An Introduction to Functional Programming and Maude

Guoqiang Li

BASICS, Shanghai Jiao Tong University

October 26, 2009

Begin With...

- λ calculus

$$M := x \mid \lambda x.M \mid MM$$

- π calculus

$$\pi := a(b) \mid \bar{a}b \mid \tau$$

$$\varphi := \top \mid \bot \mid x = y \mid x \neq y \mid \varphi \wedge \varphi$$

$$P := \sum_{i \in I} \varphi_i \pi_i . P_i \mid P|P \mid (x)P \mid !P$$

Implementation

- Traditional approaches
  - parser: Yacc.
  - represented by some data structure: list, tree, acyclic graph. etc.
  - search..
- What if a natural number 327
  - Naive, since we have type of int.
- What if we define a type of λ calculus and π calculus?

Implementation

- Traditional approaches
  - parser: Yacc.
  - represented by some data structure: list, tree, acyclic graph. etc.
  - search..
- What if a natural number 327
  - Naive, since we have type of int.
- What if we define a type of λ calculus and π calculus?

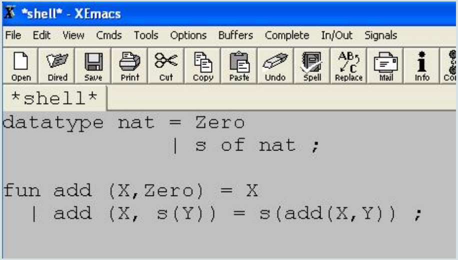
Implementation

- Traditional approaches
  - parser: Yacc.
  - represented by some data structure: list, tree, acyclic graph. etc.
  - search..
- What if a natural number 327
  - Naive, since we have type of int.
- What if we define a type of λ calculus and π calculus?

Implementation

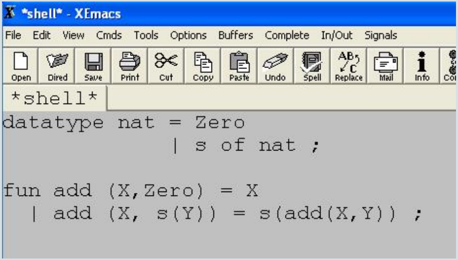
- Traditional approaches
  - parser: Yacc.
  - represented by some data structure: list, tree, acyclic graph. etc.
  - search..
- What if a natural number 327
  - Naive, since we have type of int.
- What if we define a type of λ calculus and π calculus?

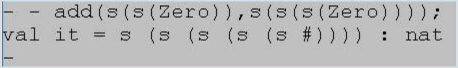
Type and Pattern Matching




```
- - add(s(s(Zero)),s(s(s(Zero))));
val it = s (s (s (s (s #)))) : nat
-
```

Type and Pattern Matching





How to define a set of variables?

Function

- Mathematical view: a function is a relation, where

$$x\,R\,y \wedge x\,R\,z \rightarrow y = z$$

- Logical/Rewriting view: confluence, describing that terms in this system can be rewritten in more than one way, to yield the same result.
- Programming view: a function is a program procedure that you can work out.
  - Such a function can be regarded as a term with only one redex.

Function

- Mathematical view: a function is a relation, where

$$x \, R \, y \land x \, R \, z \rightarrow y = z$$

- Logical/Rewriting view: confluence, describing that terms in this system can be rewritten in more than one way, to yield the same result.
- Programming view: a function is a program procedure that you can work out.
  - Such a function can be regarded as a term with only one redex.

Function

- Mathematical view: a function is a relation, where

$$x\,R\,y \wedge x\,R\,z \rightarrow y = z$$

- Logical/Rewriting view: confluence, describing that terms in this system can be rewritten in more than one way, to yield the same result.
- Programming view: a function is a program procedure that you can work out.
  - Such a function can be regarded as a term with only one redex.

What Can Functional Programming Do

- Programming Language: SML, Haskel, OCaml, SML#, Visual SML, Erlang?,...
- Theorem Proving: Isabelle/HOL, Coq, CafeOBJ,...
- Model Checking: Maude

What Can Functional Programming Do

- Programming Language: SML, Haskel, OCaml, SML#, Visual SML, Erlang?,...
- Theorem Proving: Isabelle/HOL, Coq, CafeObJ,...
- Model Checking: Maude

What Can Functional Programming Do

- Programming Language: SML, Haskel, OCaml, SML#, Visual SML, Erlang?,. . .
- Theorem Proving: Isabelle/HOL, Coq, CafeObJ,. . .
- Model Checking: Maude

What Is Maude

- Maude is a rewriting system...
  - $f(x, y) \hookrightarrow g(x)$
  - $f(f(a, z), b) \hookrightarrow g(f(a, z))$
  - $f(f(a, z), b) \hookrightarrow f(g(a), b)$
- Maude encodes both equational logic and rewriting logic...
  - An equational logic theory: $(\Sigma, E \cup A)$
  - a rewriting Logic theory: $(\Sigma, E \cup A, \phi, R)$
- Maude is a (programmable) model checker...
  - Maude provides `search` and LTL engines, which can do model checking on an established system.
- Maude is a functional programming language.

Categories of Maude

- Core Maude: functional module + system module
- Full Maude: Core Maude + object-oriented module
- Real-Time Maude: Full Maude + timed module
- Mobile Maude
- ...

	functional module	system module
syntax	`fmod ...endfm`	`mod ...endm`
rewriting	confluent & terminated	divergent & non-terminated
logic	equational logic	rewriting logic
programming lang.	sequential	concurrent

The First example

shell - XEmacs

```
datatype nat = Zero
             | s of nat ;

fun add (X,Zero) = X
  | add (X, s(Y)) = s(add(X,Y)) ;
```

```
- - add(s(s(Zero)),s(s(s(Zero))));
val it = s (s (s (s (s #)))) : nat
-
```

Nat.maude - XEmacs

```
fmod Nat is

sort Nat .

op  0 : -> Nat [ctor] .
op  s  : Nat -> Nat [ctor] .

op  add : Nat Nat -> Nat .

vars X Y : Nat .

eq add (X, 0) = X .
eq add (X, s(Y)) = s(add (X,Y)) .

endfm
```

```
Maude> in Nat.maude
Advisory: defining module Nat.
==========================================
fmod Nat
Maude> reduce add (s(s(0)), s(s(s(0)))) .
==========================================
reduce in Nat : add(s(s(0)), s(s(s(0)))) .
rewrites: 4 in -1520340230ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(s(0)))))
Maude>
```

Functional modules

- A basic functional module mainly has four parts: sorts, operations, variables and equations. For example:
  - ```
    fmod NAT is
         sort Nat .
         op 0 :  -> Nat [ctor] .
         op s :  Nat -> Nat [ctor] .
         op add :  Nat Nat -> Nat .
         vars X Y : Nat .
         eq add (X, 0) = X .
         eq add (X, s(Y)) = s( add(X,Y) ) .
    endfm
    ```
    ```

Functional modules

- A basic functional module mainly has four parts: sorts, operations, variables and equations. For example:
  - ```
    fmod NAT is
         sort Nat .
         op 0 :  -> Nat [ctor] .
         op s :  Nat -> Nat [ctor] .
         op add :  Nat Nat -> Nat .
         vars X Y : Nat .
         eq add (X, 0) = X .
         eq add (X, s(Y)) = s( add(X,Y) ) .
    endfm
    ```
    ```

Functional modules

- A basic functional module mainly has four parts: sorts, operations, variables and equations. For example:
  - ```
    fmod NAT is
          sort Nat .
          op 0 :  -> Nat [ctor] .
          op s :  Nat -> Nat [ctor] .
          op add :  Nat Nat -> Nat .
          vars X Y : Nat .
          eq add (X, 0) = X .
          eq add (X, s(Y)) = s( add(X,Y) ) .
    endfm
    ```
    ```

Functional modules

- A basic functional module mainly has four parts: sorts, operations, variables and equations. For example:
  - ```
    fmod NAT is
         sort Nat .
         op 0 :  -> Nat [ctor] .
         op s :  Nat -> Nat [ctor] .
         op add :  Nat Nat -> Nat .
         vars X Y : Nat .
         eq add (X, 0) = X .
         eq add (X, s(Y)) = s( add(X,Y) ) .
    endfm
    ```
    ```

Functional modules

- A basic functional module mainly has four parts: sorts, operations, variables and equations. For example:
    - ```
      fmod NAT is
          sort Nat .
          op 0 :  -> Nat [ctor] .
          op s :  Nat -> Nat [ctor] .
          op add :  Nat Nat -> Nat .
          vars X Y : Nat .
          eq add (X, 0) = X .
          eq add (X, s(Y)) = s( add(X,Y) ) .
      endfm
      ```
      ```

Sorts and Variables

- Maude can define a sort or several sorts each a time, with the key words `sort` or `sorts`.
  - `sort Nat .`
  - `sorts Nat Integer Real .`
- Maude can also declare subsorts, which is defined as follows:
  - `subsort Nat < Integer .`
  - `subsorts Nat < Integer < Real .`
- Maude can define kinds for handling subsorts.
- Variables are declared with the key words `var` or `vars`.
  - `var X : Nat .`
  - `vars C1 C2 C3 :  Integer .`

Operations

- There are two uses of operations: as the constructor of a sort, and as the declaration of a function.
- The latter needs to be implemented by some equations.
- `[ctor]` is a key attribute to a constructor,
  - ```
    sort Nat .
      op 0 :  -> Nat [ctor] .
      op s :  Nat -> Nat [ctor] .
    ```
  - ```
    sort Color .
      ops blue green red :  -> Color [ctor] .
    ```
- As a declaration of a function. It can be represented in an mix-fix notation, and _ is a specific place for a variable. For example,
  - `op _+_ :  Nat Nat -> Nat .`
  - `oCheck :  Message Message -> Bool .`

Attributes for Operations

- Equational Attribute: `assoc`, `comm`, `idem`, `id: <term>...`
  - `op _XOR_ : Term Term -> Term [assoc comm id: ZERO] .`
- Memorized Attribute: `memo`, which instructs Maude to memorize the result.
  - `op fibo : Nat -> Nat [memo] .`
- Frozen Attribute: `frozen`, which forbids to apply rules to the proper subitems of a term.
- Special Attribute: `special`, which is associated with appropriate C++ code by hooks.

Equations

- A function can be implemented by a set of equations. The use of variables in equations do not carry actual values. Rather, they stand for any instance of a certain sort.
  - ```
    op _+_ :  Nat Nat -> Nat .
    vars M N : Nat .
      eq 0 + N = N .
      eq s(M) + N = s(M + N).
    ```
- A conditional equation can be defined in two ways:
  - `ceq isdifferent (M, N) = true if M =/= N .`
  - ```
    eq isdifferent (M, N) = if M == N then true
                                       else false fi .
    ```
- A default equation is defined by a key attribute `[owise]`
  - `eq oCheck (M1, M2) = false [owise] .`

Importation

- A module can be imported in another module by using key words `protecting`, `extending` or `including`. For example:
  - ```
    fmod PARENT is
    ...
    endfm
    ```
  - ```
    fmod CHILD is
    protecting PARENT .
    ...
    endfm
    ```
- `protecting` means that the imported module can not be modified in any way. `including` means one can change the definition of the imported module. `extending` falls somewhere between these two extremes.

Lambda Calculus

The λ-calculus

$$M := x \mid \lambda x.M \mid MM$$

Full λ-calculus

$$1\ \frac{}{(\lambda x.M)N \rightarrow M\{N/x\}} \quad \beta\text{-rule}$$

$$2\ \frac{M \rightarrow M'}{MN \rightarrow M'N} \quad \text{structure rule}$$

$$3\ \frac{N \rightarrow N'}{MN \rightarrow MN'} \quad \text{eager evaluation}$$

$$4\ \frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'} \quad \text{partial evaluation}$$

Lazy λ-calculus $1 + 2$

Lambda Calculus in Maude

```
fmod LAMBDA is
   pr NAT .
   sorts Var Lambda .
   subsort Var < Lambda .
   op var : Nat -> Var [ctor] .
   op \_._ : Var Lambda -> Lambda [ctor prec 15] .
   op __ : Lambda Lambda -> Lambda [ctor prec 20] .

   op beta : Lambda Lambda -> Lambda .
   op sub  : Lambda Var Lambda -> Lambda .
   op LazybetaRed : Lambda -> Lambda .

   vars M N O : Lambda . vars V W : Var .

   eq  beta (\ V . M, N) = sub (M, V, N) .

   eq  sub (V, V, N)   = N .
   ceq sub (W, V, N) = W if W =/= V .
   eq  sub (\ W . M, V, N) = \ W . (sub (M,V,N)) .
   eq  sub (M O, V, N ) = sub(M, V, N) sub (O, V, N) .
***   eq  sub (M , V, N) = M  [owise] .

  eq LazybetaRed (\ W . M O) = beta (\ W . M, O) .
  eq LazybetaRed (M N) = LazybetaRed (M) N [owise] .
  eq LazybetaRed (M) = M [owise] .


endfm
```

Function modules VS. System modules

- Anything such as equations defined in a function module can be a system module. Besides that, it can define a transition system by a set of rewrite laws.
  - A set of equations in a function module defines a structure. These equations need to be confluent and terminating.
  - Rewrite laws define transitions between structures. They may be nonterminating.
- ```
  mod CIGARETTES is
     sort State .
     op cig :  -> State [ctor] .
     op box :  -> State [ctor] .
     op _ _ :  State State -> State [ctor assoc comm] .

     rl [smoke] :  cig => box .
     rl [makenew] :  box box box box => cig .
  endm
  ```
  ```

Rewrite laws

- A transition system can be implemented by a set of rewrite laws. We often give each law a unique name in a bracket (optional), for example, `[makenew]`.
  - ```
    rl [smoke] :  cig => box .
    rl [makenew] :  box box box box => cig .
    ```
- A conditional rewrite law can also be defined.
  - `crl [equation] :  a(X) => b(X-1) if X > 0 .`
  - `crl [rewrite] :  b(X) => c(X*2) if a(X)=>b(Y) .`
- Usually, we can define an initial state to begin the rewriting
  - ```
     op init :  -> State .
      eq init = cig cig cig cig cig cig cig .
    ```

Common commands

- For a function module, a common command is `reduce`, which can reduce the normal form of a term.
  - ```
    reduce in NAT : s(s(0)) + s(s(s(0))) .
    result Nat :  s(s(s(s(s(0)))))
    ```
- For a system module,
  - A common command is `rewrite` (may not terminate),
    - ```
      rewrite in CIGARETTES : init .
      result State:  box
      ```
  - `search` begins with a given state, and finds out a given number of states that satisfies the property.
    - ```
       search [2] in CIGARETTES : init =>* ST
               such that ( number(cig,ST) == 1 ) .
      solution 1 (state 8)
      init -> cig box box box box box box
      solution 2 (state 12)
      init -> cig box box box
      ```
      ```

What can Maude do?

- Maude itself is a versatile tool supporting:
  - Formal specification;
  - Execution of the specification.
- Model checking: Reachability problem can be performed by Maude itself. Maude also offers a LTL model checker for system modules.
- Theorem proving: It can be performed by a theorem prover ITP implemented by Maude, based on membership equational logic.

Q: Can Maude encode Maude itself?

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - . . .

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - . . .

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

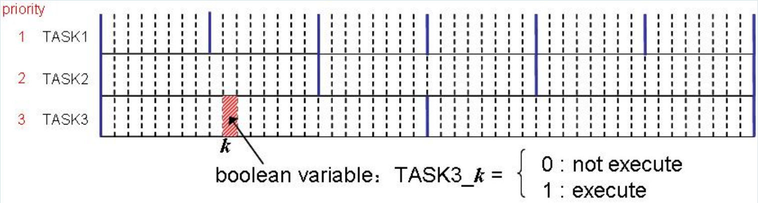
What Can We Do

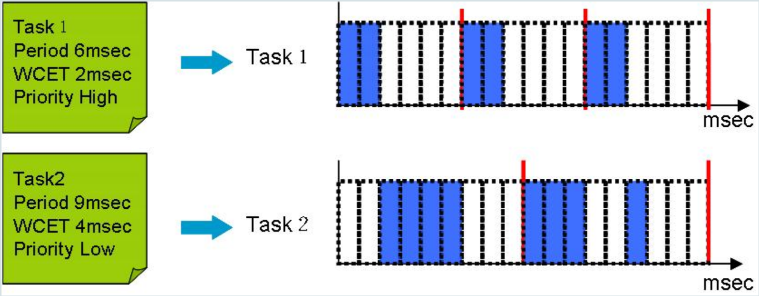
- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

What Can We Do

- Research
  - Aspect-Oriented Maude
  - Timed Automata Checker
  - Pushdown Automata Checker
  - Pi Calculus Theorem Prover
- Paper
  - Translate lambda calculus to pi calculus
  - System Simulator
  - Synthesis
  - ...

An Example: Schedulability Analysis





Clock Slot

```
fmod SLOT is
   pr NAT .
   sort Slot .
   op init : -> Slot [ctor] .
   op time : Nat -> Slot [ctor] .

   op Timeplus : Slot -> Slot .
   op getTime : Slot -> Nat .

   var N : Nat .

   eq Timeplus (init) = time(0) .
   eq Timeplus (time(N)) = time ( N + 1 ) .

   eq getTime (init) = 0 .
   eq getTime (time(N)) = N .

endfm
```

CPU

```
fmod CPU is
    pr SLOT .
    pr STRING .

    sort Cpu .
    sort CpuStatus .

    op idle : -> CpuStatus [ctor] .
    op init : -> CpuStatus [ctor] .
    op exec : String -> CpuStatus [ctor] .
    op CPU  : CpuStatus Slot -> Cpu [ctor] .
endfm
```

Task and Task Status

```
fmod TASK is
    pr NAT .
    pr STRING .
    sort Task .
    sorts Period Wcet Pri .
    op p : Nat -> Period [ctor] .
    op wcet : Nat -> Wcet [ctor] .
    op pri : Nat -> Pri  [ctor] .
    op task : String Period Wcet Pri -> Task [ctor] .
endfm
fmod TASKSTATUS is
   pr NAT .     pr STRING .
   sort TaskStatus .
   sorts Cp Tr .
   op (_,_,_,_) : String Cp Tr Bool -> TaskStatus [ctor] .
   op cp : Nat -> Cp [ctor] .
   op tr : Nat -> Tr [ctor] .
   op TaskExecutable : TaskStatus -> Bool .
endfm
```

Scheduling System

```
mod SCHEDULINGSYSTEM is
    pr CPU .  pr TASKSTATUS .
    pr TASK .
    sorts State SchedulingStatus .
    op Init : -> State .
    op exec : -> SchedulingStatus [ctor] .
    op error : -> SchedulingStatus [ctor] .
    op [_,_[_],_[_],_] : Cpu Task TaskStatus Task TaskStatus
                            SchedulingStatus -> State [ctor] .
    op getExecutedTask : Task TaskStatus Task TaskStatus Slot
                                         -> CpuStatus [memo] .
    op getTaskStatus : Task TaskStatus Slot CpuStatus ->
                                           TaskStatus [memo] .
    op getSchedulingStatus : Task TaskStatus Task TaskStatus
                          Slot CpuStatus -> SchedulingStatus .
endm
```

Scheduling System (cont.)

```
eq Init = [CPU(init,init),
           task ("TASK1", p(6), wcet(2), pri(2))
                    [("TASK1",cp(0),tr(2),true)],
           task ("TASK2", p(9), wcet(4), pri(1))
                    [("TASK2",cp(0),tr(4),true)],
                                                     exec ] .

rl [ex] : [CPU(CS,SL), T1[TS1], T2[TS2], exec ] =>
[ CPU(getExecutedTask(T1,TS1,T2,TS2,Timeplus(SL)),Timeplus(SL)),
  T1[ getTaskStatus(T1,TS1,Timeplus(SL),
                   getExecutedTask(T1,TS1,T2,TS2,Timeplus(SL))) ],
  T2[ getTaskStatus(T2,TS2,Timeplus(SL),
                   getExecutedTask(T1,TS1,T2,TS2,Timeplus(SL))) ],
  getSchedulingStatus(T1,TS1,T2,TS2,Timeplus(SL),
                    getExecutedTask(T1,TS1,T2,TS2,Timeplus(SL)))] .
```

What we have done?

- We have encoded a specification for scheduling algorithm.
- We can run the specification due to different commands (`reduce`, `rewrite`,...).
- We can perform schedulability analysis on the specification.

```
search [1] in SCHEDULINGSYSTEM : Init =>* [ CPU(CS, time(N1)),
T1[TS1], T2[TS2], exec ]
                                          such that  ( N1 == 18 )   .
```