Compiler Construction in Maude

> Introduction to metaprogramming

International School on Rewriting

Universidad Complutense de Madrid

Christiano Braga

July 12th, 2021

Universidade Federal Fluminense, Brazil

Compilers and Maude

- A compiler is a *program transformation* tool.
- Maude is a *formal tool* for system development.
- One of its distinguishing features is support for *meta programming*.
- **A compiler can be implemented as a *meta program* in Maude.**

Objective

- To present a gentle introduction to compiler construction in Maude using *meta programming*.

Approach

To teach by *example*.

1. Maude modules
2. Maude meta-modules
3. Basic elements of a compiler
4. Basic elements of a compiler and Maude (meta) elements
5. A simple compiler in Maude
6. An extension with scope

Maude

Stack datatype in Maude

```
fmod STACK is
     inc RAT .
     sort Stack .
     subsort Rat < Stack .
     op empty : -> Stack [ctor] .
     op _::_ : Stack Stack -> Stack [ctor assoc id: empty] .
     op top : Stack ~> Rat .
     op pop : Stack -> Stack .
     op push : Rat Stack -> Stack .
     var R : Rat . var S : Stack .
     eq push(R, S) = S :: R .
     eq pop(empty) = empty .
     eq pop(S :: R) = S .
     eq top(S :: R) = R .
endfm
```

```
fmod STACK is
     --- ...
endfm
```

- Defines a namespace that should be interpreted by its *initial algebra* semantics.
- Example *terms* of the initial algebra are:

  `1, 1 :: 2, push(1, 1 :: 2), pop(push(1, 1 :: 2)).`

```
fmod STACK is
     inc RAT .
     --- ...
endfm
```

- We should be careful not to *mess up* with the algebra of the module being included, when including a module.
- In the example, rational numbers should remain rational numbers after being included by the `STACK` module.

```
fmod STACK is
     inc RAT .
     sort Stack .
     op empty : -> Stack [ctor] .
     op _::_ : Stack Stack -> Stack [ctor assoc id: empty] .
     --- ...
endfm
```

- Terms of the initial algebra of `STACK` have types, or *sorts*, in Maude terminology.
- For example,
  - operator `empty` is a constant constructor operator of sort `Stack`, and
  - operator `_::_` is an infix binary (constructor) operator that relates two stacks and produces another.

Stack datatype in Maude > Equations i

```
fmod STACK is
    --- ...
    var R : Rat . var S : Stack .
    eq push(R, S) = S :: R .
    eq pop(empty) = empty .
    eq pop(S :: R) = S .
    eq top(S :: R) = R .
endfm
```

- They relate, or *identify*, elements in the initial algebra of the given module.
- Terms `push(1, 1 :: 2)`, `pop(push(1, 1 :: 2))` are *identified* with `1 :: 2 :: 1` and `1 :: 2`, respectively, by means of the equations of module `STACK`.

- Equations are *declared* with *patterns* of terms (with variables) as formal parameters (understanding equation declaration as function declaration).
- A pattern can be:
  - a ground term, such as `empty`,
  - a variable, such as `R`,
  - or a complex pattern such as `S :: R`.
- Equations are *applied* to (ground) terms of the initial algebra, the actual parameters.

Running a (functional) program in Maude

```
reduce in STACK : pop(push(1, push(2, push(3, empty)))) .
rewrites: 4 in 0ms cpu (0ms real) (266666 rewrites/second)
result Stack: 3 :: 2
```

- Command *reduce* applies equations "to the bitter end".
- Equations are assumed to implement a function, in a mathematical sense:
  - always terminate and
  - always produce the same output for a given input.

Stack datatype at the *meta-level* i

```
fmod 'STACK is
  including 'BOOL .
  including 'RAT .
  sorts 'Stack .
  subsort 'Rat < 'Stack .
  op '_::_ : 'Stack 'Stack -> 'Stack [assoc ctor id('empty.Stack)] .
  op 'empty : nil -> 'Stack [ctor] .
  op 'pop : 'Stack -> 'Stack [none] .
  op 'push : 'Rat 'Stack -> 'Stack [none] .
  op 'top : '`[Stack`] -> '`[Stack`] [none] .
  none
  eq 'pop['empty.Stack] = 'empty.Stack [none] .
  eq 'pop['_::_['S:Stack,'R:Rat]] = 'S:Stack [none] .
  eq 'push['R:Rat,'S:Stack] = '_::_['S:Stack,'R:Rat] [none] .
  eq 'top['_::_['S:Stack,'R:Rat]] = 'R:Rat [none] .
endfm
```

Stack datatype at the meta-level ii

- At the meta-level, Maude elements such as modules, module inclusion, sorts, operators, variables, and equations become *terms*.
- For example, a meta (functional) module is a term of sort `FModule`, such as the module on the previous slide.
- A meta-function is a Maude function that manipulates terms that represent Maude elements at the meta-level.

Intermezzo: descent functions

- Maude provides various functions to program at the meta-level, declared in the module `META-LEVEL`.
- Some of them are:
  - `upModule`: Given the identifier of a given module, returns its meta-representation.
  - `upTerm`: Given a term produces its associated meta-term.
  - `downTerm`: Given a meta-term returns the term associated with the given meta-term.
  - `metaReduce`: Given a meta-module and a meta-term, returns the meta-term resulting from the application of the meta-equations, in the given meta-module, to the given meta-term.

Stack datatype at the meta-level iii

```
fmod META-STACK is
     inc STACK .
     inc META-LEVEL .
endfm
```

- The module on slide `Stack datatype at the meta-level i` is an example of a meta-module obtained by the execution of `reduce in META-STACK : upModule('STACK, false) .`
  - The second argument is a Boolean value denoting wheather the `BOOL` module, for the Boolean algebra, should be included or not.

Stack datatype at the meta-level iv

```
reduce in META-STACK : upTerm(3 :: 2) .
rewrites: 1 in 0ms cpu (0ms real) (58823 rewrites/second)
result GroundTerm: '_::_['s_^3['0.Zero],'s_^2['0.Zero]]
```

- Let us look at the meta-representation of a term:
  - Constant: 0 = `'0.Zero`
  - Operator: 3 = `s^3['0.Zero]`, where `s^3['0.Zero]` is the successor of 0 composed with itself 3 times.

Stack datatype at the meta-level v

```
reduce in META-STACK :
metaReduce(
    upModule('STACK, false),
    upTerm(pop(push(1, push(2, push( 3, empty)))))
) .
rewrites: 7 in 2ms cpu (2ms real) (2784 rewrites/second)
result ResultPair: {'_::_['s_^3['0.Zero],'s_^2['0.Zero]],'Stack}
```

- Meta-function `metaReduce` simplifies the given meta-term to its simplest (or *normal*) form.
- Descent functions can be applied while defining equations, not only at the command-line. This is the basis of our compiler!

Compiler construction

Elements of a compiler

1. **A (context-free) grammar**
2. **Its parser**
3. A type-checker
4. **An intermediate-language generator**
5. An intermediate-language optimizer
6. A machine-language generator

- This tutorial seminar illustrates how to: represent a grammar (1), build its parser (2), and code an intermediate-language generator (4).

Representing a grammar in Maude i

- Let us first consider the following CFG for arithmetic expressions in BNF:

  ```
  E ::= E + E | E * E | ( E ) | <num> | <id>
  ```

  where `E` is the initial symbol and `+`, `*`, `(`, `)`, `<num>`, and `<id>` are terminals. The lexemes `<num>` and `<id>` represent numbers and identifiers in the arithmetic language.

- This grammar has an operator precedence problem. How should we parse `2 + 3 * 4`?

- This is sometimes fixed by factorizing the rules into terms and factors or by declaring an *operator precedence* among the binary arithmetic operations.

- Another problem of it is operator *associativity*.

Representing a grammar in Maude ii

```
fmod SIMP-GRAMMAR is
     inc QID .
     inc RAT .
     sorts Exp Val .
     subsort Val < Exp .
     op noExp : -> Exp .
     op num_ : Rat -> Val [prec 10] .
     op _+_ : Exp Exp -> Exp [prec 20 gather(E e)].
     op _-_ : Exp Exp -> Exp [prec 20 gather(E e)].
     op _*_ : Exp Exp -> Exp [prec 15 gather(E e)].
     op _/_ : Exp Exp -> Exp [prec 15 gather(E e)].
     op (_) : Exp -> Exp .
endfm
```

Representing a grammar in Maude iii

- Precedence and associativity can be solved by declaring operator precedence and gathering operator attributes in Maude.
- Operators with lower precedence bind tighter and the pattern `E e` specifies letf-associativity.
- Operator `num` functions as a casting operation, by injecting Rational numbers into `Val` sort.

Representing a grammar in Maude iv

This module allows us to write Rational numbers' arithmetic expressions.

```
reduce in SIMP-GRAMMAR : num 1 / num 2 * num 4 .
rewrites: 0 in 0ms cpu (0ms real) (0 rewrites/second)
result Exp: num 1 / num 2 * num 4
```

Elements of a compiler and Maude i

Elements of a compiler	Maude
A (context-free) grammar	SIMP-GRAMAR
Its parser	
An intermediate-language generator	

Intermezzo: Why to program a compiler at the meta-level?

- At this point one may have noticed that it is possible to write a compiler at the object-level (as opposed to the meta-level).
- However, using the meta-level, one can create a *formal executable environment* for a language that allows for compilation, static analysis, code-generation, verification or any other procedure one may think of.

Metaparsing Simp programs

- As we have seen in the previous slide, we can already parse terms in `SIMP-GRAMMAR`.
- However, our compiler requires meta-terms to do its job.
- Not surprisingly, there is a descent function `metaParse` that does it and produces a meta-term from a list of quoted identifiers, or *qids*, such as `'a 'b 'c`.

Metaparsing Simp programs > Simp lexer i

```
fmod SIMP-LEXER is
   inc LEXICAL . inc META-LEVEL .
   inc STRING .  inc RAT .
   op lex : String ~> QidList .
   op $lexQid : Qid ~> QidList . op $lexQidList : QidList ~> QidList .
   op $isNum : Qid -> Bool .      op $error : -> Rat .
   var S : String . var Q : Qid . var QL : QidList .
   eq lex(S) = $lexQidList(tokenize(S)) .
   eq $lexQid('+) = '+ .      eq $lexQid('-) = '- .
   eq $lexQid('*) = '* .      eq $lexQid('/) = '/ .
   eq $lexQid('`( ) = '`( .  eq $lexQid( '`) ) = '`) .
   ceq $lexQid(Q) = 'num Q if $isNum(Q) .
   eq $lexQidList(nil) = nil .
   eq $lexQidList(Q QL) = $lexQid(Q) $lexQidList(QL) .
   eq $isNum(Q) = downTerm(getTerm(
      metaParse(upModule('RAT, false), Q, 'Rat)), $error) =/= $error .
endfm
```

```
fmod SIMP-LEXER is
   inc LEXICAL . inc META-LEVEL .

   op lex : String ~> QidList .

   op $isNum : Qid -> Bool .      op $error : -> Rat .
   var S : String . var Q : Qid .
   eq lex(S) = $lexQidList(tokenize(S)) .



   ceq $lexQid(Q) = 'num Q if $isNum(Q) .


   eq $isNum(Q) = downTerm(getTerm(
      metaParse(upModule('RAT, false), Q, 'Rat)), $error) =/= $error .
 endfm
```

```
reduce in SIMP-LEXER : lex("1 + 2") .
rewrites: 23 in 4ms cpu (4ms real) (5385 rewrites/second)
result NeTypeList: 'num '1 '+ 'num '2

reduce in SIMP-LEXER : lex("a + 2") .
rewrites: 21 in 0ms cpu (0ms real) (25179 rewrites/second)
result [<_QidList Kind_>]: $lexQid('a) '+ 'num '2
```

- As a good lexer, module `SIMP-LEXER` provides a function, `lex`, that given a string produces a list of tokens.
  - Function `lex` is a partial function: it is not the case that any given string represents a Rational number arithmetic expression.
- One way to enconde tokens is as qids.
- Function `tokenize`, from module LEXICAL, helps us to do precisely that: it breaks a string into a qid list.
- Function `lex` in `SIMP-LEXER` finds a Rational number, represented by a qid `Q`, in a list of qids, and replaces it by `'num Q`.

```
fmod SIMP-PARSER is
     inc SIMP-LEXER .
     op parse : String -> Term .
     var S : String .
     eq parse(S) =
        getTerm(metaParse(upModule('SIMP-GRAMMAR, false),
                          lex(S), 'Exp)) .
endfm
```

- Function `parse`, that receives a string and returns a term in `Exp`, is quite simple to write.
- It simply calls `metaParse` passing, as actual parameters:
  1. the meta-representation of `SIMP-GRAMAR` (given by `upModule`) and
  2. the qid list representing list of tokens of the program to be parsed, returned by function `lex`.

Elements of a compiler and Maude ii

Elements of a compiler	Maude
A (context-free) grammar	SIMP-GRAMAR
Its parser	SIMP-PARSER
An intermediate-language generator	

```
fmod STACK-MACHINE-GRAMMAR is
     inc QID . inc RAT .
     sort Prog Cmd Arith .
     subsort Arith < Cmd < Prog .
     op push : Rat -> Cmd .
     ops skip pop : -> Cmd .
     ops add sub mul div : -> Arith .
     op _;_ : Prog Prog -> Prog [assoc] .
endfm
```

- This is the grammar of the target languge of our compiler.
- Sorts `Cmd` and `Prog` represent what their names imply.
- Commands are `push`, `pop`, with the usual meaning, `skip`, that "does nothing", and arithemtic operators, such as `add` that pops twice, sums the popped Rational numbers, and pushes the result back onto the stack.
- A program is either a command, an arithmetic operator or a sequence of commands separated by `;`.

Compiling Simp > Transformer i

```
fmod TRANSFORM-SIMP-STACK is
     inc META-LEVEL .
     op tau : Term -> Term .
     var C : Constant .
     var G : GroundTerm .
     vars T1 T2 : Term .
     eq tau('num_[C]) = 'push[C] .
     eq tau('num_[G]) = 'push[G] .
     eq tau('_+_[T1, T2]) =
        '_;_[tau(T1), '_;_[tau(T2), 'add.Cmd]] .
     eq tau('_-_[T1, T2]) =
        '_;_[tau(T1), '_;_[tau(T2), 'sub.Cmd]] .
     eq tau('_*_[T1, T2]) =
        '_;_[tau(T1), '_;_[tau(T2), 'mul.Cmd]] .
     eq tau('_/_[T1, T2]) =
        '_;_[tau(T1), '_;_[tau(T2), 'div.Cmd]] .
endfm
```

- Metafunction `tau` is our compiler. **As simple as that!**
- Equations for `tau` define it to be a metafunction responsible for transforming, i.e. *compiling*, expressions programs in the `SIMP` language into programs in the `STACK-MACHINE` language.

Equation `tau('num_[G]) = 'push[G]` denotes that a number is simply pushed into the stack;

```
reduce in TRANSFORM-SIMP-STACK : tau('num_['s_^2['0.Zero]]) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result GroundTerm: 'push['s_^2['0.Zero]]
```

Equation

```
tau('_/_[T1, T2]) =
    '_;_[tau(T1), '_;_[tau(T2), 'div.Cmd]]
```

transforms a division into a division of the last two elements of the stack.

```
reduce in T :
reduce in TRANSFORM-SIMP-STACK :
tau('_/_['num_['s_['0.Zero]],'num_['s_^2['0.Zero]]]) .
rewrites: 3 in 0ms cpu (0ms real) (3000000 rewrites/second)
result GroundTerm:
'_;_['push['s_['0.Zero]],'_;_['push['s_^2['0.Zero]],'div.Cmd]]
```

Elements of a compiler and Maude iii

Elements of a compiler	Maude
A (context-free) grammar	SIMP-GRAMAR
Its parser	SIMP-PARSER
An intermediate-language generator	TRANSFORM-SIMP-STACK

Building an interpreter

```
fmod STACK-MACHINE is
    inc QID . inc RAT . inc STACK .
    inc STACK-MACHINE-GRAMMAR .
    op run : Prog -> Stack .
    op $run : Prog Stack -> Stack .
    vars R R1 R2 : Rat . var S : Stack .
    var C : Cmd . var A : Arith . var P : Prog .
    eq run(P) = $run(P, empty) .
    eq $run(push(R), S) = S :: R .
    eq $run(pop, S :: R) = S .
    eq $run(add, S :: R1 :: R2) = S :: (R1 + R2) .
    eq $run(sub, S :: R1 :: R2) = S :: (R1 - R2) .
    eq $run(mul, S :: R1 :: R2) = S :: (R1 * R2) .
    eq $run(div, S :: R1 :: R2) = S :: (R1 / R2) .
    eq $run(A, empty :: R) = R .
    eq $run(A, empty) = empty .
    eq $run(C ; P, S) = $run(P, $run(C, S)) .
endfm
```

```
fmod STACK-MACHINE is



    op run : Prog -> Stack .
    op $run : Prog Stack -> Stack .
    vars R R1 R2 : Rat . var S : Stack .
    var C : Cmd .                    var P : Prog .
    eq run(P) = $run(P, empty) .
    eq $run(push(R), S) = S :: R .

    eq $run(add, S :: R1 :: R2) = S :: (R1 + R2) .





    eq $run(C ; P, S) = $run(P, $run(C, S)) .
endfm
```

- The `STACK-MACHINE` module provides a function `run` that interprets a `STACK-MACHINE` program in the context of a stack, returning the stack resulting from the applications of the given program on an stack that is initially empty..
- For example, to run a `push` command it to write a rational number to the top of the given stack.
- To run an `add` command we pop twice and push their sum back onto the stack.
  - This is accomplished by *pattern matching*.
  - Pattern `S :: R1 :: R2` matches with a stack that has Rational numbers `R1` and `R2` on its "first two tops".
  - Pathern `S :: R1 + R2` denotes the stack `S` with `R1` + `R2` replacing `R1 :: R2`.

Putting it all together > The RUN-TIME module i

```
fmod RUN-TIME is
     inc SIMP-PARSER .
     inc TRANSFORM-SIMP-STACK .
     inc STACK-MACHINE .
     op P : -> Prog .
     var T : Term .
     op compile : Term -> Prog .
     eq compile(T) = downTerm(tau(T), P) .
endfm
```

- Module RUN-TIME is a pullback of the parser, the compiler and the interpreter.

```
reduce in RUN-TIME : parse("(1 / 2) + (1 / 4)") .
rewrites: 58 in 1ms cpu (1ms real) (31815 rewrites/second)
result GroundTerm:
'_+_['_/_['num_['s_['0.Zero]],'num_['s_^2['0.Zero]]],
     '_/_['num_['s_['0.Zero]],'num_['s_^4['0.Zero]]]]

reduce in RUN-TIME : compile(parse("(1 / 2) + (1 / 4)")) .
rewrites: 67 in 1ms cpu (2ms real) (36853 rewrites/second)
result Prog:
push(1) ; push(2) ; div ; push(1) ; push(4) ; div ; add

reduce in RUN-TIME : run(compile(parse("(1 / 2) + (1 / 4)"))) .
rewrites: 87 in 1ms cpu (1ms real) (51570 rewrites/second)
result PosRat: 3/4
```

Adding scope with `let` > Lexer i

```
fmod SIMP-GRAMMAR' is
     inc SIMP-GRAMMAR .
     sort Id .
     subsort Id < Exp .
     op id_ : Qid -> Id [prec 10] .
     op let_=_in_ : Id Val Exp -> Exp [prec 30] .
endfm
fmod SIMP-LEXER' is
     inc SIMP-LEXER .
     var Q : Qid .
     eq $lexQid('let) = 'let .
     eq $lexQid('=) = '= .
     eq $lexQid('in) = 'in .
     eq $lexQid(Q) = 'id qid("'" + string(Q)) [owise] .
endfm
```

Adding scope with `let` > Lexer ii

- This extension to Simp allows for the declaration of variables as `let` expressions.
- The lexer has to know how to handle identifiers in arithmetic expressions and `let` declarations, such as `let x = (1 / 2) in x + (3 / 4)`.

Adding scope with `let` > Parser i

```
fmod SIMP-PARSER' is
     inc SIMP-LEXER' .
     op parse : String -> Term .
     var S : String .
     eq parse(S) =
        getTerm(metaParse(upModule('SIMP-GRAMMAR', false),
                lex(S), 'Exp)) .
endfm
fmod STACK-MACHINE-GRAMMAR' is
     inc STACK-MACHINE-GRAMMAR .
     op push : Qid -> Cmd .
     op load : Qid Rat -> Cmd .
endfm
```

Adding scope with `let` > Parser ii

- We need to let the parser know that the new grammar should be used: `SIMP-GRAMAR'` instead of `SIMP-GRAMMAR`.
- The stack machine has to learn how to push identifiers and how to `load` them, that is, expand the execution context with a relation between an indentifier and a (Rational) value.

Adding scope with `let` > Constant environment i

```
fmod ENV is
     inc MAP{Qid, Rat} * (sort Map{Qid, Rat} to Env,
                          sort Entry{Qid, Rat} to Bind,
                          op empty to emptyEnv) .
endfm
```

Adding scope with `let` > Constant environment ii

- The environment is a set of bindings between identifiers (qids) and values (Rational numbers).
  - This will give semantics to `load`.
- The stack machine will have to be *redefined* in order to accomodate the environment.

Adding scope with `let` > Stack machine with environment i

```
fmod STACK-MACHINE' is
     inc QID . inc RAT . inc STACK . inc ENV .
     inc STACK-MACHINE-GRAMMAR' .
     op run : Prog -> Stack . op $run : Prog Stack Env -> Stack .
     vars R R1 R2 : Rat . var S : Stack .
     var E : Env . var Q : Qid . var C : Cmd . var A : Arith .
     var P : Prog .
     eq run(P) = $run(P, empty, emptyEnv) .
     eq $run(push(R), S, E) = S :: R .
     eq $run(push(Q), S , E) = S :: E[Q] .
     eq $run(pop, S :: R, E) = S .
     eq $run(add, S :: R1 :: R2, E) = S :: (R1 + R2) .
     eq $run(sub, S :: R1 :: R2, E) = S :: (R1 - R2) .
     eq $run(mul, S :: R1 :: R2, E) = S :: (R1 * R2) .
     eq $run(div, S :: R1 :: R2, E) = S :: (R1 / R2) .
     eq $run(A, empty :: R, E) = R . eq $run(A, empty, E) = empty .
     eq $run(load(Q, R) ; P, S, E) = $run(P, S, insert(Q, R, E)) .
     eq $run(C ; P, S, E) = $run(P, $run(C, S, E), E) [owise] .
endfm
```

Adding scope with `let` > Stack machine with environment i

```
fmod STACK-MACHINE' is
     inc QID . inc RAT . inc STACK . inc ENV .
     inc STACK-MACHINE-GRAMMAR' .
     op run : Prog -> Stack . op $run : Prog Stack Env -> Stack .
     var  R        : Rat . var S : Stack .
     var E : Env . var Q : Qid .
     var P : Prog .
     eq run(P) = $run(P, empty, emptyEnv) .

     eq $run(push(Q), S , E) = S :: E[Q] .





     eq $run(load(Q, R) ; P, S, E) = $run(P, S, insert(Q, R, E)) .
```



```
endfm
```

Adding scope with `let` > Stack machine with environment ii

- Function `$run` is redefined in order to receive an environment.
- The semantics for `push` has to be extended to accomodate pushing an identifier: The value bound to the given identifier must be pushed onto the stack.
- To `load` a value means to bind the given identifier to the given value and than execute the continuation of the given program with the current environment overwritten by the new binding.

Adding scope with `let` > Transforming identifiers and `let` expressions i

```
fmod TRANSFORM-SIMP-STACK' is
     inc TRANSFORM-SIMP-STACK .
     var C : Constant . var G : GroundTerm . var Q : Qid .
     vars T T1 T2 : Term . var TL : TermList .
     op tauToRat : Term -> Term .
     op tauToRat : TermList -> Term .
     eq tauToRat(C) = C .
     eq tauToRat('num_[G]) = G .
     eq tauToRat(Q[T, TL]) = Q[tauToRat(T), tauToRat(TL)] .
     eq tau('id_[C]) = 'push[C] .
     eq tau('let_=_in_['id_[C], T, T1]) =
        '_;_['load[C, tauToRat(T)], tau(T1)] .
endfm
```

Adding scope with `let` > Transforming identifiers and `let` expressions ii

- Before, numbers were transformed into `push` commands.
- They need to be transformed to Rational numbers in order to be bound to a qid in the environment.
- A `let` expression is transformed into a composition of `load` and the transformation of its inner expression.

Adding scope with `let` > A new run-time i

```
fmod RUN-TIME' is
     inc SIMP-PARSER' .
     inc TRANSFORM-SIMP-STACK' .
     inc STACK-MACHINE' .
     op P : -> Prog .
     var T : Term .
     op compile : Term -> Prog .
     eq compile(T) = downTerm(tau(T), P) .
endfm
```

Adding scope with `let` > A new run-time ii

Now we need to pullback the extrensions so we can run programs like:

```
red run(compile(parse(
 "let x = 1 / 2 in let y = 3 / 4 in x + y * (13 / 45)"))) .

rewrites: 219 in 9ms cpu (9ms real) (23583 rewrites/second)
result PosRat: 43/60
```

Conclusion

- We have seen the basics of metaprogramming in Maude using compiler construction as an example.
- This is the end of part 1. Let us take a 15' break and come back for a slice of Π!

Compiler Construction in Maude

> Introduction to metaprogramming

International School on Rewriting

Universidad Complutense de Madrid

Christiano Braga

July 12th, 2021

Universidade Federal Fluminense, Brazil

Compiler Construction in Maude

>> A slice of Π

International School on Rewriting

Universidad Complutense de Madrid

Christiano Braga

July 12th, 2021

Universidade Federal Fluminense, Brazil

Introduction

- We have learned the basics of metaprogramming in Maude using compiler construction as our case study.
- Let us take a look now at the Maude implementation the Π framework for compiler construction.

Π framework

- Two components: Π IR and Π automaton.
- Π IR is the intermediate representation language of the Π framework.
- Π automaton is the device that recognizes Π IR programs.
  - Its states are structured by *semantic components*, such as a value stack, or the environment.
  - Computations of a Π automaton are carried on in Maude by (unconditional) **record rewriting**.

Using Π framework in Maude

- Define the (abstract) syntax of your programming language as a signature (in a functional module).
- Define the semantic equations (metafunctions, in this approach) that relate the syntax of your language with Π IR by means of equations.

A slice of Π

Π IR i

- Organized by *facets*:
  - Expressions: side-effect free constructions.
    - Their effects are noticed by observing the value stack.
  - Commands: side-effect full constructions.
    - Their effects are noticed by observing the store.
  - Declarations: build new environments.
    - Their effects are noticed by observing the environment.
  - Abstractions: compute with (recursive) functions.
    - Their effects are noticed by observing the value stack.

Π IR ii

```
fmod BASIC-PI-IR is
     sorts OpCode Statement .
endfm
```

Π automaton i

- Is a nondeterministic finite automaton whose states are structured as *variant* (or extensible) records.
- Each *field* of the record holds a semantic component.

Π automaton ii

```
fmod RECORD{X :: TRIV, Y :: TRIV} is
     sorts Field{X, Y} PreRecord{X, Y} Record{X, Y} .
     subsort Field{X, Y} < PreRecord{X, Y} .
     op __ : X$Elt Y$Elt ~> Field{X, Y} [prec 20] .
     op empty : -> PreRecord{X, Y} .
     op _,_ : PreRecord{X, Y} PreRecord{X, Y} -> PreRecord{X, Y}
                [prec 40 assoc comm id: empty format(d d nss d)] .
     op [_] : PreRecord{X, Y} -> Record{X, Y} [format (n s s d)] .
endfm
```

Π automaton iii

```
fmod PI-AUTOMATON is
     inc BASIC-PI-IR .
     inc RECORD{Idx, SemComp} *
         (sort Record{Idx, SemComp} to State,
          sort PreRecord{Idx, SemComp} to PreState) .
     sort FinalState .
     subsort FinalState < State .
     op run : State -> State [iter] .
     op run* : State -> State .
     op exec : Statement -> State .
     var S : State . var F : FinalState .
     eq run*(F) = F .
     eq run*(S) = run*(run(S)) .
endfm
```

Expressions > Syntax

```
fmod EXP-IR is
     inc BASIC-PI-IR . inc VALUE .
     sort Exp .
     subsort Exp < Statement .
     op exp : Value -> Exp .
     ops sum sub mul div lth leq
         gth geq equ lor lan : Exp Exp -> Exp [format (y o)] .
     op not : Exp -> Exp [format (y o)] .
     op ite : Exp Exp Exp -> Exp [format (y o)] .
endfm
```

```
fmod CONTROL-STACK-COMPONENT is
     inc BASIC-PI-IR . inc CONTROL . inc INDEX . inc SEM-COMP .
     inc STACK{Control} * (op [] to mtCnt) .
     subsort OpCode < Control .
     op cnt : -> Idx [format (b! o)] .
     op :_ : Stack{Control} -> SemComp .
endfm

fmod VALUE-STACK-COMPONENT is
     inc VALUE . inc INDEX .
     inc SEM-COMP .
     inc STACK{Value} * (op [] to mtVal) .
     op val : -> Idx [format (b! o)] .
     op :_ : Stack{Value} -> SemComp .
endfm
```

```
fmod EXP-AUTOMATON is
    --- <Maude code>

    --- Control stack component is a record field.
    mb (cnt : C) : Field{Idx, SemComp} .

    --- Value stack component is a record field.
    mb (val : V) : Field{Idx, SemComp} .

    --- <Maude code>
endfm
```

(Excerpt)

```
fmod EXP-AUTOMATON is
     --- <Maude code>

     eq run( [ cnt : [ L :: cnt(exp(U)) ],
               val : V ,
               ... ] ) =
             [ cnt : [ L ],
               val : push(V, U),
               ... ] .

     --- <Maude code>
endfm
```

(Excerpt)

Expressions > Automaton > Transitions > `ite` i

- ITE = if then else

```
fmod EXP-AUTOMATON is
    --- <Maude code>

    eq run( [ cnt : [ L :: cnt(ite(E1, E2, E3)) ],
              val : V,
              ... ] ) =
            [ cnt : push(push([ L ], #ITE), cnt(E1)),
              val : push(push(V, val(E3)), val(E2)),
              ... ]  .

    --- <Maude code>
endfm
```

(Excerpt)

Expressions > Automaton > Transitions > `ite` ii

```
fmod EXP-AUTOMATON is
    --- <Maude code>

    eq run( [ cnt : [ L :: #ITE ],
              val : [ M :: val(E3) :: val(E2) :: val(true) ],
              ... ] ) =
            [ cnt : [ L :: cnt(E2)],
              val : [ M ], ... ] .

    eq run( [ cnt : [ L :: #ITE ],
              val : [ M :: val(E3) :: val(E2) :: val(false) ],
              ... ] ) =
            [ cnt : [ L :: cnt(E3)],
              val : [ M ],  ... ] .

    --- <Maude code>
endfm
```


(Excerpt)

Expressions with identifiers > Syntax

```
fmod ID-IR is
     inc QID .
     inc EXP-IR .
     sort Id .
     op id : Qid -> Id [format (! o)] .
     op exp : Id -> Exp .
endfm
```

Expressions with identifiers > Automaton > Semantic component i

```
fmod ENV is
     inc MAP{Id, Bindable} * (sort Map{Id, Bindable} to Env,
                              op empty to mtEnv) .

     op update : Env Env -> Env .

     --- <Maude code>
endfm
```

(Excerpt)

- Function `update` rewrites rho1 with entries from rho2 by overwriting values of common indices with values from rho2, adding entries from rho2 which are not common to both, preserving entries from rho1 which are not common to both.

Expressions with identifiers > Automaton > Semantic component ii

```
fmod ENV-COMPONENT is
     inc INDEX . inc SEM-COMP .
     inc ENV .
     op env : -> Idx [format (b! o)] .
     op :_ : Env -> SemComp .
endfm
```

Expressions with identifiers > Automaton > Semantic component iii

```
fmod EXP-WITH-ID-AUTOMATON is
    --- <Maude code>

    --- Env component is a record field.
    mb (env : rho) : Field{Idx, SemComp} .

    --- <Maude code>

endfm
```

(Excerpt)

Expressions with identifiers > Automaton > Transition

```
fmod EXP-WITH-ID-AUTOMATON is
    --- <Maude code>

    eq run( [ cnt : [ L :: cnt(exp(I)) ],
              val : V ,
              env : rho,
              ... ] ) =
            [ cnt : [ L ],
              val : push(V, getVal(rho[I])),
              env : rho,
              ... ] .

    --- <Maude code>
endfm
```

(Excerpt)

Expressions with identifiers > ite > Example i

- Execute `maude pi-examples.maude`.

```
reduce in EXP-WITH-ID-AUTOMATON : run*(
[ cnt : [cnt(ite(exp(val(true)),
                 sum(exp(id('x)), exp(val(1))),
                 sum(exp(id('x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]) .
rewrites: 47 in 0ms cpu (0ms real) (1021739 rewrites/second)
result FinalState:
[ cnt : mtCnt,
  val : [val(6)],
  env : (id('x) |-> bnd(val(5))) ]
```

Expressions with identifiers > ite > Example ii

```
reduce in EXP-WITH-ID-AUTOMATON : run*(
[ cnt : [cnt(ite(exp(val(false)),
                 sum(exp(id('x)), exp(val(1))),
                 sum(exp(id('x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]) .
rewrites: 47 in 0ms cpu (0ms real) (47000000 rewrites/second)
result FinalState:
[ cnt : mtCnt,
  val : [val(7)],
  env : (id('x) |-> bnd(val(5))) ]
```

Declarations > Syntax

```
fmod DEC-IR is
     inc ID-IR . inc CMD-IR .
     sort Dec .
     op dec : Id Exp -> Dec [format (m o)] .
     op let : Dec Exp -> Exp [format (y o)] .

     --- <Maude code>
endfm
```

(Excerpt)

Declarations > Automaton > Transitions i

```
fmod DEC-AUTOMATON is
    --- <Maude code>

    --- Declarations
    eq run( [ cnt : [ L :: cnt(dec(I, E)) ],
              val : V, ... ] ) =
            [ cnt : push(push([ L ], #DEC), cnt(E)),
              val : push(V, val(I)), ... ] .

    eq run( [ cnt : [ L :: #DEC ],
              val : [ M :: val(I) :: U ], ... ] ) =
            [ cnt : [ L ],
              val : push([ M ], val(I |-> bnd(U))), ... ] .

    --- <Maude code>
endfm
```

(Excerpt)

Declarations > Automaton > Transitions iii

```
fmod DEC-AUTOMATON is
     --- <Maude code>

     --- Let
     eq run( [ cnt : [ L :: cnt(let(D, E)) ],
               val : V,
               ... ] ) =
             [ cnt : [ L :: #LETDEC :: cnt(D) ],
               val : push(V, val(E)),
               ... ] .

     --- <Maude code>
endfm
```

(Excerpt)

Declarations > Automaton > Transitions iv

```
fmod DEC-AUTOMATON is
     --- <Maude code>

     eq run( [ cnt : [ L :: #LETDEC ],
               val : [ M :: val(E) :: val(rho2) ],
               env : rho1,
               ... ] ) =
             [ cnt : [ L :: #LETEXP :: cnt(E) ],
               val : push([ M ], val(rho1)),
               env : update(rho1, rho2), ... ] .

     --- <Maude code>
endfm
```

(Excerpt)

Declarations > Automaton > Transitions v

```
fmod DEC-AUTOMATON is
     --- <Maude code>

     eq run( [ cnt : [ L :: #LETEXP ],
               val : [ M :: val(rho2) :: U ],
               env : rho1,
               ... ] ) =
             [ cnt : [ L ],
               val : [ M :: U ],
               env : rho2, ... ] .

     --- <Maude code>
endfm
```

(Excerpt)

Declarations > `let` > Example

```
reduce in DEC-AUTOMATON : run*(
[ cnt : [cnt(let(dec(id('x), exp(val(5))),
                 ite(exp(val(false)),
                     sum(exp(id('x)), exp(val(1))),
                     sum(exp(id('x)), exp(val(2))))))],
  val : mtVal,
  env : mtEnv ]) .
rewrites: 80 in 0ms cpu (0ms real) (509554 rewrites/second)
result FinalState:
[ cnt : mtCnt,
  val : [val(7)],
  env : mtEnv ]
```

```
fmod REC-IR is
     inc LAMBDA-IR . inc ACTUALS .
     op rec : Id Abs -> Dec [format (m o)] .
     op recall : Id Actuals -> Exp [format (y o)] .
endfm
```

Recursive abstractions > Automaton > Transitions i

```
fmod REC-ABS-AUTOMATON is
     inc ABS-AUTOMATON . inc REC-IR . sort Reclos .

     op reclosure : Formals Exp Env Env -> Reclos .
     op unfold : Env -> Env . op #RECALL : -> OpCode .

     eq run( [ cnt : [ L :: cnt(rec(I, lambda(F, E))) ],
               val : V,
               env : rho,  ... ] ) =
             [ cnt : [ L ],
               val : push(V,
                      val(unfold(I |-> bnd(val(closure(rho, F, E)))))),
               env : rho,  ... ] .
     --- <Maude code>
endfm
```

(Excerpt)

- Function `unfold` injects a recursive environment in the current one, making the declaration of a recursive function available to itself.

Recursive abstractions > Automaton > Transitions ii

```
fmod REC-ABS-AUTOMATON is
    --- <Maude code>

    eq run( [ cnt : [ L :: cnt(recall(I, A))],
              val : V, env : rho,
              ... ] ) =
            [ cnt : [ L :: #RECALL :: $mkControl(A) ],
              val : push(V, getVal(rho[I])),
              env : rho,
              ... ] .

    --- <Maude code>
endfm
```

(Excerpt)

Recursive abstractions > Automaton > Transitions iii

```
fmod REC-ABS-AUTOMATON is
    --- <Maude code>

    eq run( [ cnt : [ L :: #RECALL ],
              val : [ M1 :: val(reclosure(F, E, rho1, rho2)) :: M2 ],
              env : rho3,
              ... ] ) =
            [ cnt : [ L :: #ENDAPPLY :: cnt(E) ],
              val : [ M1 :: val(rho3) ],
              env : update(rho1, update(unfold(rho2), match(F, M2))),
              ... ] .

    --- <Maude code>
endfm
```

(Excerpt)

Recursive abstractions > Example

```
reduce in REC-ABS-AUTOMATON : run*(
[ cnt : [cnt(let(rec(id('fat), lambda(id('x),
                      ite(gth(exp(id('x)), exp(val(1))),
                          mul(exp(id('x)),
                              recall(id('fat),
                                     sub(exp(id('x)), exp(val(1))))),
                          exp(val(1))))),
                  recall(id('fat), exp(val(10)))))],
  val : mtVal,
  env : mtEnv ]) .
rewrites: 11702 in 12ms cpu (12ms real) (958473 rewrites/second)
result FinalState:
[ cnt : mtCnt,
  val : [val(3628800)],
  env : mtEnv ]
```

A com□ler for Fun

Building a com∏ler

- Take the same steps we took in the introductory part and change the intermediate representation from the stack language to Π IR.
- And that's it!

Syntax for Fun

```
fmod FUN-GRM is
     inc QID .
     inc RAT .
     inc BOOL .
     sorts Expr Idn Val .
     subsort Idn Val < Expr .
     op _,_ : Expr Expr -> Expr [prec 60 assoc] .
     op idn : Qid -> Idn .
     op rat : Rat -> Val .
     op boo : Bool -> Val .
     ops _+_ _*_ _-_ _/_
         _==_ _<_ _>_ _or_ _and_ : Expr Expr -> Expr
         [gather(E e) prec 20] .
     op !_ : Expr -> Expr [prec 5] .
     op if_then_else_ : Expr Expr Expr -> Expr [prec 40] .
     op fun_(_) =_ : Idn Idn Expr -> Expr [prec 50] .
     op _(_) : Idn Expr -> Expr [prec 10] .
endfm
```

Transforming Fun programs into Π IR i

```
fmod FUN-TRAN is
     inc META-LEVEL * (op id to ml-id) .

     op compile : Term ~> Term . op compileOp : Qid -> Qid .
     op compileActual : Term -> Term .
     op compileActuals : TermList -> TermList .

     eq compile('idn[C]) = 'id[C] . eq compile('rat[T]) = 'val[T] .
     eq compileOp('_+_) = 'sum . eq compileOp('_-_) = 'sub .

     eq compile('if_then_else_[T1,T2,T3]) =
        'ite[compileActual(T1), compileActual(T2), compileActual(T3)] .

     --- <Maude code>
endfm
```

(Excerpt)

Transforming Fun programs into Π IR ii

```
fmod FUN-TRAN is
    --- <Maude code>

    eq compile('fun_`(_`)=_[T1, T2, T3]) =
      'rec[compile(T1), 'lambda[compile(T2), compileActual(T3)]] .
    eq compile('_`(_`)[T1, T2]) =
       'recall[compile(T1), compileActual(T2)] .

    eq compile('_,_[T1, T2]) = 'let[compile(T1), compile(T2)] .

    eq compile(Op[TL]) = compileOp(Op)[compileActuals(TL)] [owise] .

    --- <Maude code>
endfm
```

(Excerpt)

An example compilation

```
reduce in FUN-RUN-TIME : downTerm(compile(upTerm(
fun idn('fat)(idn('x))=
    if idn('x) > rat(1)
    then idn('x) * idn('fat)(idn('x) - rat(1))
    else rat(1),idn('fat)(rat(100)))), E:[Actuals,Statement]) .
rewrites: 45 in 0ms cpu (0ms real) (803571 rewrites/second)
result Exp:
let(rec(id('fat), lambda(id('x),
     ite(gth(exp(id('x)), exp(val(1))),
      mul(exp(id('x)), recall(id('fat), sub(exp(id('x)), exp(val(1)))))
      exp(val(1))))),
    recall(id('fat), exp(val(100))))
```

An executable environment for Fun

```
fmod FUN-RUN-TIME is
     inc FUN-GRM .
     inc PI-IR .
     inc FUN-TRAN .
     inc REC-ABS-AUTOMATON .
     op exec : Expr -> State .
     op out : State -> Rat .
     op getRat : Value -> Rat .
     op error : ~> Exp .
     var E : Expr . var V : Stack{Value} . var ... : PreState .
     var R : Rat .
     eq getRat(val(R)) = R .
     eq exec(E) =
        run*( [cnt : [cnt(downTerm(compile(upTerm(E)), error))],
               val : mtVal, env : mtEnv ]) .
     eq out([val : V, ...]) = getRat(top(V)) .
endfm
```

An executable environment for Fun > Example

```
reduce in FUN-RUN-TIME : out(exec(
fun idn('fat)(idn('x))=
    if idn('x) > rat(1)
    then idn('x) * idn('fat)(idn('x) - rat(1))
    else rat(1),idn('fat)(rat(10)))) .
rewrites: 11751 in 12ms cpu (13ms real) (915400 rewrites/second)
result NzNat: 3628800
```

Coda

Food for thought > Some advanced topics i

- Program verification
  - Model checking: temporal logic can be used to check for reachability properties in concurrent programs using the automaton induced by the program as model to the temporal logic property.
  - Variant unification and Narrowing: Rewrite asks the question "which term is produced by rewriting term `t` in module `M`"? Narrowing asks a "backwards" question: "which term(s) produces a term `t` in module `M`"?
  - Rewriting modulo SMT: we may *condition* the term rewrite proces to the *statisfability* of formulae in standard SMT tools.

Food for thought > Some advanced topics iii

- Advanced metaprogramming: Static analysis
  - We may use metaprogramming to perform code analysis and optimization.
- Code generation to other intermediate representation languages such as LLVM.

Conclusion

- Maude is a powerful tool for compiler construction due to metaprogramming.
- When metaprogramming is combined with (symbolic) verification such as variant unification, narrowing or model checking, it gives rise to an effective formal tool for program generation.

Maude and I

Maude	I
maude.cs.uiuc.edu	www.ic.uff.br/~cbraga cbraga@ic.uff.br

Acknowledgement

- The techniques employed in the development of the Π framework were influenced by a long term collaboration with **Fabricio Chalub**, **Edward Hermann Haeusler**, **José Meseguer** and **Peter D. Mosses**, to whom I am forever thankful for their friendship and commitment.

Thank you!

Compiler Construction in Maude

>> A slice of Π

International School on Rewriting

Universidad Complutense de Madrid

Christiano Braga

July 12th, 2021

Universidade Federal Fluminense, Brazil

Appendix

Expressions > Trace > Example i

- Execute `maude trace-config.maude pi-examples.maude`.

```
reduce in EXP-WITH-ID-AUTOMATON : run*(
[ cnt : [cnt(ite(exp(val(true)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]) .
(unlabeled equation)
run*(
[ cnt : [cnt(ite(exp(val(true)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [cnt(ite(exp(val(true)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#ITE :: cnt(exp(val(true)))],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
    1))))],
```

Expressions > Trace > Example ii

```
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#ITE :: cnt(exp(val(true)))],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
    1))))],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#ITE],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
    1)))) :: val(true)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#ITE],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
    1)))) :: val(true)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [cnt(sum(exp(id('x)), exp(val(1))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ])
```

Expressions > Trace > Example iii

```
--->
run*(run(
[ cnt : [cnt(sum(exp(id('x)), exp(val(1))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#SUM :: cnt(exp(val(1))) :: cnt(exp(id('x)))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#SUM :: cnt(exp(val(1))) :: cnt(exp(id('x)))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#SUM :: cnt(exp(val(1)))],
  val : [val(5)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#SUM :: cnt(exp(val(1)))],
  val : [val(5)],
```

Expressions > Trace > Example iv

```
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#SUM],
  val : [val(5) :: val(1)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#SUM],
  val : [val(5) :: val(1)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : mtCnt,
  val : [val(6)],
  env : (id('x) |-> bnd(val(5))) ])
--->

[ cnt : mtCnt,
  val : [val(6)],
  env : (id('x) |-> bnd(val(5))) ]
rewrites: 47 in 0ms cpu (0ms real) (172794 rewrites/second)
result FinalState:
[ cnt : mtCnt,
```

Expressions > Trace > Example v

```
  val : [val(6)],
  env : (id('x) |-> bnd(val(5))) ]
==========================================
reduce in EXP-WITH-ID-AUTOMATON : run*(
[ cnt : [cnt(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]) .
(unlabeled equation)
run*(
[ cnt : [cnt(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [cnt(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#ITE :: cnt(exp(val(false)))],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
```

Expressions > Trace > Example vi

```
    1))))],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#ITE :: cnt(exp(val(false)))],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
    1))))],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#ITE],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
    1)))) :: val(false)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#ITE],
  val : [val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id('x)), exp(val(
    1)))) :: val(false)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [cnt(sum(exp(id('x)), exp(val(2))))],
  val : mtVal,
```

Expressions > Trace > Example vii

```
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [cnt(sum(exp(id('x)), exp(val(2))))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#SUM :: cnt(exp(val(2))) :: cnt(exp(id('x)))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#SUM :: cnt(exp(val(2))) :: cnt(exp(id('x)))],
  val : mtVal,
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#SUM :: cnt(exp(val(2)))],
  val : [val(5)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#SUM :: cnt(exp(val(2)))],
```

Expressions > Trace > Example viii

```
  val : [val(5)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#SUM],
  val : [val(5) :: val(2)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#SUM],
  val : [val(5) :: val(2)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : mtCnt,
  val : [val(7)],
  env : (id('x) |-> bnd(val(5))) ])
--->

[ cnt : mtCnt,
  val : [val(7)],
  env : (id('x) |-> bnd(val(5))) ]
rewrites: 47 in 0ms cpu (0ms real) (353383 rewrites/second)
result FinalState:
```

Expressions > Trace > Example ix

```
[ cnt : mtCnt,
  val : [val(7)],
  env : (id('x) |-> bnd(val(5))) ]
==========================================
reduce in DEC-AUTOMATON : run*(
[ cnt : [cnt(let(dec(id('x), exp(val(5))), ite(exp(val(false)), sum(exp(id(
    'x)), exp(val(1))), sum(exp(id('x)), exp(val(2))))))],
  val : mtVal,
  env : mtEnv ]) .
(unlabeled equation)
run*(
[ cnt : [cnt(let(dec(id('x), exp(val(5))), ite(exp(val(false)), sum(exp(id(
    'x)), exp(val(1))), sum(exp(id('x)), exp(val(2))))))],
  val : mtVal,
  env : mtEnv ])
--->
run*(run(
[ cnt : [cnt(let(dec(id('x), exp(val(5))), ite(exp(val(false)), sum(exp(id(
    'x)), exp(val(1))), sum(exp(id('x)), exp(val(2))))))],
  val : mtVal,
  env : mtEnv ]))
(unlabeled equation)
run*(
[ cnt : [#LETDEC :: cnt(dec(id('x), exp(val(5))))],
```

Expressions > Trace > Example x

```
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  env : mtEnv ])
--->
run*(run(
[ cnt : [#LETDEC :: cnt(dec(id('x), exp(val(5))))],
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2)))))],
  env : mtEnv ]))
(unlabeled equation)
run*(
[ cnt : [#LETDEC :: #DEC :: cnt(exp(val(5)))],
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2))))) :: val(id('x))],
  env : mtEnv ])
--->
run*(run(
[ cnt : [#LETDEC :: #DEC :: cnt(exp(val(5)))],
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2))))) :: val(id('x))],
  env : mtEnv ]))
(unlabeled equation)
run*(
[ cnt : [#LETDEC :: #DEC],
```

Expressions > Trace > Example xi

```
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2))))) :: val(id('x)) :: val(5)],
  env : mtEnv ])
--->
run*(run(
[ cnt : [#LETDEC :: #DEC],
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2))))) :: val(id('x)) :: val(5)],
  env : mtEnv ]))
(unlabeled equation)
run*(
[ cnt : [#LETDEC],
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2))))) :: val(id('x) |-> bnd(val(5)))],
  env : mtEnv ])
--->
run*(run(
[ cnt : [#LETDEC],
  val : [val(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))), sum(exp(id(
    'x)), exp(val(2))))) :: val(id('x) |-> bnd(val(5)))],
  env : mtEnv ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP :: cnt(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))),
```

Expressions > Trace > Example xii

```
    sum(exp(id('x)), exp(val(2)))))],
  val : [val(mtEnv)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#LETEXP :: cnt(ite(exp(val(false)), sum(exp(id('x)), exp(val(1))),
    sum(exp(id('x)), exp(val(2)))))],
  val : [val(mtEnv)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP :: #ITE :: cnt(exp(val(false)))],
  val : [val(mtEnv) :: val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id(
    'x)), exp(val(1))))],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#LETEXP :: #ITE :: cnt(exp(val(false)))],
  val : [val(mtEnv) :: val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id(
    'x)), exp(val(1))))],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP :: #ITE],
```

Expressions > Trace > Example xiii

```
  val : [val(mtEnv) :: val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id(
    'x)), exp(val(1)))) :: val(false)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#LETEXP :: #ITE],
  val : [val(mtEnv) :: val(sum(exp(id('x)), exp(val(2)))) :: val(sum(exp(id(
    'x)), exp(val(1)))) :: val(false)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP :: cnt(sum(exp(id('x)), exp(val(2))))],
  val : [val(mtEnv)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#LETEXP :: cnt(sum(exp(id('x)), exp(val(2))))],
  val : [val(mtEnv)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP :: #SUM :: cnt(exp(val(2))) :: cnt(exp(id('x)))],
  val : [val(mtEnv)],
  env : (id('x) |-> bnd(val(5))) ])
```

Expressions > Trace > Example xiv

```
--->
run*(run(
[ cnt : [#LETEXP :: #SUM :: cnt(exp(val(2))) :: cnt(exp(id('x)))],
  val : [val(mtEnv)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP :: #SUM :: cnt(exp(val(2)))],
  val : [val(mtEnv) :: val(5)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#LETEXP :: #SUM :: cnt(exp(val(2)))],
  val : [val(mtEnv) :: val(5)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP :: #SUM],
  val : [val(mtEnv) :: val(5) :: val(2)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#LETEXP :: #SUM],
  val : [val(mtEnv) :: val(5) :: val(2)],
```

Expressions > Trace > Example xv

```
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : [#LETEXP],
  val : [val(mtEnv) :: val(7)],
  env : (id('x) |-> bnd(val(5))) ])
--->
run*(run(
[ cnt : [#LETEXP],
  val : [val(mtEnv) :: val(7)],
  env : (id('x) |-> bnd(val(5))) ]))
(unlabeled equation)
run*(
[ cnt : mtCnt,
  val : [val(7)],
  env : mtEnv ])
--->

[ cnt : mtCnt,
  val : [val(7)],
  env : mtEnv ]
rewrites: 80 in 0ms cpu (0ms real) (153846 rewrites/second)
result FinalState:
[ cnt : mtCnt,
```

```
  val : [val(7)],
  env : mtEnv ]
Bye.
```