I217: Functional Programming

6. Infinite Lists

Kazuhiro Ogata, Canh Minh Do

Roadmap

- E-strategy
- Infinite Lists
- Sieve of Eratosthenes
- Hamming's Problem
- Simulator of a Mutex Protocol

E-strategy

Let us consider triples of natural numbers and how to reduce 1st((1 + 1,2 + 2,3 + 3)) that has more than one redex.

1st((1 + 1, 2 + 2, 3 + 3))

What selects one among multiple redexes is called a reduction *strategy*.

Left-most inner-most strategy

1st((1 + 1,2 + 2,3 + 3)) → 1st((2,2 + 2,3 + 3)) by (+)
→ 1st((2,4,3 + 3)) by (+)
→ 1st((2,4,6)) by (+)
→ 2 by (1st)

(Left-most) outer-most strategy

1st((1 + 1,2 + 2,3 + 3)) → 1 + 1 by (1st)
→ 2 by (+)

E-strategy

```
mod! NAT-TRIPLE-IMLM {  pr(NAT)  [NatTriple]
 op (_,_,_) : Nat Nat Nat -> NatTriple {constr} .
 op 1st : NatTriple -> Nat {strat: (1 0)} .
 vars FE SE TE : Nat .
 eq 1st((FE,SE,TE)) = FE .  }
```

the local strategy

```
set trace whole on
open NAT-TRIPLE-IMLM .
 red 1st((1 + 1,2 + 2,3 + 3)) .
close
set trace whole off
```

```
-- reduce in %NAT-TRIPLE-IMLM : (1st(((1 + 1) , (2 + 2) , (3 + 3)))):Nat
[1]: (1st(((1 + 1) , (2 + 2) , (3 + 3)))):Nat
---> (1st((2 , (2 + 2) , (3 + 3)))):Nat
[2]: (1st((2 , (2 + 2) , (3 + 3)))):Nat
---> (1st((2 , 4 , (3 + 3)))):Nat
[3]: (1st((2 , 4 , (3 + 3)))):Nat
---> (1st((2 , 4 , 6))):Nat
[4]: (1st((2 , 4 , 6))):Nat
---> (2):NzNat
(2):NzNat
```

[1] 1 + 1 is rewritten to 2.

[2] 2 + 2 is rewritten to 4.

[3] 3 + 3 is rewritten to 6.

[4] Since 1st(2,4,6) is a redex, one-step rewrite is performed.

E-strategy

```
mod! NAT-TRIPLE-OMLM {  pr(NAT)  [NatTriple]
 op (_,_,_) : Nat Nat Nat -> NatTriple {constr} .
 op 1st : NatTriple -> Nat {strat: (0 1 0)} .
 vars FE SE TE : Nat .
 eq 1st((FE,SE,TE)) = FE .  }
```

the local strategy

```
set trace whole on
open NAT-TRIPLE-OMLM .
 red 1st((1 + 1,2 + 2,3 + 3)) .
close
set trace whole off
```

```
-- reduce in %NAT-TRIPLE-OMLM : (1st(((1 + 1) , (2 + 2) , (3 + 3)))):Nat
[1]: (1st(((1 + 1) , (2 + 2) , (3 + 3)))):Nat
---> (1 + 1):NzNat
[2]: (1 + 1):NzNat
---> (2):NzNat
(2):NzNat
```

[1] Since the given term is a redex, one-step rewrite is performed.

[2] 1 + 1 is rewritten to 2.

E-strategy

op 1st : NatTriple -> Nat {**strat**: (1 0)} .

The local strategy (1 0) given to the operator 1st says that when reducing a ground term 1st(*arg*), CafeOBJ first reduces the 1st argument *arg* to *arg'* and performs one-step rewite to 1st(*arg'*) if 1st(*arg'*) is a redex; otherwise CafeOBJ returns 1st(*arg'*) as the result; after the one-step rewrite CafeOBJ reduces the contract based on the local strategy given to the top operator of the contract.

op 1st : NatTriple -> Nat {**strat**: (0 1 0)} .

(0 1 0) says that when reducing a ground term 1st(*arg*), CafeOBJ first performs one-step rewite to 1st(*arg*) if 1st(*arg*) is a redex; otherwise CafeOBJ does the same as it does for the local strategy (1 0); after the one-step rewrite CafeOBJ reduces the contract based on the local strategy given to the top operator of the contract.

E-strategy

The reduction strategy adopted by CafeOBJ is *E-strategy* that selects a redex based on local strategies given to operators as follows.

the local strategy

$$\textbf{op}\ f : S_1 \ \ldots\ S_n \text{ -> } S\ \{\textbf{strat}: (x_1\ \ldots\ x_m)\}\ .$$

where $0 \leqq x_i \leqq n$ for each $i = 1, \ldots, m$.

When reducing $f(a_1,\ldots,a_n)$ referred to as t,

Note that t may be modified in the following.

for $j = x_1, \ldots, x_m$,

(1) if $j = 0$, CafeOBJ performs one-step rewrite if t is a redex, reduces the contract and returns the result of reducing the contract as the result;

(2) otherwise, CafeOBJ reduces t_j;

Note that t may not be the same as $f(a_1,\ldots,a_n)$ in a middle of the procedure.

CafeOBJ returns t as the result.

(Note that the result may contain redexes.)

E-strategy

Operators can be given local strategies by human users.

If human users do not so, the CafeOBJ system give some adequate local strategies to operators.

```
mod! NAT-IF {  pr(NAT)
  op if_then{_}else{_} : Bool Nat Nat -> Nat .
  vars N1 N2 : Nat .
  eq if true then {N1} else {N2} = N1 .
  eq if false then {N1} else {N2} = N2 .  }
```

The CafeOBJ system gives (1 0 2 3) to if_then{_}else{_} as the local strategy.

```
show NAT-IF .
```

Infinite Lists

Lists that consist of an infinite number of elements

Based on the following

```
op nil : -> Nil {constr} .
op _|_ : Elt.E List -> NnList {constr} .
```

any lists that consist of an arbitrary finite number of elements can be constructed but any infinite lists cannot.

Infinite Lists

One way to describe infinite lists in CafeOBJ is as follows:

```
mod! INF-LIST(E :: TRIV) {
  [InfList]
  op _|_ : Elt.E InfList -> InfList {strat: (1 0)} .
}
```

Note that this local strategy is given to _|_

$e_1 \mid e_2 \mid \ldots \mid e_n \mid il$ is a term of InfList if $e_1, e_2, \ldots, e_n$ are terms of Elt.E and il is a term of InfList.

```
mod! GLIST(E :: TRIV) {
  [Nil NnList < List]
  op nil : -> Nil {constr} .
  op _|_ : Elt.E List -> List {constr} .
}
```

The module INF-LIST imports the modules GLIST and NAT:

```
pr(NAT)
pr(GLIST(E))
```

Infinite Lists

Some functions for infinite lists:

```
op take : InfList Nat -> List .
eq take(IL,0) = nil .
eq take(X | IL, NzN) = X | take(IL,p NzN) .

op drop : InfList Nat -> InfList .
eq drop(IL,0) = IL .
eq drop(X | IL, NzN) = drop(IL,p NzN) .
```

Note that the following variables are declared in INF-LIST:

```
vars X Y : Elt.E .
vars IL IL2 : InfList .
var NzN : NzNat .
var N : Nat .
var L : List .
```


Infinite Lists

```
op _@_ : List InfList -> InfList .
eq nil @ IL = IL .
eq (X | L) @ IL = X | (L @ IL) .

op zip : InfList InfList -> InfList .
eq zip(X | IL,Y | IL2) = X | Y | zip(IL,IL2) .
```

Infinite Lists

```
mod! NAT-INF-LIST {  pr(INF-LIST(NAT)) .
  op mkNILFrom : Nat -> InfList .
  op if_then{_}else{_} : Bool InfList InfList -> InfList .
  var N : Nat . vars IL1 IL2 : InfList .
  eq mkNILFrom(N) = N | mkNILFrom(N + 1) .
  eq if true then {IL1} else {IL2} = IL1 .
  eq if false then {IL1} else {IL2} = IL2 .  }

open NAT-INF-LIST .
  red mkNILFrom(0) .
  red take(mkNILFrom(0),10) .
  red drop(mkNILFrom(0),10) .
  red take(drop(mkNILFrom(0),997),10) .
  red take(take(mkNILFrom(0),10) @ drop(mkNILFrom(0),10),20) .
  red take(mkNILFrom(0),20) .
  red zip(mkNILFrom(0),mkNILFrom(0)) .
  red take(drop(zip(mkNILFrom(0),mkNILFrom(0)),997),10) .
close
```


Sieve of Eratosthenes

```
mod! ERATOSTHENES-SIEVE {
  pr(NAT-INF-LIST)
  op primes : -> InfList .
  op sieve : InfList -> InfList .
  op check : Nat InfList -> InfList .
  --
  vars X Y : Nat .
  var NzX : NzNat .
  var IL : InfList .

  -- primes
  eq primes = sieve(mkNILFrom(2)) .
  -- sieve
  eq sieve(X | IL) = X | sieve(check(X,IL)) .
  -- check
  eq check(0,IL) = IL .
  eq check(NzX,Y | IL)
     = if NzX divides Y then {check(NzX,IL)}
                        else {Y | check(NzX,IL)} .
}
```

Sieve of Eratosthenes

```
open ERATOSTHENES-SIEVE .
  red primes .
  red take(primes,10) .
  red take(primes,20) .
  red take(primes,50) .
  red take(primes,100) .
close
```


Hamming's Problem

```
mod! HAMMING {
  pr(NAT-INF-LIST)
  op ham : -> InfList .
  op 2* : InfList -> InfList .
  op 3* : InfList -> InfList .
  op 5* : InfList -> InfList .
  op merge : InfList InfList -> InfList .
  vars X Y : Nat .
  vars IL IL2 : InfList .
  -- ham
  eq ham = 1 | merge(merge(2*(ham),3*(ham)),5*(ham)) .
```

Hamming's Problem

```
 -- 2*
  eq 2*(X | IL) = 2 * X | 2*(IL) .
  -- 3*
  eq 3*(X | IL) = 3 * X | 3*(IL) .
  -- 5*
  eq 5*(X | IL) = 5 * X | 5*(IL) .
  -- merge
  eq merge(X | IL,Y | IL2)
     = if X < Y
       then {X | merge(IL,Y | IL2)}
       else {if Y < X
             then {Y | merge(X | IL,IL2)}
             else {X | merge(IL,IL2)} } .
}
```


Hamming's Problem

```
open HAMMING .
  red ham .
  red take(ham,10) .
  red take(ham,20) .
  red take(ham,50) .
  red take(ham,100) .
close
```

Simulator of a Mutex Protocol

Let us consider a multi-threaded program in which multi-threads are running simultaneously and sharing some resources, such as objects.

Some shared resources must be used mutually exclusively by at most one thread at any given moment.

A mechanism to achieve this is called a *mutual exclusion (Mutex) protocol*.

Simulator of a Mutex Protocol

Suppose that there are two threads t1 & t2 that share a Boolean variable *locked* whose initial value is false and execute the following pseudo-code:

Each thread does something that requires some shared resources in "Critical Section".

Loop: "Remainder Section"
rs: **while** *locked* = true {}
ms: *locked* := true;
"Critical Section"
cs: *locked* := false;

Each thread does something that does not require any shared resources in "Remainder Section".

Each thread *t* is at rs, ms or cs. When *t* is at rs, it can check if *locked* is false. If so, it moves to ms and sets *locked* true, entering "Critical Section" and doing something that requires some shared resources. Otherwise, it stays at rs. When *t* is at cs, it sets *locked* false, going back to "Remainder Section".

Simulator of a Mutex Protocol

Each state of the protocol is characterized by three values: *locked*, the location of t1 and the location of t2. Therefore, a state of the protocol is expressed as the following record:

$$(\text{locked: } b, \text{ pc1: } l_1, \text{ pc2: } l_2)$$

where b is a Boolean value and l_1 and l_2 are rs, ms or cs.

```
mod! STATE principal-sort State {
  pr(LOC)
  [State]
  op (locked:_,pc1:_,pc2:_) : Bool Loc Loc -> State {constr} .
}
```


Simulator of a Mutex Protocol

```
mod! TID {
  [Tid]
  ops t1 t2 : -> Tid {constr} .
  op if_then{_}else{_} : Bool Tid Tid -> Tid .
  vars T1 T2 : Tid .
  eq if true then {T1} else {T2} = T1 .
  eq if false then {T1} else {T2} = T2 .
}

mod! LOC {
  [Loc]
  ops rs ms cs : -> Loc {constr} .
}
```

Simulator of a Mutex Protocol

Given a state and a thread, the next state can be determined.

(locked: false, pc1: rs, pc2: rs) —t1→ (locked: false, pc1: ms, pc2: rs)

(locked: false, pc1: ms, pc2: rs) —t2→ (locked: false, pc1: ms, pc2: ms)

(locked: false, pc1: ms, pc2: ms) —t2→ (locked: true, pc1: ms, pc2: cs)

Simulator of a Mutex Protocol

```
mod! FMUTEX { pr(STATE) pr(TID)
 op trans : State Tid -> State .
 vars L1 L2 : Loc . var B : Bool .
 -- t1
 eq trans((locked: true,pc1: rs,pc2: L2),t1)
   = (locked: true,pc1: rs,pc2: L2) .
 eq trans((locked: false,pc1: rs,pc2: L2),t1)
   = (locked: false,pc1: ms,pc2: L2) .
 eq trans((locked: B,pc1: ms,pc2: L2),t1)
   = (locked: true,pc1: cs,pc2: L2) .
 eq trans((locked: B,pc1: cs,pc2: L2),t1)
   = (locked: false,pc1: rs,pc2: L2) .
```

Simulator of a Mutex Protocol

```
  -- t2
  eq trans((locked: true,pc1: L1,pc2: rs),t2)
     = (locked: true,pc1: L1,pc2: rs) .
  eq trans((locked: false,pc1: L1,pc2: rs),t2)
     = (locked: false,pc1: L1,pc2: ms) .
  eq trans((locked: B,pc1: L1,pc2: ms),t2)
     = (locked: true,pc1: L1,pc2: cs) .
  eq trans((locked: B,pc1: L1,pc2: cs),t2)
     = (locked: false,pc1: L1,pc2: rs) .
}
```


Simulator of a Mutex Protocol

The simulator takes a state and an infinite list of thread IDs (called a scheduling), and generates an infinite list of states (called a computation).

```
mod! COMP {
  pr(INF-LIST(STATE)
     * {sort InfList -> Comp, sort List -> FComp} )
}

mod! SCHED { pr(NAT)
  pr(INF-LIST(TID) * {sort InfList -> Sched} )
  op sched : Nat -> Sched .
  var N : Nat .
  eq sched(N) = if 2 divides N
                then {t1 | sched(N quo 2)} else {t2 | sched(N quo 2)} .
}
```

n of sched(*n*) determines when to choose which threads, namely the scheduling.

Simulator of a Mutex Protocol

```
open SCHED .
  red take(sched(123),10) .
  red take(sched(1234),10) .
  red take(sched(12345),10) .
close
```


Simulator of a Mutex Protocol

```
mod! SIM {  pr(FMUTEX)  pr(COMP)  pr(SCHED)
  op sim : State Nat -> Comp .
  op sub-sim : State Sched -> Comp .
  var S : State .  var N : Nat .  var NzD : NzNat .
  var T : Tid .  var TIL : Sched .
  eq sim(S,N) = sub-sim(S,sched(N)) .
  eq sub-sim(S,T | TIL) = S | sub-sim(trans(S,T),TIL) .
}

open SIM .
  red take(sim((locked: false,pc1: rs,pc2: rs),123),10) .
  red take(sim((locked: false,pc1: rs,pc2: rs),1234),10) .
  red take(sim((locked: false,pc1: rs,pc2: rs),12345),10) .
close
```

Simulator of a Mutex Protocol

One desired property mutex protocols should satisfy is called the *mutex property* that there is at most one thread in "Critical Section" at any given moment. The simulator is revised so that it can check in a specified depth if each state generated satisfies the property. If the simulator finds a state in which the property is broken, it returns the computation fragment leading to the state.

The property is defined as follows:

```
op mutex : State -> Bool .
vars L1 L2 : Loc .
var B : Bool .
eq mutex((locked: B,pc1: L1,pc2: L2)) = not (L1 == cs and L2 == cs) .
```


Simulator of a Mutex Protocol

The revised simulator is as follows:

```
op sim-check : State Nat Nat -> FComp .
op sub-sim-check : State Sched Nat -> FComp .
var D : Nat .  var NzD : NzNat .
eq sim-check(S,N,D) = sub-sim-check(S,sched(N),D) .
eq sub-sim-check(S,T | TIL,0) = S | nil .
eq sub-sim-check(S,T | TIL,NzD)
  = if mutex(S) then {S | sub-sim-check(trans(S,T),TIL,p NzD)}
                else {S | nil} .
open SIM .
 red sim-check((locked: false,pc1: rs,pc2: rs),123,10) .
 red sim-check((locked: false,pc1: rs,pc2: rs),1234,10) .
 red sim-check((locked: false,pc1: rs,pc2: rs),12345,10) .
close
```

As you can see, the protocol does not satisfy the property. Please see Appendix for a correct version of the protocol.

Exercises

1. Write all programs in the slides and feed them into the CafeOBJ system. Moreover, write some more test code and do some more testing for the programs.

2. Revise the simulator (including the revised one) so that it can deal with the case in which there are four threads.

3. CafeOBJ automatically gives a local strategy to an operator when a programmer does not explicitly give any local strategies to the operator. Investigate how CafeOBJ does so.

Exercises

4. Lazy evaluation corresponds to outermost strategies and is adopted by some functional programming languages, such as Haskell. Lazy evaluation is convenient because it makes it possible to deal with infinite lists as you have learned in this lecture note, but is not good from a running performance point of view. Thus, strictness analysis is often used. Investigate strictness analysis.

5. Investigate why eager evaluation (or call-by-value) can be more efficiently implemented than lazy evaluation from a running performance point of view. Come up with an efficient way to implement lazy evaluation.

Exercises

6. How to deal with infinite lists in this lecture note is rather casual. One possible way to precisely deal with infinite data structures, such as infinite lists, is co-induction. Investigate co-induction and how to deal with infinite lists with co-induction. Note that all data structures that are inductively defined are finite, such as terms and ordinary lists.

7. Investigate hidden algebra (or behavioral specifications) that has something to do with co-induction. Note that hidden algebra is one theoretical basis of CafeOBJ.

Appendix

A correct version of the protocol:

Loop: "Remainder Section"
rs: **while** test&set(*locked*) = true {}
"Critical Section"
cs: *locked* := false;

test&set(x) does the following atomically:

tmp := x;
x := true;
return *tmp*;