MMT case study: the Mini-Freja language

Fabricio Chalub Christiano Braga

July 11, 2005

Contents

1 Introduction

This technical report describes the formal specification of the Mini-Freja [2] language using the *Maude MSOS Tool* [1].

Mini-Freja is a pure functional programming language with a normal-order semantics.[1] This specification was implemented with several purposes: it does not use Mosses's CMSOS, rather, giving the semantics for the language directly in MSDF; it does not need any external tools, parsing the language directly—with some limitations, as we shall see; it is a big-step semantics; and, finally, it has pattern matching capabilities.

1.1 Abstract Syntax

The main construction of Mini-Freja is the *expression*, denoted by the set '`Exp`'.

[1] Pettersson [2] uses "call-by-name"; we follow the terminology of Reynolds [3].

```
Exp .
Exp ::= fn Var => Exp
      | Primu Exp
      | Exp :: Exp [assoc]
      | Exp Primd Exp
      | if Exp then Exp else Exp
      | Exp Exp
      | rec Exp
      | case Exp of Rules
      | let Decls in Exp .

Exp ::= Var | Const .
```

'`Exp :: Exp`' is the Mini-Freja syntax of lists, '`Exp Exp`' is the traditional syntax for application of expressions, '`rec Exp`' defines a *recursive expression*, '`Var`' is the set of variables on the language (technically speaking, they are not variables, but identifiers, but we follow the nomenclature of Pettersson), '`Const`' is the set of constants, '`Primu`' are *unary* primitive operators, and '`Primd`' are *binary* primitive operations, defined as follows:

```
Primu .
Primu ::= not | neg .

Primd .
Primd ::= lt | le | eq | ne | ge | gt | and
        | or | plus | minus | times | div | mod .

Const .
Const ::= Int | Boolean | nil .
```

Finally, we present the abstract syntax for the declaration of bindings. The syntax is slightly altered from the original, due to preregularity problems. Instead of binding variables to expression with syntax '`Var = Exp`' we use '`Var is Exp`'.

```
Decls .
Dec  .
Dec ::= Var is Exp .
Decls ::= Dec | Decls Decls [assoc] .
```

1.2 Semantics

We now describe the dynamic semantics for the Mini-Freja. This specification is based on Pettersson's and Hartel's specifications of the Mini-Freja language, in big-step operational semantics. Mini-Freja, as we mentioned, is a *normal-order* language, that is, an expression may be "suspended" and is only evaluated until it is clear that its value is needed. The only semantic component needed for this specification is the bindings environment.

```
Label = { env : Env, ... } .
```

We now add the set of values to the specification, which consist basically of constants, closures, lists and suspended expressions. Suspended expressions need an environment to be evaluated in the future.

```
Values .

Value ::= susp (Env, Exp)
        | clo (Env, Var, Exp)
        | cons (Value, Value) [assoc]
        | Const .
```

Values are a subset of expressions. We also add a new expression operator '`force e`' that forces the evaluation of a suspended expression `e`.

```
Exp ::= Value | force Exp .
```

Let us begin with the evaluation of lists in Mini-Freja, which are into a sequence of recursive applications of the '`cons`' operator.

```
[cons] (Exp1 :: Exp2) : Exp ={env = Env, -}=>
       cons (susp (Env, Exp1), susp (Env, Exp2)) .
```

The arithmetic operators are evaluated in the traditional big-step manner.

```
           (Exp1 ={X1}=> Value1), (Exp2 ={X2}=> Value2),
                         (Value1 Primd Value2 ={X3}=> Value3)
[prim-app] -- ---------------------------------------------
           (Exp1 Primd Exp2) : Exp ={X1 ; X2 ; X3}=> Value3 .
```

```
(Int1 plus Int2) : Exp ==> (Int1 + Int2) .
(Int1 times Int2) : Exp ==> (Int1 * Int2) .
(Int1 mod Int2) : Exp ==> (Int1 rem Int2) .
(Int1 minus Int2) : Exp ==> _-_(Int1, Int2) .

(Int1 eq Int2) : Exp ==> if (Int1 == Int2) then tt else ff fi .
(Int1 ne Int2) : Exp ==> if (Int1 == Int2) then ff else tt fi .
```

The following rule establishes that canonical forms always evaluate to themselves.

```
Const : Exp ==> Const .
```

In order to evaluate the conditional construction we define an auxiliary operation '`if-choose(b,`e_1`,`e_2`)`', which work as follows: if `b` is true, it evaluates to e_1, otherwise it evaluates to e_2.

```
Exp ::= if-choose (Value, Exp2, Exp3) .

[if-choose-tt] if-choose (tt, Exp2, Exp3) : Exp ==> Exp2 .
[if-choose-ff] if-choose (ff, Exp2, Exp3) : Exp ==> Exp3 .

     (Exp1 ==> Value),
     (if-choose (Value, Exp2, Exp3) ==> Exp),
     (Exp ==> Value')
[if] -- ------------------------------------------
     if Exp1 then Exp2 else Exp3 : Exp ==> Value' .
```

Closures evaluate to its value form, '`clo(`ρ`,`ν`,`$e`)`', consisting on the "captured" environment ρ, the argument ν, and the closure expression e.

```
[clo] (fn Var => Exp) : Exp ={env = Env, -}=>
      clo (Env, Var, Exp) .
```

The following rules specify the meaning of the '`force`' operator. Essentially, non-suspended expressions evaluate to themselves. Suspended expressions (rule '`[force-susp]`') '`susp(`ρ`,`$e`)`' are evaluated by replacing the current environment with ρ and evaluating e into an intermediate value ν, which is itself "forced" into the final value ν'.

```
[force-const] force Const : Exp ==> Const .

[force-clo] force clo (Env, Var, Exp) : Exp
            ==> clo (Env, Var, Exp) .

[force-cons] force cons (Value1, Value2) : Exp
        ==> cons (Value1, Value2) .

        Exp ={env = Env', -}=> Value,
                     force Value ={env = Env, -}=> Value'
[force-susp] -- ------------------------------------------
        force susp (Env', Exp) : Exp ={env = Env, -}=> Value' .
```

Application of expressions implements a type of β-reduction. It is expected that '`Exp2`' on rule '`[app]`' below evaluates to a closure.

```
      Exp1 ={env = Env, -}=> clo (Env1, Var1, Exp'),
       Env2 := (Var1 |-> susp (Env, Exp2)) / Env1,
        Exp' ={env = Env2, -}=> Value
[app] -- ------------------------------------------
      (Exp1 Exp2) : Exp ={env = Env, -}=> Value .
```

The rule for variables follows the traditional rules, with the additional requirement that the returned value must be "forced."

```
         Value := lookup (Var, Env),
         (force Value ={env = Env, ...}=> Value')
[lookup] -- ----------------------------------
         Var : Exp ={env = Env, ...}=> Value' .
```

The rule for the '`let`' operator evaluates the declarations '`Decls`' into a set of bindings '`dec(Env')`' and evaluates '`Exp`', overriding its environment with these bindings.

```
Decls ::= dec (Env) .

      Decls ={env = Env, -}=> dec (Env'),
      Env'' := Env' / Env, Exp ={env = Env'', -}=> Value
[let] -- ------------------------------------------------
      (let Decls in Exp) : Exp ={env = Env, -}=> Value .
```

In order to evaluate '`Decls`', each '`Dec`' is evaluated in turn the resulting environments are concatenated.

```
          Dec ={env = Env, -}=> dec (Env'),
          Env'' := Env' / Env,
              Decls ={env = Env'', -}=> dec(Env''')
 [decls]  -- ------------------------------------
          (Dec Decls) : Decls ={env = Env, -}=>
                            dec (Env' +++ Env''') .

              Dec ==> dec (Env')
 [decls] -- ------------------------
         Dec : Decls ==> dec (Env') .
```

The following rule is necessary to give the normal order evaluation: expressions are not evaluated as they are bound to variables; they are first converted into "suspended" values.

```
 [dec] (Var is Exp) : Dec ={env = Env, -}=>
        dec (Var |-> susp (Env, Exp)) .
```

Recursive functions in the language are implemented using a fixed point operator, following ideas present in Reynolds's book [3]. The expression '`Exp`' is expected to be a lambda-expression, such as '`fn` v `=>` e'.

```
               Exp (rec Exp) ==> Value
 [fixed-point] -- ---------------------
               rec Exp : Exp ==> Value .
```

The following rule evaluates an expression using the '`exec`' and '`strict`' constructions, the later is similar to the '`force`' construction with the exception that it operates over *values*, while '`force`' operators over *expressions*.

```
 Value ::= exec Exp | done Value .

        Exp ==> Value,
                  strict Value ==> Value'
 [exec] -- ----------------------------
        exec Exp : Exp ==> done Value' .
```

```
Value ::= strict Value .

[strict-const] strict Const : Value
                ==> Const .

[strict-clo] strict clo (Env, Var, Exp) : Value
              ==> clo (Env, Var, Exp) .

               strict Value1 ==> Value1',
                              strict Value2 ==> Value2'
[strict-cons] -- ----------------------------------
               strict cons (Value1, Value2) : Value
                         ==> cons (Value1', Value2') .

               Exp ={env = Env', -}=> Value,
               strict Value ={env = Env, -}=> Value'
[strict-susp] -- --------------------------------
               strict susp (Env', Exp) : Value
                         ={env = Env, -}=> Value' .
```

Mini-Freja has pattern matching capabilities similar to those of Standard ML. The pattern matcher is the '`case Exp of Rules`' construction, where '`Rules`' is a sequence of options to be matched, each with a resulting expression.

```
Rules .
Rule .

Rule ::= Pat => Exp .

Rules ::= Rule
        | Rules || Rules [assoc] .
```

Each rule is a pattern to be matched against, and the resulting expression. Patterns follow the same syntax of expressions: we may match against variables, constants, and lists.

```
Pat .
```

```
Pat ::= Pat :: Pat [assoc] .
Pat ::= p Const | p Var .
```

Unfortunately, due to preregularity issues, we need the coercion function '`p_`' that lifts constants and variables into the set of patterns. The problem happens with the list operator '`Pat :: Pat`': had we not used the coercion function '`p_`', a term like '`3 :: 5`' would not have a *least sort*, since it could be either a '`Pat`' or a '`Exp`'.

The following rules implement the pattern matcher of the Mini-Freja language. We begin by creating an additional operation '`case(ν,R)`', that matches a value ν (obtained from an expression) against a set `R` of rules.

```
Value ::= case (Value, Rules) .

       Exp ==> Value,
             case (Value, Rules) ==> Value'
[case] -- --------------------------------
       case Exp of Rules : Exp ==> Value' .
```

Matches are defined according to the following signature. When a match is successful, it evaluates to '`myes(ρ)`', where ρ is the binding resulting from the match. Otherwise, it evaluates to '`mno`'.

```
Match .
Match ::= myes (Env) | mno .
```

First, the base case, where the set `R` of rules consists of a single rule. The function '`match(ν,p)`' matches ν against pattern `p` and returns either '`myes(ρ)`', if the match is successful, or '`mno`' otherwise. If the value '`Value`' matches the pattern '`Pat`', then the expression '`Exp`' is evaluated by overriding the current environment with ρ.

```
Match ::= match (Value, Pat) .

       match (Value, Pat) ={env = Env, -}=> myes(Env'),
       Env'' := Env' / Env, Exp ={env = Env'', -}=> Value'
[case1] -- ---------------------------------------------
       case (Value, Pat => Exp) : Value
                                ={env = Env, -}=> Value' .
```

Now let us see the general case, in which there is at least two rules in R. The rule below specifies the following: the result of the matching of 'Value' against 'Pat' is given on to the 'case-choose' function, which will return a value 'Value''

```
       match (Value, Pat) ==> Match,
       case-choose (Match, Exp, Value, Rules) ==> Value'
[case2] -- ----------------------------------------
       case (Value, ((Pat => Exp) || Rules)) : Value
                                          ==> Value' .
```

Let's see how the auxiliary function 'case-choose(m,e,v,R)' works. If the match m is 'mno', it means that the matching against the first rule of R was unsuccessful, so the value must be matched against the remainder of the rules.

```
Value ::= case-choose (Match, Exp, Value, Rules) .

              case (Value, Rules) ==> Value'
[case-choose] -- ----------------------------------------
              case-choose (mno, Exp, Value, Rules) : Value
                                          ==> Value' .
```

Otherwise, if the matching is successful, 'case-choose' works in a similar way to the rule '[case1]'. The environment 'Env'' obtained from the successful match overrides the current environment to evaluate 'Exp' to a final value 'Value''.

```
              Env'' := Env' / Env,
              Exp ={env = Env'', -}=> Value'
[case-choose] -- ----------------------------------------
              case-choose (myes(Env'),Exp,Value,Rules) : Value
                                 ={env = Env, -}=> Value' .
```

The following rules specify each possible case of matching of values against patterns. First, matching a value against a pattern that is a variable is always successful and creates a binding from the variable to the value.

```
[match-var] match (Value, p Var) : Match
            ={env = Env, -}=> myes (Var |-> Value / Env) .
```

Matching a constant against a constant is successful only if both constants are equal.

```
                    Match := if Const1 == Const2
                             then myes (Env)
                             else mno fi
[match-const-const] -- ------------------------------
                    match (Const1, p Const2) : Match
                             ={env = Env, -}=> Match .
```

Matching a '`cons`' against a list makes needs an auxiliary function '`match-pair(m,ν,p)`' that will iterate through the list, gathering the bindings, if the matches are successful.

```
Match ::= match-pair (Match, Value, Pat)
```

Rule '`[match-cons-cons]`' states that, when matching a list against another we first attempt to match the elements at the beginning of both lists, and then use '`match-pair`' to, in a big-step manner, match the remainder of both lists.

```
                  match (Value1, Pat1) ==> Match1,
                  match-pair (Match1, Value2, Pat2) ==> Match2
[match-cons-cons] -- ----------------------------------------
                  match (cons (Value1, Value2),
                         Pat1 :: Pat2) : Match ==> Match2 .
```

Notice that, on rule '`[match-cons-cons]`', '`match-pair`' receives the matching of the first elements of both lists ('`Match1`'). If this matching is unsuccessful, then the entire list matching fails also.

```
[match-pair] match-pair (mno, Value, Pat) : Match ==> mno .
```

Otherwise it recursively matches against the remainder of the list, gathering the bindings during the process

```
              Env'' := Env' / Env,
              match (Value, Pat) ={env = Env'', -}=> Match
[match-pair] -- ----------------------------------------
              match-pair (myes (Env'), Value, Pat) : Match
                                  ={env = Env, -}=> Match .
```

Matching a constant against a list or a list against a constant always fails.

```
[match-const-cons] match (Const, Pat1 :: Pat2) : Match
                   ==> mno .
[match-cons-const] match (cons(Value1,Value2),p Const) : Match
                   ==> mno .
```

1.3 Example: sieve of Eratosthenes

Let us demonstrate the normal-order characteristics of Mini-Freja by creating a sieve of Eratosthenes using "lazy lists." The algorithm works as follows: we create an infinite list of numbers (function '`from`' below). This infinite list of numbers is filtered to keep only the prime numbers (functions '`filter`', '`sieve`', '`not-div`'). From this infinite list of primes, we take the first few (function '`take`' below). The implementation is as follows. We opted to split each function declaration into its own constant to simplify the exposition. We begin by creating all the constants that are used on the specification. We could have created these constants on an MSOS module but since they are used only for expository purposes and need equations anyway, we opted to declare them in a single Maude module.

```
ops fat n n0 xs0 x y xs pp N filter
    not-div sieve take from primes : -> Var .

op filterd : -> Dec .
op fromd : -> Dec .
op taked : -> Dec .
op not-divd : -> Dec .
op sieved : -> Dec .
op primesd : -> Dec .
op fatd : -> Dec .
```

Function '`filter`' receives as arguments (in curried form) a predicate and a list and returns only the elements from the list that satisfy the given predicate. Due to the *fixed point operator*, in order to declare a recursive function `f`, whose contents is an expression `e`, we write it as '`rec f is fn f => e`'.

```
eq filterd
= filter is rec (fn filter => fn pp => fn xs0 =>
   case xs0 of
     p nil       => nil
  || p x :: p xs => if (pp x) then
                      x :: ((filter pp) xs)
                    else
                      (filter pp) xs) .
```

Function '`from`' initiates an infinite list beginning at the value specified by its first argument

```
eq fromd
= from is rec (fn from =>
                (fn n => (n :: (from (n plus 1))))) .
```

Function '`take`' receives as arguments a number `n` and an infinite list `l` and returns the first `n` elements from `l`.

```
eq taked
= take is rec (fn take => fn n0 => fn xs0 =>
   case n0 of (p 0 => nil)
             || p n => (case xs0 of
                          p x :: p xs =>
                           (x :: ((take (n minus 1)) xs))
                        || p nil       => nil)) .
```

The function '`not-div`' is used as a predicate on function '`filter`'. It takes two arguments `x` and `y` and returns true if `y` divides `x`.

```
eq not-divd = not-div is (fn x => (fn y => ((y mod x) ne 0))) .
```

Function '`sieve`' implements the sieve by removing from the list it receives all numbers that are divisible by the rest of the numbers present on the list. The list must begin with two, for obvious reasons.

```
eq sieved
= sieve is rec (fn sieve => fn xs0 =>
                case xs0 of
```

```
                ((p x :: p xs) =>
                 (x :: (sieve ((filter (not-div x)) xs))))) .

 eq primesd = primes is sieve (from 2) .
```

Now, we may execute the program by concatenating all the declarations above and asking for the first 18 primes.

```
rewrite in PRIMES :
  < exec(let filterd taked fromd not-divd sieved primesd N is 18
         in ((take N) primes))::: 'Exp,init-rec >
result Conf :
  < done cons(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
              59,61,nil)::: 'Exp,{env = void} >
```

References

[1] Fabricio Chalub. An Implementation of Modular Structural Operational Semantics in Maude. Master's thesis, Universidade Federal Fluminense, 2005. `http://www.ic.uff.br/~frosario/dissertation.pdf`.

[2] Mikael Pettersson. *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer, 1999.

[3] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.