MMT case study: Constructive MSOS

Fabricio Chalub Christiano Braga

July 11, 2005

Contents

1	Intr	oducti	ion	2
	1.1	The C	MSOS constructions	2
		1.1.1	Expressions	3
		1.1.2	Declarations	
		1.1.3	Abstractions	
		1.1.4	Commands	
		1.1.5	Concurrency	
	1.2	ML .	· · · · · · · · · · · · · · · · · · ·	
		1.2.1	Expressions	
		1.2.2	Declarations	
		1.2.3	Imperatives	
		1.2.4	Abstractions	
		1.2.5	Concurrency	
		1.2.6	Example	
	1.3	MiniJa	ava	
		1.3.1	Expressions	
		1.3.2	Statements	
		1.3.3	Classes	
		1.3.4	Example	
Bibliography				

1 Introduction

This technical report describes the formal specification of CMSOS [9] using the Maude MSOS Tool [3].

Mosses's Constructive MSOS (CMSOS) is a technique for the specification of programming languages semantics based on the idea that each language construction should be specified in a separated module and the complete language specification is based on the combination of all these modules. A related approach, named Constructive Action Semantics, which uses the same ideas and is developed under the Action Semantics framework, is described in [5, 6, 7].

A further generalization that Constructive MSOS makes is to use *neutral* constructions with regard to a particular programming language. This allows a high degree of reuse of those modules on a wide range of programming languages. Mosses's lecture notes [9] contain a proposed set of these abstract constructions, which we have implemented and call it also as CMSOS. We also exemplify the use of these abstract constructions to give the formal semantics of two different programming languages: a subset ML described in [9] (Section 1.2) and a subset of Java, called MiniJava, based on the language described in [1] (Section 1.3).

1.1 The CMSOS constructions

This Section describes Mosses's proposed set of CMSOS constructions. Each subsection describes a particular aspect of the CMSOS language. We follow [9] on the order of presentation, which divides the semantics of CMSOS into five aspects: expressions, declarations, abstractions, commands, and concurrency. We opted not to present all of CMSOS here, selecting instead those constructions that are needed on the semantics of ML and MiniJava.

A word on terminology of CMSOS modules: Mosses's MSOS Tool library separates each module into a file in its own directory. For example, the directory 'Cons/' contains all the constructions of CMSOS. Inside 'Cons/', the directory 'Exp/' contains all the constructions related to expressions. Inside 'Exp/', one of the several directories is 'tup/', which contains the module that defines the syntax of general tuples. MSDF modules must have a name, so we opted to use the directory hierarchy as the name of the module. Also, Mosses divides each construction into three modules: one for the abstract syntax, one for the static semantics, and another for the dynamic seman-

tics. To make things simple, we opted to combine both abstract syntax and dynamic semantics in the modules implemented with MMT.

1.1.1 Expressions

Expressions are the basic building blocks of CMSOS programs and contain the following constructions: tuples, conditionals, application of operators, and applications of identifiers. We begin by showing the abstract semantics of tuples of expressions first with its most general form of tuples and then a more restricted one.

We begin by defining the general modules that define the set of expressions, 'Exp', and values, 'Value'. The module 'Value' defines the set of values along with a function 'apply-op', which receives an operation 'Op', and a sequence of values, 'Value*'. The set 'Op', defined on the module 'Cons/Op', contains all primitive operations on values. To simplify things, we avoid to define rules for extremely obvious operations, such as arithmetic operations, and comparison operations: these are defined outside the specification, as we shall see later. The important thing is that the specifier does not have to worry about defining things like the addition of two integer values, and so on.

```
msos Cons/Op is
   Op .
sosm

msos Value is
   Value .
   Value ::= apply-op (Op, Value*) .
sosm
```

Next, we define the set of expressions, 'Exp', in the module 'Cons/Exp'. Expressions evaluate into values, hence the subset inclusion.

```
msos Cons/Exp is
Exp .
Exp ::= Value .
sosm
```

The 'Cons/Exp/tup' below defines a *general* form of tuples in which each element of the tuple is selected nondeterministically to be evaluated.

A sequential variant has the additional constraint in which elements must be evaluated left to right.

The following are the modules for the application of operators and application of identifiers. Rule 'app-op1' evaluates the argument of the application—additional rules are needed for a particular case. As we mentioned earlier, rule 'app-op2' is a convenience to the user and is used in the particular case that the original argument is a tuple of values, and there is an (externally defined) equation that gives the result of the application of operation 'Op' to the values in 'Value*'.

The 'app-Id' module defines the application of an identifier to an argument. The actual rules that will govern each particular case of identifiers and arguments will be defined in subsequent modules. It is usually the case where the identifier will evaluate, by looking up on the environment, to an operator, and then the associated operation will be applied.

1.1.2 Declarations

Declarations constructions declare bindings and evaluate expressions within the context of a set of bindings. A binding associate an identifier to a "bindable" value—a value that may be part of a binding. The set 'Bindable' is the set of bindable values.

```
msos Data/Bindable is
Bindable .
sosm
```

Now the set of identifiers 'Id'. In order to solve a forwarding problem, we first define identifiers in a module 'Id', as follows:

```
msos Id is
  Id .
  Id ::= Bindable .
sosm
```

We may now define *environments* of bindings, which are used to associated identifiers with bindable values. An environment 'Env' is a parameterized set, a map from 'Id' to 'Bindable'.

```
msos Data/Env is
  Env .

Env = (Id, Bindable) Map .
sosm
```

The definition of identifiers make use of a read-only component indexed by 'env'. The evaluation of a binding looks up its value on the 'Env' component and returns the value.

```
msos Cons/Id is
Label = {env : Env, ...} .

    Bindable := lookup (Id, Env)
-------
Id : Id -{env = Env, -}-> Bindable .
sosm
```

The set 'Dec' is the set of *declarations* in the language. Declarations evaluate to bindings, hence the subset inclusion.

```
msos Cons/Dec is
Dec .
Dec ::= Env .
sosm
```

The construction 'bind' declares the binding of an identifier 'Id' and the value resulted from the evaluation of the expression 'Exp'.

Continuing, the module 'simult-seq' defines the simultaneous creation of a set of bindings. This means that the bindings are created independently of each other. This is a restricted variant of the 'simult' construction (not shown here) that evaluates the bindings sequentially, while the general version evaluates the bindings in any order. The bindings are evaluated and then joined together by the '+++' operator at the end.

The construction 'accum' defines a cumulative declaration, a counterpart of the 'simult-seq' construction, where bindings have available to themselves all bindings generated so far.

msos Cons/Dec/accum is

The 'Exp/local' construction takes a declaration and an expression, and evaluates the expression in terms of the declarations. The declaration is first evaluated into 'Env', a set of bindings that are used to override the current environment, 'Env0', that surrounds the construction. These newly created bindings, 'Env'', are used to evaluate each step of the expression 'Exp'. When this expression reaches a final value 'Value', the whole construction evaluates to this value.

```
(local Env Exp) : Exp -{env = Env0, ...}-> local Env Exp' .
  (local Env Value) : Exp --> Value .
sosm
```

1.1.3 Abstractions

We describe here the constructions associated with procedural abstractions, and recursive bindings. We begin by defining the set of abstractions and the set of its formal parameters.

```
msos Cons/Abs is
Abs .
sosm

msos Cons/Par is
Par .
sosm
```

When used to define a language in which abstractions are expressions, the following module must be included.

```
msos Cons/Exp/Abs is
Exp ::= Abs .
Value ::= Abs .
sosm
```

The abstraction is defined using the abstract syntax defined in the module 'Cons/Abs/abs-Exp'. The set 'Passable' is the set of values that may be used as arguments. The application of an abstraction to a "passable value" creates a 'local' declaration, with the passable being bound to the parameter with the 'app' construction described on the module 'Cons/Dec/app'.

```
msos Cons/Abs/abs-Exp is
see Cons/Exp/local, Cons/Dec/app .
see Cons/Exp/Abs, Cons/Exp/app .
Abs ::= abs Par Exp .
```

Closing an abstraction creates a "closure," which is, in essence, an abstraction with an environment.

```
msos Cons/Exp/close is
  see Cons/Abs/closure .

Exp ::= close Abs .

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

(close Abs) : Exp -{env = Env,-}-> (closure Env Abs) .
  sosm
```

The module 'Cons/Abs/closure' contains the definition of the application of a closure to a passable value. Essentially, the environment in the closure is used as an "outer declaration" of the 'local' construction.

The 'Cons/Dec/app' module defines the abstract syntax for the application of an argument to a parameter. This application will be converted into a declaration in which the argument will be bound to the parameter. This construction will be used on the application of closures to expressions.

```
msos Cons/Dec/app is
Dec ::= app Par Arg .
```

```
Arg -{...}-> Arg'
-- ------
(app Par Arg) : Dec -{...}-> (app Par Arg') .
sosm
```

Module 'Cons/Par/bind' defines the actual parameters of abstractions and how they become declarations when applied to arguments. The simplest form of parameter is the 'bind', which takes a single identifier. When applied to a bindable value, it is converted into a binding.

```
msos Cons/Par/bind is
see Cons/Dec/app .

Par ::= bind Id .

(app (bind Id) Bindable) : Dec --> (Id |-> Bindable) .
sosm
```

In order to bind several parameters simultaneously, the following module should be used, which defines the 'tup' parameter construction.

For recursive bindings we use the concept of *finite unfolding*, with reclosures. We define a recursive binding with the construction 'rec' applied before any declaration. This creates a special type of declaration where the first argument is always 'rec Dec'. As this declaration is evaluated to generate bindings, 'rec Dec' is evaluated over and over again, having the effect of an finite unfolding.

1.1.4 Commands

Commands are the imperative facet of CMSOS. We begin by defining the set of commands, 'Cmd'. All commands evaluate to a final value, 'skip', which is the "do nothing" command and has no meaningful value as an expression.

```
msos Cons/Cmd is
  Cmd .
  Cmd ::= skip .
sosm
```

In order to execute a sequence of commands, we use the 'seq-n' construction, which takes as parameter the non-empty list of commands, 'Cmd+', and evaluates each command in order. As each command finishes, that is, evaluates to 'skip', it is removed from this tuple.

```
msos Cons/Cmd/seq-n is
  Cmd ::= seq Cmd+ .
```

The 'effect' construction is the bridge between expressions and commands. It evaluates the expression to a value, and disposes of that value. Hopefully, the expression will have changed some read-write or write-only component, like altering a value in storage, or printing a value. Once a final value is produced by the evaluation of the expression, the command evaluates to 'skip'.

Module 'seq-Cmd-Exp' defines a construction 'seq' which evaluates the sequence of commands given as its first argument and then evaluates the expression, returning its value. It may be regarded as the body of an imperative language function where the last command is a 'return', that returns the value of 'Exp'.

```
(seq skip Exp) : Exp \longrightarrow Exp . sosm
```

The construction defined by the module 'seq-Exp-Cmd' is equivalent the one defined in 'seq-Exp-Cmd' but the expression is evaluated before the commands, but its evaluated value is returned after the evaluation of the list of commands given as the second argument.

The 'cond' construction is a command conditional, that selects which command to execute, depending on the evaluation of its first argument.

The 'while' construction is the traditional looping construction, whose meaning is based on the command sequencing and conditional commands.

```
msos Cons/Cmd/while is
  see Cons/Cmd/cond, Cons/Cmd/seq .

Cmd ::= while Exp Cmd .

(while Exp Cmd) : Cmd -->
  cond Exp (seq Cmd (while Exp Cmd)) skip .
sosm
```

Now we enter the set of constructions that deal with mutable information, modelled through 'Store', a read-write component indexed by 'st' and 'st''. In order to achieve the desired generality, CMSOS uses the set 'Var' to represent the variables that are bound to memory locations (represented by the set 'Cell').

```
msos Cons/Var is
Var .
Var ::= Cell .
sosm
```

The evaluation of a variable as an expression expects it to evaluate to a 'Cell'. Once this happens, it value bound to the cell in the store is returned.

Variables may or may not be related to identifiers. If they are, then we use the following module, which defines the evaluation of identifiers in the context of variables. It is expected that an identifier in this case evaluates to a cell. To appear in environments, a cell must be a 'Bindable' value.

The construction 'assign-seq' changes the value pointed by 'Var' to the value obtained by the evaluation of 'Exp'.

These constructions are complements of each other. The first 'ref' evaluates the variable passed as it sole argument. The second 'deref' evaluates the expression as its sole argument and expects it to be evaluated to a 'ref Cell'. These constructions are used to bring the imperative facet to a functional language by creating a special "value," 'ref' that holds a pointer. On a purely imperative language, these constructions are not necessarily needed, as variables may be used directly. They work as follows: if we want to bind an identifier i to some value ν through the store, we first create a new cell c on the store that is mapped to ν and enclose this cell by the 'ref' construction. The identifier is now bound indirectly to ν . To access this value on a subsequence computation, we will call 'deref i', which will evaluate to a 'deref (ref (c))'. The evaluation of this last expression, according to the rules from the 'Cons/Exp/Var' module, will then evaluate to the desired value ν .

The following construction returns the value bound through the variable 'Var'.

msos Cons/Exp/assigned is

The 'alloc' construction creates a new entry on the store for the value obtained from the evaluation of 'Exp' and then returns a pointer to this entry.

1.1.5 Concurrency

Now we define the concurrency aspect of CMSOS specifications. Let us begin by defining the concept of "complete" concurrent programs, or "systems," represented by the set 'Sys'.

```
msos Cons/Sys is
Sys .
sosm
```

The pool of threads running in a system is bound together by the 'conc' operator. Two rules select nondeterministically which projection should be evaluated at each step. We opted to follow Mosses's specification to the letter here. Had we defined 'conc' a commutative function, only one rule would be necessary. In either case the outcome is the definition of an interleaving model of concurrency.

The construction 'start' signals the creation of a new thread. It is expected that the body of the thread consists of an abstraction that will be applied to the empty tuple upon activation. The module 'Cons/Cmd/start' defines its meaning: after the evaluation of the expression 'Exp', the construction signalizes the creation of a new thread by "producing" the abstraction in the write-only component 'starting'.

Systems are composed of commands. The following module describes the evaluation of a command in the context of a system. If during the execution a command a new thread is detected on the 'starting' component, it is removed from that component and put into the pool of running threads, bound together by the 'conc' construction. If no thread is signalized, then the execution continues as usual. As threads end, i.e., they evaluate to the 'skip' command, they are removed from the pool.

The following modules deal with synchronous message-passing. Let us begin with the creation of channels, represented by the set 'Chan'. Each channel has an unique identifier an integer.

```
msos Data/Chan is
  Chan .
  Chan ::= chan Int .
sosm
```

The 'alloc-chan' creates a new channel to be used and adds it to the read-write 'Chans' components, that keeps track of all channels created so far. The function 'new-chan' is defined externally through equations to simplify the specification, it creates a new channel by looking at the set of channels ('Chans') and returning an unused identification.

In order to model the synchronous message passing of values, CMSOS follows the ideas of Concurrent ML. The write-only component 'event' models the production of events during a computation. Threads block after producing events, waiting for other threads to produce matching events. Concurrent ML defines several different events and their matching relationships, but in this case the only matching events are the 'sending' and 'receiving', which model the sending and receiving of values through a particular channel.

The 'send-chan-seq' function receives two expressions as arguments: the first is evaluated into a channel identification and the second is evaluated to the value that should be sent to that channel. After both expressions are evaluated, it produces the event 'sending' with the channel and value.

```
msos Cons/Cmd/send-chan-seq is
  Cmd ::= send-chan-seq Exp Exp .
```

The construction 'recv-chan' receives a value through the channel that is obtained from the evaluation of the expression 'Exp'. Here we deviate from the original MSDF specification since that used variable unification, a feature not available in Maude as of version 2.1.1. Originally, the function 'recv-chan' also used a free variable 'Value' that unifies when there is a match against a 'sending' event. Our solution is to, after evaluating 'Exp' to a 'Chan', we put a placeholder 'ph Chan' in place of the free variable. When the matching occurs we update this placeholder with the correct value, following the technique we applied while defining a Modular Rewriting Semantics of Concurrent ML in [4]

```
msos Cons/Exp/recv-chan is
  Exp ::= recv-chan Exp .

Label = {event' : Event*, ...} .

Event ::= receiving Chan .
```

Here is the module that describes the matching of events: if a 'sending' and 'receiving' event are produced by any two threads, they synchronize and the value is passed from one thread to another. This is make using a metafunction 'update-ph' that updates the placeholder with the transmitted value. This metafunction iterates over the program text at the metalevel to make the substitution, and for this reason our solution does not depend on the signature of the language and hence does not harm the modularity of the specification.

Finally, we must forbid threads to evaluate if they contain unmatched events. This is achieved by enclosing the entire system with the 'quiet' construction. This construction only let the system evolve when all events have been matched. When this happen the 'event' component is always the empty sequence '()'.

1.2 ML

ML is a subset of Concurrent ML [12] and is the first example of how to give the semantics of a language in terms of CMSOS. The conversion from ML to CMSOS described in this Section follows the one present in [9].

The transformation from ML's syntax to CMSOS abstract syntax is presented using an equation, where the CMSOS equivalent of an ML construction c is denoted between double-bracket parentheses (such as [c]). The equation specifies how each component of a construction is translated to CMSOS constructions. The transformation of an ML construction f that contains parameters x and y is written as: [f(x,y)] = m([x],[y]) meaning that the f construction is converted into an CMSOS construction 'm' that receives the converted parameters x and y from f.

1.2.1 Expressions

We begin by describing ML expressions and their CMSOS counterparts. First we need to gather all CMSOS modules that are necessary for the definitions of expressions in ML. This is done by creating a module 'Lang/ML/Exp' as follows. The module contains explicit references to all CMSOS constructions needed. It also defines that the set of values ('Value') and operators ('Op' are "bindable" in environments, that the set of values are "passable" to procedural abstractions, and that the set of operators contains the constants 'plus', 'times', etc.

The following module contains the initial dynamic basis for ML expressions. It is defined as a system module that includes the MSDF module 'Lang/ML/Exp'. Recall that we must define the operation 'apply-op' externally and this is done here for each 'Op' constant declared on the 'Lang/ML/Exp' module. Following, we create the initial environment with the default associations of identifiers to operators. We reuse the names of the operator as identifiers, created with the coercion function 'ide' on the mapping.

```
mod Lang/ML/Exp' is
 including Lang/ML/Exp .
 including QID .
 vars i1 i2 : Int .
 eq apply-op (plus, (i1, i2)) = i1 + i2.
 eq apply-op (minus, (i1, i2)) = i1 - i2.
 eq apply-op (times, (i1, i2)) = i1 * i2.
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .
 op ide : Qid -> Id .
 op ide : Op \rightarrow Id .
 op op : Qid -> Op .
 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
                ide(gt) |-> gt +++ ide(plus) |-> plus +++
                ide(times) |-> times +++ ide(minus) |-> minus) .
 eq op ('+) = plus .
                       eq op ('*) = times .
 eq op ('-) = minus . eq op ('<) = lt .
 eq op ('>) = gt.
                       eq op ('=) = eq.
endm
```

We begin the grammar by specifying "special constants" ($\langle scon \rangle$), which are currently only integers, represented by the non-terminal 'integer literal'.

► Special constants

```
\langle \operatorname{scon} \rangle \rightarrow \langle \operatorname{integer literal} \rangle
```

Special constants are converted verbatim to CMSOS constructions.

► Infix operators

Next, the rules for "infix operators" ($\langle \mathsf{inop} \rangle$) with are the operators that appear infixed in expressions. They can be either the equals sign or some symbol as defined by the non-terminal $\langle \mathsf{symbolic} \; \mathsf{id} \rangle$. We expect this non-terminal to be provided by the lexical analysis. Both are converted to an 'Op' in CMSOS through the coercion function 'op' that has as argument a quoted-identifier.

$$\langle \mathsf{inop} \rangle \rightarrow \mathsf{'='} \mid \langle \mathsf{symbolic} \mathsf{id} \rangle$$

Let s range over \langle symbolic id \rangle .

$$[=] = op('=)$$

 $[s] = op(s)$

▶ Identifiers

Identifiers defined by the nonterminal $\langle id \rangle$ (also provided by the lexical analysis phase) are converted into 'Id's using the coercion function 'ide' in the same way as the 'op' function.

$$\langle \operatorname{\mathsf{vid}} \rangle \mathop{\rightarrow} \langle \operatorname{\mathsf{id}} \rangle$$

Let i range over $\langle vid \rangle$.

$$[i] = ide(i)$$

► Atomic expressions

Being an abstract syntax version of the ML syntax, we avoid the use of different non-terminals to represent atomic, application, infix and complete expressions. However we opted to use that subdivision for ease of presentation of the specification. These are the rules for atomic expressions, which are special constants, identifiers, and tuple of expressions.

$$\langle \exp \rangle \rightarrow \langle \operatorname{scon} \rangle \mid \langle \operatorname{vid} \rangle \mid \text{`()'} \mid \text{`('} \langle \exp \rangle \text{`)'} \mid \text{`('} \langle \exp \rangle^* \text{`)'}$$

Let e_i range over $\langle \exp \rangle$.

$$[()] = tup()$$

 $[(e)] = [e]$
 $[(e*)] = tup-seq([e*])$

► Application expressions

Application expressions are used to invoke a procedural abstraction and are converted to the CMSOS construction 'app'.

$$\langle \exp \rangle \rightarrow \langle \exp \rangle \langle \exp \rangle$$

$$[e_0 \ e_1] = app \ [e_0] \ [e_1]$$

► Infix expressions

Infix expressions contains the infix operators, which are converted to the application of the operator receiving an argument the sequential tuple formed by both expressions.

$$\langle \exp \rangle \rightarrow \langle \exp \rangle \langle \operatorname{inop} \rangle \langle \exp \rangle$$

Let o range over $\langle inop \rangle$.

$$[e_0 \ o \ e_1] = app \ [o] \ tup-seq([e_0], [e_1])$$

► Complete expressions

Complete expressions are all of the above and the ML constructions for conditionals.

$$\begin{array}{l} \langle \, \exp \, \rangle \, \to \, \langle \, \exp \, \rangle \, \, \, \text{`andalso'} \, \, \langle \, \exp \, \rangle \, \, \, | \, \, \langle \, \exp \, \rangle \, \, \, \\ | \, \, \text{`if'} \, \, \langle \, \exp \, \rangle \, \, \, \text{`then'} \, \, \langle \, \exp \, \rangle \, \, \, \text{`else'} \, \, \langle \, \exp \, \rangle \end{array}$$

1.2.2 Declarations

For declarations, let us introduce the relevant MSDF module that contains all CMSOS constructions related to declarations in ML.

▶ let-Expressions

We begin by extending the atomic expressions with the 'let-in-end' expression, which is mapped into the CMSOS 'local'.

$$\langle \exp \rangle \rightarrow \text{`let'} \langle \text{dec} \rangle \text{`in'} \langle \exp \rangle \text{`end'}$$

Let d range over $\langle dec \rangle$.

$$[let d in e end] = local [d] [e]$$

► Value bindings

Next, the declarations are defined. Currently, ML supports only value bindings, that are converted into the CMSOS 'bind' construction.

$$\langle dec \rangle \rightarrow \text{`val'} \langle vid \rangle \text{'='} \langle exp \rangle$$

$$\llbracket val \ i = e \rrbracket = bind \llbracket i \rrbracket \ \llbracket e \rrbracket$$

1.2.3 Imperatives

ML does not have the concept of a "command," as everything is an expression, but it does have imperative features. Module 'Lang/ML/Cmd' defines this.

Next we add another "external" definition, which is the equation that allocates a new cell on a given store.

```
mod Lang/ML/Cmd' is
  including Lang/ML/Cmd .

var Store : Store .
  eq new-cell (Store) = cell (length (Store) + 1) .
endm
```

► Sequencing of expressions

We begin by revisiting atomic expressions, where we define the syntax of sequencing of expressions. Sequences of expressions are converted into the 'seq-Cmd-Exp' construction, which receives a sequence of commands and a final expression. Each command is an expression enclosed by the 'effect' construction. For example, the sequence of expressions '3;4;1' is converted into 'seq seq (effect (3), effect (4)), 1'.

```
\langle \exp \rangle \rightarrow \text{`('} \langle \exp \text{seq} \rangle \text{`;'} \langle \exp \rangle \text{`)'}
\langle \exp \text{seq} \rangle \rightarrow \langle \exp \rangle \text{`;'} \langle \exp \text{seq} \rangle
```

Let es range over $\langle \exp \operatorname{seq} \rangle$.

$$[(es;e)] = seq (seq [es]) [e]$$

 $[e;es] = (effect [e],[es])$

► Dereferencing of expressions

The dereferencing of expressions is straightforward: we first dereference the expression into a cell with the construction 'deref' and return the assigned value to this cell with the construction 'assigned'.

$$\langle \exp \rangle \rightarrow \text{`!'} \langle \exp \rangle$$
 $[\![!\ e]\!] = \text{assigned (deref ([\![e]\!]))}$

► Referencing of expressions

The application expressions have the additional rule of the referencing of expressions. It first allocates a new cell on the store and encloses this cell with the 'ref' construct so that it becomes a value.

$$\langle \exp \rangle \rightarrow \text{`ref'} \langle \exp \rangle$$
 $\llbracket \text{ref } e \rrbracket = \text{ref (alloc (} \llbracket e \rrbracket))$

► Assignment

The infix expressions now have the assignment operation. Since an assignment in ML does not have any final value, we use the construction 'seq-Cmd-Exp' to first execute the assignment and then to return the empty tuple ('tup()'). The assignment itself is made using the 'assign-seq' construction by first dereferencing the assigned expression.

$$\langle \exp \rangle \rightarrow \langle \exp \rangle$$
 ':=' $\langle \exp \rangle$
$$\llbracket e_0 := e_1 \rrbracket = \text{seq (effect (assign-seq (deref $\llbracket e_0 \rrbracket) \rrbracket e_1 \rrbracket)) tup()}$$$

\blacktriangleright Loops

Finally, we add a construction that is typical of imperative languages, which is the looping command.

```
\langle \exp 
angle 
ightarrow \langle \operatorname{while} 
angle \langle \exp 
angle \ \operatorname{'do'} \langle \exp 
angle \llbracket \operatorname{while} \ e_0 \ \operatorname{do} \ e_1 \rrbracket \ = \ \operatorname{seq} \ (\operatorname{while} \ \llbracket e_0 \rrbracket \ (\operatorname{effect} \ \llbracket e_1 \rrbracket)) \ \operatorname{tup}()
```

1.2.4 Abstractions

Let us introduce the relevant MSDF module, 'Lang/ML/Abs', to introduce the equations for abstractions.

```
msos Lang/ML/Abs is
see Lang/ML/Dec .

see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
see Cons/Par, Cons/Par/bind, Cons/Par/tup .
see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .
sosm
```

► Recursive functions

Our language only has recursive functions. The version here is a very simple for of recursive functions in ML that does not make use of its full pattern matching capabilities (we opted to show an example of anonymous functions—or lambda functions—and pattern matching rules in the semantics of the Mini-Freja language [2]). The new option for the 'dec' nonterminal shows the syntax of recursive functions: the first 'vid' is the name of the function, the second is the single argument, and the 'exp' is the body. It is converted into the binding of the function name to a closure.

1.2.5 Concurrency

Finally, let us present the concurrency primitives of ML. The following module, 'Lang/ML/Conc', gathers the necessary MSDF modules.

► Creating new threads

The operation 'spawn' creates a new thread of execution, and is equivalent to the 'start' construction from CMSOS.

► Creating new channels

The declaration 'chan' creates a new channel and binds to the identifier passed as its argument.

► Sending and receiving through channels

The operations 'send' and 'receive' transmit information over a channel and are implemented, respectively, by the CMSOS constructions 'send-chan-seq' and 'recv-chan'.

► Complete concurrent ML programs

Now, the final rule. All ML programs that are concurrent must be prefixed by 'cml'. It is converted to the 'quiet' CMSOS construction.

$$\langle \exp \rangle \rightarrow \text{`cml'} \langle \exp \rangle$$
 $[\![\text{cml } e]\!] = \text{quiet (effect } [\![e]\!])$

Let us discuss the actual implementation of the parser and the conversion from ML to CMSOS. The syntax of ML is given using a Bison grammar and its semantics is presented as a series of MSDF modules.

We use the Bison productions to generate the CMSOS output for each ML construction, using the 'format' function, which is a function that is similar to the C function 'sprintf', with the difference that it allocates a pointer and returns the string created according to the formatting specified. For example, let us show the actual rules for the conversion of conditional expressions in ML: The symbol '\$\$' is defined as "the semantic value of the left-hand side of the rule" by the Bison manual. In this specification it will hold the string that contains the translation from ML into CMSOS. The symbol '\$1' refers to the matched token on the rule.

When the parser reaches the topmost production on the grammar, the output string is the converted CMSOS code. The grammar of this version of ML is simple enough so that there was no need to use an abstract version in the translation to CMSOS, as it was the case with the MiniJava language (Section 1.3).

1.2.6 Example

As an example of this semantics, let us analyze a concurrent program. It starts as a single thread that creates a channel through the declaration 'chan' and binds it to the variable 'c'; then, in turn, spawns three new threads: the first sends the value 10 to the channel 'c', the second sends the value 20 through the same channel, while the third expects to receive a value through 'c'. If we search through all possible outcomes of this program it is expected that there are two final states: one in which the first thread had a successful synchronization with the third thread and another in which the second thread was the successful one.

```
cml
  let chan c in
      (spawn (fn x => send (c, 10));
      spawn (fn x => send (c, 20));
      spawn (fn x => receive c))
end
```

After the conversion to CMSOS, we have the following program.

By searching through all possible final states using the 'search' command from Maude we arrive at the expected situation. The first solution shows the remaining, unsynchronized thread stopped at the point where it is trying to send the value 10 through the channel, while the second shows the same with the value 20. (As threads end, they are removed from the configuration.)

```
search in CML-INTERPRETER : exec(...) =>!
```

```
C:Conf .
Solution 1
C:Conf <- <
 quiet
  effect (
   local (ide('c)|-> chan 1)
    local (ide('x)| \rightarrow tup())
     seq send-chan-seq chan 1 10 tup())
 ::: 'Sys, {chans = {chan 1}, env = void, starting' = (),
             event' = (), store =void}
>
Solution 2
C:Conf <- <
quiet
  effect (
   local (ide('c)|-> chan 1)
    local (ide('x)| \rightarrow tup())
     seq send-chan-seq chan 1 20 tup())
 ::: 'Sys, {chans = {chan 1}, env = void, starting' = (),
             event' = (), store =void}
>
```

No more solutions.

1.3 MiniJava

Our implementation of MiniJava into the framework of Constructive MSOS follows the idea that a (simple) object-oriented language is, in its essence, an imperative language in which classes are types, and objects are records [13, 10, 8, 11]. The actual implementation of objects is based on [14].

As an introduction, let us begin with a trivial, abstract, mapping and then expand it with more advanced features, such as recursive reference, and object instantiation. A straightforward view of objects is to consider them as records, whose fields are the methods of the object. We may simplify this even further by using tuples and keeping track of which method corresponds to each projection. As an example, let us consider an object specified in some abstract, Java-like, language:

```
object {
  int i;
  int foo() { return i; }
  int bar() { i := i + 1; return i; }
}
```

It may be represented as a closure, namely, a tuple surrounded by an environment.

```
local (bind x 1) (tup-seq f, b)
```

Here f and b are closures that represent, respectively, methods 'foo()' and 'bar()' from the object above. We omit the abstract representation of those for brevity. In this mapping, all fields of the object are declared as bindings that "surround" the tuple. With this scheme, bindings are not allowed external access directly. It is possible, however, to allow direct access to bindings by creating access functions automatically for each field.

In order to call a method in the object, we obtain the desired projection and evaluate it: if we want to call method 'bar()', we first obtain the second projection, by applying the operation 'nth(1)' to the tuple, and then evaluate it, applying the resulting closure to its arguments, the empty tuple in this example.

```
app (app nth(1) o) tup()
```

where o is an object.

This is a severely limited form of object-orientation: instance methods are not allowed to be recursive, nor access other methods on the same object. We now add a form of self-reference (e.g., 'this' in Java) so that a method may call other methods inside the same object. This is achieved by adding a recursive function 'self' which, when evaluated, returns the object. Being recursive, a method may call 'self' from within itself, thus having access to the other methods on the object, including itself.

The above code means that a closure is bound recursively to the identifier 'self' (the closure receives no arguments, specified as 'bind tup()', in CM-SOS); the closure code is, as before, the tuple containing the methods. This closure is defined in accumulation of 'bind x 1' making those fields available to the methods f and b.

Being a recursive binding, f and b may also call 'self'. Self-reference within a method is achieved by calling 'self' from within the method and evaluating the desired projection.

The remaining issue is how to instantiate an object based on the type declaration of its class. We follow the approach of object cloning using prototype objects, in the tradition of the Self language [15]. The actual implementation is as follows: each class declaration gives rise to a function declaration that, when evaluated, returns a new object. Thus, a class C is converted into the following fragment:

```
bind C (close abs (bind tup()) o)
```

where o is a prototype object created from the class declaration. Object instantiation is then converted to: (app C tup()), which returns a copy of the object o.

With this preliminary exposition, we may fully describe the mapping from MiniJava to CMSOS.

The transformation from MiniJava's abstract syntax to CMSOS constructions is presented using the same notation we used for the ML language on Section 1.2. The actual implementation of the converter was made using the SableCC framework due to the same reasons we outlined in the description of the ML language semantics.

1.3.1 Expressions

Expressions consist of mathematical operations, identifiers, method invocations (that always return a value), literals, and objects themselves.

```
\langle \exp \rangle \rightarrow \langle \text{ math operation } \rangle \mid \langle \text{id } \rangle \mid \langle \text{ method invocation } \rangle \mid \langle \text{ literal } \rangle \mid \langle \text{ this } \rangle \mid \langle \text{ new } \rangle
```

► Math operations

$$\langle$$
 math operation $\rangle \rightarrow \langle \exp \rangle \langle$ math op $\rangle \langle \exp \rangle \langle$ math op $\rangle \rightarrow \langle \&\&' | '<' | '+' | '/' | '%' | '-' | '*' | '>' | '<=' | '>=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '=' | '$

Let e_i range over $\langle \exp \rangle$, and m range over $\langle \operatorname{math} \operatorname{op} \rangle$.

$$[e_0 \ m \ e_1] = app [m] tup-seq ([e_0], [e_1])$$

► Identifiers

Let i range over $\langle id \rangle$. When i is not the left-hand side of an assignment:

$$[i]$$
 = assigned (deref i)

Otherwise, it is as follows:

$$[i]$$
 = deref i

► Method invocations

$$\langle \ \mathsf{method} \ \mathsf{invocation} \, \rangle \,{\to}\, \langle \ \mathsf{exp} \, \rangle \, \, `.\, ' \, \, \langle \ \mathsf{id} \, \rangle \, \, \text{`('} \, \, \langle \ \mathsf{exp} \, \rangle^{*`}\text{)'}$$

$$\llbracket e : i (e*) \rrbracket = app (app nth(n) \llbracket e \rrbracket) (tup-seq p)$$

The evaluation of e must return an object; n is the method number in the class of the object returned by e, obtained by looking up the method name i in the metaclass information generated in the static analysis phase of the compilation process; and p is constructed as a sequence of ref (alloc $[e_i]$) which allocates a new memory entry for each parameter e_i in e*.

► Literals

```
\langle | \text{literal} \rangle \rightarrow \langle | \text{boolean literal} \rangle | \langle | \text{integer literal} \rangle \rangle
\langle | \text{boolean literal} \rangle \rightarrow \langle | \text{true'} | \langle | \text{false'} \rangle \rangle
```

Literals are converted verbatim to CMSOS constructions.

► Self-reference

$$\langle \, {\sf this} \, \rangle \, {\to} \, {\rm `this'}$$

$$[\![{\sf this}]\!] \, = \, {\sf app \ self \ tup()}$$

▶ Object instantiations.

$$\langle \, \mathsf{new} \, \rangle \,{\to}\, {}^{\backprime} \mathsf{new} \, {}^{\backprime} \, \langle \, \mathsf{id} \, \rangle \, \, {}^{\backprime} () \, {}^{\backprime}$$

$$[new i ()] = app i tup()$$

1.3.2 Statements

MiniJava contains the usual statements of imperative programming languages: conditionals, loops, output, assignment, etc.

$$\langle \, \mathsf{statement} \, \rangle \,{\to}\, \langle \, \mathsf{if} \, \rangle \, | \, \langle \, \mathsf{while} \, \rangle \, | \, \langle \, \mathsf{block} \, \rangle \, | \, \langle \, \mathsf{print} \, \rangle \, | \, \langle \, \mathsf{assign} \, \rangle \, | \, \langle \, \mathsf{empty} \, \rangle$$

► Conditionals

$$\langle if \rangle \rightarrow 'if' \langle exp \rangle 'then' \langle statement \rangle 'else' \langle statement \rangle$$

Let s_i range over \langle statement \rangle .

$$[\![\text{if }e\text{ then }s_0\text{ else }s_1]\!] = \text{cond }[\![e]\!] [\![s_0]\!] [\![s_1]\!]$$

► Loops

$$\langle \, \mathsf{while} \, \rangle \longrightarrow \text{`while'} \, \, \text{`('} \, \, \langle \, \mathsf{exp} \, \rangle \, \, \text{`)'} \, \, \langle \, \mathsf{statement} \, \rangle$$

$$\llbracket \mathtt{while} \ (\ e\)\ s \rrbracket \ = \mathtt{while} \ \llbracket e \rrbracket \ \llbracket s \rrbracket$$

► Block statements

$$\llbracket \{ \ s * \ \} \rrbracket \ = \, \mathtt{seq} \ \llbracket s * \rrbracket$$

▶ Output

$$\langle \operatorname{print} \rangle \rightarrow \operatorname{`System.out.println'} \operatorname{`('} \langle \exp \rangle \operatorname{`)'}$$

$$[\![\operatorname{System.out.println} (e)]\!] = \operatorname{print} [\![e]\!]$$

ightharpoonup Assignment

$$\langle \operatorname{assign} \rangle \to \langle \exp \rangle$$
 '=' $\langle \exp \rangle$
$$\llbracket e_0 = e_1 \rrbracket = \operatorname{effect (assign-seq } \llbracket e_0 \rrbracket \ \llbracket e_1 \rrbracket)$$

► Empty statement

$$\langle \, \mathsf{empty} \, \rangle \,{
ightarrow} \, \, `; \, `$$

$$[;]$$
 = skip

1.3.3 Classes

► Class declaration

As described at the beginning of this Section a class declarations defines a "prototype" object, which is a closure where the fields become bindings and the methods become projections of a tuple.

```
 \begin{array}{l} \langle\, {\rm class}\,\, {\rm declaration}\, \rangle \, \rightarrow \, {\rm `class'}\,\, \langle\, {\rm identifier}\, \rangle\,\, {\rm `\{'}\,\, (\langle\, {\rm field}\,\, {\rm declaration}\, \rangle)^* \\ (\langle\, {\rm method}\,\, {\rm declaration}\, \rangle)^*{\rm `\}'} \end{array}
```

Let f_i range over \langle field declaration \rangle and m_i over \langle method declaration \rangle .

► Main class declaration

The main difference is the lack of field declarations and the existence of a single method.

► Field declarations

The type information is used only on the static analysis phase. Since the only primitive type is the integer, we bind the identifier \mathbf{i} to a newly allocated cell, with the default value of zero.

```
\label{eq:continuous} \begin{array}{l} \langle \mbox{ field declaration} \, \rangle \, \! \to \, \langle \mbox{ type} \, \rangle \, \, \langle \mbox{ identifier} \, \rangle \\ \\ \mbox{Let } t \mbox{ range over } \langle \mbox{ type} \, \rangle. \\ \\ \mathbb{[} t \mbox{ } i \mathbb{]} \mbox{ } = \mbox{ bind } i \mbox{ (ref (alloc 0))} \end{array}
```

► Method declaration

A method declaration is converted into a closure with parameters being bound simultaneously by the 'tup' construction. The closure body is a 'local' definition with the variable declarations as the declaration part the method body as the expression being evaluated. We use the 'seq-Cmd-Exp' command so that the last expression evaluated is returned as the method return value.

```
\langle \text{ method declaration } \rangle \rightarrow \langle \text{ type } \rangle \langle \text{ identifier } \rangle  (' (\langle \text{ parameter } \rangle)*')'

(\langle \text{ (} \langle \text{ var declaration } \rangle)*(\langle \text{ statement } \rangle)*'return' \langle \text{ expression } \rangle'}'
```

Let p_i range over \langle parameter \rangle .

```
\llbracket \text{t i (p*) } \{ v* s* \text{ return } e \} \rrbracket = \text{close (abs tup(} \llbracket p* \rrbracket) (local (accum } \llbracket v* \rrbracket) (seq (seq <math>\llbracket s* \rrbracket) e)))
```

► Main method declaration

```
\langle main method declaration \rangle \rightarrow 'public static void main (String arg[]) \{ '(\langle statement \rangle)*'\}'
```

```
[public static void main (String arg[]) \{s*\}] = close (abs tup(@) (local void (seq (seq [s*]) 0)))
```

The main method is similar to the generic method declaration, except that it does not take arguments, variables declarations, nor has any return expression. In this case the return expression is assumed to be zero.

► Complete program

Now we must close this specification by creating the main body of the program. It consists of a series of bindings from the class names to the prototype objects obtained from the class declarations. The body is a call to the zeroth projection of the main class, which is exactly the main method.

Let cd_i be a class, not the main, declaration and exec the body of the main program.

$$[goal] = local (accum $[cd*]) [exec]$$$

The conversion of the class declarations is a series of bindings from class names (represented by n_i) to the prototype objects (represented by o_i). Thus, for some class declaration i, $[cd_i]$ is as follows:

$$[cd_i] = bind n o_i$$

Now, the equation for exec. It is a call to the main method of the main class. Even though this method does not receive any arguments, an empty tuple with a single reference is passed as a "dummy" parameter. The prototype object of the main class is represented by o_m .

```
[exec] = (app (app nth(0) o_m) (tup-seq (ref (alloc 0))))
```

To complete the semantics and show an example of execution we show next the MSDF module that gathers all relevant CMSOS constructions necessary for the MiniJava language:

```
msos MiniJava is
 see Cons/Prog, Cons/Prog/Exp .
 see Cons/Exp, Cons/Exp/Boolean, Cons/Exp/Int, Cons/Exp/local,
     Cons/Exp/Id, Cons/Exp/cond, Cons/Exp/app-Op,
     Cons/Exp/app-Id, Cons/Exp/tup, Cons/Exp/tup-seq .
 see Cons/Arg, Cons/Arg/Exp .
 see Cons/Op .
 see Cons/Id .
 see Cons/Prog, Cons/Prog/Dec .
 see Cons/Dec, Cons/Dec/bind, Cons/Dec/simult-seq,
     Cons/Dec/accum, Cons/Dec/local .
 see Cons/Exp, Cons/Exp/local .
 see Cons/Cmd, Cons/Cmd/seq-n, Cons/Cmd/effect, Cons/Cmd/while,
     Cons/Cmd/print .
 see Cons/Exp, Cons/Exp/seq-Cmd-Exp, Cons/Exp/seq-Exp-Cmd,
     Cons/Exp/assign-seq, Cons/Exp/ref, Cons/Exp/assigned .
 see Cons/Var, Cons/Var/alloc, Cons/Var/deref .
 see Cons/Exp/ref .
 see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
 see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
 see Cons/Par, Cons/Par/bind, Cons/Par/tup .
 see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .
 Bindable ::= Value | Op .
 Op ::= nth (Int) | plus | times | minus | eq | lt | gt .
 Passable ::= Value .
 Storable ::= Value .
sosm
```

Next, the following code defines the initial basis for MiniJava programs, with the initial record and the 'apply-op' and 'new-cell' equations. We

also define an 'output' function which receives a configuration as argument and that, once the computation end with a 'skip' command, it removes the output that appears on the write-only component 'out'.

```
mod MiniJava' is
 including MiniJava .
 var I : Int .
 var V : Value .
 var VL : Seq'(Value') .
 op init-rec : -> Record .
 eq init-rec = { out' = (()).Seq'(Value'),
                 env = (void).Map'(Id'|'Bindable'),
                 store = (void).Map'(Cell'|'Storable') } .
 vars i1 i2 : Int .
 eq apply-op (plus, (i1, i2)) = i1 + i2.
 eq apply-op (minus, (i1, i2)) = i1 - i2.
 eq apply-op (times, (i1, i2)) = i1 * i2.
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .
 eq apply-op (nth (0), (V, VL)) = V.
 ceq apply-op (nth (I), (V, VL))
 = apply-op (nth (I - 1), (VL))
 if I > 0.
 op ide : Qid -> Id .
 op ide : Op \rightarrow Id .
 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
                ide(gt) |-> gt +++ ide(plus) |-> plus +++
                ide(times) |-> times +++ ide(minus) |-> minus) .
 var Store : Store .
 eq new-cell (Store) = cell (length (Store) + 1) .
```

```
sort Output .
op output : Conf -> Output .
op output : Seq'(Value') -> Output .

rl output(< V:Value ::: 'Exp, { out' = VL:Seq'(Value'),
        PR:PreRecord } >) => output(VL:Seq'(Value')) .
endm)
```

1.3.4 Example

As an example of the translation, let us consider a class that calculates the factorial of a number. This class implements actually two forms of calculating the factorial: the one implemented by the 'RecFat' is recursive, while the one implemented by 'NonRecFac' is direct, using a loop. The code is as follows:

```
class Factorial
 public static void main (String[] arg)
    System.out.println (new Fac().Test (6));
}
class Fac
  public int Test (int num)
  {
    int r;
    Fac recfac;
   Fac nonrecfac;
    recfac = this;
    nonrecfac = this;
    return recfac.RecFac(num) - nonrecfac.NonRecFac (num);
  }
 public int RecFac(int num)
  {
```

```
int num_aux;
    if (num < 1) num_aux = 1;
    else num_aux = num * (this.RecFac(num-1));
    return num_aux;
  }
 public int NonRecFac (int num)
    int i;
    int fat;
    i = num;
    fat = 1;
    while (i > 0)
        fat = fat * i;
        i = i - 1;
    return fat;
  }
}
```

And the converted code to CMSOS is:

local accum bind Fac close (abs bind @ local accum void rec (bind self close (abs bind @ tup-seq (close (abs tup bind num local accum bind r ref alloc 0 accum bind recfac ref alloc 0 accum bind nonrecfac ref alloc 0 void seq seq (effect (assign-seq deref recfac app self tup ()),effect (assign-seq deref nonrecfac app self tup ()),skip) app minus tup-seq ((app app nth(1) assigned deref recfac tup-seq ref alloc assigned deref num),app app nth(2) assigned deref nonrecfac tup-seq ref alloc assigned deref num)),close (abs tup bind num local accum bind num_aux ref alloc 0 void seq seq ((cond app lt tup-seq (assigned deref num,1) effect (assign-seq deref num_aux 1) effect (assign-seq deref num_aux app times tup-seq (assigned deref num,app app nth(1) app self tup () tup-seq ref alloc (app minus tup-seq (assigned deref num,1)))),skip)

assigned deref num_aux), close (abs tup bind num local accum bind i ref alloc 0 accum bind fat ref alloc 0 void seq seq (effect (assign-seq deref i assigned deref num), effect (assign-seq deref fat 1), (while app gt tup-seq (assigned deref i,0) seq (effect (assign-seq deref fat app times tup-seq (assigned deref fat, assigned deref i)), effect (assign-seq deref i app minus tup-seq (assigned deref i,1))), skip) assigned deref fat))) app self tup ()) void app app nth(0) local accum void rec (bind self close (abs bind @ tup-seq close (abs tup bind @@ local void seq seq (print (app app nth(0)) app fac tup () tup-seq ref alloc 0, skip) 0))) app self tup () tup-seq ref alloc 0

Executing it under Maude, the following output is produced:

Now, let summarize the actual implementation. The option to use SableCC, which is also a LALR(1) parser generator, was due to SableCC's support for the transformation from concrete syntax to abstract syntax directly in the grammar file using a somewhat "term rewriting" style, which simplifies the construction of the compiler enormously, since MiniJava has a rather complex concrete grammar. This choice also enabled us to reuse the already existing Java 1.1 grammar created by Etienne Gagnon to create the grammar of MiniJava, which is a sub-language of Java.

The process is as follows. First, the Java 1.1 grammar was modified to exclude the syntax not supported by MiniJava. Next, an abstract syntax of MiniJava, based on the abstract syntax described in Appel's book was created and the concrete grammar was modified to add the rewriting rules that convert from the concrete to the abstract syntax. When this file is processed, SableCC generates tree-walker classes that uses the *visitor* design pattern. This tree-walker is then implemented by the programmer by creating a subclass that adds the appropriate actions that must be performed as the walker visits each node.

The actual compilation process consists of two phases: the static analysis, where all types are checked, and metadata is generated. The second phase makes another visit to the tree, converting each node into its CMSOS equivalent. As in the case of the ML language, instead of constructing the CMSOS code in Java, we could have just exported the abstract syntax tree and let Maude equations do the conversion. In this first implementation, we opted to leave the MiniJava compiler self-contained.

Let us illustrate this description with a simple example, which shows how arithmetic expressions are compiled. We begin with the abstract syntax of arithmetic expressions. In the fragment shown below, the 'expression' non terminal has a rule named 'math_operation'. The rule name is important as it will be used to generate the node classes. An arithmetic expression is two expressions with an infixed mathematical operator, represented by the non-terminal 'math_op'. The prefixes '[lh]' and '[rh]' have the purpose of identifying each expression on the rule for code-generation purposes.

Now, let us show the concrete syntax for these, along with the rewriting rules that generate the abstract syntax. We show only the rule for "additive expressions," since, due to operator precedence issues, the concrete grammar has nine different rules. On the grammar below an 'additive_expression' is converted to the abstract 'expression'. It has three alternatives: it is either a 'multiplicative_expression' (not shown here), in this case it is converted to whatever 'multiplicative_expression' is converted. It can be also the 'plus' option, in which a new abstract syntax node 'math_operation' is created by converting recursively each side of the additive expression described. The same happens for the 'minus' alternative.

Now we show the fragment from the tree-walker code that passes through the 'math_operation' node. The name of the method is automatically generated by SableCC and it receives as parameter a node that is the representation of the 'math_operation' production. Through getters and setters, each component of the syntax tree is accessed. The conversion is achieved by keeping a dictionary 'cmsos' that maps every node to its CMSOS equivalent. Thus, by looking up the 'lh' and 'rh' expressions (using the functions 'n.getLH()' and 'n.getRh()'), we obtain the representation of each expression. After we have all this information at hand, we put back into the dictionary the mapping from the current node, the mathematical operation, to its CMSOS equivalent.

```
public
void outAMathOperationExpression (AMathOperationExpression n)
{
   String l, r;
   PMathOp o = n.getMathOp ();

   String[] args = { (String) cmsos.get (n.getMathOp ()),
        (String) cmsos.get (n.getLh ()),
        (String) cmsos.get (n.getRh ()) };

   cmsos.put (n, printf ("(app {0} tup-seq ({1}, {2}))", args));
}
```

References

- [1] Andrew W. Appel. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge University Press, Cambridge, UK, February 1997.
- [2] Fabricio Chalub. MMT case study: the Mini-Freja language. http://mmt.ic.uff.br/case-study-mf.pdf.
- [3] 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.
- [4] Fabricio Chalub and Christiano Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789-807, July 2004. http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics.
- [5] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, April 2003.
- [6] Jørgen Iversen. Formalisms and tools supporting Constructive Action Semantics. PhD thesis, Univ. of Aarhus, 2005.
- [7] Jørgen Iversen and Peter D. Mosses. Constructive action semantics for core ML. *IEE Proceedings*, 152(2), April 2005. Special issue on Language definitions and tool generation.
- [8] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.
- [9] Peter D. Mosses. Fundamental Concepts and Formal Semantics of Programming Languages—an introductory course. Lecture notes, available at http://www.daimi.au.dk/jwig-cnn/dSem/, 2004.
- [10] Benjamin C. Pierce. Types and Programming Languages. MIT Press, 2002.

- [11] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 289–297, Snowbird, Utah, July 1988.
- [12] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.
- [13] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.
- [14] Lars Thorup and Mads Tofte. Object-oriented programming and Standard ML. In John H. Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida*, number 2265 in Rapport de recherche, pages 41–49. INRIA, June 1994.
- [15] David Ungar and Craig Chambers. Self: The power of simplicity. In Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices, number 12, pages 227–242. ACM, 1987.