} else {LE2} = LE .
eq if false then {LE} else {LE2} = LE2 .
  }

```

Generic Lists with Error Handling

```

mod! NATLIST8 {
  pr(GLIST-ERR(E <= TRIV-ERR2NAT-ERR)
    * {sort List -> NatList,
      sort Nil -> NLNil,
      sort NnList -> NnNatList,
      sort ErrList -> ErrNatList,
      sort List&Err -> NatList&Err,
      op nil -> nlnil,
      op errList -> errNatList } )
}

open NATLIST8 .
red hd(nlnil) .
red hd((4 | 3 | nlnil) @ (2 | 1 | 0 | nlnil)) .
red tl(nlnil) .
red tl((4 | 3 | nlnil) @ (2 | 1 | 0 | nlnil)) .
close

```

Generic Lists with Error Handling

```

view TRIV-ERR2NATLIST8 from TRIV-ERR to NATLIST8 {
  sort Elt -> NatList,
  sort Err -> ErrNatList,
  sort Elt&Err -> NatList&Err,
  op err -> errNatList, }
  mod! NATLISTLIST1 {
    pr(GLIST-ERR(E <= TRIV-ERR2NATLIST8))
    * {sort List -> NatListList,
      sort Nil -> NLLNil,
      sort NnList -> NnNatListList,
      sort ErrList -> ErrNatListList,
      sort List&Err -> NatListList&Err,
      op nil -> nllnil,
      op errList -> errNatListList } ) }

open NATLISTLIST1 .
  red hd(nllnil) .
  red hd((4 | 3 | nlnil) | (2 | 1 | 0 | nlnil) | nllnil) .
  red tl(nlnil) .
  red tl((4 | 3 | nlnil) | (2 | 1 | 0 | nlnil) | nllnil) .
close

```

Generic Quick Sort

```

mod* TRIV-ERR-ORD {
  [Elt Err < Elt&Err]
  op err : -> Err .
  op ord : Elt Elt -> Bool .
}

-- qsort
eq qsort(nil) = nil .
eq qsort(X | nil) = X | nil .
eq qsort(X | Y | L) = partition(X,Y | L,nil,nil) .
-- partition
eq partition(X,nil,LL,RL) = qsort(LL) @ (X | qsort(RL)) .
eq partition(X,Y | L,LL,RL)
  = if ord.E(Y,X) then {partition(X,L,Y | LL,RL)}
    else {partition(X,L,LL,Y | RL)} . }

mod! GQSORT(E :: TRIV-ERR-ORD) {
  -- imports
  pr(GLIST-ERR(E))
  -- signature
  op qsort : List -> List .
  op partition : Elt.E List List List -> List .
  -- CafeOBJ vars
  vars X Y : Elt.E .
  vars L LL RL : List .

```

Generic Quick Sort

```

view TRIV-ERR-ORD2NAT-ERR from TRIV-ERR-ORD to NAT-ERR {
  sort Elt -> Nat,
  sort Err -> ErrNat,
  sort Elt&Err -> Nat&Err,
  op err -> errNat,
  op ord -> _<_,
}

open GQSORT(E <= TRIV-ERR-ORD2NAT-ERR) .
  red qsort(4 | 7 | 5 | 1 | 0 | 3 | 6 | 2 | nil) .
close

```

Generic Quick Sort

```

mod! STRING-ERR { pr(STRING)
  [String ErrString < String&Err]
  op errStr : -> ErrString {constr} . }

view TRIV-ERR-ORD2STRING-ERR from TRIV-ERR-ORD to STRING-ERR {
  sort Elt -> String,
  sort Err -> ErrString,
  sort Elt&Err -> String&Err,
  op err -> errStr,
  op ord -> string<,
}

open GQSORT(E <= TRIV-ERR-ORD2STRING-ERR) .
  red qsort("Lisp" | "Python" | "Pascal" | "CafeOBJ" | "C" |
    "Java" | "Prolog" | "Fortran" | nil) .
close

```

Generic Quick Sort

What if GQSORT is instantiated with NATLISTLIST1, creating a module in which **lists of lists of natural numbers** are sorted?

```
view TRIV-ERR-ORD2NATLISTLIST1
  from TRIV-ERR-ORD to NATLISTLIST1 {
    sort Elt -> NatListList,
    sort Err -> ErrNatListList,
    sort Elt&Err -> NatListList&Err,
    op err -> errNatListList,
    op ord -> ???, }
```

NATLISTLIST1 does not have any operator that can adequately replace **ord**.

Generic Quick Sort

```
mod! GLIST-ERR-ORD(E :: TRIV-ERR-ORD) {
  ...
  op ord : List List -> Bool .
  ...
  eq ord(nil,nil) = false .
  eq ord(nil,Y | L2) = true .
  eq ord(X | L,nil) = false .
  eq ord(X | L,Y | L2) = ord.E(X,Y) or ((not ord.E(Y,X)) and ord(L,L2)) .
}
```

The remaining parts are the same as what are declared in GLIST-ERR.

Generic Quick Sort

```

mod! NATLIST9 {
  pr(GLIST-ERR-ORD(E <= TRIV-ERR-ORD2NAT-ERR)
    * {sort List -> NatList,
      sort Nil -> NLNil,
      sort NnList -> NnNatList,
      sort ErrList -> ErrNatList,
      sort List&Err -> NatList&Err,
      op nil -> nlnil,
      op errList -> errNatList } )
}

```

Generic Quick Sort

```

view TRIV-ERR-ORD2NATLIST9 from TRIV-ERR-ORD to NATLIST9 {
  sort Elt -> NatList,
  sort Err -> ErrNatList,
  sort Elt&Err -> NatList&Err,
  op err -> errNatList,
  op ord -> ord,
}

open GQSORT(E <= TRIV-ERR-ORD2NATLIST9) .
  red qsort((1 | 2 | nlnil) | (3 | 0 | 2 | nlnil) | (2 | nlnil) |
    (0 | nlnil) | nlnil | (1 | 1 | nlnil) | (3 | 0 | 1 | nlnil) |
    (1 | 0 | nlnil) | nil) .
close

```

Generic Quick Sort

```

mod! STRLIST1 {
  pr(GLIST-ERR-ORD(E <= TRIV-ERR-ORD2STRING-ERR))
    * {sort List -> StrList,
      sort Nil -> SLNil,
      sort NnList -> NnStrList,
      sort ErrList -> ErrStrList,
      sort List&Err -> StrList&Err,
      op nil -> slnil,
      op errList -> errStrList } )
}

```

Generic Quick Sort

```

view TRIV-ERR-ORD2STRLIST1 from TRIV-ERR-ORD to STRLIST1 {
  sort Elt -> StrList,
  sort Err -> ErrStrList,
  sort Elt&Err -> StrList&Err,
  op err -> errStrList,
  op ord -> ord,
}

open GQSORT(E <= TRIV-ERR-ORD2STRLIST1) .
  red qsort(("CafeOBJ" | "Fortran" | slnil) |
    ("Java" | "C" | "Fortran" | slnil) | ("Fortran" | slnil) |
    ("C" | slnil) | slnil | ("CafeOBJ" | "CafeOBJ" | slnil) |
    ("Java" | "C" | "CafeOBJ" | slnil) | ("CafeOBJ" | "C" | slnil) | nil) .
close

```

Exercises

1. Type each module used in the slides and some test code (enclosed with **open** and **close**) in one file and feed it into the CafeOBJ system.
2. Write a parametrized module in which the generic merge sort is described and some test code for lists of natural numbers, lists of strings, lists of lists of natural numbers and lists of lists of strings.
3. Write a parametrized module in which the generic bubble sort is described and some test code for lists of natural numbers, lists of strings, lists of lists of natural numbers and lists of lists of strings.

Exercises

4. Write a parametrized module in which the generic insertion sort is described and some test code for lists of natural numbers, lists of strings, lists of lists of natural numbers and lists of lists of strings.
5. Write a parametrized module in which the generic selection sort is described and some test code for lists of natural numbers, lists of strings, lists of lists of natural numbers and lists of lists of strings.

Exercises

6. Modern programming languages are equipped with what corresponds to parametrized modules, such as functors of Standard ML and templates of C++. Investigate them and make a comparison of parameterized modules with them.

7. Parameterized modules can be used to partially implement what can be done by higher-order functions. Investigate it and write some examples in CafeOBJ. See the following:

<https://cseweb.ucsd.edu/~goguen/pps/utyop.pdf>

Exercises

8. Terms can be defined in terms of operators in CafeJOB, which needs to use parameterized modules, and term rewriting can be implemented in CafeOBJ. Do them. See the following: <https://dl.acm.org/doi/10.1145/338407.338558>

9. Modules including parameterized ones can be defined in terms of operators in CafeJOB. Do this and implement some functions with which modules can be handled. This has something to do with reflective (or meta) programming. See the following: <https://maude.lcc.uma.es/maude30-manual-html/maude-manual.html>, especially Chapter 17.

Exercises

10. The theoretical background of parameterized module is Theory of Institution. Investigate Theory of Institution, for example, by reading the following paper:

<https://dl.acm.org/doi/10.1145/147508.147524>

Note that you need to learn Category Theory to understand Theory of Institution.

11. Theory of Institution can be used to describe logics. Investigate how to use Theory of Institutions to describe logics.